EDISI PERTAMA

DART 3.x

PANDUAN BELAJAR BAHASA PEMROGRAMAN CROSS-PLATFORM DENGAN DART

BAGUS AJI SANTOSO
ngoding.dev/dart

Belajar Dart 3.x

Panduan belajar bahasa pemrograman cross-platform dengan Dart.

Bagus Aji Santoso

This book is available at http://leanpub.com/belajardart

This version was published on 2024-10-29

Contents

Persiapan

Sebelum bisa menulis satu baris kode Dart, kita perlu menyiapkan perangkat lunak yang dibutuhkan. Bab ini ditulis untuk membantu pembaca menyiapkan *development environment* yang dibutuhkan agar bisa mengikuti pembahasan pada bab-bab selanjutnya.

Apa Saja yang Dibutuhkan?

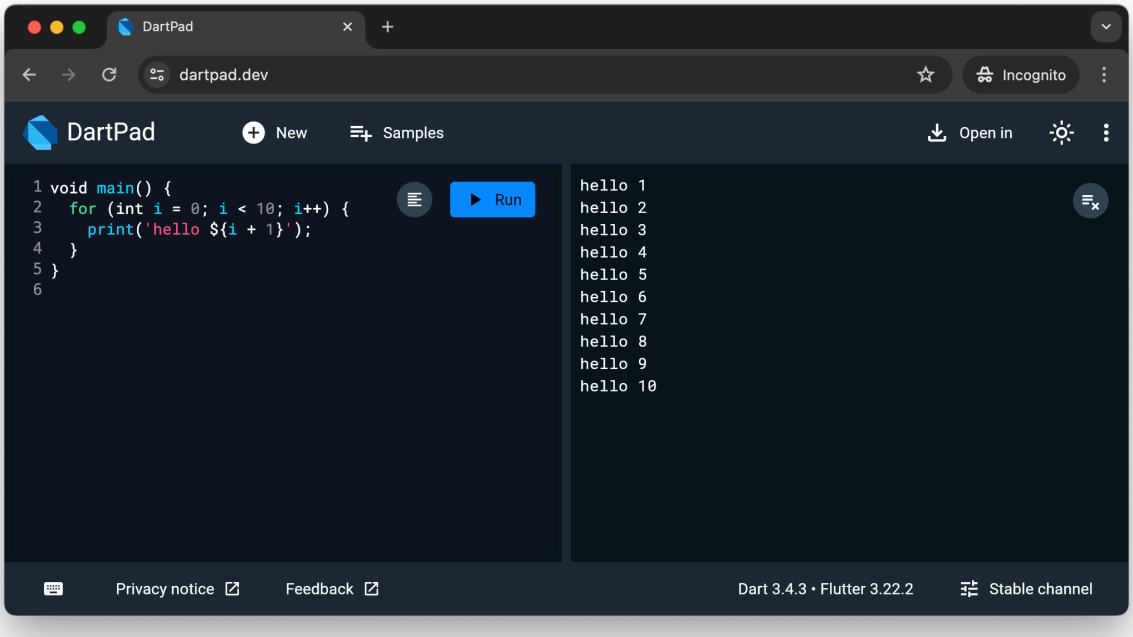
- Komputer atau laptop dengan spesifikasi yang cukup.
- Editor
- Flutter SDK
- Smartphone atau bisa juga menggunakan iOS Simulator dan Android emulator bila tidak memiliki perangkat smartphone.

Editor

Ada beberapa opsi yang bisa dipakai untuk menulis program Dart/Flutter diantaranya:

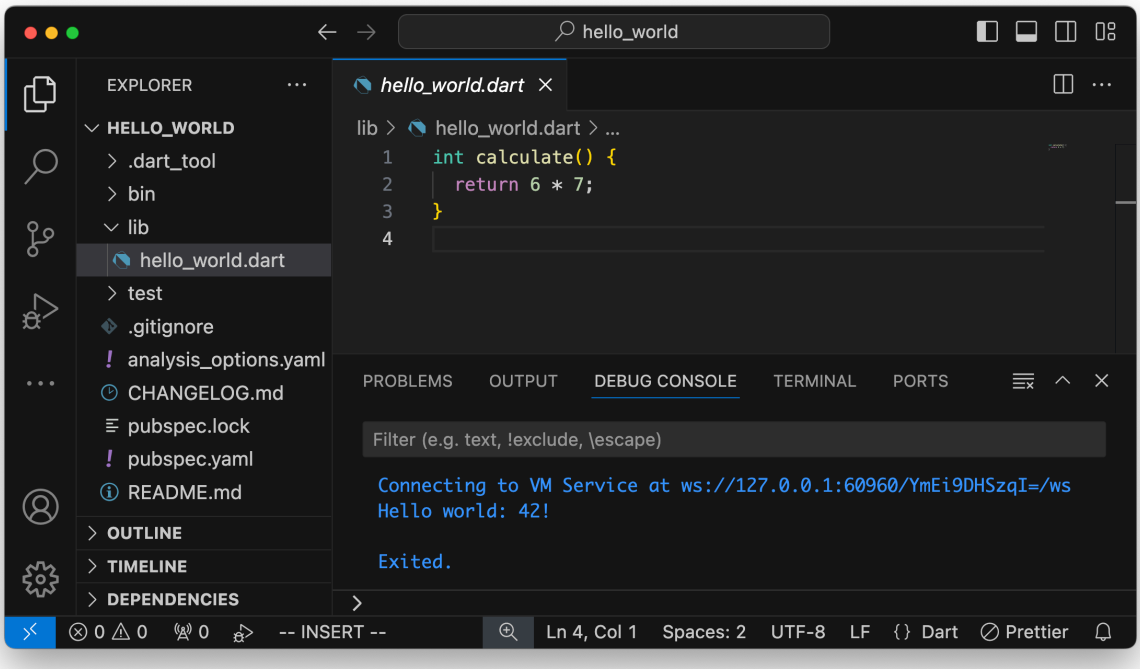
DartPad adalah aplikasi web yang bisa dibuka lewat browser. Web ini dikembangkan langsung oleh tim Google. Cocok untuk sekedar memeriksa potongan kode Dart, namun kurang baik untuk dipakai untuk menulis program yang lebih kompleks. Akses lewat dartpad.com[1].

[1] https://dartpad.dev/

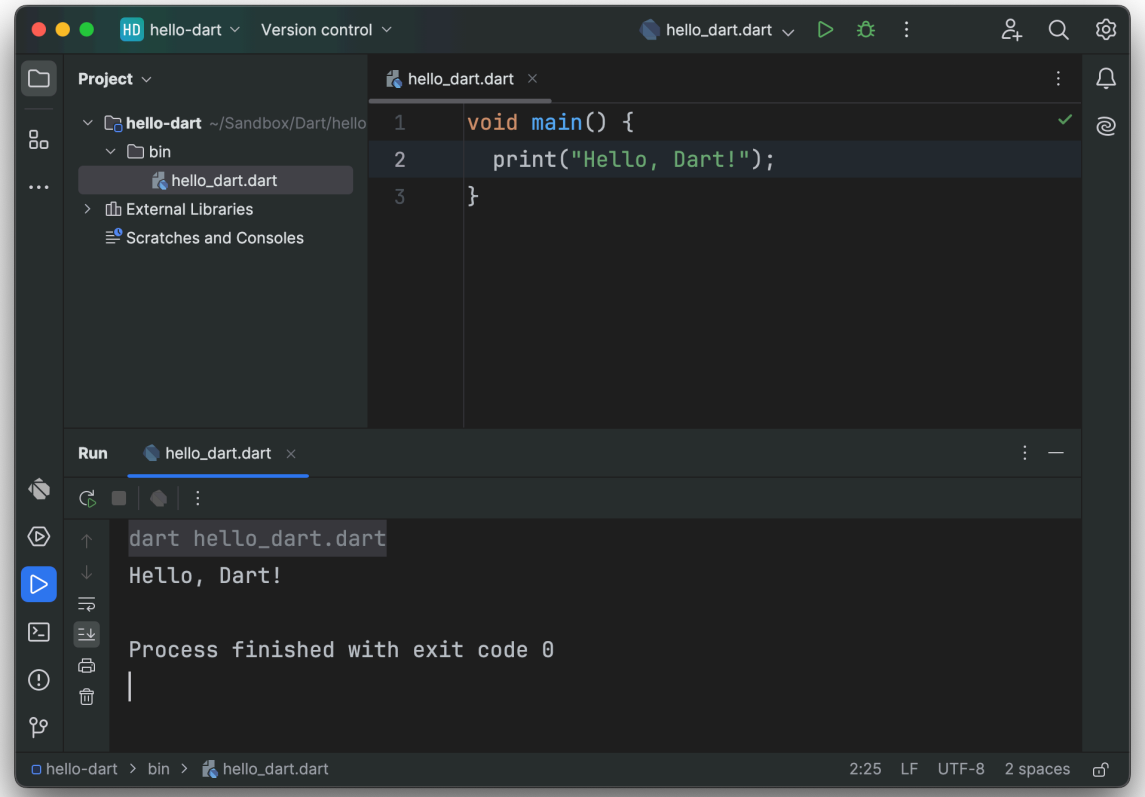


Visual Studio Code adalah teks editor yang cukup populer untuk menulis program dalam bahasa pemrograman apapun. Berkat *extension* Dart Language Support[2] menulis program Dart di editor ini menjadi lebih mudah.

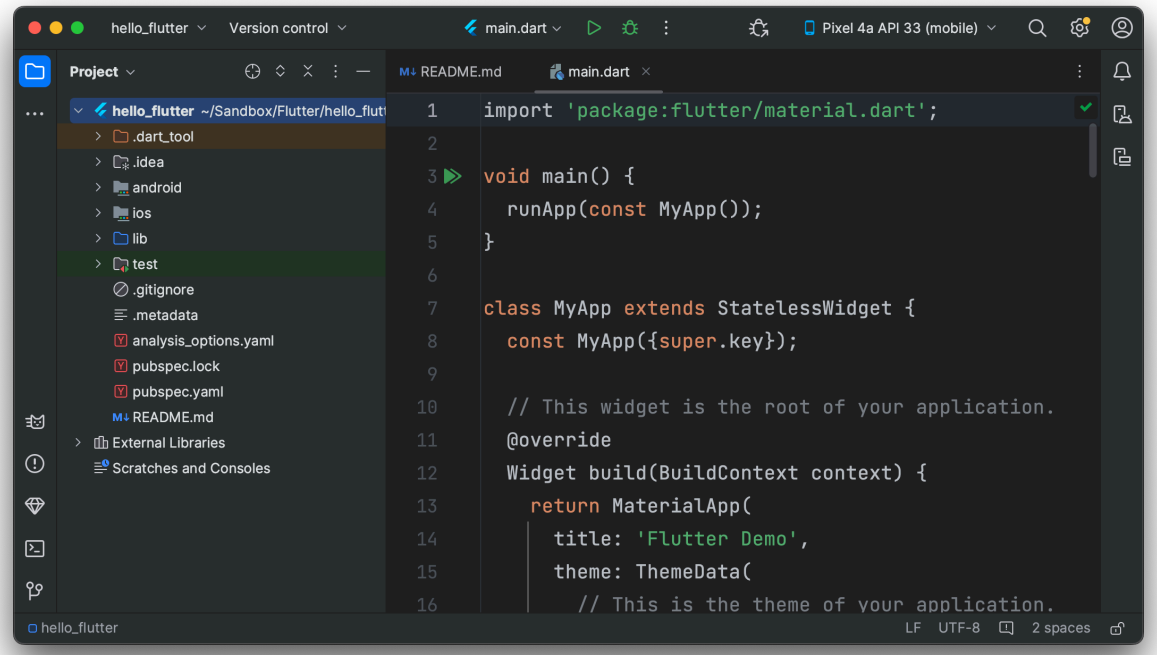
[2] https://dartcode.org/



IntelliJ IDEA merupakan *integrated development environment* (IDE) yang mendukung pengembangan aplikasi Dart dengan tambahan plugin. IntelliJ merupakan basis yang dipakai oleh Android Studio, sehingga kemampuannya tidak perlu diragukan.



Android Studio merupakan editor yang dikembangkan khusus oleh Google untuk pengembangan aplikasi native Android dengan Kotlin atau Java. Namun, dengan tambahan plugin, kita bisa menggunakan Android Studio untuk pengembangan aplikasi Flutter.



Xcode merupakan editor *official* dari Apple untuk pengembangan aplikasi iOS dengan Swift atau Objective C. Kita tidak bisa menggunakan Xcode untuk menuliskan program Flutter, tapi bila ingin agar program Flutter kita bisa diakses dari perangkat Apple, maka kita tetap membutuhkan Xcode.

Untuk melewati pembahasan pada bagian pertama buku ini (Dart), penulis menyarankan penggunaa penggunaan Visual Studio Code. Selain lebih ringan, contoh-contoh program yang relatif ringkas pada bagian pertama ini akan lebih cepat dan mudah di tulis dengan VS Code.

Sementara itu, untuk pembahasan pada bagian kedua buku ini (Flutter) penulis akan menggunakan Android Studio. Pembaca tetap bisa mengikutinya dengan menggunakan VS Code, namun seluruh tangkapan layar dan contoh pada bagian kedua akan mengasumsikan penggunaan Android Studio.

Memasang Visual Studio Code

Visual Studio Code merupakan editor *cross-platform*, artinya ia tersedia untuk berbagai sistem operasi baik itu Windows, Linux maupun MacOS. VS Code tersedia secara gratis tanpa *embel-embel* apapun, tidak ada batasan fitur, tidak ada batasan waktu, dan tidak ada *popup* yang akan meminta pembelian lisensi pro.

VS Code bisa diunduh langsung di laman code.visualstudio.com/[3]. Cara memasangnya sangat mudah, cukup ikuti proses pemasangan yang standar dilakukan sama seperti aplikasi-aplikasi lain.

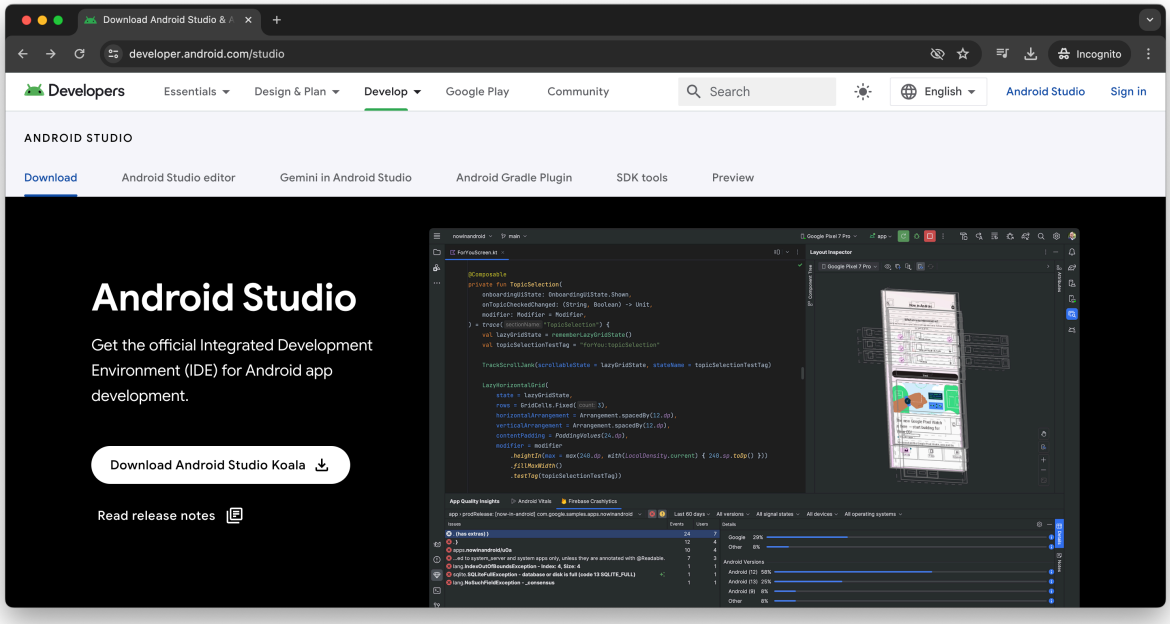
[3] https://code.visualstudio.com/

Penulis yakin, pembaca akan mampu memasang Visual Studio Code secara mandiri meskipun tanpa panduan langkah demi langkah.

Memasang Android Studio

Flutter secara resmi mendukung tidak editor yaitu Android Studio, Visual Studio Code dan Emacs. Contoh tangkapan layar pada bagian kedua buku ini (Flutter) secara khusus menggunakan Android Studio, namun semua contoh kode tersebut tetap bisa dijalankan di Visual Studio Code tanpa perubahan apapun.

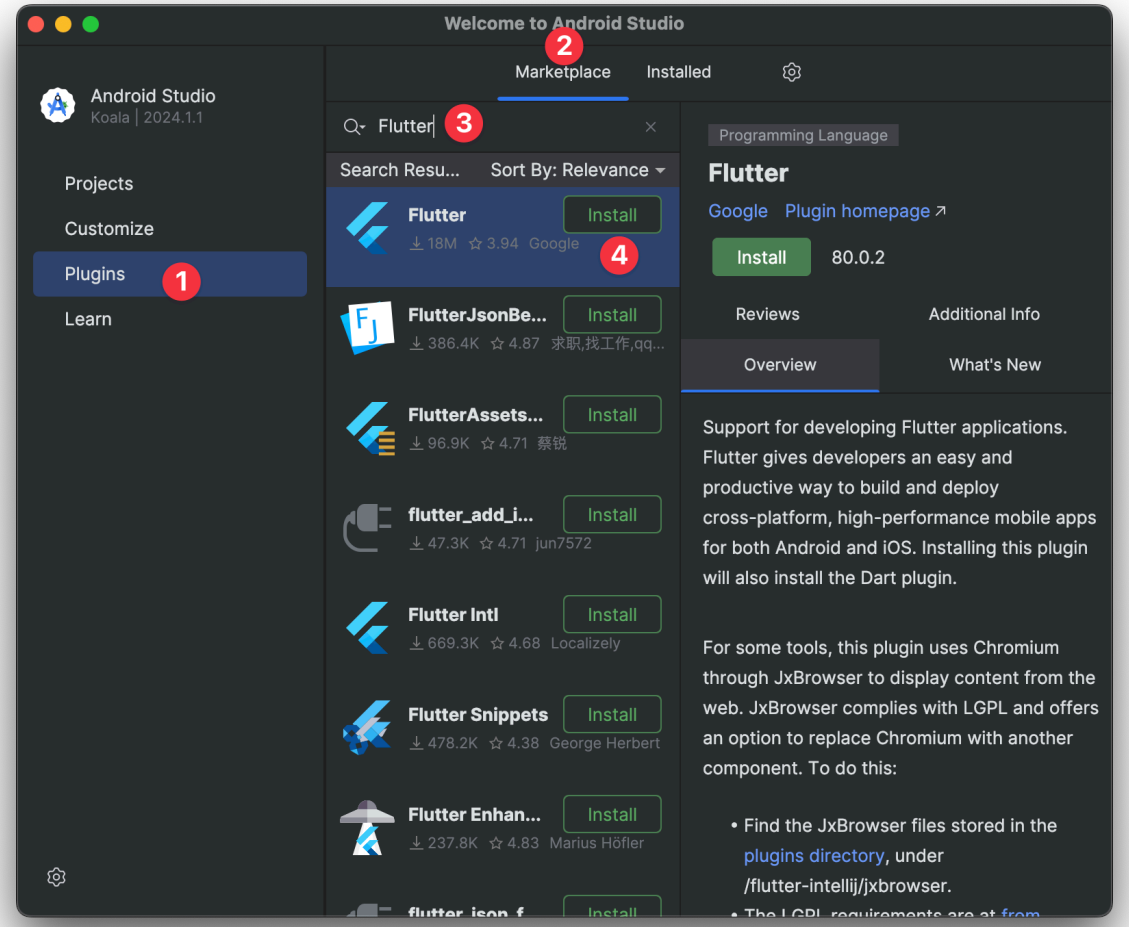
Kunjungi laman resmi developer.android.com/studio untuk mengunduh Android Studio versi terbaru. Ketika tulisan ini dibuat, versi terakhirnya adalah **Android Studio Koala | 2024.1.1**.



Panduan pemasangan Android Studio bagi Windows, MacOS, maupun Linux tersedia dalam bentuk teks maupun video tutorial di halaman Install Android Studio[4] sehingga tidak akan penulis tulis ulang di sini.

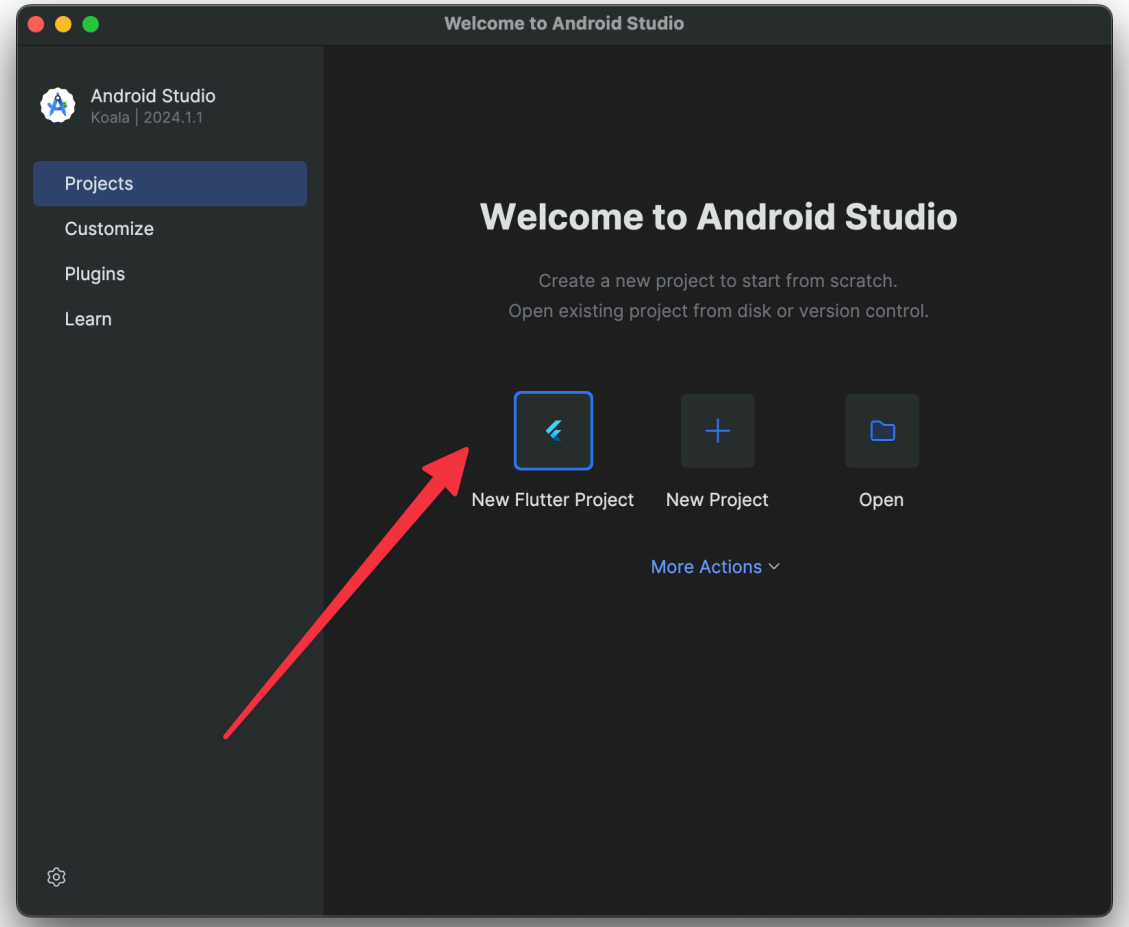
Setelah Android Studio terpasang, selanjutnya pasang **plugin Flutter** yang sekaligus akan menambahkan **plugin Dart** secara otomatis. Akses menu **Plugin** pada bagian sebelah kiri, klik tab **Marketplace**, lalu cari dengan kata kunci **Flutter*.

[4] https://developer.android.com/studio/install



Klik tombol **Install** yang berwarna hijau, tunggu proses selesai lalu klik tombol **Restart IDE** yang muncul.

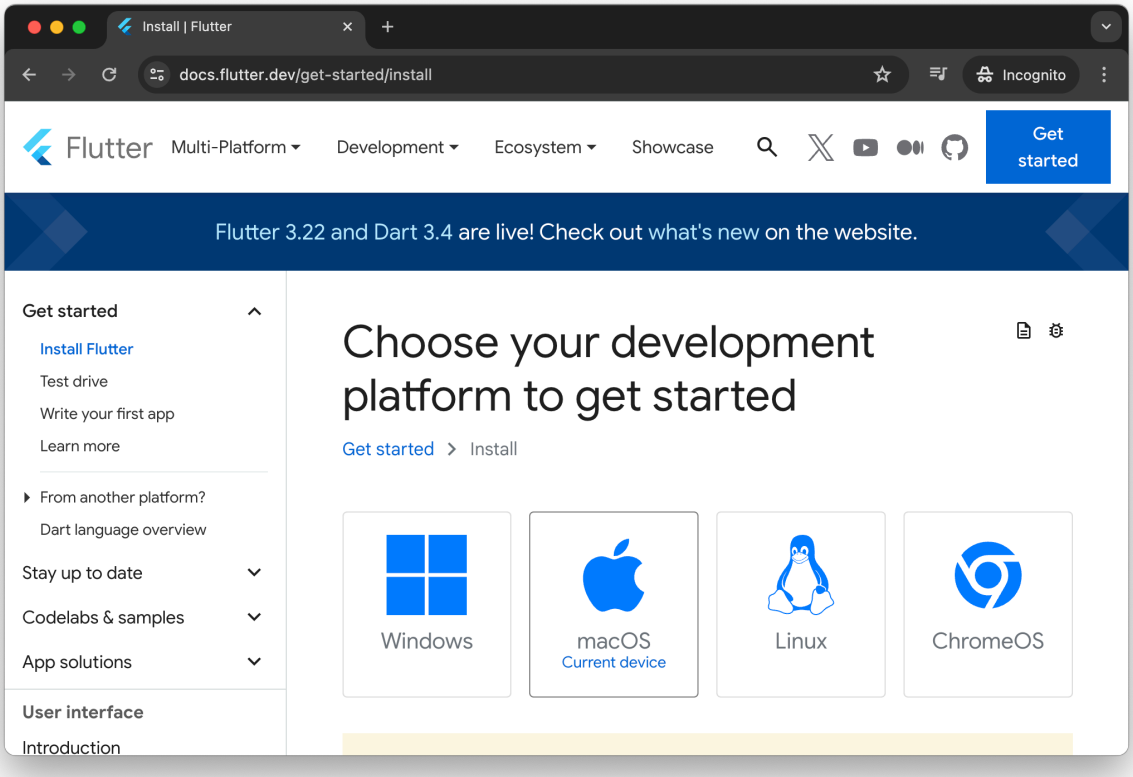
Bila muncul menu baru **New Flutter Project** di jendela awal Android Studio, artinya proses pemasangan Android Studio dan *plugin Flutter* telah selesai.



Menyiapkan Flutter SDK

Laman resmi flutter.dev[5] menawarkan file yang kita butuhkan untuk menyiapkan Flutter SDK dan memberikan langkah demi langkah yang sangat lengkap dan mudah untuk diikuti. Untuk membuat buku ini tetap ringkas, penulis tidak memberikan langkah pemasangan Flutter SDK secara lengkap, namun bila pembaca menemui kesulitan saat mengikuti panduan yang diberikan oleh **flutter.dev**, silahkan hubungi penulis untuk melakukan konsultasi tatap muka secara online atau bertanya lewat grup tanya-jawab yang telah disediakan.

[5] https://flutter.dev

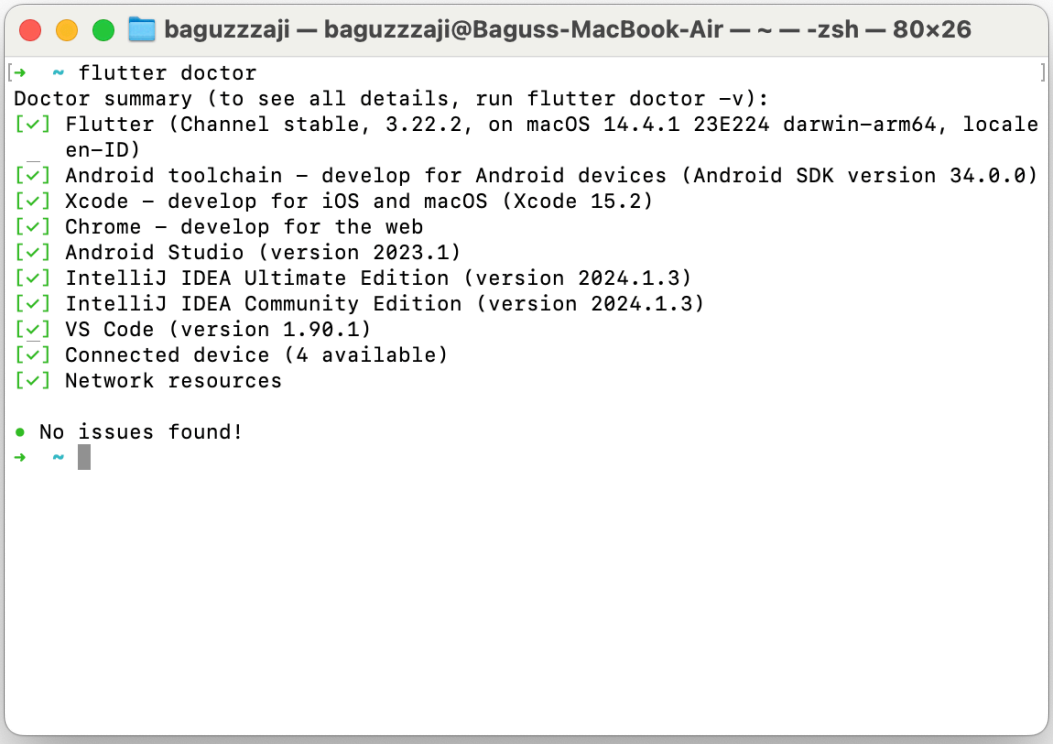


Pembaca perlu catat bahwa Flutter memiliki beberapa *release channel* untuk distribusi versi SDK yang berbeda. Pada umumnya yang paling banyak dipakai secara luas adalah channel **beta** dan **stable**. Fitur-fitur baru yang masih dalam tahap pengujian sebelum dirilis secara resmi akan hadir lebih dulu di *channel beta*. Apabila fitur tersebut sudah teruji, barulah versi Flutter yang baru tersebut ditambahkan ke *channel stable*.

Gunakan selalu versi **stable** untuk pengembangan aplikasi pada umumnya untuk menghindari hal-hal yang tidak diinginkan. Bila memang ada kebutuhan untuk menguji fitur baru yang belum hadir di versi **stable**, barulah gunakan versi **beta**.

Setelah langkah-langkah di atas selesai dilewati, kita akan bisa mengakses perintah `flutter` dari *command-line*. Untuk mengetahui apakah komputer kita sudah siap untuk dipakai bertempur dengan Flutter, jalankan perintah berikut:

```
1  flutter doctor
```



Pemasangan Flutter SDK sudah termasuk dengan Dart SDK sehingga kita tidak membutuhkan step khusus untuk menyiapkannya dan bisa langsung berkutat dengan pembahasan di bagian pertama.

Bagi pengguna Windows, harap ketahui bahwa bagian Xcode akan selalu merah karena Apple tidak merilis Xcode diluar sistem MacOS. Meskipun tidak bisa menguji atau merilis aplikasi untuk sistem Apple (iOS dan macOS), selama bagian *Android toolchain* mendapat centang hijau, kita tetap dapat mengembangkan aplikasi Flutter untuk perangkat Android.

Halo Dunia!

Bagi yang sudah familiar dengan bahasa pemrograman berorientasi objek atau bahasa pemrograman lain seperti Java, Kotlin, Swift, atau JavaScript, maka mempelajari Dart akan menjadi lebih mudah. Meskipun Dart memiliki keunggulan fiturnya sendiri, namun mayoritas aturan-aturan di dalamnya mirip dengan bahasa lain. Apabila Dart adalah bahasa pemrograman pertama pembaca, jangan berkecil hati. Konsep-konsep yang akan dibahas pada bagian pertama buku ini akan membantu memberikan dasar yang cukup untuk bisa mengikuti pelajaran pada bagian yang kedua.

Dart tergolong bahasa pemrograman baru. Bahasa ini diperkenalkan pada tahun 2011 dengan rilis versi stabil yang pertama (1.0) pada tahun 2013. Pada awal perkembangannya, Dart bukan ditujukan untuk pengembangan aplikasi mobile tapi sebagai tandingan JavaScript untuk pemrograman web. Sayangnya, Dart gagal mencapai tujuan utama dan sempat terpilih sebagai bahasa pemrograman terburuk untuk dipelajari karena tidak ada tempat untuk Dart.

Semua berubah saat Flutter datang. Flutter bukan framework *cross-platform* mobile yang pertama. Pada saat itu sudah ada Ionic dan React Native yang berkuasa. Berkat sponsor resmi dari Google yang memasang Dart sebagai bahasa pemrograman untuk Flutter, popularitas Dart menjadi meroket.

Dart *virtual machine* memungkinkan proses *rebuild* yang sangat cepat saat proses pengembangan aplikasi. Proses pengembangan yang cepat tentu harus dibarengi dengan aplikasi yang cepat pula. Sistem compiler *ahead-of-time* memungkinkan Dart membuat aplikasi *native* yang efisien untuk semua jenis platform.

Hello World!

Tradisi dalam belajar bahasa pemrograman baru adalah dengan mencetak teks **Hello**, **World!** ke layar. Berkat pemasangan Flutter SDK yang telah dilakukan pada bab sebelumnya, kita tidak perlu menyiapkan *environment* khusus untuk bisa menulis dan memproses file-file Dart.

Menjalankan File Dart

Compiler Dart bisa membaca satu file Dart dan mengeksekusinya langsung di terminal. Siapkan folder untuk menyimpan file Dart di manapun, lalu buatlah satu file baru bernama `hello_world.dart`.

Berikutnya, tuliskan kode Dart berikut ke file tadi:

```dart
void main() {
    print("Hello, World!");
}
```

Program Dart memperbolehkan *top-level function* seperti Kotlin. Artinya kita boleh membuat suatu fungsi tanpa memiliki sebuah `class` tidak seperti pada pemrograman java. Mencetak sesuatu ke layar terminal dilakukan dengan fungsi `print()` dan setiap baris harus diakhiri dengan titik koma.

Selanjutnya jalankan perintah berikut diterminal, pastikan berada di folder yang sama dengan file `hello_world.dart`.

```
dart run hello_world.dart
```

Pembaca seharusnya akan dapat melihat teks "Hello, World!" di layar seperti pada gambar di bawah ini.

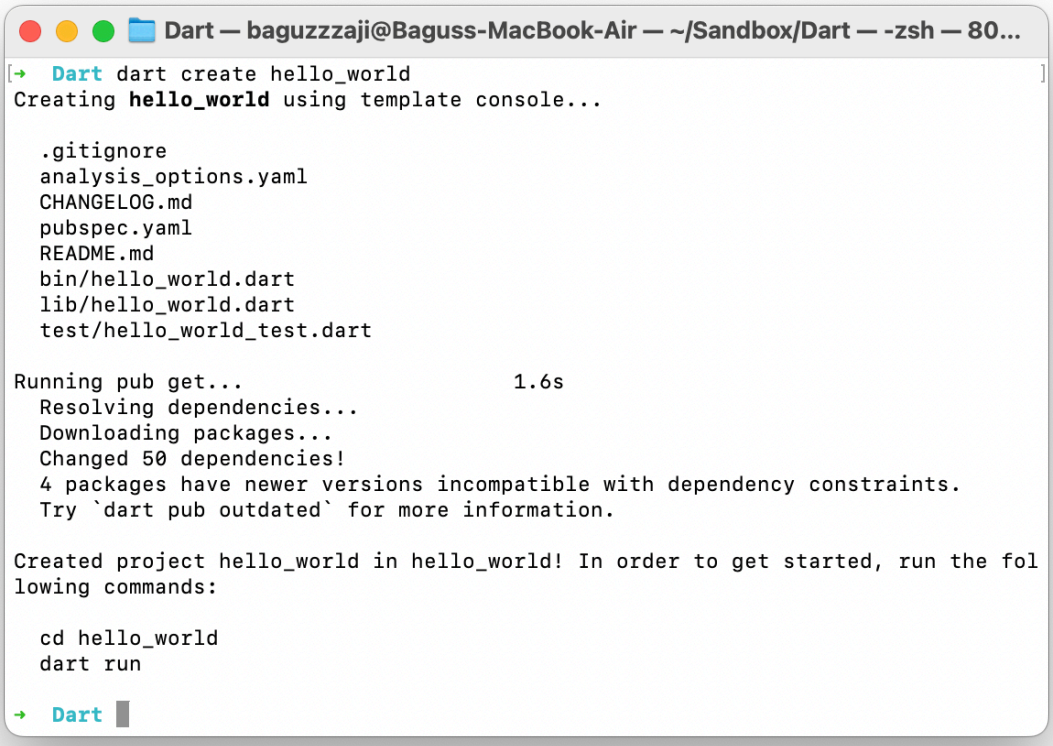
Membuat Project Dart

Program-program kecil seperti mayoritas contoh kode pada bagian pertama buku ini, mungkin sudah cukup dijalankan dengan cara yang telah kita pelajari sebelumnya. Akan tetapi, semakin besar project yang dibuat, mungkin pembaca perlu untuk memisahkan program dalam file-file tersendiri dan menyusun project dengan struktur yang lebih rapi. Untuk melakukan hal tersebut, disarankan untuk membuat project Dart yang utuh dengan perintah `dart create`.

Akses folder yang diinginkan lalu jalankan perintah berikut ini di terminal.

```
1 dart create hello_world
```

Perintah di atas akan membuat project kosong dengan beberapa kode standar.



Menjalankan Project Dart

Untuk menjalankan project yang sudah dibuat, masuk ke folder yang telah dibuatkan oleh perintah `dart create`:

```
cd hello_world
```

Berikutnya jalankan project ini dengan perintah:

```
dart run bin/hello_world.dart
```

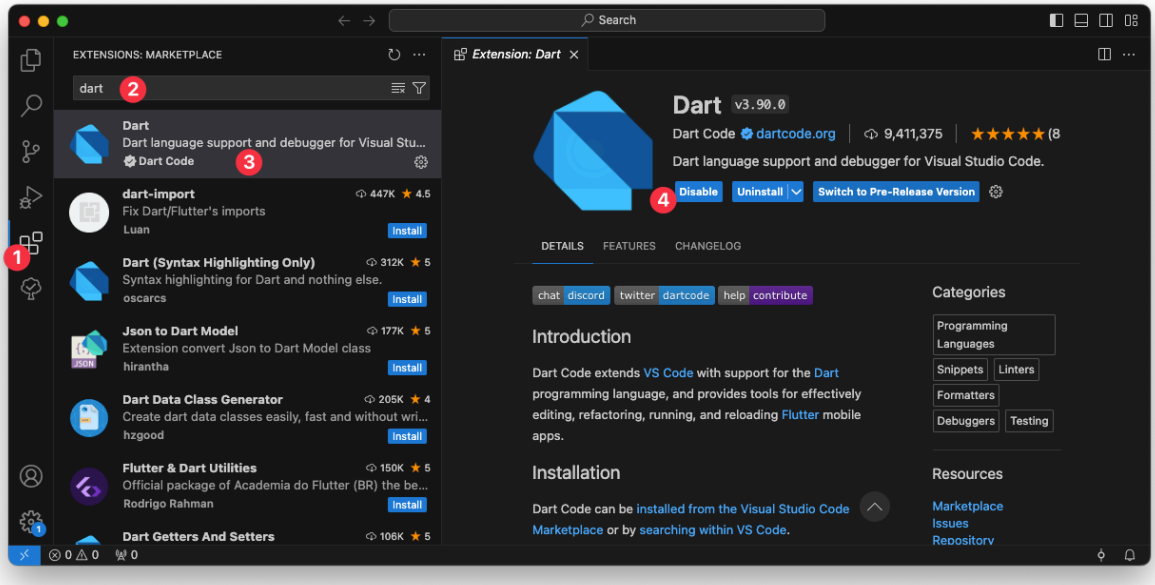
Pembaca akan melihat teks **Hello world: 42!** yang dibuatkan oleh `create`. Perintah `run` bersifat opsional, kita bisa menjalankan program Dart tanpa perintah ini.

```
dart bin/hello_world.dart
```

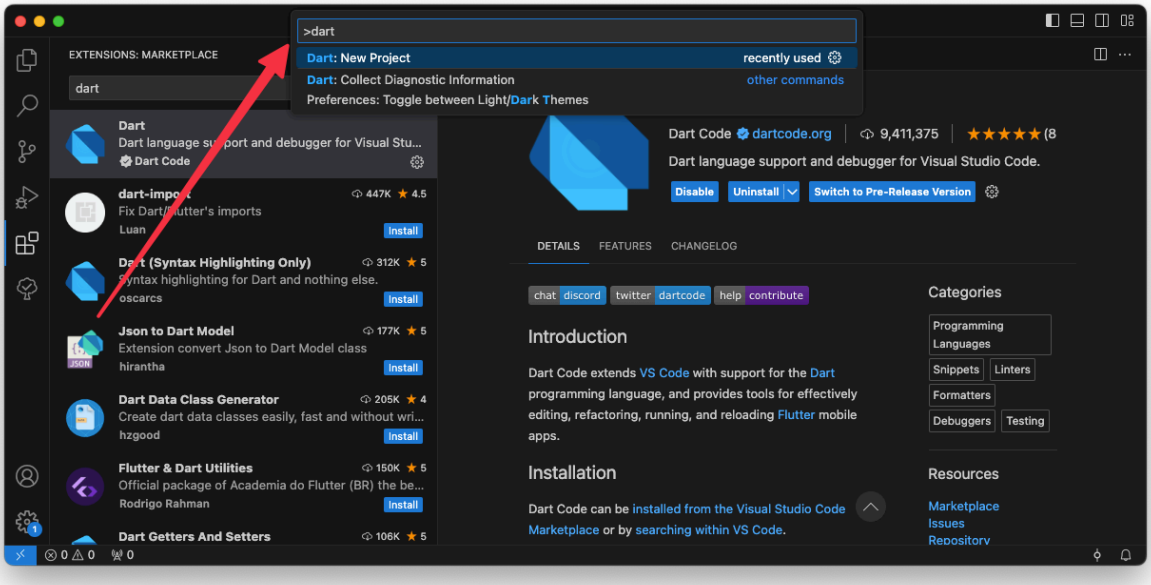
Membuat Project Dart dengan Visual Studio Code

Visual Studio Code bisa membuatkan project Dart baru untuk kita bila sudah dipasangkan ekstensi **Dart language support**.

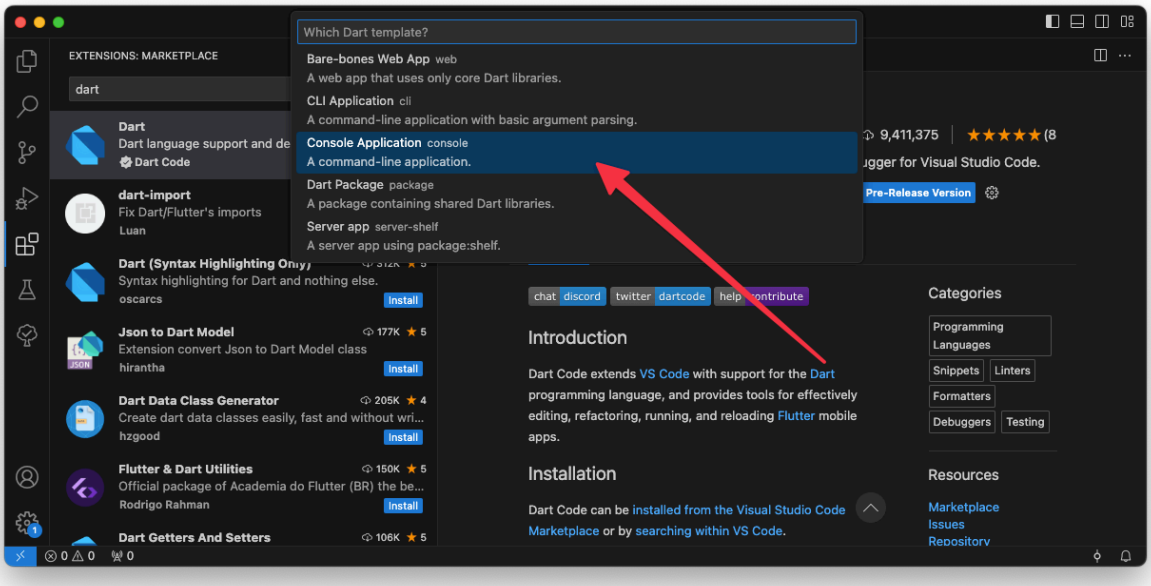
Buka jendela Visual Studio Code lalu buka menu **Extensions** (1), tuliskan kata kunci **dart** di kotak pencarian kemudian pilih hasil teratas (2 & 3), terakhir klik tombol **Install** (4). Restart jendela Visual Studio Code bila diperlukan.



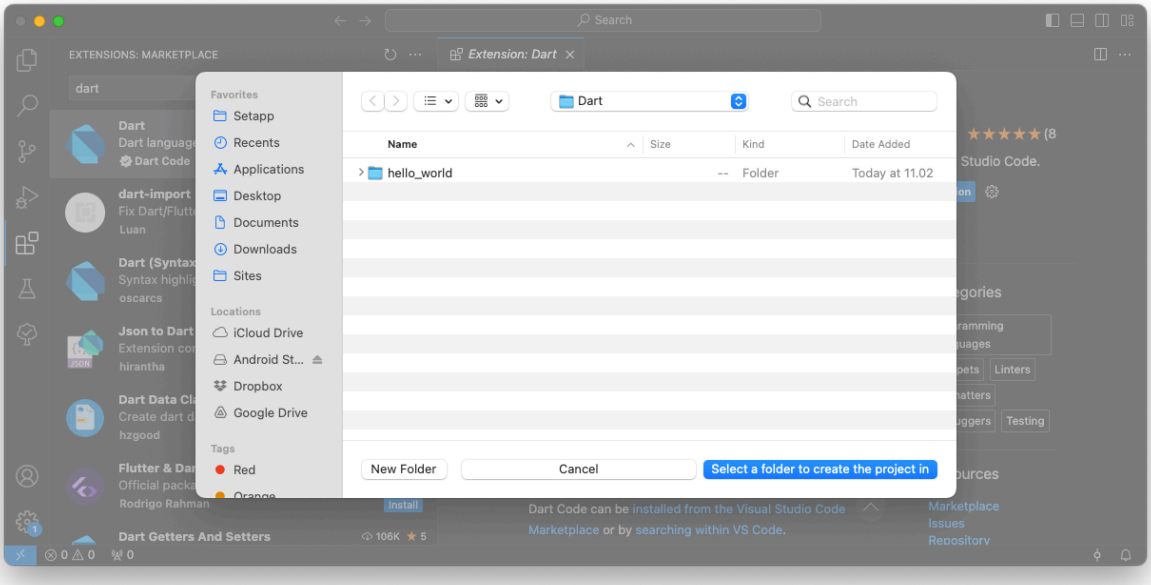
Untuk membuat project Dart baru lewat Visual Studio Code, tekan tombol **Cmd + Shift + P** di MacOS atau **Ctrl + Shift + P** di Windows untuk memunculkan **Command Palette**. Pada popup yang muncul tuliskan dart.



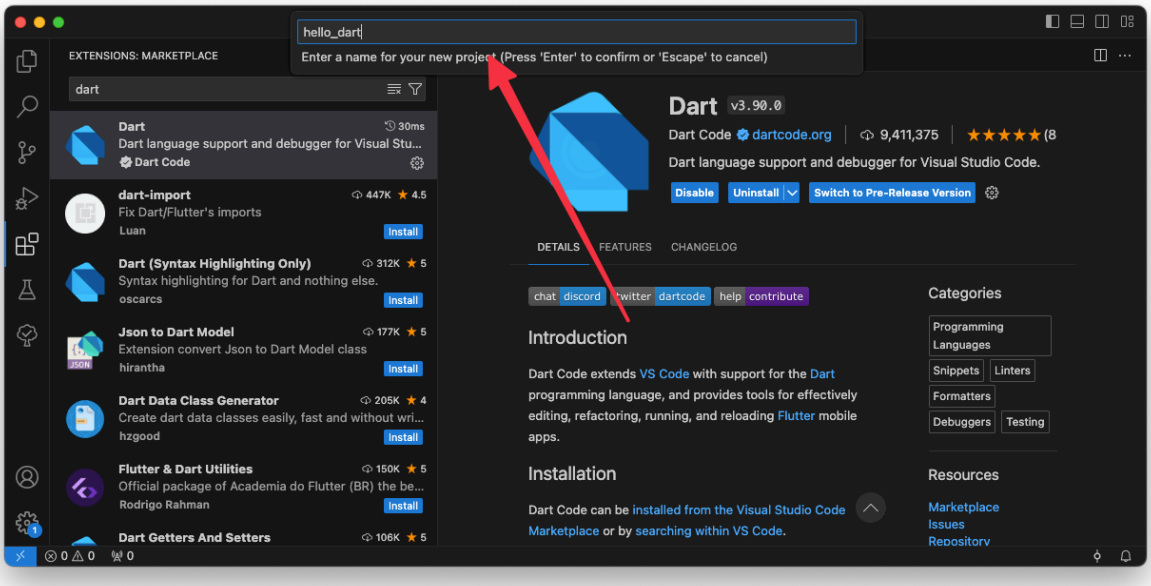
Pastikan memilih opsi **New project** lalu di menu berikutnya pilih **Console Application**.



Visual Studio Code akan meminta kita memilih folder tempat menyimpan project baru. Klik tombol **Select a folder to create the project in** bila telah selesai memilih.



Langkah selanjutnya adalah memberikan nama project. Karena sebelumnya sudah membuat project lewat terminal bernama `hello_world`, maka di sini penulis membuat project baru bernama `hello_dart`.



Pastikan menggunakan huruf kecil untuk nama project dengan simbol _ bila ingin memisahkan masing-masing kata. Jangan menggunakan simbol lain seperti -.

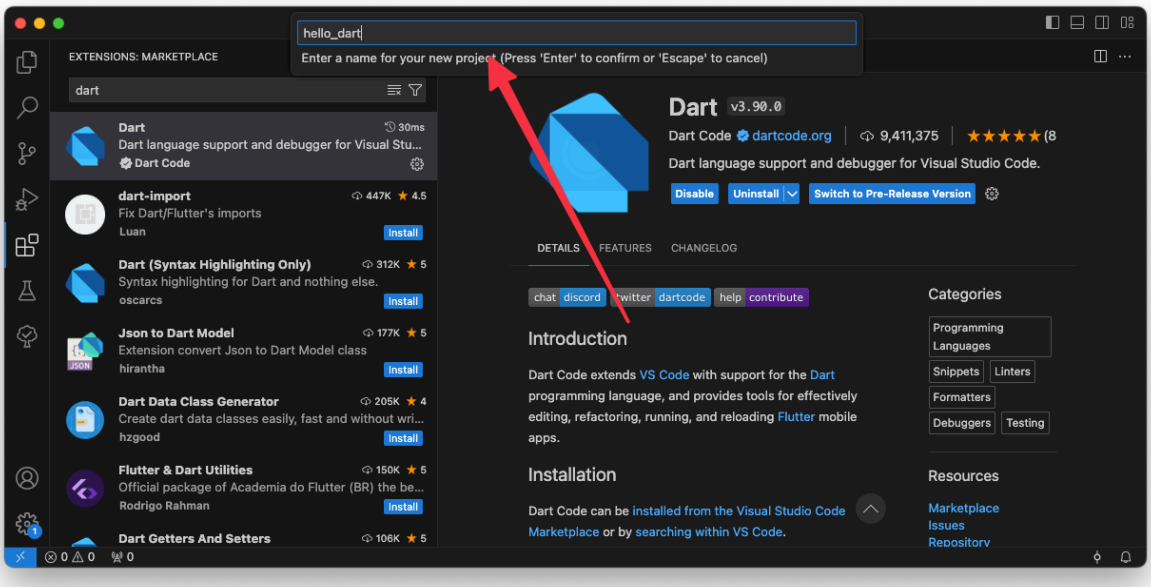


Figure 1. Struktur project baru yang dibuat dengan VS Code

Visual Studio Code akan memuat folder berisi project Dart yang baru. Kita bisa menjalankan project ini langsung dari jendela VS Code lewat menu **Run and Debug**.

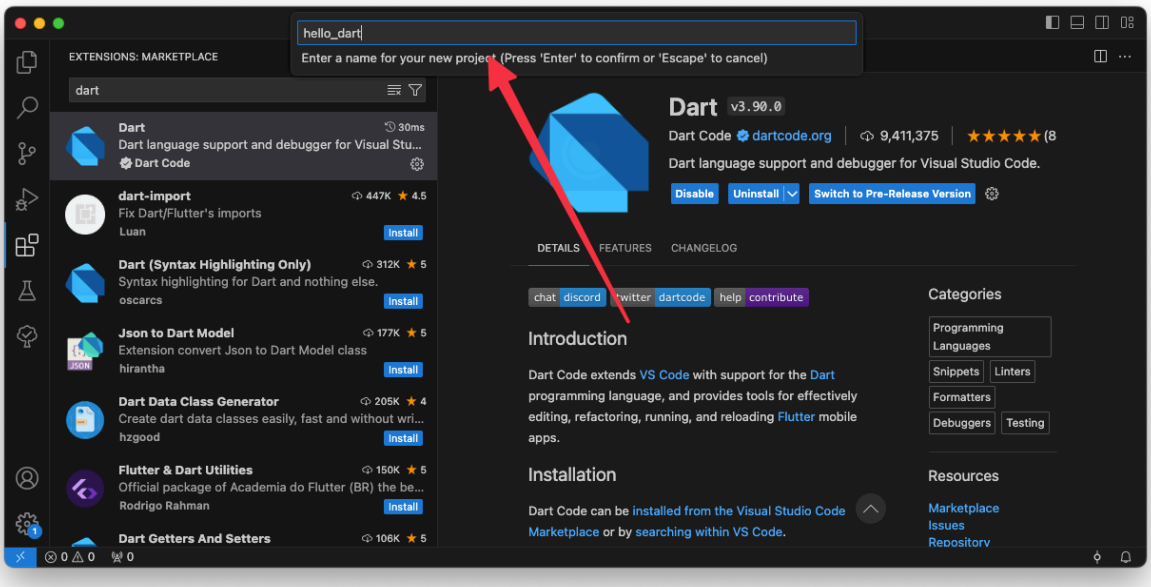


Figure 2. Menjalankan project langsung dari VS Code

Struktur Project Dart

Project yang dibuatkan baik oleh perintah `dart create` maupun Visual Studio Code akan memiliki struktur seperti pada gambar.

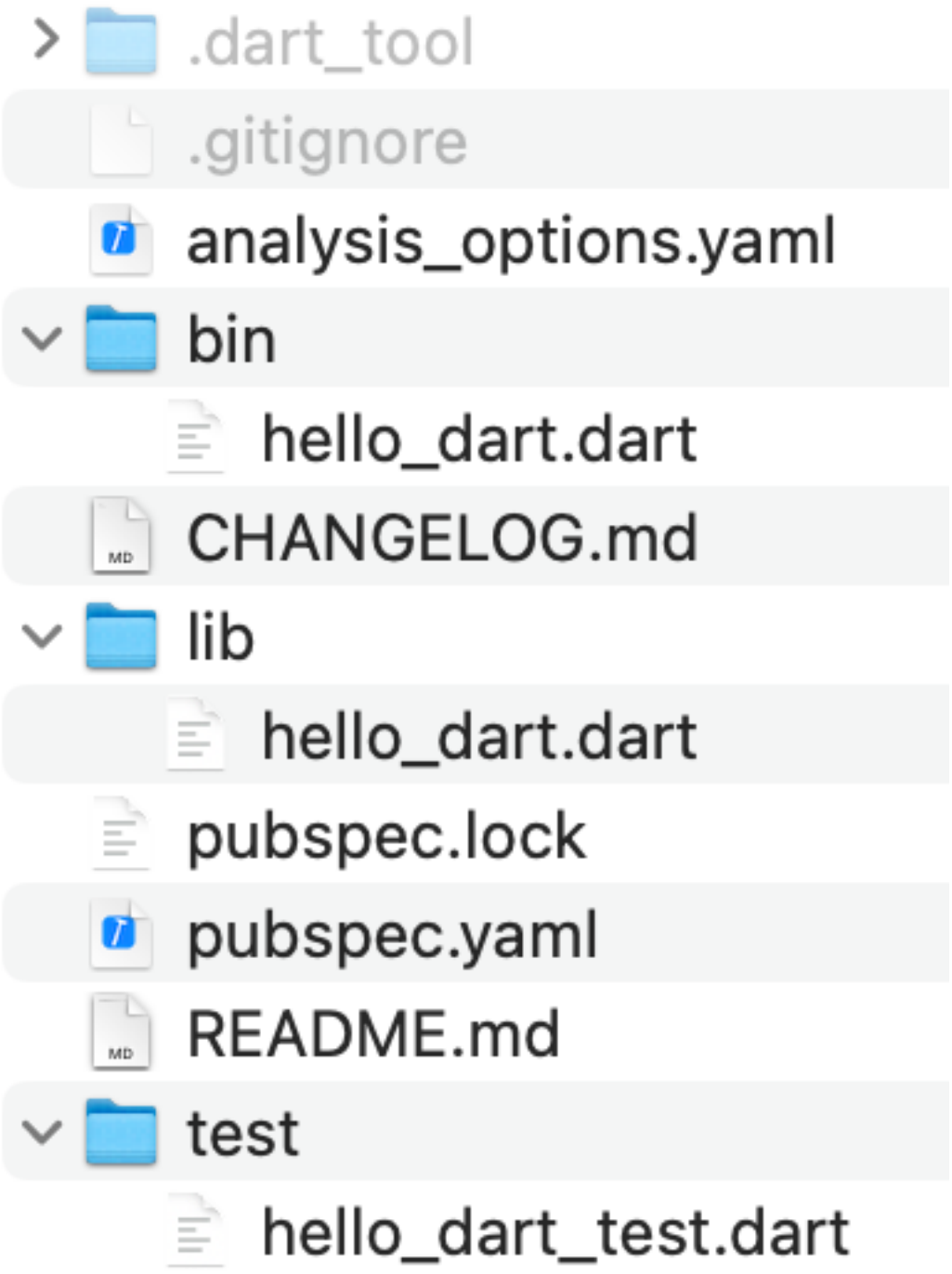


Figure 3. Menjalankan project langsung dari VS Code

Tujuan masing-masing file/folder di atas adalah sebagai berikut:

- **.dart_tool**: berisi pengaturan yang dibutuhkan oleh *compiler* Dart.

- **.gitignore**: memberitahukan Git file-file apa saja yang tidak perlu di simpan di repository sehingga tidak ikut di upload ke Github atau yang sejenisnya.
- **analysis_option.yaml**: menyimpan pengaturan untuk melakukan sebuah proses bernama **linting** yang secara singkat dipakai mendeteksi kesalahan penulisan program sebelum di *compile*.
- **bin**: berisi kode Dart yang bisa di ekeskusi.
- **hello_dart.dart**: merupakan program utama yang akan dieksekusi.
- **lib**: berisi banyak file .dart yang bisa dipakai diberbagai tempat atau di rilis sebagai sebuah *library*.
- **pubspec.yaml**: berisi dependensi terhadap *library* pihak ketiga yang ingin dipakai di project ini.
- **pubspec.lock**: file yang memastikan nomor versi dari *library* yang tertulis di `pubspec.yaml` oleh project ini.
- **README.md**: file yang mendeskripsikan tentang project ini. Tidak berelasi langsung dengan Dart/Flutter, lebih sebagai informasi bagi developer lain.
- **test**: berisi file-file untuk melakukan *testing*.

Variabel & Tipe Data

Bab ini akan membahas bagaimana mendefinisikan suatu variabel dan jenis-jenis tipe data yang ada dalam bahasa pemrograman Dart. Pembaca yang sudah pernah belajar bahasa pemrograman lain seperti Java, Kotlin, atau Swift mungkin akan sedikit familiar dengan tipe data yang ada di Dart.

Berikut beberapa poin yang akan kita bahas di bab ini:

- Variabel
- Constant
- Number
- String
- Boolean
- *Type Inference*
- *Late Variable*
- *Constant*
- List
- Set
- Map

Variabel

Dalam suatu program kita akan bekerja dengan banyak jenis data. Terkadang data tersebut berupa angka seperti umur, uang, jarak, terkadang juga berupa teks seperti nama, alamat, dan lain sebagainya.

Agar bisa mengelola banyak data, kita memerlukan suatu cara untuk menampung data tersebut. Perhatikan kode berikut:

```
1  int age = 17;
```

Pada contoh di atas kita membuat sebuah variabel bernama `age` yang menampung data beruba angka bernilai 17. Karena angka 17 merupakan integer, maka kita perlu menggunakan tipe data `int` yang ditulis sebelum nama variabel yang diinginkan.

Suatu variabel bisa dipanggil lagi untuk mengambil data yang ia tampung dari tempat yang berbeda, untuk di proses lagi atau dicetak ke layar.

```dart
int age = 17;

print(age); // mencetak angka 17

// mencetak teks berdasarkan nilai dari variabel age
if (age >= 17) {
    print("adult");
} else {
    print("kids");
}
```

Jika pembaca mencari arti dari *variable* di dalam kamus, maka akan menmukan terjemahan "dapat berubah-ubah". Dalam pemrograman **variabel** merupakan sesuatu karena nilainya bisa diubah (dengan beberapa pengecualian)

```dart
int age = 17;

print(age); // 17

age = 21;

print(age); // 21
```

Dalam bahasa pemrograman Dart pada umumnya menggunakan *camelCase* seperti Java dan Kotlin. Meskipun bisa menggunakan *snake_case*, tapi mengikuti standar yang umum dipakai akan lebih baik.

Number

Angka menjadi jenis data yang paling sering digunakan disebuah aplikasi. Dart memberikan variasi tipe data yang bisa dipakai untuk bekerja dengan angka.

num

Tipe data `num`, merupakan *superclass* dari tipe data lain yaitu `double` dan `int`. Namun, dalam praktiknya, kita akan jarang sekali menggunakan `num` secara langsung tapi menggunakan `double` atau `int` karena lebih spesifik.

> ***CATATAN:*** Superclass akan dibahas di bab Class & Object.

Berikut contoh penggunaan num:

```dart
num pi = 3.14;

// atau

num age = 17;
```

Tipe data num memiliki beberapa fungsi yang bisa dimanfaatkan untuk memproses angka di dalam suatu variabel.

```dart
pi.toString() // mengubah angka menjadi string 3.14

pi.toStringAsPrecision(2) // mengubah angka  dengan presisi di belakang koma, pada c\
ontoh ini akan menghasilkan 3.1

p.ceil() // Pembulatan ke atas (4)

p.floor() // Pebulatan ke bawah (3)
```

Karena `double` dan `int` merupakan *subclass* dari `num`, maka mereka juga mendapat akses pada fungsi-fungsi di atas.

Berdasarkan pengalaman penulis, tipe data `num` jarang dipakai secara langsung. Satu contoh kasus dimana `num` terpaksa dipakai adalah saat *backend api* yang memberikan data secara dinamis berupa `int` tapi terkadang `double`.

int

Tipe data `int` menyimpan angka berupa bilangan bulat baik positif maupun negatif diantara -2^{63} sampai $^{2}63-1$.

Contoh deklarasi variabel int adalah sebagai berikut:

```dart
int age = 17;
```

Seringkali kita akan mendapatkan suatu data angka tapi berupa string. String tersebut bisa kita ubah menjadi `int` dengan memanggil method `parse`.

```dart
int length = int.parse('413');
```

Meskipun jarang, tapi jika suatu hari membutuhkan variabel yang bisa menampung apa yang `int` bisa, kita bisa menggunakan class `BigInt` untuk angka yang lebih besar.

double

Variabel bertipe `double` bisa menyimpan angka berupa bilangan desimal dengan limit yang kurang lebih sama seperti `int`.

```dart
double pi = 3.14;
```

Perlu diketahui juga bahwa Flutter bisa memproses sebuah bilangan bulat dan menganggapnya sebagai *double* secara otomatis.

```dart
double width = 8;
```

Dalam bahasa pemrograman lain seperit Java dan Kotlin, kode di atas akan menghasilkan *compiler error* karena `8` bukan sebuah `double`, harus dilengkapi dengan `8.0` atau dengan `8d` atau `8f`.

Sama halnya dengan `int`, double memiliki akses ke fungsi seperti `toString()`, `ceil()`, `floor()` dan `parse()` untuk mengubah string menjadi double.

```dart
double width = double.parse('3.14');
```

String

Data berupa teks disimpan sebagai sebuah `String`, mirip dengan bahasa pemrograman lain. Dart memungkinkan kita untuk menggunakan tanpa petik satu maupun petik dua.

```dart
String name = "Bagus Aji";

// atau

String name = 'Bagus Aji';
```

Jika kita ingin menyelipkan variabel di dalam suatu teks, kita bisa menggunakan fitur *string interpolation* dibandingkan menggunakan *string concatenation*.

```dart
String name = "Bagus Aji";
String welcomeConcatenation = "Selamat pagi " + name + ". Selamat datang kembali.";
String welcomeInterpolation = "Selamat pagi $name. Selamat datang kembali.";
```

Karena seringnya flutter developer akan bekerja dengan String, kita akan membahasnya secara lebih mendalam di bab tersendiri

bool

Sebuah variabel yang hanya bisa menyimpan dua nilai `true` atau `false` alias *boolean*, dalam pemrograman Dart menggunakan kata kunci `bool`.

Variabel bertipe data boolean sering dipakai saat membuat kondisi percabangan atau perulangan.

List

`List` merupakan tipe data khusus yang berguna untuk menyimpan lebih dari satu data dalam tipe yang sejenis. Dalam bahasa pemrograman lain, *list* berfungsi mirip dengan sebuah *array*.

```
List<int> integers = [1, 2, 3, 4, 5];
List<double> doubles = [1.1, 1.2, 1.3];
```

List memiliki banyak method yang bisa dipakai untuk menambah, menghapus, mengurutkan atau mengosongkannya.

Sama seperti bahasa pemrograman pada umumnya, List menggunakan index untuk mengakses elemen di dalamnya dimulai dari 0 sampai dengan jumlah data dikurangi 1.

```
// Menambah item baru diujung
// [1, 2, 3, 4, 5, 9];
integers.add(9);

// Mengakses indeks ke 2
// 3
integers[2];

// Menambah item baru di indeks ke-1
// [1, 7, 2, 3, 4, 5, 9];
integers.insert(1, 7);

// Menghapus elemen berdasarkan nilainya
// [7, 2, 3, 4, 5, 9];
integers.remove(1);

// Mengurutkan list
integers.sort();

// Mengosongkan list
integers.clear();
```

List akan dibahas secara lebih mendalam di bab tersendiri.

Set

Secara sederhana, kita bisa menganggap bahwa `Set` merupakan variasi `List` yang hanya bisa menyimpan satu data yang sama (tidak boleh ada dua data yang sama di dalamnya). Set bisa dideklarasikan dengan menggunakan simbol `{}` berisi nilai-nilai awal:

```
Set<String> vegetables = {"Apple", "Durian", "Strawberry"};

vegetables.add("Pineapple");
// {Apple, Durian, Strawberry, Pineapple}

vegetable.add("Apple");
// {Apple, Durian, Strawberry, Pineapple}
// Apple tetap ada 1, tidak ada penambahan.
```

Map

Tipe data ini memungkinkan kita untuk memiliki *key-value pair* atau kita bisa memasangkan suatu data pada tipe data yang lain. Contoh berikut menunjukkan bagimana kita menggunakan nomor punggung pemain sepak bola karena satu pemain hanya akan memiliki satu nomor punggung:

```
Map<int, String> players = {
    7: "Rafael Struick",
    12: "Pratama Arhan",
    22: "Ernando Ari S."
};
```

Dalam deklarasi Map, bagian `<int, String>` menentukan apa yang menjadi *key* (nomor punggung berupa integer) dan apa yang menjadi *value*-nya (nama pemain berupa string). Baik *key* maupun *value* bisa menggunakan tipe data apa saja.

Kita bisa mengakses nama pemain dengan memanfaatkan nomor punggunya (namun tidak sebaliknya). Cara mengakses isi map berdasarkan key adalah sebagai berikut:

```
players[12];
```

Type Safety & Type Inference

Sebagai bahasa pemrograman modern yang canggih, Dart memiliki fitur *type inference*, artinya kita tidak perlu menuliskan tipe data saat mendeklarasikan suatu variabel. Pada umumnya, saat mendeklarasikan tipe data, programmer tidak secara langsung menulis tipe data seperti yang telah kita lakukan pada contoh-contoh sebelumnya.

Dengan menggunakan keyword `var` seperti contoh di bawah, kita memberitahu Dart untuk "menggunakan tipe data yang paling cocok":

```dart
// variabel age mendapatkan tipe data int
var age = 17;

// variabel pi mendapatkan tipe data double
var pi = 3.14;

// variabel isAdult mendapatkan tipe ada bool
var isAdult = true;

// variabel arr mendapatkan tipe data List<int>
var arr = [1, 2, 3];

// variabel name mendapatkan tipe data String
var name = "Bagus Aji";

// variabel players mendapatkan tipe data Map<int, String>
var players = {
    7: "Rafael Struick",
    12: "Pratama Arhan",
    22: "Ernando Ari S."
};
```

Dart akan tahu bahwa angka `10` adalah sebuah integer, angka `3.14` adalah sebuah double, dan lain sebagainya. Jadi meskipun tidak secara eksplisit tipe datanya ditulis tapi Dart bisa memahami nya saat deklarasi variabel dengan `var`.

Meskipun kita tidak menuliskan tipenya saat mendeklarasikan variabel, bukan berarti kita bisa bebas menggunakan tipe data yang lain sesudahnya variabel tersebut dideklarasikan.

```dart
var age = 17;
age = 17.5;
```

Dart sudah mengasumsikan tipe data int untuk variabel `age`. Saat variabel age akan diubah dengan nilai yang bukan berupa integer misalnya double, maka akan muncul pesan error:

```
Error: A value of type 'double' can't be assigned to a variable of type 'int'.
```

Dart merupakan bahasa pemrograman yang *type-safe*, artinya sekali dideklarasikan, maka tidak bisa diubah lagi seperti pada contoh di atas. Fitur *type safety* ini akan membantu kita menghindari error yang terjadi akibat data yang tidak seharusnya sehingga bisa menyebabkan aplikasi mengalami *crash*.

Fitur *type safety* ini bisa dihindari dengan menggunakan tipe data khusus bernama `dynamic`. Semua tipe data yangn ada di dart pada dasarnya merupakan `dynamic` sehingga data yang ditampung bisa diubah dengan tipe data yang lain.

```dart
dynamic age = 17;
age = 17.5;

print(age);
// 17.5

age = "seventeen";
print(age);
// seventeen
```

Meskipun bisa, penggunaan tipe data `dynamic` tidak direkomendasikan tanpa alasan khusus. Memakai tipe ini akan menyulitkan proses bug fixing yang terkait dengan tipe data kaerna tipe data dynamic bisa berubah-ubah.

Pembaca perlu berhati-hati saat menggunakan *type inference* di tipe data *collection* misalnya List.

```dart
var lst = [];

print(lst.runtimeType); // List<dynamic>
```

Constant & Final

Dart bisa memiliki dua jenis variabel yang setelah diisi satu kali, tidak bisa diubah lagi sesudahnya. Dua jenis variabel ini dibuat dengan `const` dan `final`.

const

Variabel yang datanya bisa diubah seperti pada contoh-contoh sebelumnya disebut sebagai *mutable data*. Sementara itu, ada beberapa kasus di mana kita memerlukan variabel yang datanya tidak bisa diubah. Variabel yang datanya tidak bisa diubah lagi ini disebut juga dengan *immutable data*.

Untuk membuat sebuah variabel konstan, gunakan `const`:

```dart
const pi = 3.14;
```

Dalam matematika, pi sudah memiliki ketetapan dimana nilainya akan selalu `3.14`, bila nilai pi diubah maka hasil perhitungannya tentu akan ikut berubah juga. Untuk mencegah agar variabel pi tidak diubah disuatu tempat, deklarasikan sebagai sebuah variabel konstan. Sama seperti `var`, saat kita menggunakan `const`, Dart juga akan cukup pintar untuk menentukan tipe data nya lewat *type inference*.

```dart
const pi = 3.14;
pi = 3;
```

Kode di atas tidak akan bisa di *compile* dan menghasilkan pesan error:

```
Error: Can't assign to the const variable 'pi'.
```

final

Secara konsep `final` sama dengan `const`, hanya saja saat menggunakan const, nilainya harus sudah ditentukan sebelum aplikasi di *compile*. Apabila kita menginginkan sebuah `const` tapi belum ada nilainya saat aplikasi di *compile*, itulah saat kita membutuhkan yang namanya *runtime constant*.

Cara mudah membedakan kapan menggunakan `const` dan `final` adalah dengan melihat nilainya.

```dart
const age = 17;
const name = "Bagus";
const pi = 3.14;
```

Pada contoh di atas kita sudah tahu saat mendeklarasikan variabel, apa nilai yang akan dituliskan setelah `=`. Apabila data tersebut tidak bisa dituliskan secara langsung dan perlu menunggu suatu proses lain setelah aplikasi dijalankan seperti data yang datang dari suatu api atau query database, maka saat itulah kita menggunakan `final`.

```dart
final now = DateTime.now();
print(now);
// 2024-06-23 13:52:50.722594
```

Method `DateTime.now()` adalah fungsi yang akan memberikan jam dan tanggal saat program diakses. Karena nilainya baru akan didapat bila program sudah jalan, maka variabel now yang kita miliki tidak bisa dideklarasikan dengan `const` sehingga harus menggunakan `final`.

Percabangan

Percabangan dipakai untuk mengatur alur program tersebut sehingga terkadang disebut juga dengan kontrol alur atau *control flow*. Program paling sederhana sekalipun pasti akan memiliki bagian untuk memutuskan apa yang perlu dilakukan pada suatu kondisi:

- Apakah user sudah dewasa (berumur lebih dari 17 tahun)?
- Apakah user sudah terdaftar?
- Apakah user sudah login?
- Apakah email sudah terverifikasi?
- Apakah status koneksi internet *online*?

Pertanyaan-pertanyaan di atas merupakan kondisi yang akan sering ditemukan pada sebuah aplikasi mobile. Semuanya merupakan pertanyaan dengan jawaban ya atau tidak atau dalam bahasa pemrograman pertanyaan yang bisa dijawab dengan `true` atau `false`.

Selanjutnya, bagaimana cara memeriksa suatu kondisi dalam bahasa pemrograman Dart?

Percabangan `if`

Metode yang paling dasar untuk melakukan percabangan dalam bahasa pemrograman apapun adalah perintah `if`. Ia dipakai untuk mengeksekusi suatu blok apabila komparasinya berhasilkan nilai `true`.

```
1  var age = 22;
2
3  if (age > 17) {
4      print("Adult user");
5  }
```

Blok `if` pada kondisi di atas memeriksa apakah user sudah dewasa atau belum berdasarkan umurnya. Apabila umurnya lebih dari 17 tahun, maka user dianggap sudah dewasa.

Lalu bagimana bila kita ingin melakukan sesuatu bila user belum berusia 17 tahun?

Percabangan dengan `else`

Blok `else` dipakai untuk melakukan sesuatu apabila kondisi pada blok `if` tidak terpenuhi.

```
var age = 22;

if (age > 17) {
    print("Adult user");
} else {
    print("Non-adult user");
}
```

Saat menggunakan percabangan `if-else`, hanya akan ada salah satu blok saja yang dieksekusi berdasarkan hasil komparasinya, apakah memenuhi atau tidak memenuhi syarat.

Percabangan dengan `else if`

Untuk beberapa kasus, kita memerlukan kondisi tambahan lainnya seperti pada contoh:

- Baby: 0-2 tahun
- Child: 3-12 tahun
- Teen: 13-17 tahun
- Young Adult: 18-27 tahun
- Adult: lebih dari 27 tahun

Kondisi di atas bisa dikontrol dengan memanfaatkan `else if`.

```
var age = 15;

if (age > 27) {
    print("An adult");
} else if (age > 17) {
    print("A young-adult");
} else if (age > 12) {
    print("A teenager");
} else if (age > 3) {
    print("A child");
} else {
    print("A baby");
}
```

Sama seperti sebelumnya, dalam suatu deretan `if-else if-else`, hanya akan ada satu blok yang dieksekusi. Program akan menguji kondisinya satu per satu sampai ada kondisi yang bernilai `true`. Bila tidak ada, maka blok `else` akan selalu dieksekusi. Ubah-ubah nilai variabel `age` untuk mendapatkan hasil yang berbeda-beda.

Saat membuat kondisi dengan banyak percabangan, perhatikan urutan yang ditulis karena bisa menentukan blok mana yang akan dieksekusi terlebih dahulu.

Operator Boolean

Pada contoh penggunaan percabangan `if`, kita melakukan komparasi dengan menguji apakah sautu nilai bernilai `true` atau tidak dengan memakai *operator boolean*. Operator inilah yang akan menghasilkan nilai `true` atau `false` dengan melakukan perbandingan.

Berikut bagaimana cara melakukan perbandingan dalam bahasa pemrograman Dart.

Menguji Nilai Sama Dengan

Menguji nilai sama dengan dilakukan menggunakan *equality operator* yang dipakai dengan simbol ==, dua kali tanda sama dengan. Salah satu kesalahan yang paling sering dilakukan pemula adalah menggunakan satu kali tanda sama dengan saat menulis kondisi `if`.

```
1  var department = "Computer Science";
2
3  print(department == "Mathematics"); // false
```

Menguji Nilai Tidak Sama Dengan

Apabila simbol == dipakai untuk menguji nilai sama dengan, maka untuk menguji nilai tidak sama dengan dilakukan dengan simbol `!=` yang disebut juga sebagai *inequality operator*.

```
1  var department = "Computer Science";
2
3  print(department != "Mathematics"); // true
```

Menguji Nilai Lebih Dari dan Kurang Dari

Untuk membandingkan dua angka dan memeriksa apakah salah satunya bernilai lebih besar atau lebih kecil dapat dilakukan dengan simbol `>` atau `<`. Kedua simbol ini bersifat *exclusive*, artinya angka yang sama akan dianggap `false`.

```
1  print(4 < 5); // true
2  print(5 < 5); // false
3  print(4 > 5); // false
4  print(5 > 5); // false
```

Untuk memeriksa kondisi lebih dari sama dengan, dilakukan dengan simbol `>=`.

```
print(4 <= 5); // true
print(5 <= 5); // true
```

Sebaliknya memeriksa kondisi kurang dari sama dengan dilakukan dengan simbol `<=`.

```
print (5 >= 5); // true
```

Operator AND

Pada beberapa contoh kasus di atas, kita selalu memeriksa satu kondisi saja. Pada mayoritas kasus, memeriksa satu kondisi saja cukup. Namun tidak jarang juga akan ada dua kondisi yang perlu diperiksa sekaligus. Dua kondisi ini bisa digabungkan dalam satu komparasi, salah satunya dengan operator AND atau simbol `&&`.

```
var age = 16;
var withParent = true;

if (age < 17 && withParent) {
    print("Can buy ticket");
} else {
    print("Can't buy ticket");
}
```

Pada contoh di atas kita memeriksa apakah user berusia dibawah 17 tahun dan datang bersama orang tua. Bila keduanya bernilai `true` maka akan menghasilkan nilai `true` juga. Namun bila salah satu `false`, maka nilai akhirnya menjadi `false` juga.

Untuk memeriksa kondisi `== true`, bisa disingkat tanpa menuliskannya. Pada contoh di atas cara yang lebih lengkap adalah dengan menulis `withParent == true`, tapi karena variabel ini bertipe boolean, maka bisa disingkat saja.

Operator OR

Dalam kondisi AND, semua komparasi harus bernilai `true` sementara pada operator OR cukup satu saja ondisi bernilai `true` maka hasil ahirnya akan bernilai `true`. Penulisan operator or menggunakan simbol `||`.

```dart
var status = "closed";

var isStatusAccepted = status == "closed" || status == "approved"

print (isStatusAccepted)
```

Operator NOT

Secara sederhana operator *not* akan menyangkal suatu nilai boolean. Operator not dipakai dengan menambahkan simbol `!` didepan suatu variabel.

```dart
var age = 20;
var isAdult = age > 17;

print(!isAdult)
```

Pada contoh di atas `isAdult` akan bernilai true karena komparasi nilai variabel `age` lebih besar dari 17. Namun, dengan adanya operator not, maka nilai pada komparasi `if` menjadi `false` (not true berarti false dan not false berarti true).

Prioritas Operator dengan `()`

Mirip seperti pada operasi aritmatika, penggunaan operator di bahasa pemrograman Dart bisa diatur urutan prioritasnya, mana kondisi yang akan diuji terlebih dahulu dengan menggunakan simbol `()`. Penambahan simbol dalam kurung tersebut akan membantu kode menjadi lebih mudah dibaca ketika satu proses komparasi melibatkan banyak kondisi.

```dart
print((6 > 2 && 1 < 3) || 1 < 3); // true
```

Ternary Operator

Saat memiliki kondisi `if-else` seperti pada contoh di bawah:

```
var level = 101;
String message;

if (level > 100) {
    message = "You win!";
} else {
    message = "Play more games";
}
```

Bisa kita tulis ulang dengan *ternary operator* untuk mendapatkan kode yang lebih ringkas.

```
String message = (level > 100) ? "You win!" : "Play more games";
```

Sintaks menggunakan *ternary operator* adalah:

```
(kondisi yang ingin diperiksa) ? nilai jika true : nilai jika false;
```

Terkadang menggunakan *ternary operator* membuat program menjadi tidak begitu jelas di baca. Kita harus lebih cermat memahami alur kode yang ditulis daripada menggunakan `if-else` biasa yang meskipun lebih panjang namun *logic* masing-masing blok akan terlihat lebih jelas.

Switch

Selain menggunakan `if-else` juga *ternary operator*, cara lain untuk melakukan kontrol alur program adalah menggunakan perintah `switch`. Aturan penulisannya sebagai berikut:

```
var command = 'OPEN';
switch (command) {
  case 'CLOSED':
    // do something
    break;
  case 'PENDING':
    // do something
    break;
  case 'APPROVED':
    // do something
    break;
  case 'DENIED':
    // do something
    break;
  case 'OPEN':
```

```
16      // do something
17      break;
18    default:
19      print("Command is not valid");
20  }
```

Setelah menuliskan perintah `switch`, tulis nama variabel yang ingin diperiksa isinya di dalam tanda kurung. Setiap `case` akan memeriksa apakah variabel yang ada di dalam tanda kurung nilainya sama dengan yang ditentukan oleh `case`. Jika sama, maka kode yang berada di antara titik dua dan `break;` akan dieksekusi.

Perintah `break` berfungsi untuk mengelompokkan satu `case` dengan `case` lain. Bila suatu `case` tidak memiliki `break` dan di bawahnya ada `case` lain, maka blok `case` di bawahnya akan ikut dieksekusi.

Blok `default` berfungsi sama dengan blok `else`, dieksekusi hanya bila tidak ada `case` yang sesuai.

Seperti percabangan `if`, dengan `switch` kita juga bisa memakai *logical operator*:

```
1  switch (value) {
2    case value == 2 || (value / 2) == 0 : // or
3    case value > 0 && value % 2 == 1: // and
4    default:
5      print("Error");
6  }
```

Perulangan

Pada pembahasan bab-bab esbelumnya kita telah belajar tentang bagaimana mendeklarasikan suatu variabel, bagaimana menentukan tipe data dari suatu variabel, serta bagaimana membuat program yang kita buat bisa memutuskan sesuatu berdasarkan suatu nilai. Program-program tersebut ditulis dalam program main yang dieksekusi baris demi baris sampai dengan selesai. Bab ini akan membahas tentang bagaimana suatu program bisa mengulang-ulang suatu proses sesuai dengan apa yang kita inginkan. Perulangan sering dipakai saat melakukan pencarian, melakukan pengurutan data, atau proses otomatisasi yang harus dilakukan berulang kali.

Perulangan For

Perulangan `for` merupakan salah satu konsep yang ada hampir disemua bahasa pemrograman. Aturan penulisan perulangan `for` di pemrograman Dart mirip seperti pada bahasa pemrograman turunan C lainnya.

```
for (var i = 0; i < 5; i++) {
    print(i);
}
```

Bila pembaca pernah belajar bahasa pemrograman lain sebelumnya, pasti familiar dengan blok `for` di atas. Jika ini pertama kalinya, jangan risau, mari kita bahas satu demi satu.

Sebelum proses perulangan berlangsung, kita akan memulainya dengan menentukan sebuah variabel yang menentukan kapan dan berapa kali proses perulangan terjadi. Deklarasi `var i = 0` pada contoh di atas bisa dinamakan apa saja, `i` di sini diambil dari kata *index*. Variabel tersebut diberikan sebuah nilai awal, biasanya angka `0`, tapi juga bisa diatur sesuai kebutuhan. Jangan lupa perhatikan titik komanya.

Setelah menentukan nilai awal, tahap berikutnya adalah memberikan kondisi kapan perulangan ini terus berjalan. Selama kondisi bernilai `true`, maka perulangan akan terus berjalan sampai ia hasilnya menjadi `false`. Pada contoh ini, program akan terus berjalan selama variabel `i` nilainya di bawah lima, ketika variabel `i` nilainya menjadi 5, perulangan akan berhenti.

Tahap berikutnya adalah melakukan *increment* atau *decrement* yaitu proses untuk mengubah nilai variabel ini. Bila variabel `i` nilainya tidak berubah, maka perulangan kita beresiko untuk tidak pernah berhenti atau istilah kerennya *looping forever*/*invite loop*.

Kode `i++` pada contoh sama artinya dengan `i = i + `, artinya kita ingin menambahkan angka satu ke variabel `i`. Bila sekarang `i` bernilai `3`, maka `i++` akan membuat variabel `i` bernilai `4`. Proses

increment baru akan terjadi ketika semua kode pada blok `for` telah tereksekusi, artinya setelah kode `print(i)` dijalankan, nilai `i` tersebut baru di *increment*-kan.

Bila contoh program di atas dibalik untuk menuliskan angka `4` sampai dengan `0` dengan *decrement* maka hasilnya adalah:

```
1  for (var i = 4; i >= 0; i--) {
2      print(i);
3  }
```

Perulangan While

Perulangan dengan `while` memiliki struktur yang sedikit berbeda dengan `for`. Contoh penulisan blok `while` adalah sebagai berikut:

```
1  while(true) {
2      // kode program yang lain
3  }
```

Perulangan while akan memeriksa kondisi yang diberikan di dalam tanda kurung. Selaam bernilai `true` maka program akan jalan terus. Apabila contoh pada perulangan `for` ditulis ulang dengan ‘while, maka hasilnya akan menjadi:

```
1  var i = 0;
2
3  while (i < 5) {
4      print(i);
5      i++;
6  }
```

Pertama kita deklarasikan dulu variabel `i` untuk mengontrol perulangan sehingga tidak terjadi *infinite loop*. Selanjutnya, `while` akan memeriksa kondisi apabila variabel `i` nilainya di bawah `i` maka lakukan perintah `print`, lalu *increment* variabel `i`. Ketika i sudah bernilai `4`, maka program akan berhenti.

Perulangan Do While

Perulangan dengan `do while` memiliki sedikit perbedaan dengan `while`. Perbedaannya, kondisi perulangan diperiksa diakhir blok bukan di awal. Oleh karena itu, ketika menggunakan `do while`, blok perulangan pasti akan dijalankan dulu minimal satu kali.

Contoh program perulangan sebelumnya bila ditulis dengan `do while` akan menjadi:

```
var i = 0;
do {
    print(i);
    i++;
} while (i < 5);
```

Program di atas akan menghasilkan keluaran yang sama, hanya saja struktur penulisannya yang berbeda.

Break, Keluar dari Loop

Sebuah yang memiliki perulangan tidak harus selalu menunggu perulangan itu untuk selesai. Dalam kasus tertentu kita perlu untuk keluar dari proses perulangan meskipun belum mencapai kondisi dimana seharusnya perulangan itu berhenti.

```
  var i = 1;

  while (i < 10000) {
    print(i);

    if (i == 999) {
      break;
    }

    i++;
  }
```

Pada contoh kode di atas, terdapat dua kondisi untuk berhenti melakukan perulangan. Kondisi yang pertama saat variabel `i` bernilai `10000` dan kondisi yang kedua saat variabel i bernilai `999`.

Saat nilai i bernilai `999`, meskipun belum memenuhi kondisi yang seharusnya, tapi kode `break` akan menyebabkan perulangan untuk behenti.

Continue, Lanjut ke Perulangan Berikutnya

Terkadang ada keperluan dimana kita tidak perlu menghentikan proses perulangan, mungkin kita cukup melewati proses perulangan tertentu. Perhatikan contoh berikut:

```
for (var i = 0; i < 100; i++) {
  if (i % 10 == 0) {
    continue;
  }
  print(i);
}
```

Program di atas akan mencetak setiap angka dari 0 sampai 999, tapi setiap angka kelipatan 10 akan dilewati. Perintah `continue` akan langsung memanggil proses *increment* (`i++`), lalu kembali ke awal program.

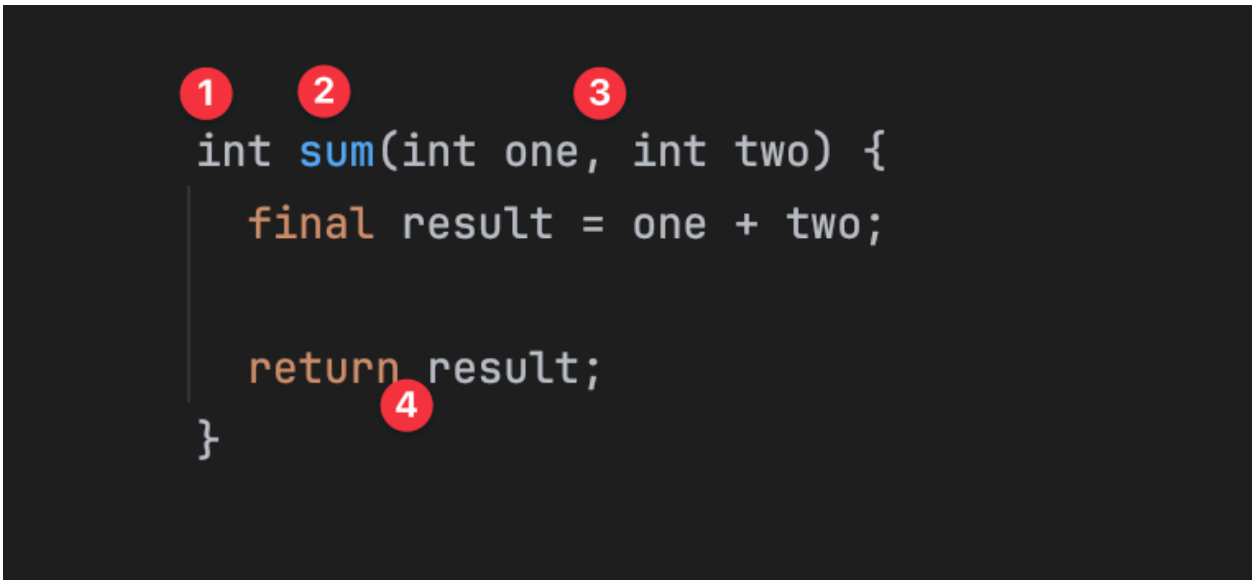
Fungsi

Sampai dengan bab perulangan, aplikasi Dart yang kita tulis selalu berada dalam sebuah fungsi bernama `main`. Tapi, apa itu fungsi?

Fungsi merupakan sebuah blok kode yang bisa dipanggil atau dipakai ulang (*reuseable*) tanpa menuliskan kodenya dua kali. Sebuah program yang kompleks akan terdiri dari sekelompok fungsi yang masing-masing memiliki tugasnya sendiri.

Dalam pemrograman, ada sebuah konsep bernama *don't repeat yourself*. Saat ada kode yang sama tapi dipakai berulang kali (dengan *copy paste* contohnya), maka itu lah saat kita membutuhkan sebuah fungsi.

Sebuah fungsi terdiri dari empat komponen yaitu *return type* (1), nama fungsi (2), parameter (3), dan *return value* (4).



Sebuah fungsi dibuat dengan menentukan *return_type*-nya. Apa yang akan di berikan oleh suatu fungsi? Ketika kita menggunakan pemanggang roti maka yang akan di dapatkan adalah roti yang telah matang. Saat menggunakan *juicer* maka yang akan kita dapatkan adalah jus. Bila menggunakan penanak nasi, maka hasil yang akan kita dapatkan adalah nasi yang matang. Saat menulis sebuah fungsi, kita tentukan data apa yang akan dihasilkan.

```dart
int sum(int one, int two) {
    return one + two;
}

String generateOrderCode() {
    return "ORD/25052024/0001";
}

void greetings() {
    print("Welcome back");
}

void main() {
    greetings();

    var result = sum(1, 2);

    var code = generateCode();
}
```

Pada tiga contoh di atas kita memiliki empat buah fungsi yang masing-masing akan memberikan sebuah data berupa integer, string dan dua fungsi tidak memiliki return type.

Saat suatu fungsi memiliki return type, hasilnya bisa kita tampung dalam sebuah variabel yang bisa dipakai untuk proses lainnya. Tipe data yang ditulis pada definisi fungsi harus sama dengan data yang di-return.

```dart
int circle(double r) {
    var result = 3.14 * r * r;

    return result;
}
```

Program di atas tidak akan bisa di *compile* karena variabel `result` akan menampung data bertipe `double`, sementara fungsi `circle` memiliki return `int`. Agar bisa di compile, tipe data dan apa yang di *return* harus disesuaikan.

Parameter Fungsi

Saat menggunakan penanak nasi, ia akan meminta dua hal, beras dan air. Beras dan air dalam konsep fungsi merupakan parameter. Tanpa kedua parameter ini, penanak nasi tidak akan berfungsi. Ketika kita menyatakan bahwa suatu fungsi dalam program Dart membutuhkan suatu data, maka tulislah apa saja yang dibutuhkan dalam definisi fungsi tersebut.

```
double circle(double r) {
    var result = 3.14 * r * r;

    return result;
}
```

Pada contoh program penghitung luas lingkaran di atas, kita membutuhkan data berupa jari-jari. Bila jari-jari tidak ada, maka luas lingkaran tidak mungkin bisa di hitung. Kita juga menentukan bahwa jari-jari ini harus bertipe integer, user tidak bisa mengirimkan tipe data yang lain.

```
void main() {
    circle("14"); // Error
    circle(3); // Error
}
```

Multiple Parameter

Parameter boleh ditulis lebih dari satu buah. Setiap penulisannya dipisahkan oleh koma.

```
double rectangle(double length, double width) {
    return length * width;
}

void main() {
    final length = 4.0;
    final width = 10.0;
    print("Luas persegi panjang = ${rectangle(length, width)}");
}
```

Saat memanggil fungsi, kita perlu memberikan urutan data yang sesuai dengan permintaan parameternya. Salah mengirimkan urutan posisi parameter bisa saja memberikan hasil yang tidak sesuai. Karena parameter meminta data yang harus sesuai posisi, maka penulisan di atas juga biasa disebut dengan *positional parameter*.

Optional Parameter

Saat menggunakan positional parameter, semua parameternya harus lengkap. Bila kita hanya mengirimkan satu parameter saja, maka program Dart tidak bisa dijalankan.

```
rectangle(4.0);
//Error: Too few positional arguments: 2 required, 1 given.
```

Pesan error di atas menunjukkan jenis errornya, harus ada 2 parameter, tapi hanya 1 yang diberikan. Argumen hanyalah istilah bagi parameter yang ditulis saat memanggil fungsi.

Anggaplah kita ingin agar fungsi penghitung luas persegi panjang di atas bisa menerima hanya satu parameter. Bila hanya 1 parameter yang diberikan, kita anggap user akan menghitung luas persegi karena panjang sisinya sama.

```
double rectangle(double length, [double? width]) {
    if (width == null) return length * length;

    return length * width;
}
```

Untuk menentukan bahwa suatu parameter bersifat *optional* atau boleh tidak diberikan, tuliskan di dalam tanda kurung siku. Karena boleh dikosongkan, maka tipe datanya wajib menggunakan tanda tanya diakhir yang menandakan variabel tersebut merupakan *nullable*. Sekarang, program di atas bisa kita panggil dengan 2 cara:

```
rectangle(4.0);
rectangle(4.0, 10.0);
```

Yang perlu pembaca ingat, *optional parameter* harus disimpan terakhir. Kita tidak bisa menuliskan `[double? width]` sebelum `length`. *Optional parameter* juga boleh lebih dari satu, bahkan semua parameter boleh dijadikan *optional*.

Default Value

Saat menggunakan *optional paramater*, sistem Dart akan memberikan nilai null pada variabel `width` saat fungsi `rectangle` hanya mengirimkan satu argumen. Dart juga memungkinkan kita untuk menentukan sendiri apa nilai *default* bagi suatu parameter.

```
int multiply(int a, [int b = 1]) {
    return a * b;
}

multiply(2, 2); // 4
multiply(5); // 5
```

Program di atas akan mengalikan variabel `a` dengan variabel `b`. Apabila fungsi `multiply` hanya memberikan satu argument untuk variabel `a` saja, maka variabel `b` secara otomatis menampung nilai `1` sesuai dengan *default value* yang tertulis.

Berbeda dengan optional parameter yang sebelumnya bisa bernilai null, saat diberikan *default value*, parameter ini pasti akan mendapatkan nilai meskipun tidak dikirim. Oleh karena itu tipe datanya boleh tidak berupa nullable.

Named Parameter

Fungsi di pemrograman Dart memiliki fitur bernama *named parameter* untuk membuat nama parameter pada fungsi yang dipanggil lebih jelas. Untuk membuat *named parameter*, berikan tanda kurung kurawal pada parameter yang diinginkan:

```
1  void printTime(int hour, {int minute = 0, int second = 0}) {
2    var  hourStr = hour.toString().padLeft(2, "0");
3    var  minStr = minute.toString().padLeft(2, "0");
4    var  secStr = second.toString().padLeft(2, "0");
5
6    print("$hourStr:$minStr:$secStr");
7  }
8
9  void main() {
10     printTime(5); 05:00:00
11     printTime(5, second: 33, minute: 10); // 05:10:33
12 }
```

Variabel `minute` dan `second` berada di dalam tanda kurung kurawal, artinya kita harus menuliskan nama variabelnya saat memanggil fungsi tersebut. Perhatikan juga bahwa Dart tidak melihat urutan *named parameter* karena kita sudah menentukan apa parameter yang dituju. Sementara itu, variabel yang bukan *named parameter* harus ditulis sesuai urutan penulisannya.

```
1  void printTime({int hour, int minute = 0, int second = 0}) {
2  }
3
4  void main() {
5     printTime();
6  }
```

Pada contoh berikutnya, kita membuat `hour` menjadi *named parameter*. Akan tetapi setiap *named parameter* merupakan *optional* sehingga bila tidak diberikan *default value* ia harus dideklarasikan sebagai nullable (`int?  hour`). Bila kita tidak menentukan *default value*, maka kita bisa menambahkan `required` agar saat fungsi dipanggil, parameter tersebut tetap wajib dituliskan. Agar suatu *named parameter* wajib dikirim saat fungsi dipanggil, kita bisa menambahkan `required`.

```dart
void printTime({required int hour, int minute = 0, int second = 0}) {
}

void main() {
    printTime(hour: 3); // 03:00:00
}
```

Parameter Tanpa Tipe Data

Suatu parameter bila tipe datanya tidak dituliskan akan otomatis diberikan tipe data `dynamic`.

```dart
double circle(r) {
    var result = 3.14 * r * r;

    return result;
}
```

Program akan tetap bisa di *compile*, kita pun bisa memanggil fungsi ini dengan mengirimkan `int` maupun `double`, tapi bila mengirimkan data yang tidak sesuai, maka aplikasi akan crash.

```dart
void main(){
    circle("15");
    // Unhandled exception:
    // type 'String' is not a subtype of type 'num'
}
```

Visual Studio Code atau editor lainnya tidak bisa mendeteksi error sebelum di *compile* karena parameter `r` tidak ditentukan tipe datanya.

```
void main() {
  print(circle("Hello"));
}

double circle(r) {
  print(r.runtimeType);
  var result = 3.14 * r * r;

  return result;
}
```

Hal yang sama juga berlaku untuk *return type* fungsi. Secara umum penulis lebih menyarankan untuk menuliskan tipe data secara eksplisit untuk menghindari berbagai error yang mungkin muncul.

Arrow Function

Fungsi yang isinya hanya terdiri dari satu baris bisa ditulis dengan *arrow function*.

```
1 int add(int a, int b) {
2
3 }
```

Fungsi di atas bisa ditulis ulang menjadi:

```
int add(int a, int b) => a + b;
```

Blok dalam kurung kurawal dan perintah `return` digantikan oleh tanda panah `=>`. Penggunaan tanda panah hanya berlaku bagi fungsi yang memiliki *return type* dan bisa ditulis dalam satu baris saja. Bila blok kodenya lebih dari satu baris, maka tidak bisa dituliskan sebagai *arrow function*.

Class dan Object

Tipe data yang sudah kita gunakan sampai bab ini merupakan tipe data yang sudah ditentukan sebelumnya seperti `String`, `int`, atau `bool`. Kali ini kita akan mendiskusikan bagaimana membuat tipe data sendiri dengan menggunakan `class`.

Class bagaikan sebuah template yang di dalamnya terdapat variabel, fungsi dan komponen-komponen lain. Dalam konsep pemrogramn berorientasi objek/*object oriented programming (OOP)*, class sering dicontohkan sebagai sebuah *blueprint* sebuah desain awal yang produksinya bisa menghasilkan berbagai jenis objek. Contoh, Mitsubishi memiliki satu desain mobil Expander, tapi dari satu desain itu mereka bisa memproduksi mobil dengan berbagai macam warna, berbagai macam jenis tempat duduk, tapi juga masih memiliki mesin yang sama, rangka yang sama, setir yang sama dan lain sebagainya.

Semua nilai dalam pemrograman Dart merupakan sebuah objek. Objek dalam konsep OOP adalah variabel yang dibuat dari sebuah class. Sejatinya tipe data seperti int, double dan bool semuanya merupakan objek. Class merupakan komponen mendasar pemrogaman berorientasi objek, saat nanti membuat aplikasi mobile dengan Flutter, kita akan memakai class di mana-mana.

Cara Membuat Class

Berikut ini merupakan contoh class sederhana dengan dua buah properti:

```
1  class Student {
2      int age = 0;
3      String name = '';
4  }
```

Sebuah class didefinisikan dengan menulis *keyword* **class** diikuti dengan nama yang diinginkan. Nama suatu class ditulis dengan format CamelCase lalu diikuti oleh blok kurung kurawal. Pada contoh di atas, kelas `Student` memiliki dua buah properti yaitu `age` dengan tipe data `int` serta `name` dengan tipe data `String`. Properti adalah istilah bagi suatu variabel yang dideklarasikan di dalam suatu `class`. Istilah lain bagi properti yang mungkin akan sering pembaca temukan adalah `fields`.

Membuat Objek dari Sebuah Class

Seperti yang sudah pernah disinggung sebelumnya, sebuah objek adalah sesuatu yang dibuat dari suatu class atau istilah lain bagi sebuah objek adalah suatu `instance`. Berikut contoh membuat sebuah objek dari class `Student`.

```
var andi = Student();
```

Kita membuat sebuah *instance* dari class `Student` dengan nama `andi`. Perhatikan tanda kurung setelah menuliskan nama class. Tanda kurung ini wajib dituliskan dan berfungsi untuk memanggil *constructor* dari kelas `Student` yang akan kita bahas selanjutnya.

Membuat objek dengan cara yang dicontohkan baru bisa dilakukan pada versi Dart 2.0. Pada versi-versi sebelumnya kita wajib menuliskan keyword `new` sebelum menulis nama kelas. Pada saat itu kode yang harus kita tuliskan adalah sebagai berikut:

```
var andi = new Student();
```

Meskipun masih bisa dipakai namun penulisan keyword `new` tidak begitu disarankan. Bahkan editor seperti Android Studio akan menyarankan kita untuk tidak menuliskannya.

Mengubah Nilai Properti

Kelas `Student` yang kita buat memiliki dua buah properti yaitu `age` dan `name`. Untuk mengisi nilai baru pada properti kita akan mengaksesnya menggunakan *dot notation*.

```
void main() {
  final andi = Student();
  andi.name = "Andi Malarangeng";
  andi.age = 23;
  print(andi.name);
}

class Student {
  int age = 0;
  String name = '';
}
```

Bila kode di atas dijalankan maka kita akan mendapat `Andi Malarangeng` di cetak oleh perintah print. Apa yang terjadi bila yang kita print adalah objeknya?

```
print(andi);
// Instance of 'Student'
```

Hasil yang kita dapat tentu tidak memiliki informasi yang berharga. Umumnya, saat mencetak suatu objek kita juganingin melihat semua atau sebagian dari properti yang dimiliki objek tersebut.

Setiap objek yang ada di Dart memiliki method `toString` untuk mengatur teks yang didapatkan saat objek di print.

```
class Student {
  int age = 0;
  String name = '';

  @override
  String toString() {
    return "Student(age: $age, name: $name)";
  }
}
```

Sesuatu yang di awali dengan tanda `@` disebut dengan *annotation*. *Annotation* berfungsi untuk memberitahu *compiler* informasi mengenai kode yang ada di bawahnya. Khusus annotation `@override`, ia memberi tahu bahwa method `toString` diturunkan dari class lain dan class yang kita miliki akan memiliki implementasi yang berbeda.