

# Pengenalan Dart



# Dart

## Apa itu Dart?

Pernahkah kalian mendengar atau bahkan mengembangkan aplikasi Flutter? Jika iya, berarti kalian sudah tahu bahwa Flutter ditulis menggunakan bahasa Dart. Jika belum, maka selamat berkenalan dengan Dart.

**Dart** adalah bahasa pemrograman yang *open source* dan *general purpose*. Dart dikembangkan oleh Google dan ditujukan untuk membuat aplikasi *multiplatform* seperti *mobile*, *desktop*, dan *web*.

Dart awalnya dikenalkan pada *GOTO conference* pada tahun 2011. *Project* ini didirikan oleh Lars Bak dan Kasper Lund dari Google, sampai akhirnya versi Dart 1.0 dirilis pada 14 November 2013. Bulan Agustus 2018, Dart 2.0 dirilis dengan perubahan bahasa seperti perubahan *type system*. Saat ini versi Dart terbaru yang *stable* adalah 2.7 dengan fitur baru *extension methods* yang nanti akan kita bahas lebih lanjut.

Jauh sebelum ada Flutter, Dart awalnya digunakan untuk membuat web yang ada di Google. Tujuan awal pembuatan Dart adalah untuk menggantikan JavaScript yang dinilai memiliki banyak kelemahan. Sejak saat itu, rilisnya Flutter SDK untuk pengembangan iOS, Android, dan web menjadi sorotan baru pada bahasa Dart.

## Kenapa perlu belajar Dart?

Dalam buku “*The Pragmatic Programmer*” disebutkan bahwa untuk menjadi *professional software developer* kita perlu belajar setidaknya satu bahasa baru setiap tahunnya. Lalu adakah alasan yang bagus untuk memulai belajar pemrograman dengan Dart?

- Pertama, ***Dart adalah bahasa pemrograman yang fleksibel***. Dart bisa berjalan di mana pun baik itu Android, iOS, maupun web. Sebagai *developer*, tentunya sebuah keuntungan jika bisa menuliskan kode dan bisa berjalan di mana saja.
- ***Dart adalah project open-source***. Dart dibuat oleh Google, lalu bersama dengan komunitas *developer* Dart mengembangkan teknologi dan fitur-fitur menarik yang bisa ditambahkan pada Dart. Jika Anda menemukan *bug* atau masalah pada Dart, Anda dapat melaporkannya atau bahkan memperbaikinya sendiri. Selain itu Anda tidak perlu khawatir dalam masalah lisensi ketika menggunakan bahasa Dart. Anda dapat ikut berkontribusi pada bahasa Dart pada repositori berikut: <https://github.com/dart-lang>.
- ***Dart digunakan oleh Flutter***. Sejak kemunculan Flutter, Dart kembali menjadi perhatian. Saat ini ada banyak perusahaan yang menggunakan Flutter pada aplikasinya. Flutter bisa dibilang merupakan proyek yang revolusioner dari Google untuk mengembangkan aplikasi multiplatform dengan tampilan UI yang menarik. Untuk itu, jika Anda tertarik mengembangkan aplikasi dengan Flutter, maka menguasai Dart adalah hal yang fundamental.
- ***Dart memiliki dukungan tools yang lengkap***. Hampir setiap teks editor atau IDE memiliki dukungan besar untuk Dart. Anda dapat menggunakan IDE seperti IntelliJ IDEA, Webstorm, Android Studio maupun *editor* sederhana seperti VS Code, Sublime text, Atom, atau yang lainnya sesuai kenyamanan Anda.
- ***Dart mudah dipelajari dan bagus sebagai first language***. Anda akan bisa memahami Dart dengan cepat khususnya jika sudah familiar dengan bahasa pemrograman populer lain seperti Java, Python, JavaScript, dll. Bahkan jika Anda baru memulai pemrograman, Dart adalah bahasa yang bagus. Anda tidak perlu menginstal apapun, cukup memanfaatkan *online compiler* dari Dart, Anda sudah bisa menulis dan menjalankan aplikasi Dart. Selain itu, dokumentasi dan tutorial Dart yang disediakan Google cukup mudah untuk diikuti, ditambah dengan sintaks yang sederhana, dan komunitas yang bersahabat dalam membantu kita mempelajari Dart.

## Alasan

### Dart

Dart merupakan bahasa modern dan berfitur lengkap. Dart juga memiliki banyak kemiripan dengan bahasa lain yang sudah banyak dikenal seperti Java, C#, Javascript, Swift, dan Kotlin. Salah satu rancangan utama dari Dart adalah supaya bahasa ini familiar bagi *developer* Javascript dan Java/C#. Artinya, yang telah familiar dengan kedua bahasa tersebut dapat memulai belajar bahasa Dart dengan lebih mudah. Namun, jika Anda adalah calon *developer* yang baru memulai belajar pemrograman dan memutuskan Dart sebagai *first language*, tenang saja. Dart adalah bahasa yang nyaman dan mudah dipelajari untuk memulai pemrograman.

Kita ambil contoh potongan kode Dart berikut:

```
1. main() {  
2.   var name = 'Dicoding';
```

```
3. String language = 'Dart';
4. print('Hello $name. Welcome to $language!');
5. }
```

Jika Anda telah familiar dengan bahasa pemrograman lain seperti Java, Kotlin, atau Swift, tentu Anda telah paham bagaimana kode di atas bekerja. Jadi, kode di atas akan menampilkan “***Hello Dicoding. Welcome to Dart!***” pada konsol.

Dart sebagai bahasa memiliki beberapa karakteristik berikut:

- *Statically typed*,
- *Type inference*,
- *String expressions*,
- *Multi-paradigm: OOP & Functional*.

Dart adalah bahasa yang ***statically typed***, artinya kita perlu mendefinisikan variabel sebelum bisa menggunakannya. Potongan kode berikut adalah contoh deklarasi variabel pada Dart.

```
1. var name = 'Dicoding';
2. String language = 'Dart';
```

Bisa dilihat bahwa pada Dart kita tidak perlu mendefinisikan tipe data variabel secara eksplisit. Ini disebabkan karena Dart juga mendukung ***type inference***, di mana tipe data akan secara otomatis terdeteksi ketika suatu variabel diinisialisasi. Sebagai contoh variabel **name** di atas akan terdeteksi sebagai **String**. Selain itu, Dart juga memiliki *dynamic variable*. Apa itu? Bahasan ini akan kita dalami pada modul *variable*.

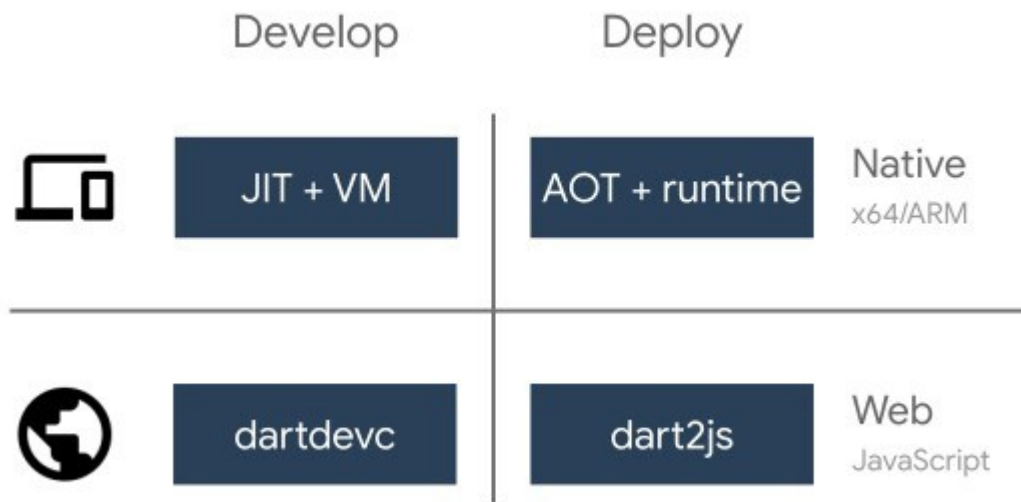
```
1. print('Hello $name. Welcome to $language!');
```

Kode di atas juga menunjukkan kalau Dart memiliki fitur ***String interpolation***. Ini adalah fitur di mana kita bisa menyisipkan variabel ke dalam sebuah objek String tanpa *concatenation* (penggabungan objek String menggunakan +). Dengan fitur ini, tentunya kita menjadi lebih mudah dalam membuat objek String yang dinamis.

## Dart Platform

Pada modul sebelumnya kita telah mempelajari bahwa Dart digunakan untuk menulis aplikasi *multi-platform*. Teknologi *compiler* yang fleksibel memungkinkan kode Dart dapat dijalankan dengan cara yang berbeda, tergantung target platform yang dituju.

- **Dart Native** : Ditujukan untuk program yang menargetkan perangkat seluler, *desktop*, *server*, dan lainnya. Dart Native mencakup **Dart VM** dengan kompilasi **JIT** (*just-in-time*) dan kompiler **AOT** (*ahead-of-time*) untuk menghasilkan kode mesin.
- **Dart Web** : Ditujukan untuk program yang menargetkan web. Dart Web menyertakan kompiler *development* (**dartdevc**) dan kompiler *production* (**dart2js**).



Dart Native (VM JIT dan AOT)

*Dart Native* memungkinkan kode Dart dijalankan dan dikompilasi dengan kode mesin ARM atau X64 native untuk aplikasi *mobile*, *desktop*, dan *server*.

Dart VM dilengkapi dengan ***just-in-time compiler (JIT)*** yang mendukung interpretasi murni dan optimasi runtime. Lalu apa keuntungan dari menggunakan JIT? *Compiler* bertugas untuk mengubah bahasa *high-level* yang kita tulis menjadi bahasa *low-level* yang dimengerti oleh mesin. *JIT compiler* akan mengubah bahasa pemrograman yang kita tulis menjadi *intermediate language* atau *bytecode* seperti pada Java, selanjutnya instruksi ke mesin akan dilakukan hanya ketika dibutuhkan, sehingga disebut *just-in-time*. Metode ini akan membuat proses iterasi program menjadi lebih efisien.

Saat aplikasi siap digunakan untuk *production*, Anda dapat memanfaatkan ***Dart AOT compiler***. Apa ini? Apa bedanya dengan JIT? *AOT compilation* akan mengubah bahasa *high-level* atau *intermediate-level* menjadi kode mesin pada mesin atau server sebelum aplikasi dijalankan. Kompilasi AOT akan menghasilkan *rendering* aplikasi yang lebih cepat dan ukuran yang lebih kecil karena kode telah dikompilasi sebelum aplikasi dijalankan.

Dart Web (JavaScript)

*Dart Web* memungkinkan kode Dart dijalankan pada platform web yang didukung oleh JavaScript. Dengan Dart Web, kode Dart akan dikompilasi ke kode JavaScript sehingga nantinya bisa berjalan di *browser*.

Dart Web menggunakan ***Dart dev compiler (dartdevc)***, *compiler* yang mengonversi kode Dart menjadi JavaScript. Alih-alih menggunakan *dartdevc* secara langsung, Anda dapat menggunakan ***webdev***, yakni alat yang mendukung tugas inti *developer* seperti menjalankan, membangun, dan *debugging*.

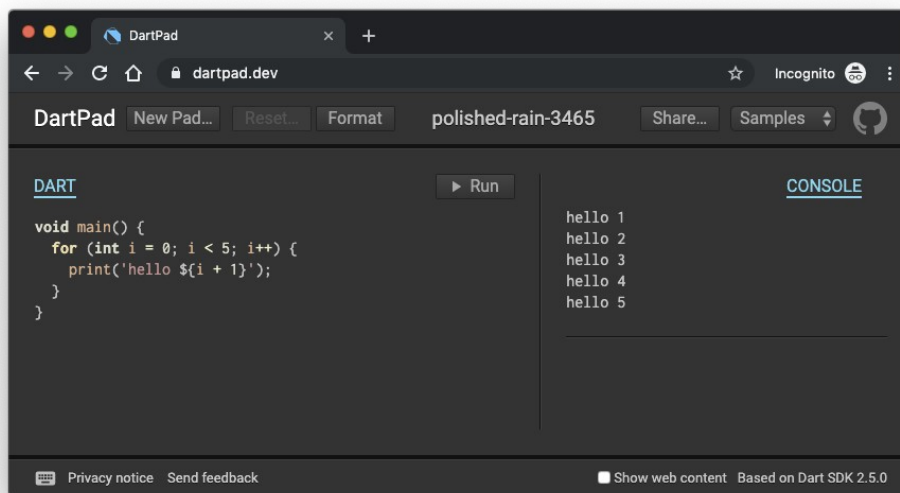
Untuk mengoptimalkan kode pada lingkungan *production*, terdapat ***dart2js compiler*** yang dapat mengompilasi kode Dart dengan cepat dan ringkas. *Dart2js* menggunakan teknik yang bisa mengeliminasi kode yang tidak perlu.

## Memulai Pemrograman dengan Dart

Pada modul ini kita akan mulai menyiapkan *tools* apa saja yang dibutuhkan untuk membuat dan menjalankan aplikasi Dart pertama kita.

Setiap *tools* yang akan digunakan bertujuan agar kita mampu mengembangkan aplikasi Dart secara lokal pada laptop atau komputer masing-masing. Namun, jika Anda ingin melakukannya secara *online*, Anda bisa memanfaatkan [DartPad](https://dartpad.dev/). Sebuah *tool* yang memungkinkan Anda menulis dan menjalankan kode Dart pada jendela *browser*. DartPad ini dapat Anda akses pada tautan <https://dartpad.dev/>.

Berikut ini adalah tampilan DartPad:



# Instalasi Dart SDK

Untuk bisa menjalankan Dart pada perangkat lokal, kita perlu menginstal Dart SDK. Dart SDK adalah kumpulan *library* dan *command-line tools* yang dibutuhkan untuk mengembangkan aplikasi *web*, *command-line*, dan *server* menggunakan Dart. Untuk pengembangan aplikasi *mobile*, Anda bisa langsung menggunakan Flutter SDK yang telah termasuk Dart SDK di dalamnya.

Dart memiliki dua versi rilis, yaitu *stable* dan *dev*. *Stable releases* adalah versi Dart SDK yang stabil dan biasanya diperbarui setiap 6 minggu. Saat ini *stable release* Dart adalah versi 2.7.1 sedangkan *dev channel* adalah versi *preview* yang biasanya di-*update* setiap satu minggu.

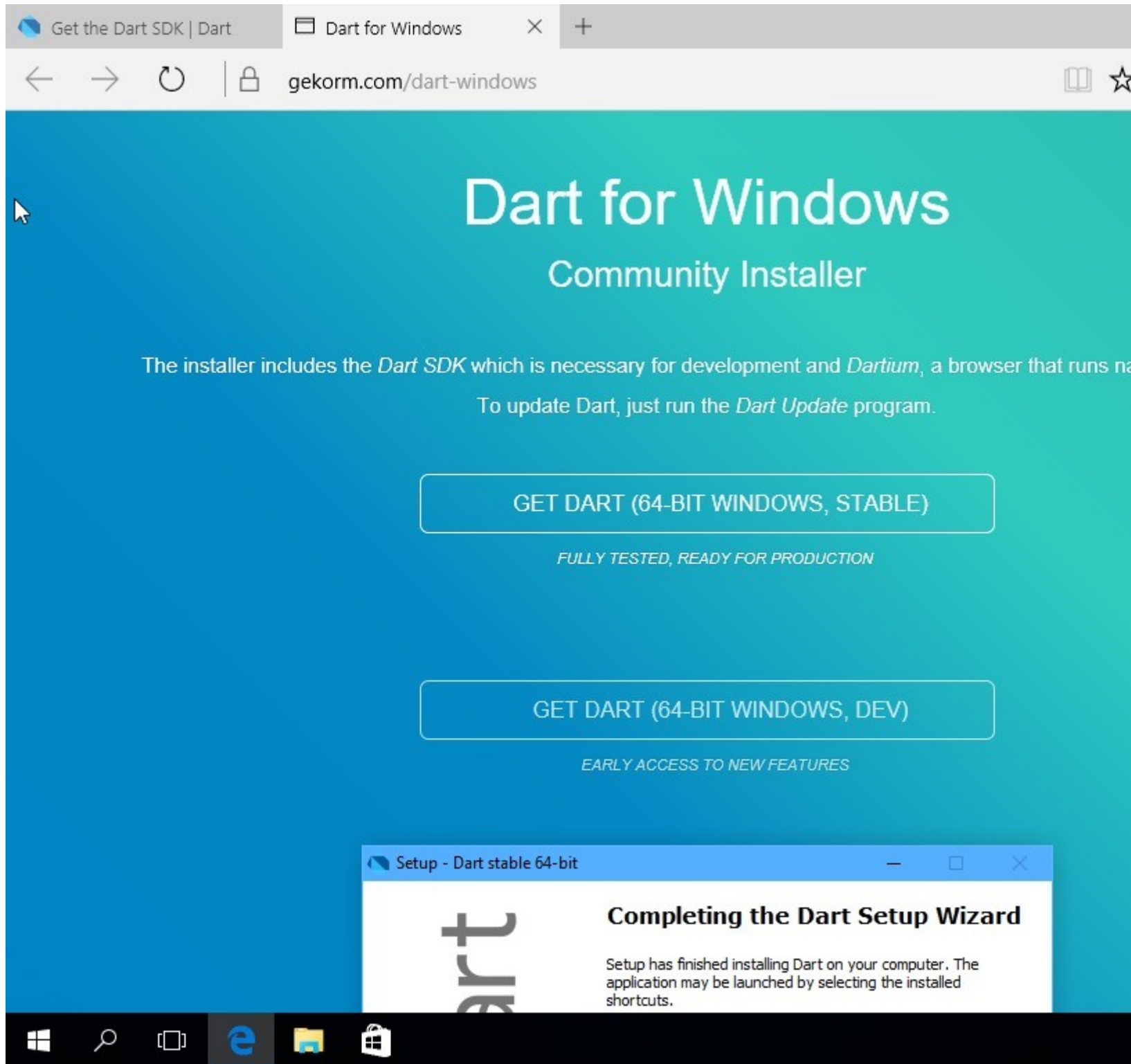
Anda dapat memilih untuk menggunakan versi *dev* jika ingin mendapatkan akses lebih awal ke berbagai fitur dan perbaikan baru.

Dart SDK dapat Anda instal sesuai dengan sistem operasi yang Anda gunakan.

## Windows

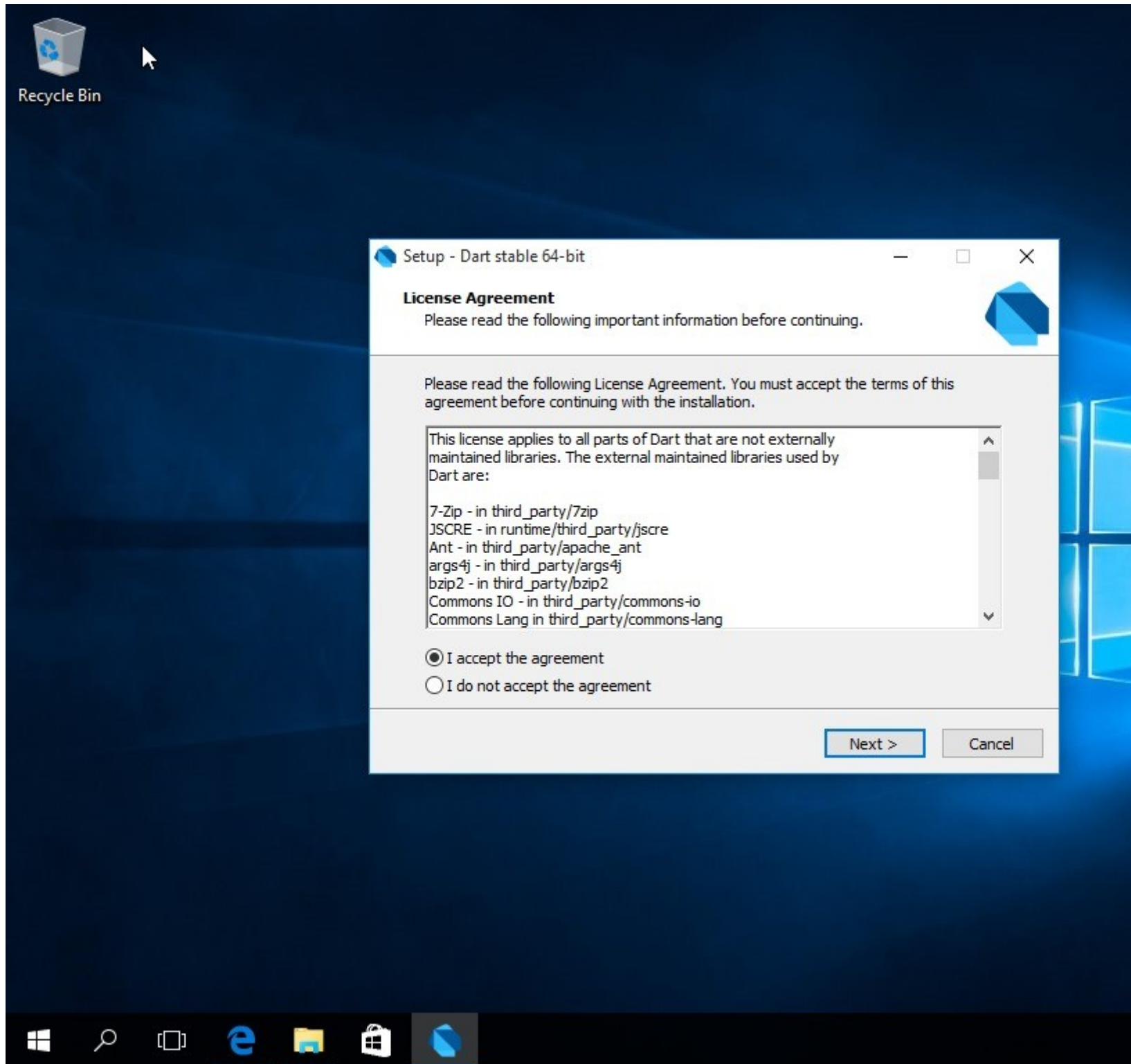
Jika Anda menggunakan sistem operasi Windows, Anda dapat mengunduh Dart SDK pada tautan berikut: <https://www.gekorm.com/dart-windows>.

Anda dapat memilih menggunakan Dart SDK versi *stable* atau pun *dev*.



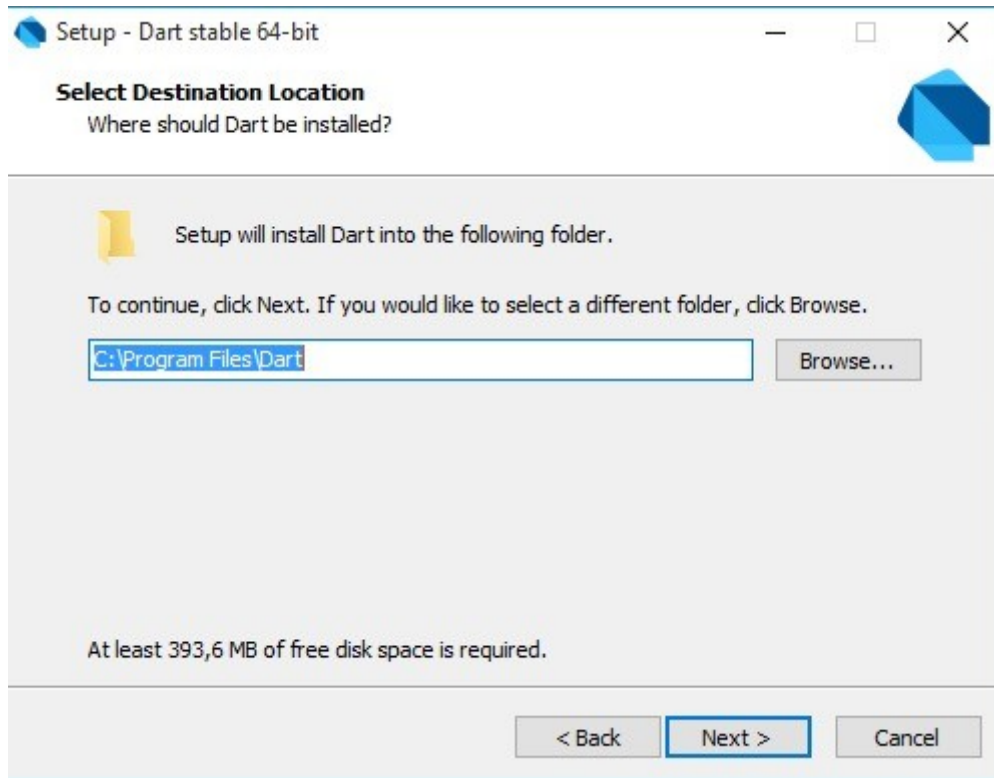
Setelah diunduh, *double click* berkas tadi lalu ikuti langkah-langkah instalasi.





Tentukan lokasi instalasi Dart SDK.

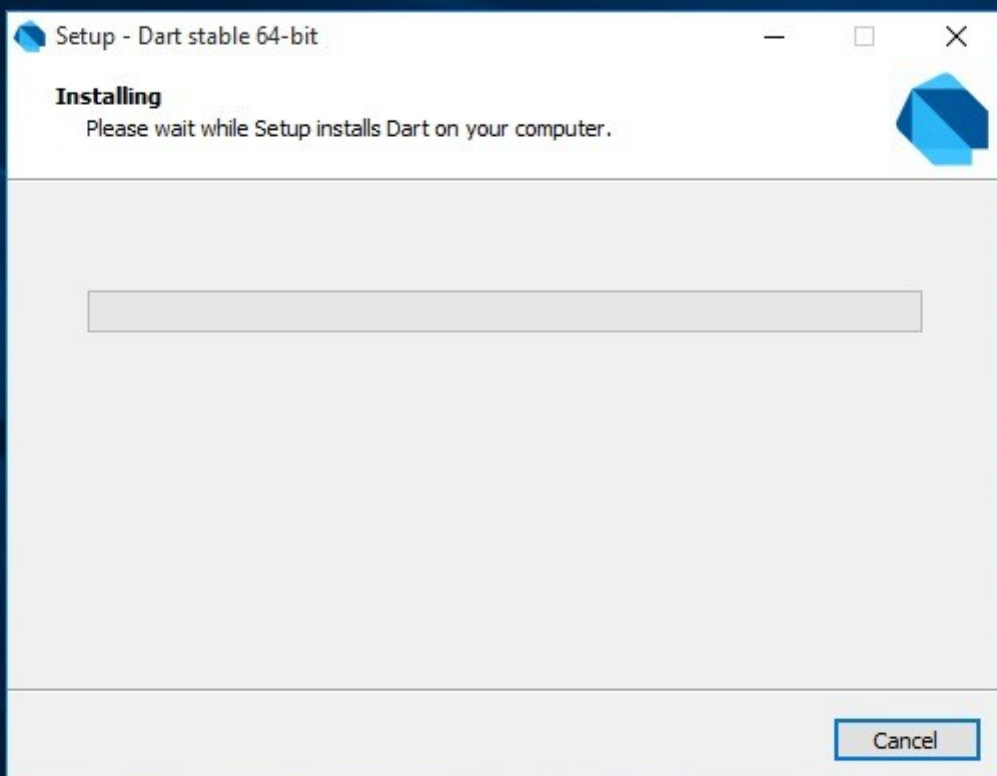




Tunggu proses instalasi hingga selesai.

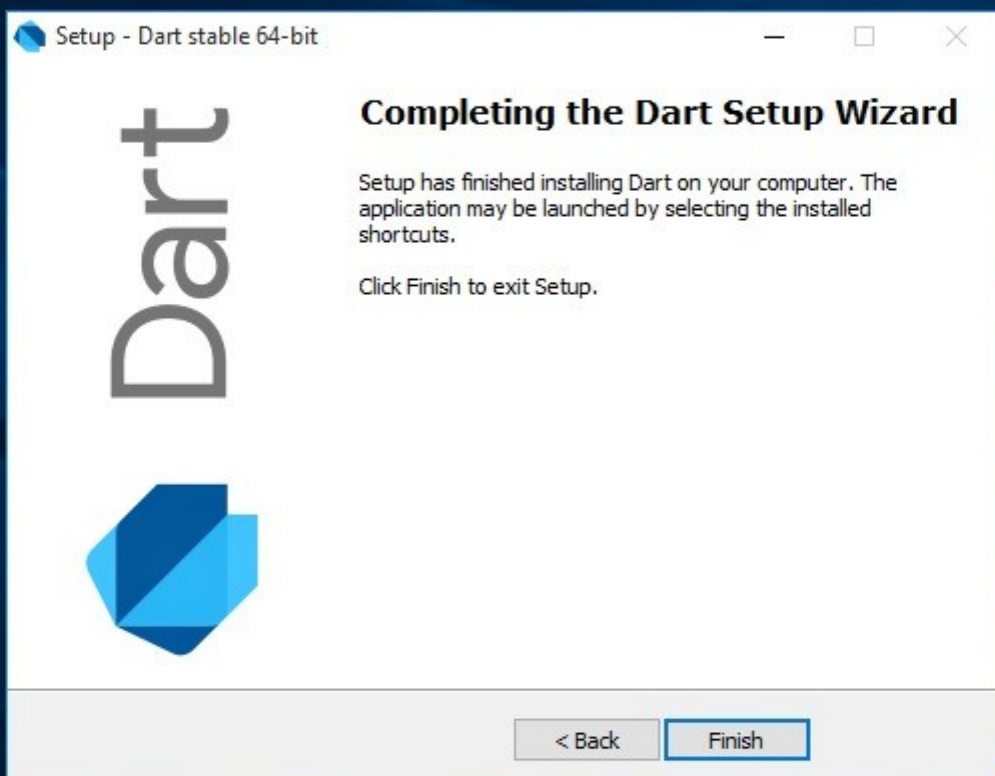


Recycle Bin





Recycle Bin



## Linux

Bagi pengguna linux, Anda dapat menginstal Dart SDK melalui *terminal* dengan perintah `apt-get`.

Jalankan perintah berikut untuk *set up* Dart SDK:

```
1. sudo apt-get update
2. sudo apt-get install apt-transport-https
3. sudo sh -c 'wget -qO- https://dl-ssl.google.com/linux/linux_signing_key.pub | apt-key add -'
4. sudo sh -c 'wget -qO- https://storage.googleapis.com/download.dartlang.org/linux/debian/dart_stable.list > /etc/apt/sources.list.d/dart_stable.list'
```

Kemudian jalankan perintah berikut untuk menginstal Dart SDK versi stable:

```
1. sudo apt-get update
2. sudo apt-get install dart
```

Jika Anda ingin untuk menginstal versi dev, gunakan perintah berikut:

```
1. sudo sh -c 'wget -qO- https://storage.googleapis.com/download.dartlang.org/linux/debian/dart_unstable.list > /etc/apt/sources.list.d/dart_unstable.list'
2. sudo apt-get update
3. sudo apt-get install dart
```

Selanjutnya kita perlu menambahkan folder bin dari Dart SDK yang telah terpasang ke environment PATH. Caranya jalankan perintah berikut:

```
1. echo 'export PATH="$PATH:/usr/lib/dart/bin"' >> ~/.profile
```

## Mac

Untuk Mac, Anda dapat menginstal Dart SDK menggunakan [Homebrew](#).

Jalankan perintah berikut:

```
1. brew tap dart-lang/dart
2. brew install dart
```

Jika Anda ingin menggunakan versi dev, tambahkan perintah `--devel`:

```
1. brew install dart -- --devel
```

## Instalasi IDE

Dart adalah bahasa yang fleksibel dengan *tools* pendukung yang cukup banyak. Mulai dari DartPad di mana Anda dapat bereksperimen dengan fitur-fitur Dart secara *online* tanpa perlu mengunduh *software* apa pun.

Namun jika Anda memiliki kebutuhan untuk menyimpan *project* dan mengerjakannya secara lokal, menggunakan teks editor atau *Integrated Development Environment (IDE)* adalah opsi yang tepat. Anda dapat menggunakan Dart pada IDE populer seperti Android Studio, IntelliJ IDEA, atau Visual Studio Code dengan memanfaatkan *plugin* yang disediakan. Selain itu *plugin* Dart juga bisa digunakan pada *editor* seperti Emacs, Atom Vim, dan Eclipse.

Untuk kemudahan dan meningkatkan produktivitas dalam mengembangkan aplikasi, kita akan menggunakan IDE IntelliJ IDEA. IDE ini memiliki beberapa fitur yang sangat membantu kita sebagai *developer* seperti *code suggestions*, *shortcut*, *version control*, dll. Guna mengenal IntelliJ IDEA lebih lanjut, silakan baca blog berikut: [Berkenalan Dengan IntelliJ IDEA](#).

Sebelum melakukan instalasi IntelliJ IDEA, ketahui terlebih dahulu beberapa syarat di masing-masing sistem operasi sebagai berikut:

- [Windows](#)
- [macOS](#)
- [Linux](#)
- Microsoft Windows 10/8/7/Vista/2003/XP (incl.64-bit)
- 2 GB RAM minimum, 4 GB RAM direkomendasikan
- 1.5 GB ruang kosong pada penyimpanan
- 1024x768 minimal resolusi layar

Pastikan komputer Anda memenuhi syarat di atas. Jika sudah, unduhlah berkas instalasi IntelliJ IDEA berdasarkan sistem operasi yang digunakan pada tautan <https://www.jetbrains.com/idea/download/>. Anda bisa mengunduh versi *Community* untuk penggunaan lisensi secara gratis. Selanjutnya ikuti langkah-langkah instalasi sesuai sistem operasi yang digunakan.

## Linux

Setelah berhasil mengunduh berkas, ekstrak berkas tersebut dengan menggunakan perintah yang dijalankan pada terminal berikut:

```
1. tar xvf ideaIC-2019.3.2.tar.gz
```

Lalu, masuk ke dalam folder bin dengan menggunakan perintah:

```
1. cd idea-IC-193.6015.39/bin/
```

Terakhir, jalankan *shell script* `idea.sh` dengan menggunakan perintah:

```
1. ./idea.sh
```

Ikuti instruksi pada jendela yang tampil untuk menyelesaikan proses instalasi.

#### Windows dan macOS

Berbeda dengan Linux, jika menggunakan Windows atau macOS, Anda tidak perlu mengekstrak hasil unduhan. Anda bisa langsung menjalankan berkas `idealC-2019.3.2.exe` (Windows) atau `idealC-2019.3.2.dmg` (macOS). Lalu ikuti instruksi pada jendela yang tampil untuk menyelesaikan proses instalasi.

#### Memasang Dart SDK




Setelah proses instalasi IntelliJ IDEA selesai, langkah selanjutnya kita perlu menambahkan *plugin*Dart agar bisa kita gunakan membuat *project* Dart. Caranya klik **Configure** -> **Plugins**. Pada tab **Marketplace** cari *plugin* **Dart** lalu klik **Install**.

Plugins

MarketplaceInstalled⚙️

Q Dart


Search Results (30)Sort By: Relevance



Dart

↓ 3.2M ☆ 4.4 JetBrains


Install



JSON To Dart Class (JsonT...

↓ 11.4K ☆ 4.5 typ0520


Install



JsonToDart (JSON To Dart)

↓ 6.5K ☆ 4.7 Ankit Mahadik


Install



Json2Dart

↓ 38K ☆ 4.5 Azoft Ltd.


Install



FlutterJsonBeanFactory

↓ 56.1K ☆ 4.8 ruiyu-QQGroup(963752388)


Install



AngularDart Folding

↓ 12.3K


Install



Dart Data Class

↓ 9.4K Andras Ferenczi

Install



Dart

↓ 3.2M ☆ 4.4 JetBrains

Languages193.6015.53Jan 29, 2020

Install

Plugin homepage

Support for Dart

Features

- Smart coding assistance for Dart that includes code completion, formatting, navigation, intentions, refactorings, and more
- Integration with pub and the Dart Analysis Server
- The IDE detects the problems with your code on-the-fly and suggests ways to automatically fix them
- Run and debug Dart command-line and web applications right in the IDE using the built-in debugger
- Run and debug tests
- Create new Dart projects from the IDE Welcome screen

Find more information on

getting started with Dart in the IDE

in our docs.

Size: 2M

OK

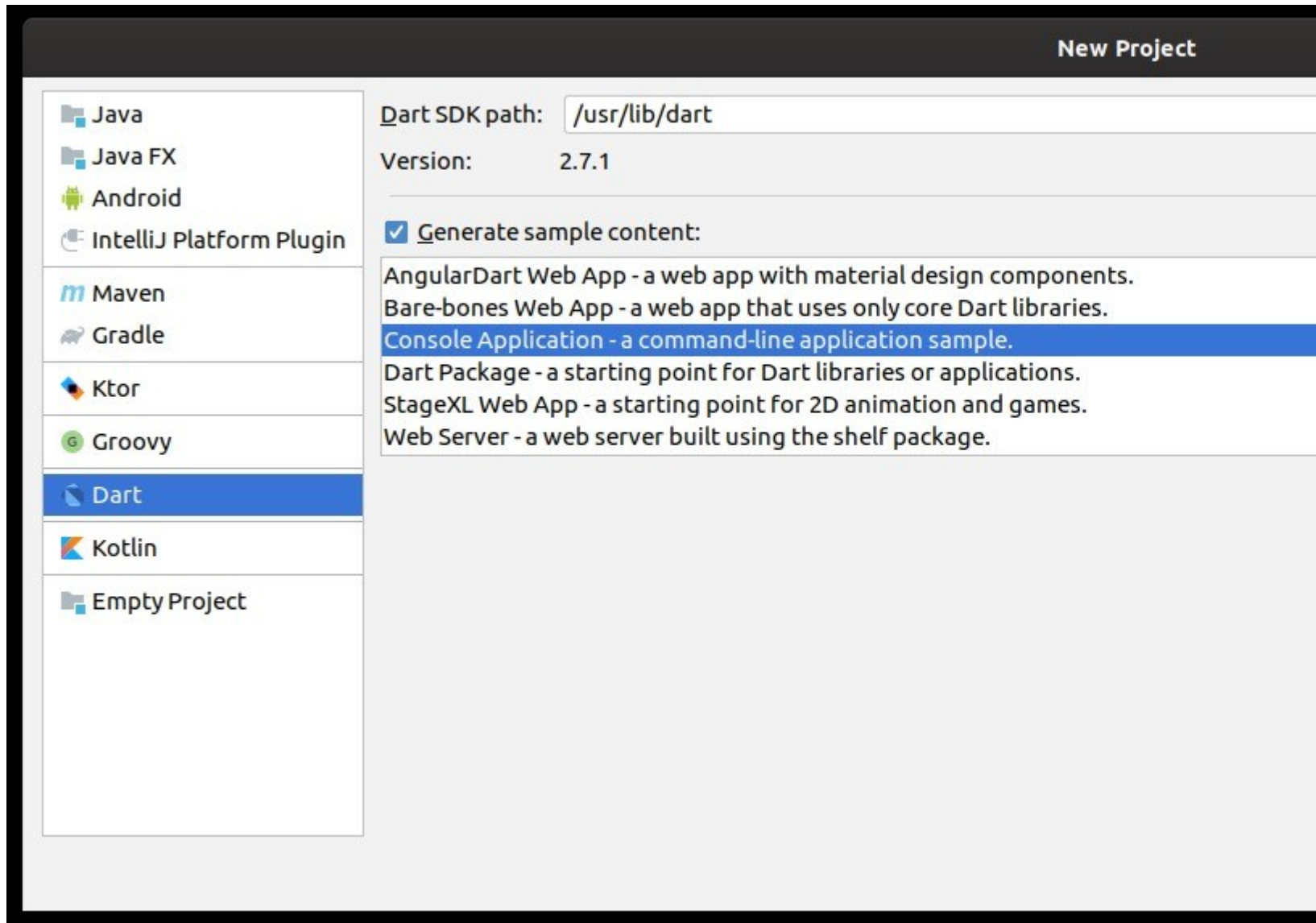
Cancel

Help

Setelah *plugin* terinstal, Anda perlu untuk me-*restart* IDE.

Selamat! Anda telah siap untuk membuat aplikasi dengan Dart. Untuk mulai membuat project klik **Create New Project**. Anda perlu memasukkan *directory* dari tempat Anda menginstall Dart SDK. Jika Anda pengguna Linux dan macOS, SDK Anda biasanya akan terletak di direktori `/usr/lib/dart`.

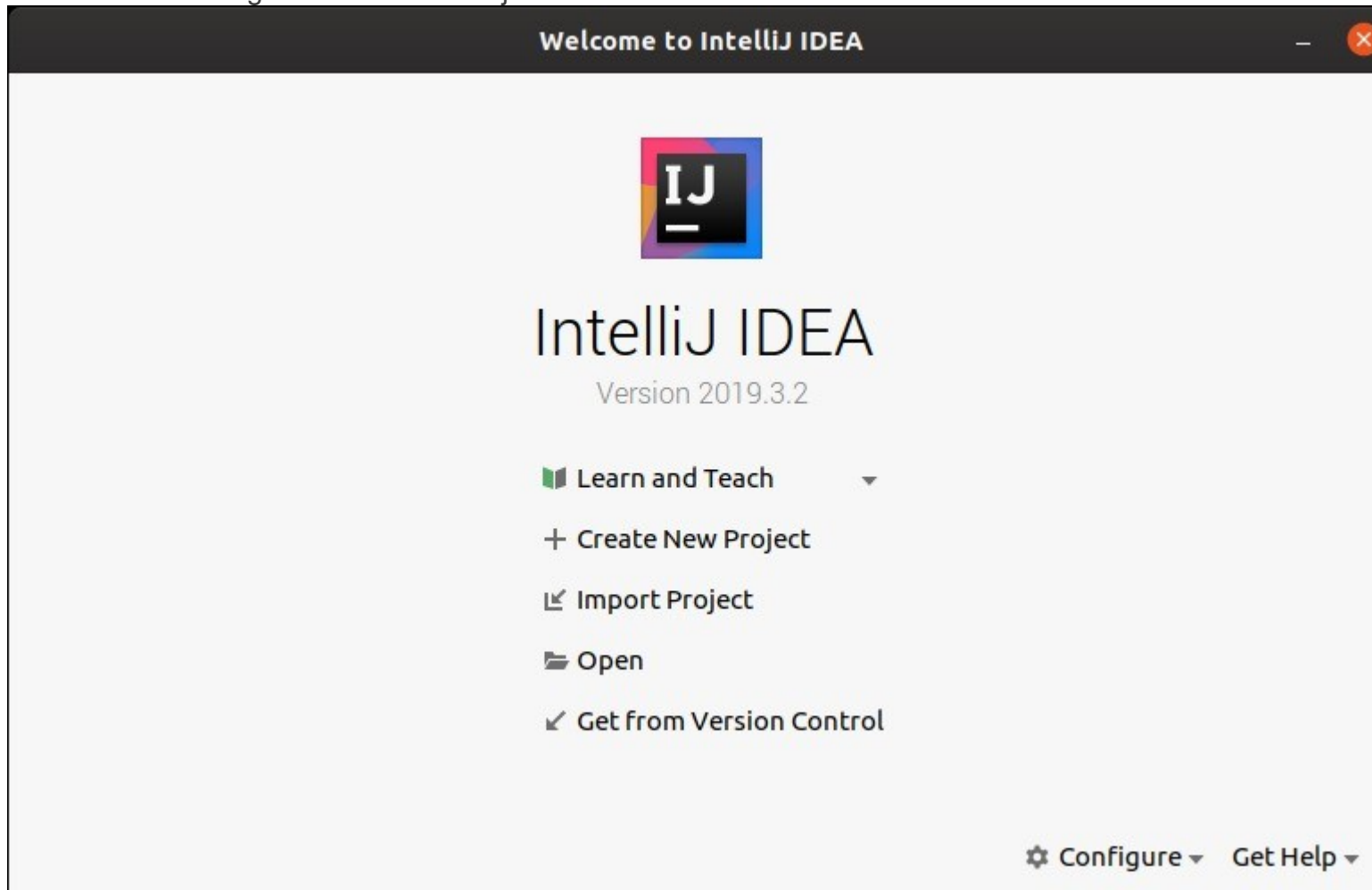




## Membuat Program Dart Pertamamu

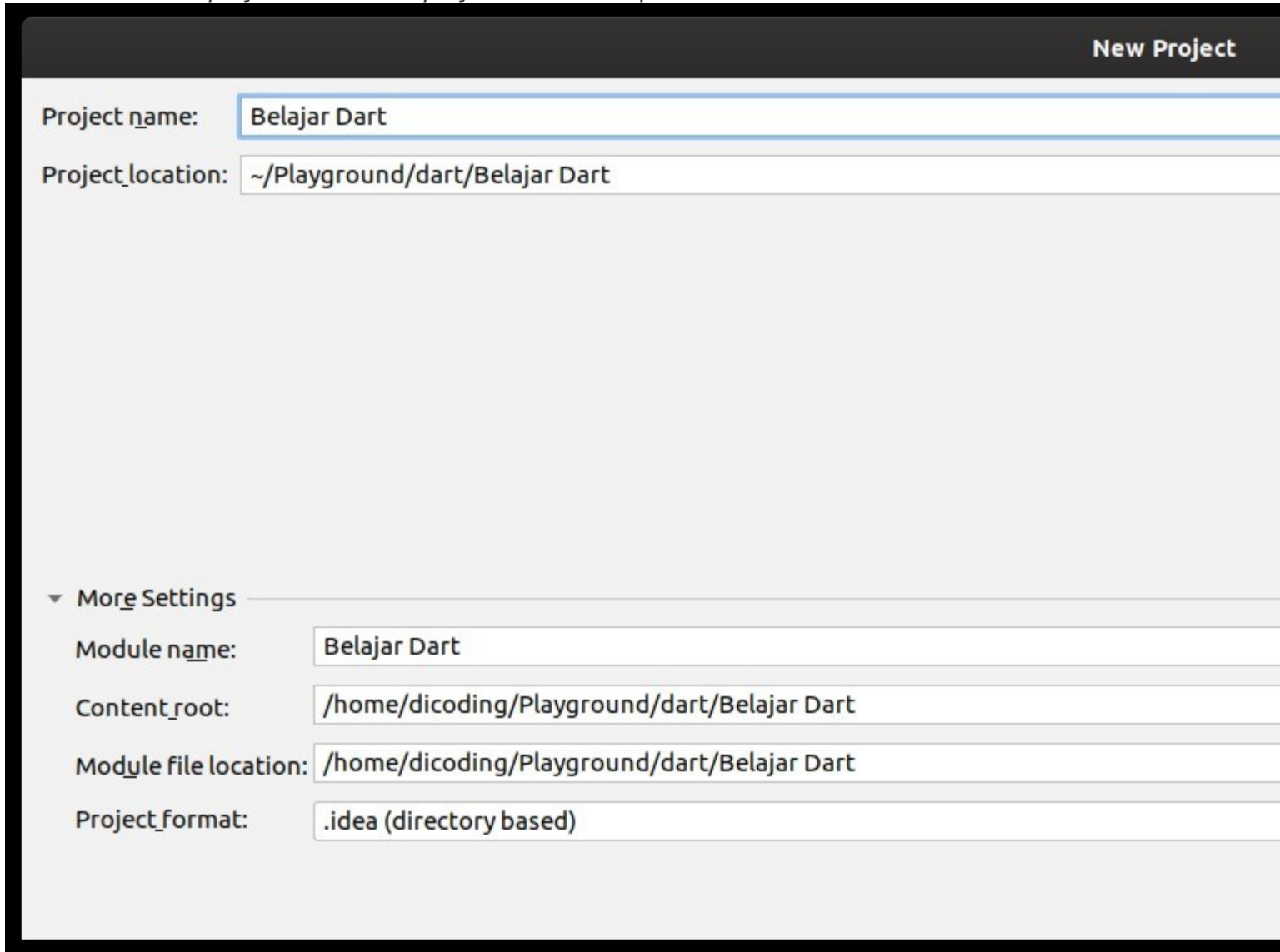
Setelah semua peralatan terpasang, Anda pun telah siap untuk membuat program Dart. Sudah jadi tradisi di setiap belajar bahasa pemrograman bahwa aplikasi yang pertama kali dibuat untuk memulai sebuah bahasa adalah menampilkan ucapan "Hello, World!"

1. Mari kita mulai dengan membuka IntelliJ IDEA.



2. Klik **Create New Project**.
3. Pilih **Dart** dan masukkan *path* Dart SDK.
4. Gunakan *template* **Console Application**.

5. Masukkan nama *project* dan lokasi *project* akan disimpan.



New Project

Project name: Belajar Dart

Project location: ~/Playground/dart/Belajar Dart

▼ More Settings

Module name: Belajar Dart

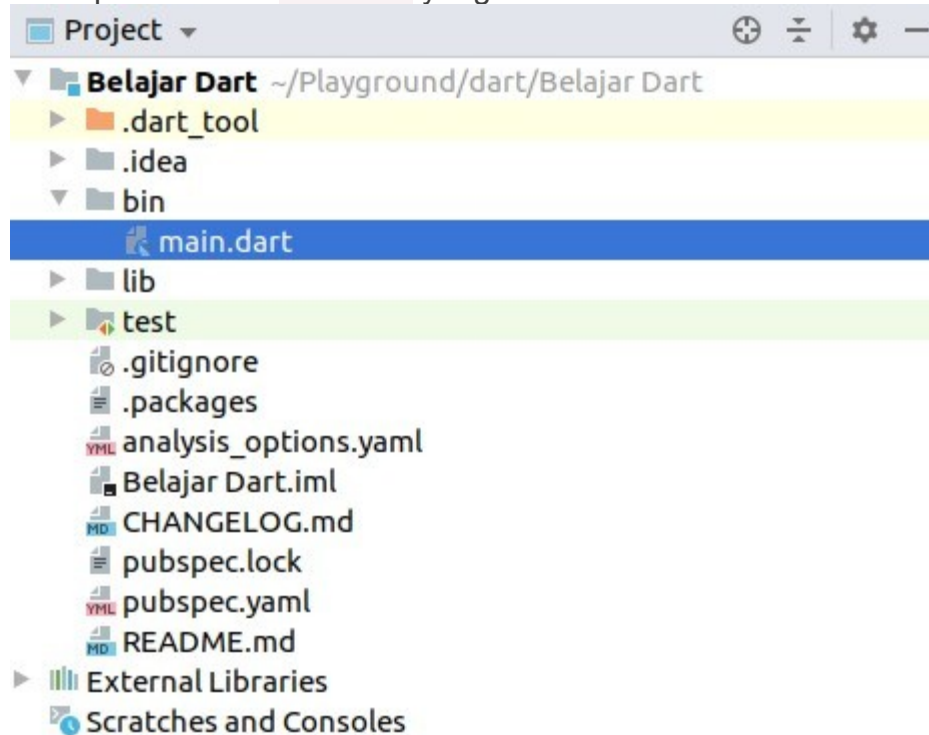
Content root: /home/dicoding/Playground/dart/Belajar Dart

Module file location: /home/dicoding/Playground/dart/Belajar Dart

Project format: .idea (directory based)

6. Jika sudah klik ***Finish***.

7. IntelliJ IDEA akan menyimpan *project* Dart. Setelah siap, Anda dapat menuliskan kode program Anda pada berkas **main.dart** yang berada di folder bin.



8. Pada berkas **main.dart** telah tersedia kode yang bisa langsung kita jalankan.

```
1. import 'package:Belajar_Dart/Belajar_Dart.dart' as Belajar_Dart;
2.
3. void main(List<String> arguments) {
4.   print('Hello world: ${Belajar_Dart.calculate()}!');
5. }
```

Untuk menjalankan aplikasi pada IntelliJ IDEA, klik ikon segitiga hijau pada menu bagian atas atau dengan shortcut **Shift + F10**.

9. Konsol Dart akan menampilkan teks seperti berikut:

```
1. Hello world: 42!
```

Selamat! Anda telah berhasil membuat dan menjalankan program Hello World dengan Dart. Namun tahukah Anda bagaimana aplikasi berjalan dan bisa menampilkan output seperti pada konsol?

Pertanyaan

Apakah yang dimaksud dengan IDE?

Jawaban Anda



Teks editor untuk menuliskan kode program.

- ☐ Semua salah.
- ☐ Lingkungan pengembangan aplikasi yang mempunyai beberapa fasilitas yang dibutuhkan dalam pembangunan perangkat lunak.
- ☐ Kumpulan library dan command-line tools yang dibutuhkan untuk mengembangkan Dart web, command-line, dan server apps.

### Anda menjawab

Lingkungan pengembangan aplikasi yang mempunyai beberapa fasilitas yang dibutuhkan dalam pembangunan perangkat lunak.

**Jawaban Anda benar. Berikut adalah penjelasannya:**

IDE bisa dibilang merupakan teks editor yang mewah karena memiliki banyak fasilitas untuk mendukung pemrograman.

## Dart Fundamental

Kita telah belajar bagaimana membuat aplikasi sederhana “Hello, World!” menggunakan Dart. Pada modul ini kita akan mempelajari konsep-konsep dasar pada Dart. Beberapa topik yang akan kita pelajari antara lain variabel, tipe data, *operator*, hingga *functions*.

## Comments

Sebelum mulai mengembangkan aplikasi yang lebih kompleks, ada satu hal penting lagi yang perlu kita tahu, yaitu instruksi kepada komputer untuk mengabaikan bagian dari suatu program. Kode yang ditulis dalam suatu program tetapi tidak dijalankan oleh komputer disebut “*comments*.”

Sebuah komentar akan dilewatkan ketika proses kompilasi, sehingga tidak akan memengaruhi alur program yang kita tulis. Komentar bisa digunakan sebagai dokumentasi yang menjelaskan kode yang kita tulis.

Terdapat tiga jenis komentar yang bisa digunakan pada Dart. Pertama adalah ***single-line comment*** atau komentar satu baris yang diawali dengan tanda `//` dan berakhir pada akhir baris tersebut.

```
1. // Single line comment
```

Selanjutnya ada ***multi-line comment*** di mana kita bisa menuliskan beberapa baris komentar. Komentar ini dimulai dari tanda `/*` dan diakhiri dengan `*/`.

```
1. /*
2.   multi
```

```
3. line
4. comment
5. */
```

Kemudian terakhir adalah **documentation comment**. Komentar ini adalah *single-line comment* atau *multi-line comment* yang diawali dengan `///` atau `/**`.

Di dalam *documentation comment*, kompiler Dart akan mengabaikan semua teks kecuali yang tertutup dalam kurung siku (`[]`). Di dalam kurung siku kita dapat memasukkan referensi dari nama kelas, variabel, atau fungsi. Berikut ini adalah contoh penggunaan komentar:

```
1. import 'package:Belajar_Dart/Belajar_Dart.dart' as Belajar_Dart;
2.
3. /// Fungsi [main] akan menampilkan 2 output
4. /// Output pertama menampilkan teks dan output kedua menampilkan hasil perkalian pada library
   [Belajar_Dart]
5. void main(List<String> arguments) {
6.   // Mencetak Hello Dart! Dart is great. pada konsol
7.   print('Hello Dart! Dart is great. ');
8.   // Testing documentation comment with [].
9.   print('6 * 7 = ${Belajar_Dart.calculate()}');
10.
11.
12.
13. /*
14.   output:
15.     Hello Dart! Dart is great.
16.     6 * 7 = 42
17. */
```

Jika Anda menggunakan IDE lalu menekan **Ctrl** dan mengarahkan ke teks di dalam kurung siku, maka Anda akan dapat klik dan mengarah ke kode referensinya.

Menulis komentar pada kode adalah praktik yang baik. Namun perlu diperhatikan, “Kode yang baik memiliki banyak komentar, sementara kode yang buruk memerlukan banyak komentar.”

## Variables

Ketika menulis sebuah program, kita memberi tahu komputer cara memproses informasi seperti mencetak teks ke layar atau melakukan operasi perhitungan. Untuk lebih memudahkan penggunaan dan pemanggilan data ini maka kita bisa memanfaatkan variabel. Variabel bisa dibayangkan sebagai sebuah kotak atau wadah yang menyimpan nilai. Di dalam komputer variabel ini disimpan di dalam memori komputer. Setiap variabel memiliki nama yang dapat kita panggil dan gunakan.

Pada Dart kita mendefinisikan sebuah variabel dengan *keyword* **var**.

```
1. var greetings = 'Hello Dart!';
```

Perhatikan tanda sama dengan (=) di atas. Simbol tersebut dikenal dengan **assignment operator**. Kode di atas berarti kita memasukkan nilai 'Hello Dart!' ke dalam sebuah kotak atau variabel yang bernama *greetings*. Proses *assignment* nilai ke variabel ini disebut inisialisasi.

Selanjutnya mari coba tampilkan nilai variabel ini ke konsol. Ubah kode fungsi main Anda menjadi seperti berikut:

```
1. void main() {  
2.   var greetings = 'Hello Dart!';  
3.   print(greetings);  
4. }
```

Pada contoh kode di atas kita memasukkan data berupa teks ke dalam variabel. Lalu, bagaimana dengan data numerik atau berupa angka? Tentu saja bisa. Anda cukup menginisialisasi variabel dengan nilai angka.

```
1. var myAge = 20;  
2. print(myAge);
```

Lalu bagaimana jika kita ingin membuat variabel namun tidak ingin langsung menginisiasiasinya? Misalnya Anda ingin menunggu beberapa operasi selesai lalu menginisiasiasinya ke dalam variabel.

Proses ini disebut dengan deklarasi variabel. Deklarasi variabel akan menyimpan nama dan ruang di dalam memori tanpa memberikan nilai. Anda dapat menginisialisasi nilai setelah deklarasi sesuai dengan kebutuhan Anda.

```
1. var myAge;  
2. myAge = 20;  
3. print(myAge);
```

Sekarang cobalah *comment* kode inisialisasi variabel di atas lalu jalankan programnya, apa yang terjadi?

Apakah konsol Anda menampilkan *null*? Setiap deklarasi variabel akan memberikan nilai *default* berupa *null*. Ini berarti variabel tersebut belum terinisialisasi atau bisa dibilang variabel Anda tidak memiliki nilai atau *null*.

## Data Types

Pada materi sebelumnya kita telah mempelajari tentang variabel yang dapat menyimpan nilai. Jadi bagaimana komputer membedakan antara variabel yang bernilai angka atau teks? Dan kenapa penting untuk bisa membedakannya?

Dart memiliki banyak tipe data yang mewakili jenis data yang dapat kita gunakan dan bagaimana data tersebut dioperasikan. Dengan tipe data, komputer dapat menghindari operasi yang tidak mungkin serta bisa menghasilkan *bug*, misalnya seperti perkalian karakter alfabet atau mengubah angka menjadi kapital.



Dart adalah bahasa yang mendukung *type inference*. Ketika Anda mendeklarasikan variabel dengan `var`, Dart akan secara otomatis menentukan tipe datanya. Misalnya :

```
1. var greetings = 'Hello Dart!'; // String
2. var myAge = 20;                // integers
```

Komputer akan tahu bahwa variabel `greetings` memiliki nilai berupa `String` atau teks dan variabel `myAge` bernilai angka atau `integers` meskipun kita tidak mendefinisikannya secara eksplisit.

Anda tetap bisa mendeklarasikan tipe data variabel secara eksplisit untuk menghindari kebingungan dan memudahkan proses *debugging*.

```
1. String greetings = 'Hello Dart!';
2. int myAge = 20;
```

Beberapa tipe data yang didukung oleh Dart antara lain:

Tipe	Deskripsi	Contoh
int	Integer (bilangan bulat)	5, -7, 0
double	Bilangan desimal	3.14, 18.0, -12.12
num	Bilangan bulat dan bilangan desimal	5, 3.14, -99.00
bool	Boolean	true, false
String	Teks yang terdiri dari 0 atau beberapa karakter	'Dicoding', 'Y', "
List	Daftar nilai	[1, 2, 3], ['a', 'b', 'c']
Map	Pasangan key-value	{"x": 4, "y": 10}
dynamic	Tipe apa pun	

Dart mendukung *type inference*, menariknya ketika kita mendeklarasikan variabel tanpa melakukan inisialisasi, variabel akan memiliki tipe **dynamic**. Karena sebuah variabel bernilai *dynamic* bisa berisi tipe apa pun, maka tidak ada masalah jika kita mengubah nilai di dalamnya.

```
1. var x; // dynamic
2. x = 7;
3. x = 'Dart is great';
4. print(x);
```

Kode di atas tetap bisa berjalan dan menampilkan pesan *'Dart is great'* tanpa ada masalah. Berbeda jika kita menginisialisasi nilai variabel `x` secara langsung. Akibatnya, *editor* akan menampilkan eror karena terjadi perubahan tipe data.

```
1. var x = 7; // int
2. x = 'Dart is great'; // Kesalahan assignment
3. print(x);
```

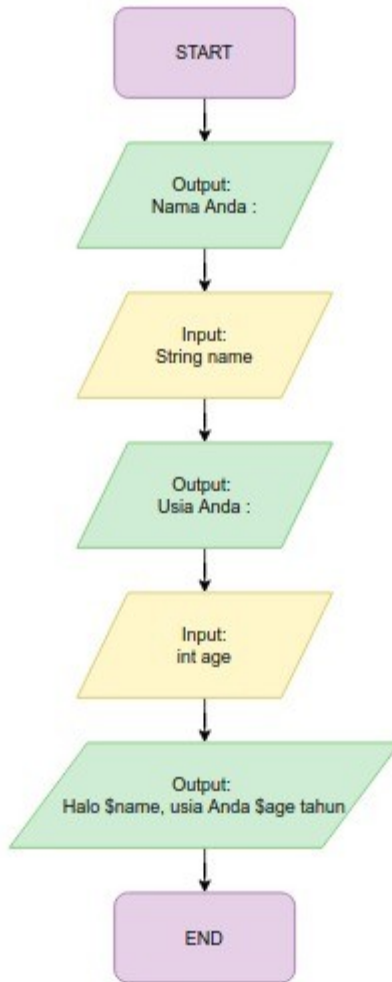
Menerima input pengguna

Selain menampilkan pesan ke konsol, kita juga dapat menerima input pengguna untuk selanjutnya diproses dan ditampilkan. Ini memungkinkan kita dapat membuat aplikasi yang interaktif dengan pengguna.

Untuk bisa menerima input, kita akan menggunakan statement `stdin.readLineSync()`. Fungsi ini merupakan bagian dari *library* `dart:io`, sehingga kita perlu mengimpor *library* tersebut.

```
1. import 'dart:io';
```

Kali ini kita akan membuat aplikasi sederhana yang menerima input nama dan usia dari pengguna lalu menampilkan pesan ke konsol. Sebelum melihat kode solusi di bawah bisakah Anda menerjemahkan *flowchart* berikut menjadi kode aplikasi?



#### Kode solusi

Pada persoalan di atas kita perlu menampilkan output, menerima beberapa input, dan menampilkan output lagi sesuai input yang diberikan. Kode yang perlu Anda tulis kurang lebih adalah seperti berikut:

```
1. import 'dart:io';
2.
3. void main() {
4.   stdout.write('Nama Anda : ');
5.   String name = stdin.readLineSync();
6.   stdout.write('Usia Anda : ');
7.   int age = int.parse(stdin.readLineSync());
8.   print('Halo $name, usia Anda $age tahun');
9. }
```

Jika kode Anda berbeda karena menggunakan `print()` maka tidak masalah. `Statement print()` dan `stdout.write()` memiliki fungsi yang sama yaitu untuk menampilkan suatu

objek ke konsol. Yang membedakan adalah `print()` akan mencetak baris baru setelah menampilkan sesuatu sehingga selanjutnya Anda perlu memasukkan input pada baris baru. Sementara `stdout.write()` hanya menampilkan objeknya saja dan ketika ada input atau output baru lagi masih akan ditampilkan di baris yang sama.

Kode baru lain adalah `int.parse()`. Kita menggunakan kode ini untuk mengkonversi tipe data `String` menjadi `int`. Input yang diambil dari `stdin.readLineSync()` akan memiliki tipe data berupa `String`. Sehingga ketika ingin menyimpan dan mengoperasikan input dalam tipe data lain kita perlu melakukan konversi terlebih dahulu.

## Numbers

Tipe data angka pada Dart dapat disimpan ke dalam dua jenis: *int* dan *double*.

*Integers* adalah nilai bilangan bulat yang tidak lebih besar dari 64 *bit* tergantung *platform* yang digunakan. Untuk Dart VM variabel *integer* dapat menyimpan nilai mulai dari  $-2^{63}$  hingga  $2^{63} - 1$ , sementara jika dikompilasi ke JavaScript *integer* memiliki nilai dari  $-2^{53}$  sampai  $2^{53} - 1$ .

*Integer* atau bilangan bulat adalah bilangan yang tidak memiliki titik desimal. Contohnya seperti berikut:

```
1. var number = 2020;
2. var hex = 0xDEADBEEF;
```

Jika sebuah bilangan adalah bilangan desimal, maka bilangan tersebut termasuk tipe data *double*. *Double* juga memiliki ukuran sebesar 64 bit. Berikut ini adalah contoh variabel *double*:

```
1. var decimal = 1.2;
2. var pi = 3.14;
```

Sejak versi Dart 2.1, kita bisa menuliskan tipe data *double* tanpa menuliskan angka di belakang koma secara literal. Sebelum versi tersebut, kita harus menuliskan bilangan desimal secara literal.

```
1. double withoutDecimal = 7; // Sama dengan double withoutDecimal = 7.0
```

*Int* dan *double* adalah subtype dari tipe data *num*. Ketiga tipe data ini dapat kita gunakan untuk melakukan perhitungan dasar seperti penjumlahan, perkalian, hingga menggunakan fungsi seperti `abs()`, `ceil()`, `floor()`, dan banyak fungsi lainnya. Jika Anda membutuhkan operasi perhitungan namun tidak tersedia pada tiga tipe data ini maka Anda bisa memanfaatkan *library* `dart:math`.

Pada materi sebelumnya kita memiliki kebutuhan untuk mengonversi tipe *String* menjadi *int*. Konversi tipe data ini adalah hal yang penting dan kita akan banyak membutuhkannya. Selain konversi *String* menjadi *int*, kita juga bisa melakukan konversi *double* menjadi *String* atau sebaliknya.

```
1. // String -> int
2. var eleven = int.parse('11');
3.
4. // String -> double
5. var elevenPointTwo = double.parse('11.2');
6.
7. // int -> String
8. var elevenAsString = 11.toString();
9.
10. // double -> String
11. var piAsString = 3.14159.toStringAsFixed(2); // String piAsString = '3.14'
```

## Strings

Kita telah banyak menggunakan String pada materi-materi sebelumnya dan seperti kita tahu, *String* digunakan untuk menyimpan data dalam bentuk teks. *String* pada Dart dapat didefinisikan dengan menggunakan tanda petik satu maupun petik dua.

```
1. String singleQuote = 'Ini adalah String';
2. String doubleQuote = "Ini juga String";
```

Anda dapat menggunakan tanda petik secara bergantian, khususnya jika Anda memiliki teks yang mengandung tanda petik.

```
1. print('What do you think of Dart?' he asked');
```

Lalu bagaimana jika teks kita memiliki kedua tanda petik ini?

```
1. print('I think it's great! I answered confidently');
```

Tentunya kode di atas akan menghasilkan error. Solusinya, gunakanlah *backslash* (\) untuk mengurangi ambiguitas dalam tanda petik. Mekanisme ini juga dikenal dengan nama **escape string**. Sehingga kode di atas akan menjadi seperti berikut:

```
1. print('I think it\'s great! I answered confidently');
```

*Backslash* sebelum tanda petik ini akan mengatakan kepada Dart bahwa itu hanyalah tanda petik dan tidak boleh ditafsirkan sebagai pembatas *string*. Selain tanda petik, *backslash* juga berguna untuk mengabaikan simbol lain yang menimbulkan ambiguitas di dalam *string*, contohnya seperti *backslash* itu sendiri.

```
1. print("Windows path: C:\\Program Files\\Dart");
```

Fitur lain dari *String* adalah ***String interpolation*** di mana kita bisa memasukkan nilai dari variabel atau *expression* ke dalam *string*. *Interpolation* ini bisa kita terapkan dengan simbol `$`.

```
1. var name = 'Messi';
2. print('Hello $name, nice to meet you.');
```

Jika Anda perlu menampilkan harga dalam dollar, maka apa yang akan Anda lakukan? *Yup, backslash* akan memberitahu Dart mana tanda `$` sebagai *interpolation* atau sebagai mata uang. Selain itu Anda juga menggunakan cara seperti berikut:

```
1. print(r'Dia baru saja membeli komputer seharga $1,000.00');
```

Huruf `'r'` sebelum *String* akan memberitahu Dart untuk menganggap *String* sebagai *raw*, yang berarti akan mengabaikan *interpolation*.

Selain itu, kita juga bisa menambahkan sebuah *Unicode* ke dalam *String*. Pada Dart *Unicode* ini dikenal dengan ***runes***. *Unicode* mendefinisikan nilai numerik unik untuk setiap huruf, angka, dan simbol yang digunakan dalam semua sistem penulisan dunia. Cara umum untuk mengekspresikan unicode adalah `\uXXXX`, di mana ***XXXX*** adalah nilai heksadesimal 4 digit. Misalnya karakter hati (♥) adalah `\u2665`.

```
1. print('Hi \u2665');
2.
3. /*
4.   output :
5.   Hi ♥
6. */
```

## Booleans

Setelah angka dan teks, ada satu tipe data utama lagi yang penting untuk dipelajari, yaitu ***boolean***. Nama *boolean* ini diambil dari nama seorang matematikawan asal Inggris yang bernama *George Boole*. Beliau dikenal karena penciptaan aljabar boolean, yakni cabang aljabar di mana nilai variabel selalu benar atau salah.

*Boolean* pada Dart dideklarasikan dengan kata kunci `bool`. Sesuai dengan penjelasan di atas, variabel boolean hanya bisa menyimpan dua nilai, yaitu `true` dan `false`.

```
1. bool alwaysTrue = true;
2. var alwaysFalse = false;
3. var notTrue = !true;
4. bool notFalse = !false;
```

Tanda **!** di atas disebut dengan operator “*not*” atau “*bang*”. Operator ini berfungsi untuk menegasikan nilai *boolean*, sederhananya membalik nilai *boolean*. Misalnya **!true** sama saja bernilai **false**.

Kita akan banyak menggunakan boolean dalam pengembangan aplikasi untuk operasi perbandingan dan juga pengondisian aplikasi.

```
1. if(true) {
2.   print("It's true");
3. } else {
4.   print("It's False");
5. }
```

Kita akan segera mempelajari tentang ini pada materi-materi selanjutnya.

# Operators

Istilah operator dipinjam dari matematika dengan pengertian yang sedikit berbeda. Pada Dart operator menginstruksikan komputer untuk melakukan operasi.

Sebenarnya kita telah menerapkan beberapa contoh operator pada materi sebelumnya. Sebagai contoh operator *assignment* (=) yang kita gunakan untuk inisialisasi nilai variabel.

```
1. var firstNumber = 4;
```

Pada kode di atas kita menginstruksikan komputer untuk memasukkan nilai 4 ke dalam variabel **firstNumber**.

## Operator aritmatika

Contoh operator lain yang telah Anda lihat adalah operator aritmatika yang digunakan untuk operasi seperti penjumlahan atau perkalian. Lihatlah contoh kode berikut:

```
1. var firstNumber = 4;
2. var secondNumber = 13;
3. var sum = 4 + 13;
4. print(sum);
5.
6. /*
7. Output :
8.   17
9.   */
```

Dart mendukung operator aritmatika umum sebagai berikut:

Operator	Deskripsi
+	Penjumlahan



Operator	Deskripsi
-	Pengurangan
*	Perkalian
/	Pembagian
~/	Pembagian, mengembalikan nilai int
%	Modulo atau sisa hasil bagi

```

1. print(5 + 2); // int add = 7
2. print(5 - 2); // int subtract = 3
3. print(5 * 2); // int multiply = 10
4. print(5 / 2); // double divide = 2.5
5. print(5 ~/ 2); // int intDivide = 2
6. print(5 % 2); // int modulo = 1

```

Operator aritmatika pada pemrograman memiliki aturan yang sama dengan matematika, di mana perkalian dan pembagian akan didahulukan sebelum penjumlahan atau pengurangan.

```
1. print(2 + 4 * 2); // output: 10
```

Jika Anda ingin melakukan operasi penjumlahan terlebih dahulu, gunakan tanda kurung atau *parentheses*.

```
1. print((1 + 3) * (4 - 2)); // output: 8
```

Selain itu Dart mendukung *increment* dan *decrement*. Contohnya adalah seperti berikut:

```

1. var a = 0, b = 5;
2. a++;
3. b--;
4. print(a); // output = 1
5. print(b); // output = 4

```

Expression `a++` di atas dapat diartikan dengan `a = a + 1`. Komputer akan mengambil nilai dari `a` kemudian menambahkan 1 lalu memasukkannya kembali ke variabel `a`. Bentuk *increment* lainnya adalah seperti berikut:

```

1. var c = 0;
2. c += 5; // c = c + 5 atau c = 0 + 5
3. print(c); // output 5

```

Operator ini juga bisa digunakan pada operator aritmatika lain seperti perkalian dan pembagian.

```

1. var d = 2;
2. d *= 3; // d = d * 3 atau d = 2 * 3
3. print(d); // output = 6

```

### Operator perbandingan

Dart juga mendukung operasi perbandingan untuk membandingkan nilai-nilai yang dijadikan sebagai *operands*. Berikut ini adalah contoh operator perbandingan pada Dart:

Operator	Deskripsi
==	Sama dengan
!=	Tidak sama dengan
>	Lebih dari
<	Kurang dari
>=	Lebih dari sama dengan
<=	Kurang dari sama dengan

Operator perbandingan ini akan mengembalikan nilai dalam bentuk *boolean*.

```
1. if (2 <= 3) {
2.   print('Ya, 2 kurang dari sama dengan 3');
3. } else {
4.   print('Anda salah');
5. }
6.
7. /*
8. Output:
9.   Ya, 2 kurang dari sama dengan 3
10. */
```

### Operator logika

Kita juga dapat menggabungkan ekspresi *boolean* atau membalikinya dengan operator logika. Operator ini meliputi:

Operator	Deskripsi
	OR
&&	AND
!	NOT

Kita telah membahas operator *NOT* atau *bang* pada materi *boolean*. Sementara itu operator *OR* atau *AND* digunakan untuk menguji logika dari beberapa nilai *boolean*. Operator *AND* akan menghasilkan nilai *true* jika semua *operand*-nya bernilai *true*, sedangkan *OR* jika salah satu saja dari *operand* bernilai *true* maka operator akan mengembalikan nilai *true*. Contohnya seperti kode berikut:

```
1. if (2 < 3 && 2 + 4 == 5) {
2.   print('Untuk mencetak ini semua kondisi harus benar');
3. } else {
```

```

4.   print('2 kurang dari 3, tapi 2 + 4 tidak sama dengan 5, maka ini akan tampil');
5. }
6.
7. if (false || true || false) {
8.   print('Ada satu nilai true');
9. } else {
10.  print('Jika semuanya false, maka ini akan tampil');
11. }
12.
13. /*
14. Output:
15.  2 kurang dari 3, tapi 2 + 4 tidak sama dengan 5, maka ini akan tampil
16.  Ada satu nilai true
17. */

```

## Exceptions

Sebuah aplikasi yang sudah berjalan mungkin mengalami eror dan *crash*. Kondisi eror pada saat aplikasi berjalan (*runtime*) ini dikenal dengan **exception**. Ketika *exception* terjadi aplikasi akan dihentikan dan program setelahnya juga tidak akan dieksekusi.

Salah satu contoh *exception* yang mungkin terjadi adalah pada aplikasi kalkulator. Di mana menurut prinsip matematika kita tidak bisa membagi bilangan lain dengan bilangan nol (0).

```

1. var a = 7;
2. var b = 0;
3. print(a ~/ b);

```

Coba jalankan kode di atas. Anda akan mendapatkan eror seperti berikut:

```

Unhandled exception:
IntegerDivisionByZeroException
#0 int~/ (dart:core-patch/integers.dart:24:7)
#1 main (file:///home/dicoding/Playground/dart/Belajar%20Dart/bin/main.dart:24:11)
#2 _startIsolate.<anonymous closure> (dart:isolate-patch/isolate_patch.dart:305:32)
#3 _RawReceivePortImpl._handleMessage (dart:isolate-patch/isolate_patch.dart:174:12)

```

Sebagai *developer*, tentunya menjadi tugas kita untuk memastikan aplikasi tetap berjalan bagaimana pun kondisinya, termasuk mengatasi ketika terjadi *exception*. Lantas bagaimana?

Untuk menangani *exception*, mari gunakan **try** dan **catch**. Pindahkan kode yang berpotensi menimbulkan *exception* ke dalam blok **try**.

```

1. try {
2.   var a = 7;
3.   var b = 0;
4.   print(a ~/ b);
5. }

```

Kode di atas masih belum lengkap karena di dalam sebuah blok `try` ada kode yang diasumsikan berpeluang menjadi *exception* sehingga perlu ditangani. Pada eror sebelumnya telah diketahui bahwa *exception* yang terjadi adalah `IntegerDivisionByZeroException`. Sehingga kita bisa memanfaatkan keyword `on` untuk mengatasi ketika *exception* tersebut terjadi.

```
1. try {
2.   var a = 7;
3.   var b = 0;
4.   print(a ~/ b);
5. } on IntegerDivisionByZeroException {
6.   print('Can not divide by zero.');
```

Nah, kita telah berhasil membuat program kita aman dari *crash* ketika berusaha membagi bilangan dengan nol. Namun bagaimana dengan *exception* lainnya yang belum kita ketahui? Pada Dart sendiri telah tersedia beberapa *exception* seperti `IOException`, `FormatException`, dsb. Untuk menangani *exception* yang tidak diketahui secara spesifik, kita bisa menggunakan keyword `catch` setelah blok `try`.

```
1. try {
2.   var a = 7;
3.   var b = 0;
4.   print(a ~/ b);
5. } catch(e) {
6.   print('Exception happened: $e');
```

Kode `catch` membutuhkan satu parameter yang merupakan objek *exception*. Kita dapat mencetak *exception* tersebut ke layar untuk menampilkan *exception* apa yang terjadi. Output kode di atas adalah:

```
1. Exception happened: IntegerDivisionByZeroException
```

Selain itu, kita juga dapat menambahkan satu parameter lagi di dalam `catch` yang merupakan objek *stack trace*. Dari *stack trace* ini kita bisa melihat detail *exception* dan di baris mana *exception* tersebut terjadi.

```
1. try {
2.   var a = 7;
3.   var b = 0;
4.   print(a ~/ b);
5. } catch(e, s) {
6.   print('Exception happened: $e');
```

```
7.   print('Stack trace: $s');
```

```
8. }
```

```
9.
```

```
10. /*
```

```
11. Output :
```

```
12. Exception happened: IntegerDivisionByZeroException
```

```
13. Stack trace: #0      int.~/ (dart:core-patch/integers.dart:24:7)
```

```
14. #1      main (file:///home/dicoding/Playground/dart/Belajar%20Dart/bin/main.dart:25:13)
```

```
15. #2      _startIsolate.<anonymous closure> (dart:isolate-patch/isolate_patch.dart:305:32)
16. #3      _RawReceivePortImpl._handleMessage (dart:isolate-patch/isolate_patch.dart:174:12)
17. */
```

Blok **catch** dapat digabungkan dengan **on**. **Catch** akan menangkap apabila tidak ada *exception* yang memenuhi blok **on** yang terpasang.

Finally

Selain *try*, *on*, dan *catch*, ada satu blok lagi yang ada dalam mekanisme *exception handling*, yaitu **finally**. Blok **finally** akan tetap dijalankan tanpa peduli apa pun hasil yang terjadi pada blok *try-catch*.

```
1. try {
2.   var a = 7;
3.   var b = 0;
4.   print(a ~/ b);
5. } catch(e, s) {
6.   print('Exception happened: $e');
7.   print('Stack trace: $s');
8. } finally {
9.   print('This line still executed');
10. }
```

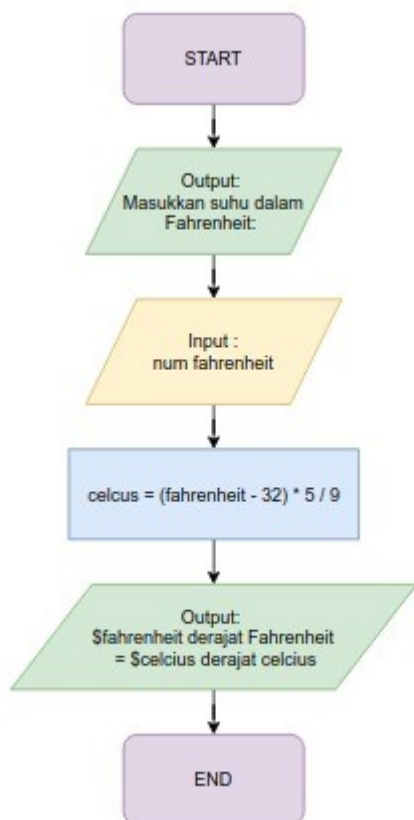
## Latihan - Aplikasi Konversi Suhu

Katakanlah Anda diundang untuk menghadiri acara konferensi developer di Amerika, namun Anda kebingungan karena ternyata Amerika menggunakan standar suhu Fahrenheit. Tentu jika Anda memiliki aplikasi yang dapat mengonversi suhu dari Fahrenheit menjadi Celcius, akan sangat berguna, bukan?

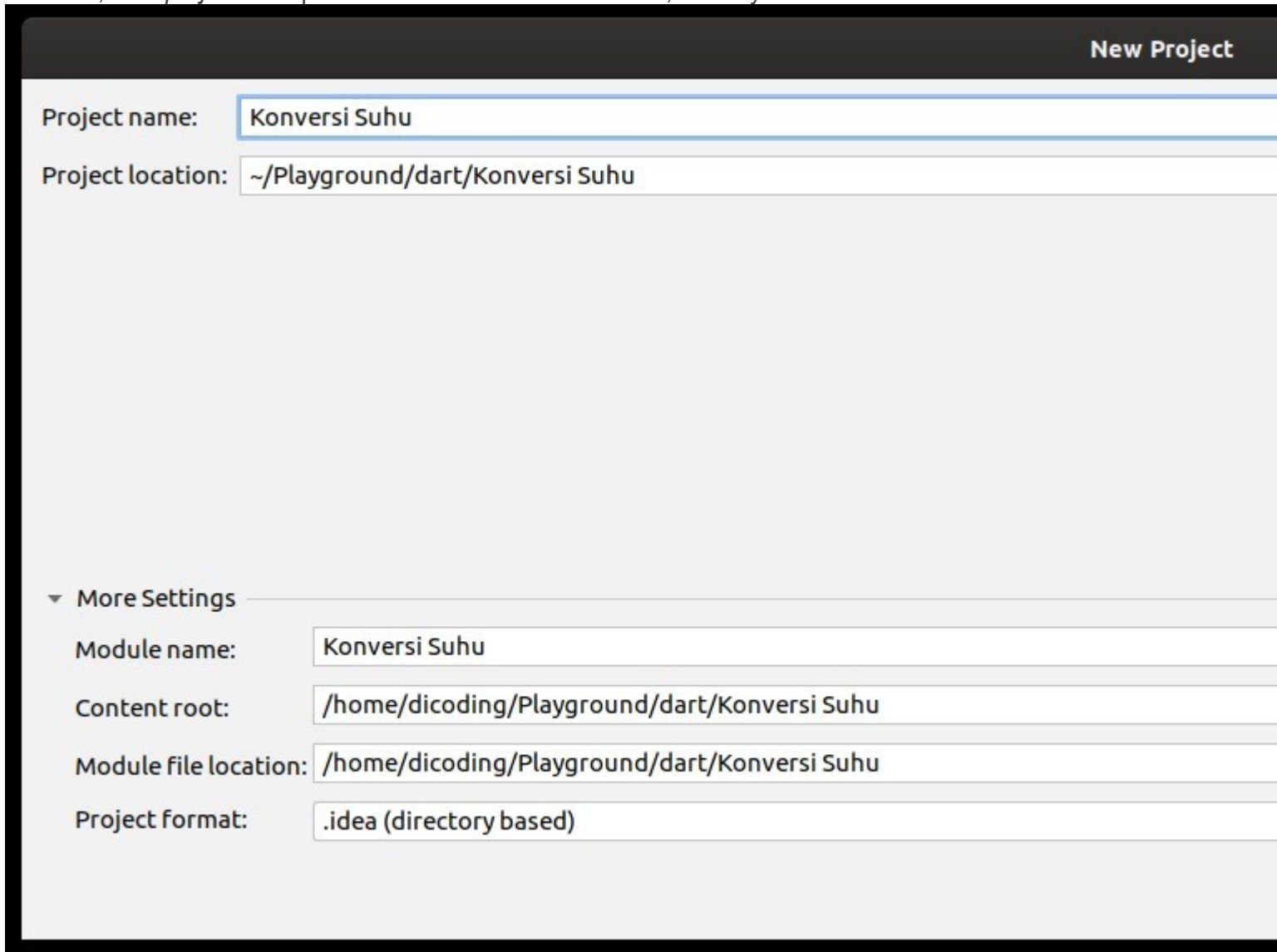
Sebelum masuk ke pembuatan aplikasi, tentu kita harus tahu rumus konversi suhu terlebih dulu. Alur aplikasi konversi suhu kita akan seperti berikut:

1. Menerima input suhu fahrenheit dari pengguna.
2. Melakukan konversi sesuai rumus.
3. Menampilkan hasil konversi.

Seperti inilah jika proses tersebut dituangkan ke dalam *flowchart*:



1. Pertama, buat *project* baru pada IDE Anda lalu berikan nama, misalnya Konversi Suhu.



2. Selanjutnya kita perlu menerima input dari pengguna dan jangan lupa untuk menampilkan informasi apa yang perlu diinputkan. Hapus semua kode pada berkas `main.dart` lalu tambahkan kode berikut:

```
1. import 'dart:io';
2.
3. void main() {
4.   stdout.write('Masukkan suhu dalam Fahrenheit: ');
5.   var fahrenheit = num.parse(stdin.readLineSync());
6. }
```
3. Lakukan konversi dengan memasukkan rumus konversi suhu.

```
1. var celsius = (fahrenheit - 32) * 5 / 9;
```
4. Terakhir tampilkan hasil konversi Anda ke konsol.

```
1. print('$fahrenheit derajat Fahrenheit = $celsius derajat celsius');
```
5. Jalankan dan uji apakah aplikasi Anda telah berjalan dengan sesuai.



### Challenge

Sebagai tantangan, buatlah aplikasi Anda dapat mendukung konversi suhu lain seperti Reamur, Kelvin, dll.

## Functions

*Functions* pada Dart digunakan untuk menghasilkan output berdasarkan input tertentu yang diberikan, selain itu juga sebagai blok kode atau prosedur yang dapat digunakan kembali. Sadar atau tidak, sebenarnya kita telah mengimplementasikan beberapa *functions* pada kode kita. Semua program Dart dimulai dari fungsi `main()`. `main()` adalah contoh fungsi utama yang selalu kita gunakan. Selain itu, `print()` juga termasuk fungsi.

```
1. print('Hello Dart!');
```

Fungsi `print()` akan mengambil nilai *String* atau objek lainnya dan menampilkannya ke konsol. Untuk mencetak sesuatu ke konsol sebenarnya dibutuhkan beberapa instruksi yang lebih *low-level*, namun kita menjadi sangat terbantu dengan adanya fungsi `print()` ini dan dapat menggunakannya secara berulang.

Untuk mendeklarasikan fungsi, caranya sama dengan penulisan fungsi `main()` yaitu dengan menentukan tipe nilai balik atau *return value* lalu nama fungsi dan *parameter* inputnya.

```
1. returnType functionName(type param1, type param2, ...) {  
2.   return result;  
3. }
```

Setiap fungsi Dart selalu mengembalikan nilai. Namun ada satu tipe data khusus yang bisa kita lihat pada fungsi `main` yaitu *return type* **`void`**. Keyword **`void`** berarti fungsi tersebut tidak menghasilkan output atau nilai kembali. Biasanya fungsi seperti ini digunakan untuk kumpulan instruksi atau prosedur yang berulang dan sering digunakan.

Setelah fungsi dibuat, selanjutnya kita bisa memanggilnya pada fungsi `main()` atau pada bagian program lain yang Anda inginkan.

```
1. void main() {  
2.   greet(); // output : Hello!  
3. }  
4.  
5. void greet() {  
6.   print('Hello!');  
7. }
```

Pada contoh sederhana di atas fungsi `greet()` memang belum menghemat banyak kode yang Anda tulis. Namun, apabila Anda memiliki 30 instruksi `greet` dan ternyata versi terbaru aplikasi Anda memerlukan perubahan teks yang ditampilkan, Anda cukup ubah satu baris kode saja, tak perlu 30 baris kode yang berbeda. Selain itu, jika Anda memiliki kode yang cukup panjang akan lebih baik jika kode tersebut dimasukkan ke dalam fungsi supaya lebih mudah dibaca.

### Function parameters

Pada beberapa kasus fungsi bisa memerlukan input data untuk diproses. Input data ini kita kenal sebagai **parameter**. Untuk menambahkan *parameter* ke dalam fungsi, kita bisa memasukkannya ke dalam tanda kurung. Sebuah fungsi bisa menerima nol, satu, atau beberapa parameter.

Contoh penggunaan parameter pada fungsi yang pernah kita lihat adalah pada fungsi `print()`.

```
1. print('Hello Dart!');
```

Berikut ini adalah contoh fungsi dengan dua parameter:

```
1. void main() {
2.   greet('Dicoding', 2015); // output : Halo Dicoding! Tahun ini Anda berusia 5 tahun
3. }
4.
5. void greet(String name, bornYear) {
6.   var age = 2020 - bornYear;
7.   print('Halo $name! Tahun ini Anda berusia $age tahun');
8. }
```

Sebuah fungsi juga bisa menghasilkan output atau mengembalikan nilai. Fungsi yang mengembalikan nilai ditandai dengan definisi *return type* selain `void` dan memiliki *keyword* `return`. Contohnya seperti berikut:

```
1. void main() {
2.   var firstNumber = 7;
3.   var secondNumber = 10;
4.   print('Rata-rata dari $firstNumber & $secondNumber adalah ${average(firstNumber, secondNumber)}');
5. }
6.
7. double average(num num1, num num2) {
8.   return (num1 + num2) / 2;
9. }
```

Jika fungsi hanya memiliki satu baris kode atau instruksi di dalamnya, maka bisa disingkat dengan anotasi `=>`. Ini juga dikenal dengan nama **arrow syntax**.

```
1. double average(num num1, num num2) => (num1 + num2) / 2;
2. void greeting() => print('Hello');
```

### Optional parameters

Anda memiliki fungsi seperti berikut:

```
1. void greetNewUser(String name, int age, bool isVerified)
```

Satu-satunya cara untuk bisa memanggil fungsi di atas adalah dengan cara berikut:

```
1. greetNewUser('Widy', 20, true);
```

Namun, Dart mendukung **optional parameter**, di mana kita tidak wajib mengisi parameter yang diminta oleh fungsi. Untuk bisa membuat parameter menjadi opsional, kita perlu memasukkannya ke dalam kurung siku seperti contoh berikut:

```
1. void greetNewUser([String name, int age, bool isVerified])
```

Cara ini disebut dengan **positional optional parameters**. Dengan *optional parameter* seperti di atas kita bisa memanggil fungsi seperti berikut:

```
1. greetNewUser('Widy', 20, true);  
2. greetNewUser('Widy', 20);  
3. greetNewUser('Widy');  
4. greetNewUser();
```

Setiap parameter yang tidak dimasukkan akan memiliki nilai **null**. Dengan cara ini, urutan parameter masih perlu diperhatikan sehingga jika kita hanya ingin mengisi parameter terakhir, kita perlu mengisi parameter sebelumnya dengan **null**.

```
1. greetNewUser(null, null, true);
```

Untuk mengatasi masalah di atas kita bisa memanfaatkan **named optional parameters**. Pada opsi ini kita menggunakan kurung kurawal pada parameter.

```
1. void greetNewUser({String name, int age, bool isVerified})
```

Dengan cara ini Anda bisa memasukkan parameter tanpa mepedulikan urutan parameter dengan menyebutkan nama parameternya.

```
1. greetNewUser(name: 'Widy', age: 20, isVerified: true);  
2. greetNewUser(name: 'Widy', age: 20);  
3. greetNewUser(age: 20);  
4. greetNewUser(isVerified: true);
```

Terkadang **null** bukanlah pilihan yang kita inginkan ketika menggunakan *optional parameter*. Sebagai solusi, kita bisa menggunakan **default value parameter**. Kita akan memberikan nilai **default** pada parameter lalu nilai ini akan digunakan jika kita tidak memasukkan parameternya.

```
1. void greetNewUser({String name = "Dicoding", int age = 5, bool isVerified = false})
```

# Variable Scope

Setelah Anda memisahkan kode Anda ke dalam blok atau fungsi yang terpisah, perlu Anda ketahui bahwa hal tersebut akan mempengaruhi bagaimana suatu variabel digunakan. Setiap variabel memiliki *scope* atau lingkungannya masing-masing. Sebuah variabel dianggap satu lingkup selama masih berada di satu blok kurung kurawal yang sama. Lingkup ini menentukan bagian kode mana yang dapat membaca dan menggunakan variabel tersebut.

Perhatikan kode berikut ini:

```
1. void main() {  
2.   var price = 300000;  
3.   var discount = 0;  
4.   print('You need to pay: ${price - discount}');  
5. }
```

Pada kode di atas variabel `discount` masih bisa diakses dari dalam kode `if` karena masih berada di dalam satu *scope* fungsi `main()`. Bagaimana jika Anda ingin memisahkan kode di atas menjadi dua fungsi untuk menghitung diskonnya?

```
1. void main() {  
2.   var price = 300000;  
3.   var discount = checkDiscount(price);  
4.   print('You need to pay: ${price - discount}');  
5. }  
6.  
7. num checkDiscount(num price) {  
8.   num discount = 0;  
9.   if (price >= 100000) {  
10.    discount = 10 / 100 * price;  
11.  }  
12.  
13.  return discount;  
14. }
```

Variabel `discount` dideklarasikan pada fungsi `checkDiscount()` sehingga memiliki *scope* pada fungsi tersebut dan menyebabkan eror pada fungsi `main()`. Maka untuk mengatasinya kita tetap perlu membuat variabel di kedua fungsi.

Selain berada dalam lingkup fungsi, suatu variabel juga bisa menjadi variabel global, yaitu variabel yang dideklarasikan di luar blok kode apa pun. Variabel ini bisa diakses di mana pun selama masih berada di berkas yang sama.

```
1. var price = 300000;  
2.  
3. void main() {  
4.   var discount = checkDiscount(price);  
5.   print('You need to pay: ${price - discount}');  
6. }
```

```

7.
8. num checkDiscount(num price) {
9.   num discount = 0;
10.  if (price >= 100000) {
11.    discount = 10 / 100 * price;
12.  }
13.
14.  return discount;
15. }

```

Variabel juga dapat memiliki *scope* yang sespesifik mungkin hingga ke level *control flow*.

```

1. num checkDiscount(num price) {
2.   num discount = 0;
3.   if (!discountApplied) { // Error
4.     if (price >= 100000) {
5.       discount = 10 / 100 * price;
6.       var discountApplied = true;
7.     }
8.   }
9.
10.  return discount;
11. }

```

## Constants & Final

Terkadang kita butuh menyimpan sebuah nilai yang tidak akan pernah berubah selama program berjalan. Variabel telah membantu kita untuk menyimpan nilai dan bisa diakses dengan nama yang deskriptif. **Constants** adalah hal baru yang akan kita pelajari dan berguna untuk menyimpan nilai yang tidak akan berubah selama program berjalan.

Sesuai pengertian di atas, kita bisa mendefinisikan nilai yang konstan pada program kita. Salah satu contoh paling mudah yang bisa kita ambil adalah nilai  $\pi = 3.14$ . Untuk mendefinisikan variabel konstan, gunakanlah *keyword* **const**.

```

1. const pi = 3.14;

```

*Type inference* dari Dart akan secara otomatis mendeteksi tipe data dari **const pi** di atas sebagai **double**, namun Anda dapat menentukan tipe data secara eksplisit.

```

1. const num pi = 3.14;

```

Sehingga pada sebuah aplikasi kalkulator luas lingkaran, kode Anda akan menjadi seperti berikut:

```

1. const num pi = 3.14;
2.
3. void main() {
4.   var radius = 7;
5.   print('Luas lingkaran dengan radius $radius = ${calculateCircleArea(radius)}');

```

```
6. }  
7.  
8. num calculateCircleArea(num radius) => pi * radius * radius;
```

Selain `const`, opsi lain untuk menghindari perubahan nilai variabel setelah diinisialisasi adalah `final`. Apa bedanya `final` dan `const`?

Variabel yang dideklarasikan sebagai `const` bersifat *compile-time constants*, artinya nilai tersebut harus sudah diketahui sebelum program dijalankan. Sedangkan `final`, nilainya masih bisa diinisialisasi ketika *runtime* atau ketika aplikasi berjalan. Sebagai contoh kode berikut:

```
1. final firstName = stdin.readLineSync();  
2. final lastName = stdin.readLineSync();  
3.  
4. // lastName = 'Dart'; Tidak bisa mengubah nilai  
5.  
6. print('Hello $firstName $lastName');
```

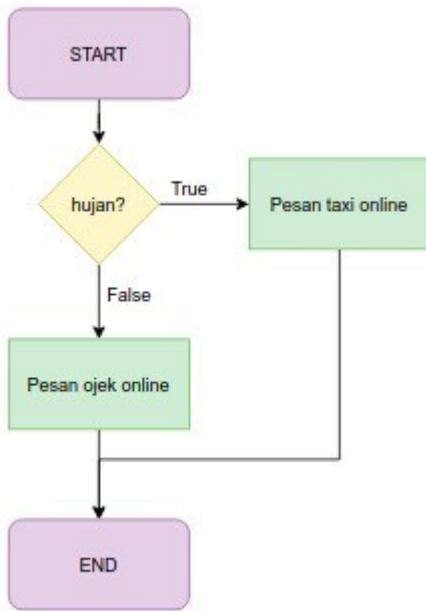
Kita masih bisa menerima input dan menyimpannya pada variabel `firstName` dan `lastName`, namun nilai variabel tersebut tidak bisa diubah setelah diinisialisasi.

Variabel yang nilainya tidak bisa berubah dikenal dengan *immutable variable*. *Mutability* ini memungkinkan kita terhindar dari *bug* yang tidak terduga karena terjadi perubahan nilai. Maka dari itu jika Anda yakin bahwa variabel Anda nilainya tetap, gunakanlah `const` atau `final`.

Jadi kapan kita harus menggunakan `const` dan kapan `final`? Kapan pun memungkinkan, selalu gunakan `const` karena *compile-time constant* memiliki performa yang lebih baik dan menggunakan memori lebih sedikit. Jika tidak memungkinkan untuk menggunakan `const`, gunakan `final` untuk melindungi variabel agar tidak berubah.

## Control Flow

Setiap hari kita melakukan perhitungan dan perbandingan guna membuat keputusan, apapun itu. Contohnya apakah perlu mencuci kendaraan ketika cuaca sedang cerah? Apa saja transportasi *online* yang bisa dipesan ketika hujan untuk sampai di tempat tujuan?



Sebuah program juga perlu membuat keputusan. Pada modul ini kita akan belajar bagaimana memberikan instruksi bagi komputer untuk mengambil keputusan dari kondisi yang diberikan serta bagaimana melakukan instruksi yang berulang.

## If and Else

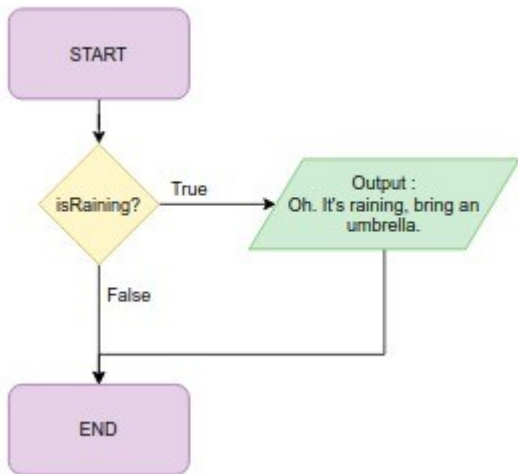
Ketika mengembangkan sebuah program, kita akan bertemu dengan alur yang bercabang tergantung kepada kondisi yang terjadi. Untuk mengakomodasi dan mengecek sebuah kondisi pada Dart kita menggunakan kata kunci `if`.

Ekspresi `if` akan menguji suatu kondisi. Jika hasil ekspresi tersebut bernilai *true*, maka blok kode di dalamnya akan dijalankan. Sebaliknya, jika bernilai *false* maka proses yang ditentukan akan dilewatkan.

```
1. void main() {
2.   var isRaining = true;
3.
4.   print('Prepare before going to office.');
```

```
5.   if (isRaining) {
6.     print("Oh. It's raining, bring an umbrella.");
7.   }
8.   print('Going to the office.');
```

```
9. }
```



Kode di atas akan menampilkan output:

1. Prepare before going to office.
2. Oh. It's raining, bring an umbrella.
3. Going to the office.

Jika Anda mengubah nilai `isRaining` menjadi `false`, maka kode di dalam blok `if` akan dilewatkan dan program Anda tidak akan mengingatkan untuk membawa payung.

Lalu bagaimana jika Anda ingin melakukan operasi lain ketika kondisi bernilai `false`? Jawabannya adalah dengan menggunakan `else`. Pada contoh kode berikut kita akan melakukan pengecekan kondisi pada operator perbandingan dan operator logika.

```
1. void main() {
2.     var openHours = 8;
3.     var closedHours = 21;
4.     var now = 17;
5.
6.     if (now > openHours && now < closedHours) {
7.         print("Hello, we're open");
8.     } else {
9.         print("Sorry, we've closed");
10.    }
11. }
```

Anda juga dapat mengecek beberapa kondisi sekaligus dengan menggabungkan `else` dan `if`. Contohnya seperti program konversi nilai berikut:

```
1. void main() {
2.     stdout.write('Inputkan nilai Anda (1-100) : ');
3.     var score = num.parse(stdin.readLineSync());
4.
5.     print('Nilai Anda: ${calculateScore(score)}');
```



```

6. }
7.
8. String calculateScore(num score) {
9.   if (score > 90) {
10.    return 'A';
11.  } else if (score > 80) {
12.    return 'B';
13.  } else if (score > 70) {
14.    return 'C';
15.  } else if (score > 60) {
16.    return 'D';
17.  } else {
18.    return 'E';
19.  }
20. }

```

Fitur menarik lain dari Dart adalah ***conditional expressions***. Dengan ini kita bisa menuliskan *if-else statement* hanya dalam satu baris:

```

1. // condition ? true expression : false expression
2.
3. var shopStatus = now > openHours ? "Hello, we're open" : "Sorry, we've closed";

```

Selain itu Dart juga mendukung *conditional expressions* seperti berikut:

```

1. expression1 ?? expression2
2. var buyer = name ?? 'user';

```

Pada kode di atas jika variabel **name** tidak bernilai **null**, maka **buyer** akan menyimpan nilai dari **name**. Namun jika tidak, **buyer** akan berisi **'user'**.

Berbeda dengan modul-modul sebelumnya yang menggunakan *flowchart*, bisakah kali ini Anda membuat program atau memahami maksud gambar berikut?

## Being a Programmer

**Mom said:** "Please go to the shop and buy 1 bottle of milk. If they have eggs, bring 6"

**I came back with 6 bottle of milk.**

**She said:** "Why the hell did you buy 6 bottles of milk?"

**I said:** "BECAUSE THEY HAD EGGS"



WebDevelopersNotes.com

## For Loops

Ketika menulis program komputer, akan ada situasi di mana kita perlu melakukan hal sama berkali-kali. Misalnya kita ingin menampilkan semua nama pengguna yang terdaftar di aplikasi kita, atau sesederhana menampilkan angka 1 sampai 10. Tentunya tidak praktis jika kita menulis kode seperti berikut:

```
1. print(1);
2. print(2);
3. print(3);
4. print(4);
5. print(5);
6. print(6);
7. print(7);
8. print(8);
9. print(9);
10. print(10);
```

Bagaimana jika kita perlu menampilkan angka 1 sampai 100?

Dart memiliki banyak opsi untuk melakukan perulangan kode, salah satunya adalah **for**. **For** cocok digunakan pada kondisi perulangan di mana kita membutuhkan variabel indeks dan tahu berapa kali perulangan yang kita butuhkan. Sebagai contoh jika kita ingin menampilkan angka 1 sampai 100, kita bisa menuliskan seperti berikut:

```
1. for (int i = 1; i <= 100; i++) {
2.   print(i);
3. }
```

Lebih ringkas bukan? Terdapat tiga bagian utama dalam sintaks `for` di atas:

- Pertama, variabel *index* yang seringkali diberi nama `i` yang berarti *index*. Pada variabel ini kita menginisialisasi nilai awal dari perulangan yang kita lakukan.
- Kedua, operasi perbandingan. Pada bagian ini komputer akan melakukan pengecekan kondisi apakah perulangan masih perlu dilakukan. Jika bernilai `true` maka kode di dalam blok `for` akan dijalankan.
- Ketiga, *increment/decrement*. Di sini kita melakukan penambahan atau pengurangan variabel *index*. Jadi pada contoh di atas variabel indeks akan ditambah dengan 1 di setiap akhir perulangan.

Jika dituliskan dalam bentuk *pseudocode*, maka kode di atas bisa dimaknai dengan “Jika `i` kurang dari sama dengan 100, maka jalankan kode berikut.”

### Challenge

Kini saatnya menguji pemahaman Anda tentang materi *for loops*. Bisakah Anda membuat program Dart yang menampilkan output seperti berikut?

```
1. *
2. **
3. ***
4. ****
5. *****
6. ****
7. ***
8. **
9. *
10. 
```

## While and do-while

Metode lain untuk melakukan perulangan adalah dengan `while`. Sama seperti `for`, instruksi `while` mengevaluasi ekspresi *boolean* dan menjalankan kode di dalam blok `while` ketika bernilai `true`.

Untuk menampilkan angka 1 sampai 100 dengan `while` kita bisa menulis kode seperti berikut:

```
1. var i = 1;
2.
3. while (i <= 100) {
4.   print(i);
5.   i++;
6. }
```

Bisa dilihat pada kode di atas bahwa perulangan dengan *while* tidak memiliki ketergantungan dengan variabel index seperti pada *for loop*. Karena itu, meskipun *while* dapat melakukan perulangan yang sama dengan *for*, *while* lebih cocok digunakan pada kasus di mana kita tidak tahu pasti berapa banyak perulangan yang diperlukan.

Bentuk lain dari `while` adalah perulangan `do-while`.

```
1. do {
2.   print(i);
3.   i++;
4. } while (i <= 100);
```

Kondisi pada **while** akan dievaluasi sebelum blok kode di dalamnya dijalankan, sedangkan **do-while** akan mengevaluasi *boolean expression* setelah blok kodenya dijalankan. Ini artinya kode di dalam **do-while** akan dijalankan setidaknya satu kali.

Salah satu skenario umum dari penggunaan **do-while** adalah pada validasi *user*.

```
1. String username;
2. bool notValid = false;
3.
4. do {
5.   stdout.write('Masukkan nama Anda (min. 6 karakter): ');
6.   username = stdin.readLineSync();
7.
8.   if (username.length < 6 ) {
9.     notValid = true;
10.    print('Username Anda tidak valid');
11.  }
12. } while (notValid);
```

Pada contoh di atas jika **username** yang dimasukkan oleh *user* kurang dari 6 karakter, maka input tersebut tidak valid dan *user* akan diminta lagi untuk memasukkan *username*.

## Infinite loops

Ketika menerapkan perulangan pada program kita, ada satu kondisi yang perlu kita hindari yaitu **infinite loop**. *Infinite loop* atau *endless loop* adalah kondisi di mana program kita *stucked* di dalam perulangan. Ia akan berjalan terus hingga menyebabkan *crash* pada aplikasi dan komputer kecuali ada intervensi secara eksternal, seperti mematikan aplikasi.

Kode berikut ini adalah contoh di mana kondisi *infinite loop* dapat terjadi:

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

Dapatkah Anda mencari apa yang salah dari kode di atas sehingga terjadi *infinite loop*?

Jawabannya adalah karena variabel **i** selalu bernilai 1. Alhasil kondisi **i < 5** akan selalu bernilai **true** dan akibatnya aplikasi akan terus mencetak 1 ke konsol sehingga mengalami *crash*.

### Challenge

Sebenarnya program *input username* di atas masih belum lengkap karena memiliki *bug* yang bisa menimbulkan *infinite loop*. Dapatkah Anda menemukan *bug* tersebut dan cara mengatasinya?

## Break and Continue

Anda memiliki aplikasi yang menyimpan data 20 bilangan prima pertama. Pengguna dapat mencari bilangan prima lalu komputer akan menampilkan urutan berapa bilangan tersebut. Ketika bilangan prima tersebut sudah ditemukan tentunya komputer tidak perlu melanjutkan proses perulangan lagi. Nah, di sinilah kita bisa menggunakan **break** untuk menghentikan dan keluar dari proses iterasi.

```
1. void main() {
2.     // 20 bilangan prima pertama
3.     var primeNumbers = [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71];
4.     stdout.write('Masukkan bilangan prima : ');
5.     var searchNumber = int.parse(stdin.readLineSync());
6.
7.     for (int i = 0; i < primeNumbers.length; i++) {
8.         if (searchNumber == primeNumbers[i]) {
9.             print('$searchNumber adalah bilangan prima ke-${i+1}');
10.            break;
11.        }
12.        print('$searchNumber != ${primeNumbers[i]}');
13.    }
14. }
```

Ketika kode di atas dijalankan, proses iterasi akan dihentikan ketika angka yang diinputkan pengguna sama dengan bilangan prima yang dicari.

```
1. Masukkan bilangan prima : 13
2. 13 != 2
3. 13 != 3
4. 13 != 5
5. 13 != 7
6. 13 != 11
7. 13 adalah bilangan prima ke-6
```

*Keyword* lain yang berguna pada proses perulangan adalah **continue**. Dengan *continue* kita bisa melewati proses iterasi dan lanjut ke proses iterasi berikutnya. Misalnya Anda ingin menampilkan angka 1 sampai 10 kecuali angka kelipatan 3. Anda dapat menuliskannya seperti berikut:

```
1. void breakContinue() {
2.     for (int i = 1; i <= 10; i++) {
3.         if (i % 3 == 0) {
4.             continue;
5.         }
6.         print(i);
7.     }
```

```
8. }
9.
10. /*
11. output :
12. 1
13. 2
14. 4
15. 5
16. 7
17. 8
18. 10
19. */
```

## Switch and Case

Sebelumnya kita telah mempelajari bagaimana mengondisikan logika komputer dengan menggunakan *if*. Namun, bagaimana jika ada banyak kondisi yang perlu dicek menggunakan *if*? Tentu akan membingungkan dan kode kita pun jadi sulit dibaca.

Dart mendukung *statement switch* untuk melakukan pengecekan banyak kondisi dengan lebih mudah dan ringkas.

```
1. switch (variable/expression) {
2.     case value1:
3.         // do something
4.         break;
5.     case value2:
6.         // do something
7.         break;
8.     ...
9.     ...
10.    default:
11.        // do something else
12. }
```

Tanda kurung setelah *keyword switch* berisi variabel atau ekspresi yang akan dievaluasi. Kemudian untuk setiap kondisi yang mungkin terjadi kita masukkan *keyword case* diikuti dengan nilai yang valid. Jika kondisi pada *case* sama dengan variabel pada *switch*, maka blok kode setelah titik dua (:) akan dijalankan. *Keyword break* digunakan untuk keluar dari proses *switch*. Terdapat satu *case* bernama *default* yang memiliki fungsi yang sama dengan *keyword else* pada *control flow if-else*. Jika tidak ada nilai yang sama dengan variabel pada *switch* maka blok kode ini akan dijalankan.

Berikut ini adalah contoh aplikasi kalkulator yang menerapkan *switch-case*.

```
1. stdout.write('Masukkan angka pertama : ');
2. var firstNumber = num.parse(stdin.readLineSync());
3. stdout.write('Masukkan operator [ + | - | * | / ] : ');
4. var operator = stdin.readLineSync();
5. stdout.write('Masukkan angka kedua : ');
6. var secondNumber = num.parse(stdin.readLineSync());
```

```

7.
8. switch (operator) {
9.   case '+':
10.    print('$firstNumber $operator $secondNumber = ${firstNumber + secondNumber}');
11.    break;
12.   case '-':
13.    print('$firstNumber $operator $secondNumber = ${firstNumber - secondNumber}');
14.    break;
15.   case '*':
16.    print('$firstNumber $operator $secondNumber = ${firstNumber * secondNumber}');
17.    break;
18.   case '/':
19.    print('$firstNumber $operator $secondNumber = ${firstNumber / secondNumber}');
20.    break;
21.   default:
22.    print('Operator tidak ditemukan');
23. }

```

## Collections

Kita telah mempelajari tentang tipe data dan *control flow*. Lanjut, kini kita akan menghadapi masalah yang lebih kompleks. Untuk bisa memecahkannya secara efisien, kita membutuhkan struktur data yang lebih canggih.

Selain *string*, *number*, dan *boolean*, Dart masih memiliki tipe data lain yang dapat menyimpan banyak data sekaligus yang dalam istilah pemrograman dikenal sebagai ***collections***. *Collections* merupakan sebuah objek yang bisa menyimpan kumpulan objek lain. Contoh *collections* pada Dart antara lain *List*, *Set*, dan *Map*.

## List

List sesuai namanya dapat menampung banyak data ke dalam satu objek. Dalam kehidupan sehari-hari kita menggunakan list untuk menyimpan daftar belanja, nomor telepon, dsb. Begitu pula dengan Dart kita bisa menyimpan bermacam-macam tipe data seperti *string*, *number*, dan *boolean*. Cara penulisannya pun sangat mudah. Perhatikan saja contoh berikut:

```

1. List<int> numberList = [1, 2, 3, 4, 5];

```

Kode di atas adalah contoh dari satu objek List yang berisi kumpulan data dengan tipe *integer*. Karena kompiler bisa mengetahui tipe data yang ada dalam sebuah objek List, maka tak perlu kita menuliskannya secara eksplisit.

```

1. var numberList = [1, 2, 3, 4, 5];
2. var stringList = ['Hello', 'Dicoding', 'Dart'];

```

Sesuai contoh di atas, kita mendefinisikan tipe data yang bisa dimasukkan ke dalam List di dalam tanda kurung sudut (<>). Sama seperti variabel, jika kita tidak mendefinisikan nilai secara eksplisit ke dalam List, maka List akan menyimpan tipe *dynamic* atau bisa menyimpan semua tipe data.

```
1. List dynamicList = [1, 'Dicoding', true]; // List<dynamic>
```

Ketika bermain dengan sebuah List, tentunya ada saat di mana kita ingin mengakses posisi tertentu dari List tersebut. Untuk melakukannya, kita bisa menggunakan fungsi *indexing* seperti berikut:

```
1. print(dynamicList[1]);
```

Perhatikan kode di atas. Fungsi indexing ditandai dengan tanda []. Jika Anda mengira bahwa konsol akan menampilkan angka 1, maka tebakan Anda kurang tepat. Karena dalam sebuah List, indeks dimulai dari 0. Maka ketika kita akan mengakses data pada *dynamicList* yang berada pada indeks ke-1, artinya data tersebut merupakan data pada posisi ke-2. Jadi data yang akan ditampilkan pada konsol adalah **Dicoding**.

Lalu apa yang akan terjadi jika kita berusaha menampilkan item dari List yang berada di luar dari ukuran List tersebut? Sebagai contoh, Anda ingin mengakses indeks ke-3 dari *dynamicList*:

```
1. print(dynamicList[3]);
```

Hasilnya adalah eror! Kompiler akan memberitahukan bahwa perintah itu tidak bisa dijalankan. Berikut pesan eror yang akan muncul:

Unhandled exception:  
RangeError (index): Invalid value: Not in range 0..2, inclusive: 3

Pesan di atas memberitahu kita bahwa List telah diakses dengan indeks ilegal. Ini akan terjadi jika indeks yang kita inginkan negatif atau lebih besar dari atau sama dengan ukuran List tersebut.

Masih ingat *looping*? Untuk menampilkan seluruh item dari *list* kita bisa memanfaatkan *looping*. Contohnya perhatikan kode berikut:

```
1. for(int i = 0; i < stringList.length; i++) {  
2.   print(stringList[i]);  
3. }
```

Pada kode di atas kita memanfaatkan perulangan sebanyak jumlah data di dalam *list* untuk mencetak data yang ada di dalam *list*. Banyaknya data di dalam *list* bisa kita panggil melalui properti *.length*.

Selain itu kita juga bisa menggunakan fungsi *foreach* untuk menampilkan data di dalam *list*.

```
1. stringList.forEach((s) => print(s));
```



Mekanisme di atas dikenal sebagai **lambda** atau **anonymous function**. Kita akan mempelajarinya lebih dalam pada modul yang akan datang.

Sejauh ini kita baru belajar menginisialisasikan dan mengakses data dari sebuah List. Pasti Anda bertanya, “Bagaimana kita memanipulasi data pada List tersebut?” Nah, untuk menambahkan data ke dalam *list*, kita bisa menggunakan fungsi **add()**.

```
1. stringList.add('Flutter');
```

Fungsi **add** ini akan menambahkan data di akhir *list*. Sehingga ketika dicetak, konsol akan menampilkan data berikut:

```
1. [Hello, Dicoding, Dart, Flutter]
```

Lalu bagaimana jika kita ingin menambahkan data namun tidak di akhir List? Jawabannya adalah dengan fungsi **insert**. Di dalam fungsi **insert** kita perlu memasukkan 2 parameter, yaitu indeks *list* dan data yang akan dimasukkan.

```
1. stringList.insert(0, 'Programming');  
2. // stringList = [Programming, Hello, Dicoding, Dart, Flutter]
```

Untuk mengubah nilai di dalam *list*, kita bisa langsung menginisialisasikan nilai baru sesuai indeks yang diinginkan.

```
1. stringList[1] = 'Application';
```

Sedangkan untuk menghapus data terdapat beberapa fungsi **remove** yang bisa kita gunakan, antara lain:

```
1. stringList.remove('Programming'); // Menghapus list dengan nilai Programming  
2. stringList.removeAt(1);           // Menghapus list pada index ke-1  
3. stringList.removeLast();          // Menghapus data list terakhir  
4. stringList.removeRange(0, 2);      // Menghapus list mulai index ke-0 sampai ke-1 (indeks 2 masih dipertahankan)
```

## Set

Selanjutnya kita akan membahas jenis *collection* yang kedua, yaitu **Set**. **Set** merupakan sebuah *collection* yang hanya dapat menyimpan nilai yang unik. Ini akan berguna ketika Anda tidak ingin ada data yang sama alias duplikasi dalam sebuah *collection*. Kita bisa mendeklarasikan Set dengan beberapa cara berikut:

```
1. var numberSet = {1, 4, 6};  
2. Set<int> anotherSet = new Set.from([1, 4, 6, 4, 1]);
```

Perhatikan kode di atas. Di sana terdapat beberapa angka yang duplikat, yaitu angka 1 dan 4. Silakan tampilkan pada konsol dan lihat hasilnya.

```
1. print(anotherSet);
2.
3. // Output: {1, 4, 6}
```

Secara otomatis Set akan membuang angka yang sama, sehingga hasilnya adalah {1, 4, 6}.

Untuk menambahkan data ke dalam Set kita dapat memanfaatkan fungsi `add()` atau `addAll()`.

```
1. numberSet.add(6);
2. numberSet.addAll({2, 2, 3});
```

Fungsi `add` akan menambah satu *item* ke dalam Set, sementara `addAll` digunakan untuk menambahkan beberapa *item* sekaligus. Nilai yang duplikat akan diabaikan.

Lalu gunakan fungsi `remove()` untuk menghapus objek di dalam set yang diinginkan.

```
1. numberSet.remove(3);
```

Kode di atas akan menghapus nilai 3 di dalam Set, bukan indeks ke-3.

Kemudian untuk menampilkan data pada indeks tertentu, gunakanlah fungsi `elementAt()`.

```
1. var numberSet = {1, 4, 6};
2.
3. numberSet.add(6);
4. numberSet.addAll({2, 2, 3});
5. numberSet.remove(3);
6.
7. print(numberSet.elementAt(2));
8.
9. // Output: 6
```

Selain itu, Dart juga memiliki fungsi ***union*** dan ***intersection*** untuk mengetahui gabungan dan irisan dari 2 (dua) buah Set. Sebagai contoh:

```
1. var setA = {1, 2, 4, 5};
2. var setB = {1, 5, 7};
3.
4. var union = setA.union(setB);
5. var intersection = setA.intersection(setB);
6.
7. print(union);
8. print(intersection);
9.
10. // union: {1, 2, 4, 5, 7}
11. // intersection: {1, 5}
```

# Map

*Collection* ketiga adalah Map, yakni sebuah *collection* yang dapat menyimpan data dengan format *key-value*. Perhatikan contoh berikut:

```
1. var capital = {  
2.   'Jakarta': 'Indonesia',  
3.   'London': 'England',  
4.   'Tokyo': 'Japan'  
5. };
```

*String* yang berada pada sebelah kiri titik dua (:) adalah sebuah *key*, sedangkan yang di sebelah kanan merupakan *value*-nya. Lalu untuk mengakses nilai dari Map tersebut, kita bisa menggunakan *key* yang sudah dimasukkan. Misalnya, kita bisa menggunakan *key* “Jakarta” untuk mendapatkan *value* “Indonesia”:

```
1. print(capital['Jakarta']);  
2.  
3. // Output: Indonesia
```

Kita dapat menampilkan *key* apa saja yang ada di dalam Map dengan menggunakan *property keys*.

```
1. var mapKeys = capital.keys;  
2.  
3. // mapKeys = (Jakarta, London, Tokyo)
```

Sedangkan untuk mengetahui nilai apa saja yang ada di dalam Map kita bisa menggunakan *property values*.

```
1. var mapValues = capital.values;  
2.  
3. // mapValues = (Indonesia, England, Japan)
```

Untuk menambahkan *key-value* baru ke dalam Map, kita bisa melakukannya dengan cara berikut:

```
1. capital['New Delhi'] = 'India';  
2. print(capital);  
3.  
4. // Output: {Jakarta: Indonesia, London: England, Tokyo: Japan, New Delhi: India}
```

# Object Oriented Programming

Pada modul awal kita sudah mengetahui bahwa Dart adalah bahasa yang mendukung pemrograman berorientasi objek. OOP adalah paradigma pemrograman yang banyak digunakan saat ini untuk mengembangkan aplikasi.

Paradigma OOP berdasarkan pada konsep objek yang memiliki atribut serta dapat melakukan operasi atau prosedur tertentu. Dengan OOP, kita bisa dengan mudah memvisualisasikan objek yang ada di dunia nyata ke dalam program komputer. Sebagai contoh, bayangkan kucing sebagai sebuah objek. Objek kucing ini memiliki karakteristik seperti warna bulu, usia kucing, dan berat badan. Ciri-ciri ini disebut dengan *attributes* atau *properties*. Selain itu kucing juga bisa melakukan beberapa hal seperti makan, tidur, dan bermain. Perilaku pada objek kucing ini adalah sebuah *method*.

Terdapat 4 (empat) pilar dalam pemrograman berorientasi objek, antara lain: *encapsulation*, *abstraction*, *inheritance*, dan *polymorphism*.

### Encapsulation

Enkapsulasi adalah kondisi di mana status atau kondisi di dalam *class*, dibungkus dan bersifat privat. Artinya objek lain tidak bisa mengakses atau mengubah nilai dari *property* secara langsung. Pada contoh kasus kucing kita tidak bisa langsung mengubah berat badan dari kucing, namun kita bisa menambahkannya melalui fungsi atau *method* makan.

### Abstraction

Abstraksi bisa dibilang merupakan penerapan alami dari enkapsulasi. Abstraksi berarti sebuah objek hanya menunjukkan operasinya secara *high-level*. Misalnya kita cukup tahu bagaimana seekor kucing makan, namun kita tidak perlu tahu seperti apa metabolisme biologis dalam tubuh kucing yang membuat berat badannya bertambah.

### Inheritance

Beberapa objek bisa memiliki beberapa karakteristik atau perilaku yang sama, namun mereka bukanlah objek yang sama. Di sinilah *inheritance* atau pewarisan berperan. Kucing memiliki sifat dan perilaku yang umum dengan hewan lain, seperti memiliki warna, berat, dsb. Maka dari itu kucing sebagai objek turunan (*subclass*) mewarisi semua sifat dan perilaku dari objek induknya (*superclass*). Begitu juga dengan objek ikan juga mewarisi sifat dan perilaku yang sama, namun ikan bisa berenang sementara kucing tidak.

### Polymorphism

*Polymorphism* dalam bahasa Yunani berarti “banyak bentuk.” Sederhananya objek dapat memiliki bentuk atau implementasi yang berbeda-beda pada satu metode yang sama. Semua hewan bernafas, namun tentu kucing dan ikan memiliki cara bernafas yang berbeda. Perbedaan bentuk atau cara pernafasan tersebut merupakan contoh dari *polymorphism*.

## Class

Salah satu fitur utama dari OOP adalah fitur seperti **class**. *Class* merupakan sebuah *blueprint* untuk membuat objek. Di dalam kelas ini kita mendefinisikan sifat (*attribute*) dan perilaku (*behaviour*) dari objek yang akan dibuat. Sebagai contoh kelas *Animal* memiliki atribut berupa nama, berat, dan umur, dll. Kemudian perilakunya adalah makan, tidur, dsb.

Setiap kelas memiliki *attribute* dan *behaviour*. Pada Dart kita mendefinisikan *attribute* dengan variabel, sedangkan *behaviour* sering direpresentasikan sebagai *function*.

Animal
+ String name + int age + double weight
- eat() - sleep() - poop()

Untuk mendefinisikan kelas dalam Dart, cukup gunakan keyword `class` diikuti dengan nama kelas yang akan dibuat.

```
1. class Animal {  
2. }
```

Kemudian kita bisa menambahkan variabel dan fungsi pada kelas tersebut.

```
1. class Animal {  
2.   String name;  
3.   int age;  
4.   double weight;  
5.  
6.   Animal(this.name, this.age, this.weight);  
7.  
8.   void eat() {  
9.     print('$name is eating.');10.    weight = weight + 0.2;  
11.  }  
12.  
13.  void sleep() {  
14.    print('$name is sleeping.');15.  }  
16.  
17.  
18.  void poop() {  
19.    print('$name is pooping.');20.    weight = weight - 0.1;  
21.  }  
22. }
```

Kemudian untuk membuat sebuah objek dari suatu *class*, gunakan sintaks berikut:

```
1. var nameOfObject = NameOfClass(property1, property2);
```

Sebuah objek sama seperti variabel, artinya kita bisa menggunakan `var` untuk menginisialisasikan sebuah objek. Objek yang disimpan ke dalam variabel ini juga dikenal dengan *instance* atau *instance of a class*. *Instance* ini menyimpan *reference* atau alamat memori dari objek. Proses membuat variabel *instance* seperti di atas disebut instansiasi (*instantiation*).

```
1. var dicodingCat = Animal('Gray', 2, 4.2);
```

Kita menggunakan nama kelas serta diikuti dengan tanda kurung. Di dalam tanda kurung ini kita bisa memasukkan parameter untuk menginisialisasi nilai di dalam objek. Tanda kurung ini merupakan sebuah *constructor* yang akan kita pelajari pada materi selanjutnya.

Setelah objek terbuat kita bisa menjalankan fungsi atau menampilkan nilai dari *property* yang ada di dalamnya.

```
1. void main() {  
2.   var dicodingCat = Animal('Gray', 2, 4.2);  
3.   dicodingCat.eat();  
4.   dicodingCat.poop();  
5.   print(dicodingCat.weight);  
6. }
```

Ketika program di atas dijalankan, konsol akan menampilkan hasil sebagai berikut:

```
1. Gray is eating.  
2. Gray is pooping.  
3. 4.3000000000000001
```

## Properties & Methods

Kita telah mempelajari variabel untuk menyimpan nilai dan *function* untuk menuliskan set instruksi yang bisa digunakan kembali. Di dalam class, variabel dan fungsi ini dikenal dengan ***property*** dan ***method***.

Seperti yang telah disebutkan pada materi *Class*, *property* merepresentasikan atribut pada sebuah objek sementara *method* menggambarkan perilaku dari objek.

Sama seperti variabel, kita mendeklarasikan *property* di dalam kelas dengan menentukan tipe datanya atau menginisialisasikan nilainya secara eksplisit.

```
1. class Animal {  
2.   String _name = '';  
3.   int _age;  
4.   double _weight = 0;  
5. }
```

OOP memiliki konsep enkapsulasi di mana kita bisa menyembunyikan informasi di dalam objek sehingga status atau data di dalam objek tidak bisa diubah atau bahkan dilihat. Umumnya bahasa pemrograman memiliki *visibility modifier* untuk menentukan siapa saja yang bisa mengakses *property* atau *method*. Namun, Dart tidak memiliki *keyword visibility modifier* seperti *private* atau *public*. Bagaimana cara mengatasinya?

Jadikanlah *class* sebagai *library* lalu panggilah ia dengan keyword **import**. Untuk membuat *class* sebagai *library* Anda cukup membuat berkas baru, sehingga Anda akan memiliki 2 buah berkas:

- [main.dart](#)

```
1. import 'Animal.dart';
2.
3. void main() {
4.   var dicodingCat = Animal('Gray', 2, 4.2);
5.
6.   dicodingCat.eat();
7.   dicodingCat.poop();
8.   print(dicodingCat.weight);
9. }
```

- [Animal.dart](#)

```
1. class Animal {
2.   String name = '';
3.   int age;
4.   double weight = 0;
5.
6.   Animal(this.name, this.age, this.weight);
7.
8.   void eat() {
9.     print('$name is eating.');
```

```
10.    weight = weight + 0.2;
11.  }
12.
13.  void sleep() {
14.    print('$name is sleeping.');
```

```
15.  }
16.
17.  void poop() {
18.    print('$name is pooping.');
```

```
19.    weight = weight - 0.1;
20.  }
21. }
```

*Property* yang *private* artinya hanya bisa diakses pada berkas atau library yang sama. Kita akan membutuhkan *private property* ini di saat kita tidak ingin objek diubah dari luar. Karena Dart tidak

memiliki *modifier private*, sebagai gantinya kita perlu menambahkan *underscore* (`_`) sebelum nama *property*.

```
1. String _name = '';
2. int _age;
3. double _weight = 0;
```

Setelah menambahkan *underscore* pada nama variabel, Anda akan mendapatkan eror di berkas `main.dart` ketika mengakses *property* `weight`. Apa pasal? Kini `weight` bersifat *private* dan tidak bisa diakses dari luar berkasnya. Solusinya, Anda bisa menambahkan *setter* dan *getter* untuk mendapatkan nilai serta mengubahnya dari luar berkas. Jika menggunakan IntelliJ IDEA Anda bisa menggunakan shortcut **Alt + Insert** lalu pilih **Getter and Setter**.

```
1. // Setter
2. set name(String value) {
3.     _name = value;
4. }
5.
6. // Getter
7. double get weight => _weight;
```

Selain dengan *setter*, Anda juga bisa mengubah nilai dengan *property* dari pemanggilan *method*. Pada contoh kelas hewan tentunya kita tidak bisa langsung mengubah nilai berat badan, namun kita bisa menambah dan mengubah nilainya melalui proses makan atau buang air besar (BAB).

```
1. void eat() {
2.     print('$_name is eating. ');
3.     _weight = _weight + 0.2;
4. }
5.
6. void poop() {
7.     print('$_name is pooping. ');
8.     _weight = _weight - 0.1;
9. }
```

## Constructor

Ketika suatu objek dibuat, semua properti pada kelas tersebut harus memiliki nilai. Kita dapat langsung menginisialisasi pada properti tertentu atau menginisiasiasinya melalui **constructor**. *Constructor* adalah fungsi spesial dari sebuah kelas yang digunakan untuk membuat objek.

Sesuai namanya, *constructor* digunakan untuk mengonstruksi objek baru.

Jadi kenapa *constructor* disebut sebagai fungsi yang spesial? Apa bedanya dengan fungsi lain pada *class*? Beberapa perbedaan antara *constructor* dan fungsi biasa adalah:

1. *Constructor* memiliki nama yang sama dengan nama kelas.
2. *Constructor* tidak memiliki nilai kembalian (*return type*).



3. *Constructor* akan secara otomatis dipanggil ketika sebuah objek dibuat.
4. Jika kita tidak mendefinisikan *constructor*, *default constructor* tanpa argumen akan dibuat.

Secara *default* sebuah kelas memiliki *constructor* yang tidak menerima argumen.

```
1. var dicodingCat = Animal();
```

Karena kita tidak memasukkan nilai ketika membuat objek, maka nilai *default* dari properti atau variabel akan digunakan. Anda perlu berhati-hati jika tidak memberikan nilai pada properti, karena akan membuat properti bernilai *null* sehingga bisa menyebabkan error.

Untuk memberikan nilai pada properti, silakan akses properti yang ada di dalam sebuah kelas.

```
1. var dicodingCat = Animal();
2. dicodingCat.name = 'Gray';
3. dicodingCat.age = 2;
4. dicodingCat.weight = 4.2;
```

Dengan membuat *constructor*, kita tidak hanya bisa menginisialisasikan nilai namun juga menjalankan instruksi tertentu ketika objek dibuat.

```
1. Animal(String name, int age, double weight) {
2.     this.name = name;
3.     this.age = age;
4.     this.weight = weight;
5.     // other instructions
6. }
```

Keyword **this** di atas menunjuk pada objek yang ada di kelas tersebut. *Keyword this* ini umumnya digunakan untuk menghindari ambiguitas antara atribut dari *class* dan parameter yang memiliki nama yang sama.

Jika *constructor* hanya digunakan untuk menginisialisasi nilai properti, maka kode konstruktor dapat diringkas menjadi seperti berikut:

```
1. Animal(this.name, this.age, this.weight);
```

### Named Constructor

Pada beberapa kasus kita mungkin akan membutuhkan beberapa *constructor* untuk skenario yang berbeda-beda. Pada situasi ini kita bisa memanfaatkan ***named constructor***.

Dengan menggunakan *named constructor*, kita dapat membuat beberapa *constructor* pada kelas yang sama. Setiap *constructor* akan memiliki nama yang unik.

```
1. class_name.constructor_name (arguments){
2.     // Statements
3. }
```

Contoh pada `class Animal` adalah seperti berikut:

```
1. Class Animal {
2.     ...
3.     Animal.Name(this._name);
4.     Animal.Age(this._age);
5.     Animal.Weight(this._weight);
6.     ...
7. }
```

## Inheritance

Beberapa objek bisa memiliki beberapa karakteristik atau perilaku yang sama, namun sebenarnya mereka bukanlah objek yang sama. Di sini hadir lah peran *inheritance* atau pewarisan. Apa definisi keduanya? **Inheritance** adalah kemampuan suatu program untuk membuat kelas baru dari kelas yang ada. Konsep *inheritance* ini bisa dibayangkan layaknya seorang anak mewarisi sifat dari orang tuanya. Di dalam OOP kelas yang menurunkan sifat disebut sebagai kelas induk (*parent class/superclass*) sementara kelas yang mewarisi kelas induknya disebut sebagai kelas anak (*child class/subclass*).

Yuk kembali lagi pada contoh objek kucing. Selain kucing ada jenis hewan lain yang bersifat sama. Misalnya ikan dan burung juga memiliki nama, berat, dan umur. Selain itu mereka juga melakukan aktivitas seperti makan dan tidur. Yang membedakan objek tersebut adalah cara mereka bernafas dan bergerak. Untuk lebih memahami, perhatikanlah tabel kelas berikut:

Cat	Fish	Bird
+ name + weight + age + furColor	+ name + weight + age + skinColor	+ name + weight + age + featherColor
- eat() - sleep() - poop() - walk()	- eat() - sleep() - poop() - swim()	- eat() - sleep() - poop() - fly()

Bisa kita lihat pada tabel di atas bahwa objek `Cat`, `Fish`, dan `Bird` memiliki beberapa *property* dan *method* yang sama seperti `name`, `weight`, `age`, `eat()`, dan `sleep()`.

Dibandingkan membuat 3 kelas dan menuliskan ulang properti yang sama, kita bisa memanfaatkan teknik *inheritance* dengan mengelompokkan properti dan fungsi yang sama. Caranya buat sebuah kelas baru yang nantinya akan diturunkan sifatnya.

Animal
+ name + weight

+age		
- eat() - sleep() - poop()		
Cat	Fish	Bird
+ furColor	+ skinColor	+ featherColor
- walk()	- swim()	- fly()

Setelah membuat kelas **Animal**, kita dapat membuat kelas lainnya lalu melakukan **extends** ke kelas induknya. Untuk menerapkan *inheritance* gunakan keyword **extends** seperti contoh berikut:

```
1. class ChildClass extends ParentClass {
2.
3. }
```

Dengan begitu kita bisa membuat kelas **Cat** mewarisi kelas **Animal**.

- [Cat.dart](#)

```
1. import 'Animal.dart';
2.
3. class Cat extends Animal {
4.   String furColor;
5.
6.   Cat(String name, int age, double weight, String furColor) : super(name, age, weight) {
7.     this.furColor = furColor;
8.   }
9.
10.  void walk() {
11.    print('$name is walking');
12.  }
13.
14. }
```

- [Animal.dart](#)

```
1. class Animal {
2.   String _name = '';
3.   int _age;
4.   double _weight = 0;
5. }
```

```

6.  Animal(this._name, this._age, this._weight);
7.
8.  String get name => _name;
9.  double get weight => _weight;
10.
11. void eat() {
12.     print('$_name is eating.');
```

```

13.     _weight = _weight + 0.2;
14. }
```

```

15.
16. void sleep() {
17.     print('$_name is sleeping.');
```

```

18. }
19. }
```

Karena kelas **Cat** adalah turunan dari kelas **Animal**, maka kita bisa mengakses sifat dan perilaku dari **Animal** melalui kelas **Cat**.

```

1. import 'Cat.dart';
2.
3. void main() {
4.     var dicodingCat = Cat('Grayson', 2, 2.2, 'Gray');
```

```

5.     dicodingCat.walk();
6.     dicodingCat.eat();
7.     print(dicodingCat.weight);
8. }
```

```

9.
10. /*
11. Output :
```

```

12. Grayson is walking
13. Grayson is eating.
14. 2.4000000000000004
15. */
```

## Inheritance constructor

Karena kelas **Animal** memiliki *constructor* untuk menginisialisasi properti di dalamnya, maka semua kelas turunannya juga perlu mengimplementasikan *constructor* tersebut. Oleh sebab itu ketika membuat kelas **Cat** tanpa mendefinisikan *constructor* kita akan mendapatkan eror. IntelliJ IDEA akan memberikan saran untuk membuat *constructor*.

```

1. Cat(String name, int age, double weight) : super(name, age, weight);
```

Keyword **super** di atas akan diarahkan ke *constructor* dari kelas **Animal**.

Jika ingin menginisialisasikan nilai **furColor** melalui *constructor*, maka kita bisa menambahkan parameter di dalam *constructor*.

```

1. Cat(String name, int age, double weight, String furColor) : super(name, age, weight) {
2.     this.furColor = furColor;
```

```
3. }
```

Atau, kita bisa meringkasnya seperti yang telah kita pelajari pada materi *constructor*.

```
1. Cat(String name, int age, double weight, this.furColor) : super(name, age, weight);
```

## Abstract Class

Sesuai namanya, *abstract* merupakan gambaran umum dari sebuah kelas. Ia tidak dapat direalisasikan dalam sebuah objek. Pada modul sebelumnya kita sudah mempunyai kelas **Animal**. Secara harfiah hewan merupakan sebuah sifat. Kita tidak tahu bagaimana objek hewan tersebut. Kita bisa melihat bentuk kucing, ikan, dan burung namun tidak untuk hewan. Maka dari itu konsep *abstract class* perlu diterapkan agar kelas **Animal** tidak dapat direalisasikan dalam bentuk objek namun tetap dapat menurunkan sifatnya kepada kelas turunannya.

Untuk menjadikan sebuah kelas menjadi *abstract*, kita hanya perlu menambah keyword **abstract** sebelum penulisan kelas:

```
1. abstract class Animal {  
2.   String name;  
3.   int age;  
4.   double weight;  
5.  
6.   // ...  
7. }
```

Dengan begitu kelas **Animal** tidak dapat diinisialisasikan menjadi sebuah objek.

```
1. var dicodingCat = Animal('Gray', 2, 4.2); // Error: The class 'Animal' is abstract and can't be  
   instantiated.
```

## Implicit Interface

Selain *abstract class*, cara lain yang bisa kita gunakan untuk menerapkan abstraksi dalam OOP adalah dengan **interface**. *Interface* atau antarmuka merupakan set instruksi yang bisa diimplementasi oleh objek. Secara umum, *interface* berfungsi sebagai penghubung antara sesuatu yang abstrak dengan sesuatu yang nyata.

Bayangkan *remote TV* atau tombol yang ada di HP Anda. Tombol-tombol ini bisa kita sebut sebagai *interface*. Kita tak perlu tahu dan peduli tentang bagaimana fungsi yang ada di dalamnya.

Dart tidak memiliki *keyword* atau *syntax* untuk mendeklarasikan *interface* seperti bahasa pemrograman OOP lainnya. Setiap *class* di dalam Dart dapat bertindak sebagai *interface*. Maka dari itu *interface* pada Dart dikenal sebagai **implicit interface**. Untuk mengimplementasikan *interface*, gunakan keyword **implements**. Kita bisa mengimplementasikan beberapa *interface* sekaligus pada satu kelas.

```
1. class ClassName implements InterfaceName
```

Setelah kelas mengimplementasikan *interface*, maka kelas tersebut wajib mengimplementasikan semua metode yang ada di dalam *interface*. Misalnya kita buat baru kelas baru bernama **Flyable** yang akan bertindak sebagai *interface*.

```
1. class Flyable {  
2.   void fly() { }  
3. }
```

Kita dapat membiarkan *body* dari *method fly()* tetap kosong karena fungsi implementasinya akan dilakukan oleh **class**. Selanjutnya buat kelas baru yang mengimplementasi *interface Flyable*.

```
1. class Bird extends Animal implements Flyable {  
2.   String featherColor;  
3.  
4.   Bird(String name, int age, double weight, this.featherColor) : super(name, age, weight);  
5.  
6. }
```

Anda akan mendapati eror yang memberikan pesan “*Missing concrete implementation of Flyable.fly*”. Ini artinya kita harus mengimplementasi fungsi **fly** yang terdapat pada *interface Flyable*.

```
1. class Bird extends Animal implements Flyable {  
2.   String featherColor;  
3.  
4.   Bird(String name, int age, double weight, this.featherColor) : super(name, age, weight);  
5.  
6.   @override  
7.   void fly() {  
8.     print('$name is flying');  
9.   }  
10.  
11. }
```

**Keyword** atau anotasi **@override** menunjukkan fungsi tersebut mengesampingkan fungsi yang ada di *interface* atau kelas induknya, lalu menggunakan fungsi yang dalam kelas itu sendiri sebagai gantinya.

## Enumerated Types

Bagaimana kita bisa menyimpan banyak nilai konstan di satu tempat dan menanganinya secara bersamaan? Solusinya, Dart menyediakan **Enums**. *Enum* mewakili kumpulan konstan yang membuat kode kita lebih jelas dan mudah dibaca.

```
1. enum Rainbow {  
2.   red, orange, yellow, green, blue, indigo, violet  
3. }  
4.  
5. enum Status {
```

```
6.   Todo, In_Progress, In_Review, Done
7. }
```

Enum pada Dart memiliki beberapa *property* bawaan yang dapat kita gunakan untuk menampilkan seluruh nilai dalam bentuk list serta menampilkan item dan indeks dari item tersebut.

```
1. print(Rainbow.values);
2. print(Rainbow.blue);
3. print(Rainbow.orange.index);
```

Ketika kode di atas dijalankan, maka konsol akan tampil seperti berikut:

```
1. [Rainbow.red, Rainbow.orange, Rainbow.yellow, Rainbow.green, Rainbow.blue, Rainbow.indigo,
   Rainbow.violet]
2. Rainbow.blue
3. 1
```

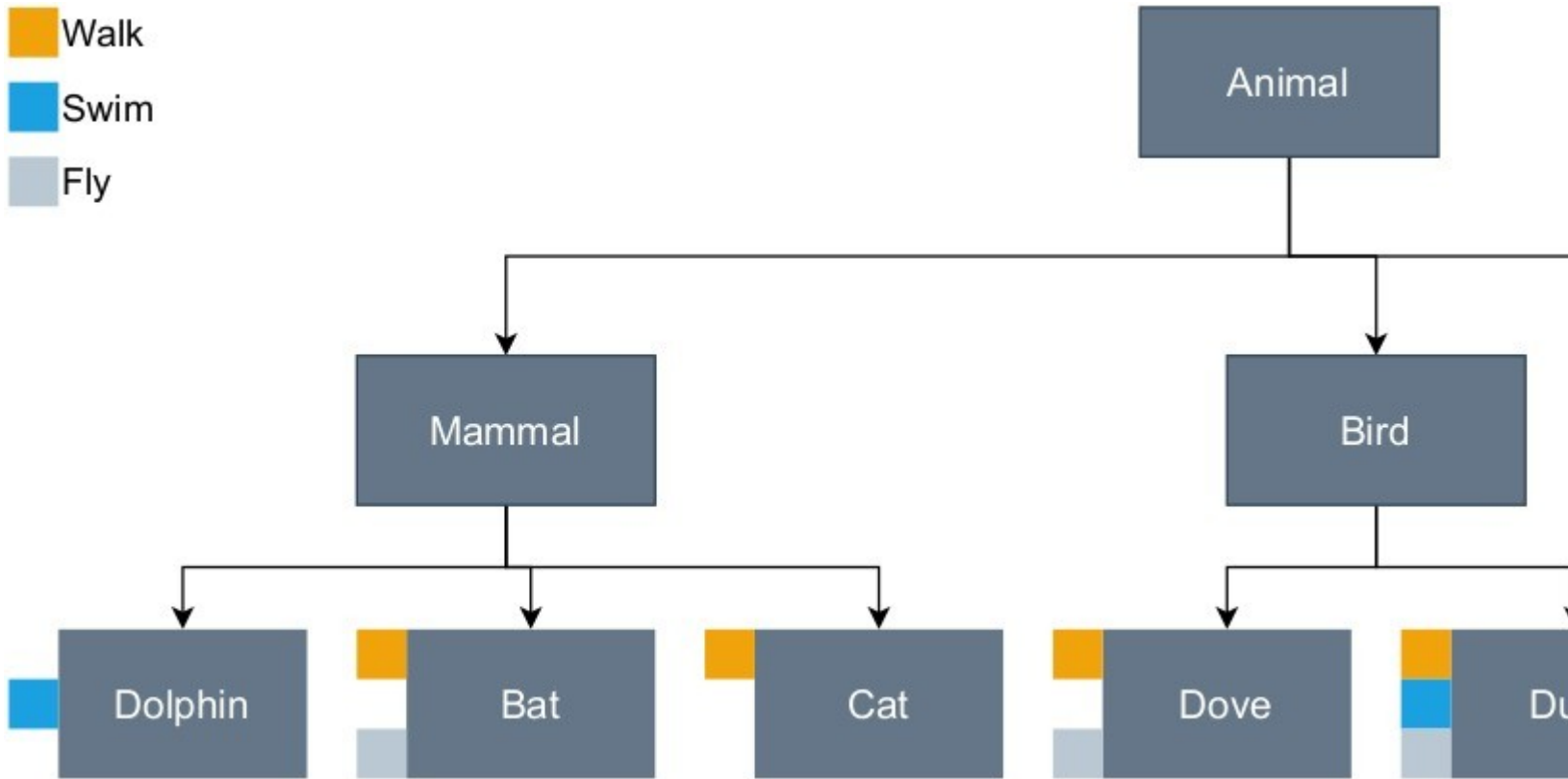
Kita juga bisa menggunakan `enum` ke dalam `switch` statements. Namun kita perlu menangani semua kemungkinan nilai *enum* yang Ada.

```
1. var taskStatus = Status.In_Progress;
2.
3. switch(taskStatus) {
4.   case Status.TODO:
5.     print('Task is still in To do');
6.     break;
7.   case Status.In_Progress:
8.     print('Task is in progress');
9.     break;
10.  case Status.In_Review:
11.    print('Task is currently under review');
12.    break;
13.  case Status.Done:
14.    print('Task is done');
15.    break;
16. }
```

## Mixins

**Mixin** adalah cara menggunakan kembali kode kelas dalam banyak hirarki kelas. Konsep *mixin* mungkin adalah konsep yang baru bagi Anda karena konsep ini tidak ada pada bahasa C# atau Java. Jadi kenapa dan kapan kita perlu menggunakan *mixin*?

Kita kembali pada contoh hewan. Perhatikan diagram berikut:



Kita memiliki *superclass* **Animal** dengan tiga *subclass*. Di bawahnya ada beberapa kelas turunan yang memiliki perilaku berbeda-beda. Beberapa hewan memiliki perilaku yang sama, seperti Cat dan Duck sama-sama bisa berjalan. Kita bisa saja membuat kelas seperti **Walkable**, **Swimmable**, dan **Flyable**. Sayangnya, Dart tidak mendukung *multiple inheritance*, sehingga sebuah kelas hanya bisa mewarisi (*inherit*) satu kelas induk. Kita bisa saja membuat *interface* lalu mengimplementasikannya ke ke kelas **Cat** atau **Duck**. Namun, implementasi *interfacemengharuskan* kita untuk meng-*override method* dan membuat implementasi fungsi di masing-masing kelas.

```
1. mixin Dapat Terbang {Flyable {
2.   void fly() {
3.     print("I'm flying");
4.   }
5. }
6.
7. mixin Walkable {
8.   void walk() {
9.     print("I'm walking");
10.  }
11. }
12.
13. mixin Swimmable {
14.   void swim() {
15.     print("I'm swimming");
16.   }
17. }
```



Kelas *mixin* dapat didefinisikan dengan keyword `class` seperti kelas pada umumnya. Jika Anda tidak ingin kelasnya bertindak seperti kelas biasa misalnya seperti bisa diinstansiasi menjadi objek, gunakan saja keyword `mixin`. Setelah membuat kelas seperti di atas kita bisa menambahkan sebagai *mixin* dengan keyword `with` dan diikuti dengan satu atau beberapa kelas *mixin*.

```
1. class Cat extends Mammal with Walkable { }
2.
3. class Duck extends Bird with Walkable, Flyable, Swimmable { }
```

Dengan *mixin* ini memungkinkan objek `cat` untuk memanggil metode `walk()`. Sementara objek `duck` bisa memanggil metode `walk()`, `fly()`, dan `swim()`.

```
1. void main() {
2.     var donald = Duck();
3.     var garfield = Cat();
4.
5.     garfield.walk();
6.
7.     donald.walk();
8.     donald.swim();
9. }
```

Jika diperhatikan *mixin* ini memang mirip dengan *multiple inheritance*. Namun kelas *mixin* ini tidak termasuk ke dalam hirarki *parent-child* atau *inheritance*. Oleh sebab itu *mixin* memungkinkan kita terhindar dari masalah yang sering terjadi pada *multiple inheritance* yang dikenal dengan [diamond problem](#), yaitu ada dua *parent class* yang memiliki *method* dengan nama yang sama sehingga *child class*-nya ambigu dalam menjalankan *method* yang mana.

Sebagai contoh kita punya kelas bernama `Performer`.

```
1. abstract class Performer {
2.     void perform();
3. }
```

Lalu kita punya dua kelas turunan dari `Performer`.

```
1. class Dancer extends Performer {
2.     @override
3.     void perform() {
4.         print('Dancing');
5.     }
6. }
7.
8. class Singer extends Performer {
9.     @override
10.    void perform() {
11.        print('Singing');
12.    }
13. }
```

Kita asumsikan Dart memiliki dukungan terhadap *multiple inheritance* sehingga kita punya 1 kelas lagi seperti berikut:

```
1. class Musician extends Dancer, Singer {  
2.   void showTime() {  
3.     perform();  
4.   }  
5. }
```

Kira-kira *method* `perform()` mana yang akan dijalankan? Beruntung dengan Dart kita bisa menghindari situasi seperti ini dengan *mixin*.

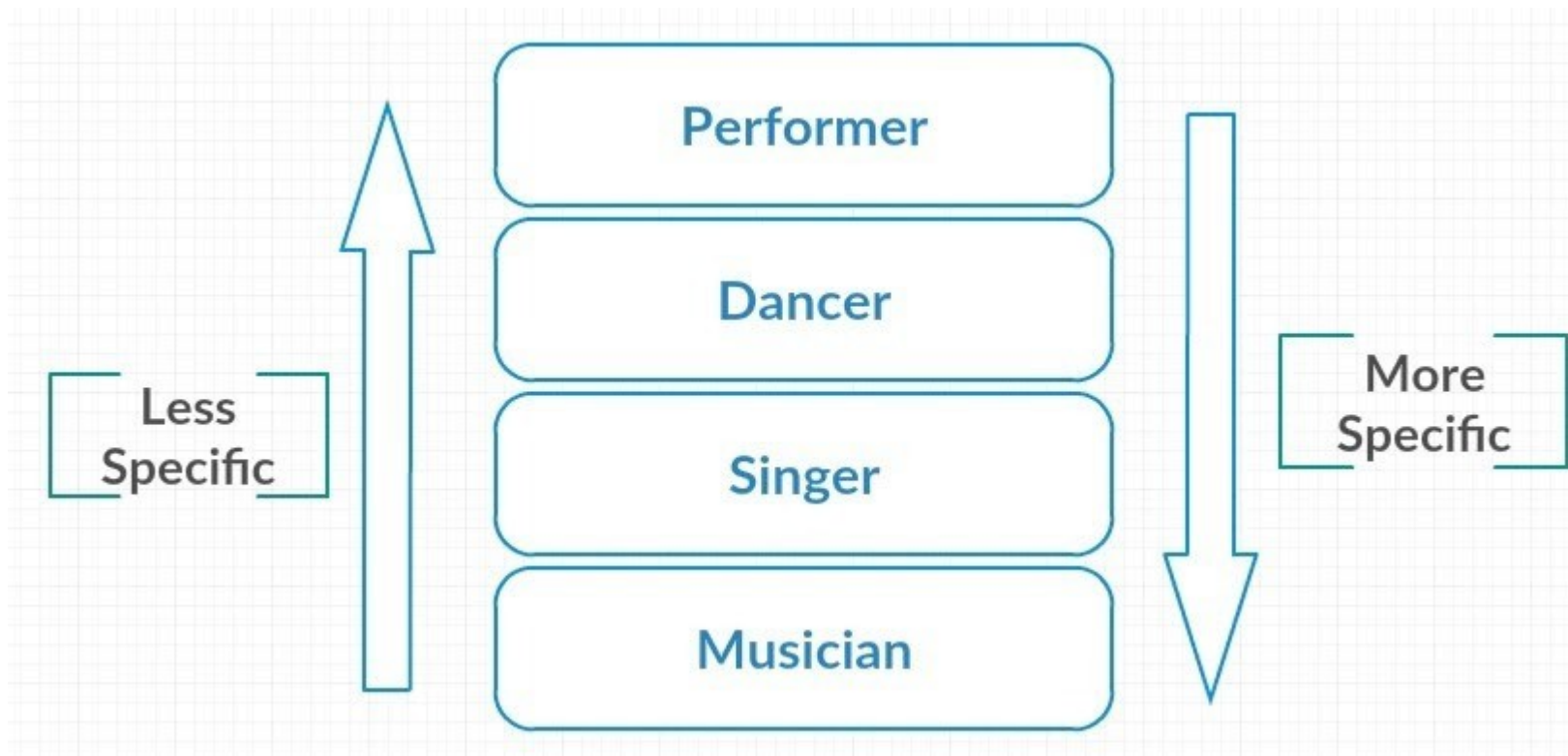
Ketika mencampur (*mixing*) kelas, kelas yang digunakan sebagai *mixin* tidak paralel namun saling bertumpuk. Itulah mengapa kelas atau method pada *mixin* tidak ambigu satu sama lain. Karena itu jugalah, urutan menjadi hal yang penting dalam menerapkan *mixin*. Misalnya kita telah menerapkan *mixin* pada kelas `Musician`.

```
1. mixin Dancer implements Performer {  
2.   @override  
3.   void perform() {  
4.     print('Dancing');  
5.   }  
6. }  
7.  
8. mixin Singer implements Performer {  
9.   @override  
10.  void perform() {  
11.    print('Singing');  
12.  }  
13. }  
14.  
15. class Musician extends Performer with Dancer, Singer {  
16.   void showTime() {  
17.     perform();  
18.   }  
19. }
```

Lalu buatlah objek yang akan menjalankan *method* `perform()`.

```
1. void main() {  
2.   var arielNoah = Musician();  
3.   arielNoah.perform();  
4. }
```

Coba jalankan fungsi main di atas, apakah yang akan tampil pada konsol? Mengapa demikian? Seperti yang telah dijelaskan, kelas *mixin* bersifat *stack* atau bertumpuk. Kelas-kelas ini berurutan dari yang paling umum hingga paling spesifik. Sehingga sesuai urutan *mixin* di atas kelas `Musician` akan menampilkan *method* dari `Singer` karena berada di urutan terakhir atau paling spesifik.



## Extension Methods

Pada versi 2.7 Dart mengenalkan fitur baru yaitu **extension methods**. Tujuan dari fitur ini adalah supaya kita bisa membuat fungsionalitas tambahan dari *library* yang sudah ada.

Ketika Anda menggunakan *library*, baik itu *library* bawaan Dart atau pun *library* milik orang lain, ada kemungkinan *library* tersebut kurang lengkap sehingga kita perlu menambahkan beberapa fungsionalitas. Namun akan jadi PR kita untuk mengubah *library* yang sudah ada. Dengan *extension method*, kita dapat membuat fungsi atau *method* tambahan lalu menggunakannya sesuai dengan kebutuhan aplikasi kita.

Contoh sederhananya, kita memiliki variabel *list integer*.

```
1. var unsortedNumbers = [2, 5, 3, 1, 4];
```

Kita memiliki kebutuhan untuk mengurutkan nilai di dalam *list* tersebut namun **List** pada Dart tidak memiliki fungsi untuk mengurutkannya (Dart memiliki fungsi **sort()** namun bersifat mentransformasi *list* dan tidak mengembalikan nilai). Kita bisa membuat *extension method* dari objek **List** dengan sintaks seperti berikut:

```
1. /* extension <extension name> on <type> {  
2.   (<member definition>)*  
3. } */
```

```

4.
5. extension Sorting on List<int> {
6.     List<int> sortAsc() {
7.         var list = this;
8.         var length = this.length;
9.
10.
11.         for (int i = 0; i < length - 1; i++) {
12.             int min = i;
13.             for (int j = i + 1; j < length; j++) {
14.                 if (list[j] < list[min]) {
15.                     min = j;
16.                 }
17.             }
18.
19.
20.             int tmp = list[min];
21.             list[min] = list[i];
22.             list[i] = tmp;
23.         }
24.
25.
26.         return list;
27.     }
28. }

```

Pada *extension method* di atas kita telah membuat *method* atau fungsi untuk mengurutkan data di dalam *list* menggunakan *selection sort algorithm*. Selanjutnya kita bisa memanggil *method* ini dari objek *list*.

```

1. void main() {
2.     var unsortedNumbers = [2, 5, 3, 1, 4];
3.     print(unsortedNumbers);
4.     var sortedNumbers = unsortedNumbers.sortAsc();
5.     print(sortedNumbers);
6.
7.
8.     /*
9.     Output: [2, 5, 3, 1, 4]
10.            [1, 2, 3, 4, 5]
11.     */
12. }

```

Kita juga bisa menggunakan kembali *extension method* ini di beberapa berkas yang berbeda sebagai *library*.

- [main.dart](#)

```

1. import 'extension.dart';
2.
3. void main() {
4.     var unsortedNumbers = [2, 5, 3, 1, 4];
5.     print(unsortedNumbers);

```

```
6.   var sortedNumbers = unsortedNumbers.sortAsc();
7.   print(sortedNumbers);
8. }
```

- [extensions.dart](#)

```
1. extension Sorting on List<int> {
2.   List<int> sortAsc() {
3.     var list = this;
4.     var length = this.length;
5.
6.     for (int i = 0; i < length - 1; i++) {
7.       int min = i;
8.       for (int j = i + 1; j < length; j++) {
9.         if (list[j] < list[min]) {
10.          min = j;
11.        }
12.      }
13.
14.      int tmp = list[min];
15.      list[min] = list[i];
16.      list[i] = tmp;
17.    }
18.
19.    return list;
20.  }
21. }
22.
23. extension NumberParsing on String {
24.   int parseInt() {
25.     return int.parse(this);
26.   }
27. }
```

## Functional Styles

Seperti yang sudah disampaikan di awal, Dart adalah bahasa yang mendukung *multiparadigm*. Artinya selain merupakan bahasa pemrograman berorientasi objek, penulisan *syntax* Dart juga menggunakan gaya *functional programming*.

*Functional programming* adalah paradigma pemrograman di mana proses komputasi didasarkan pada fungsi matematika murni. *Functional programming* (FP) ditulis dengan gaya deklaratif yang berfokus pada “*what to solve*” dibandingkan “*how to solve*” pada gaya imperatif.

Berikut ini beberapa konsep dan karakteristik *functional programming*:

## Pure functions

*Pure functions* berarti sebuah fungsi bergantung pada argumen atau parameter yang dimasukkan ke dalamnya. Sehingga pemanggilan fungsi dengan nilai argumen yang sama akan selalu memberikan hasil yang sama pula. Contohnya pada fungsi `sum()` berikut nilai yang dikembalikan akan bergantung pada argumen yang diberikan.

```
1. int sum(int num1, int num2) {  
2.     return num1 + num2;  
3. }
```

## Recursion

Pada *functional programming* tidak ada konsep perulangan seperti `for` atau `while`. Iterasi pada *functional programming* dilakukan melalui rekursi atau pemanggilan fungsi dari fungsi itu sendiri, hingga mencapai kasus dasar.

```
1. int fibonacci(n) {  
2.     if (n <= 0) {  
3.         return 0;  
4.     } else if (n == 1) {  
5.         return 1;  
6.     } else {  
7.         return fibonacci(n - 1) + fibonacci(n - 2);  
8.     }  
9. }
```

## Immutable variables

Variabel pada *functional programming* bersifat *immutable*, artinya kita tidak bisa mengubah sebuah variabel ketika sudah diinisialisasi. Alih-alih mengubah nilai variabel, kita bisa membuat variabel baru untuk menyimpan data. Mekanisme ini bertujuan agar kode kita menjadi lebih aman karena *stated* dari aplikasi tidak akan berubah sepanjang aplikasi berjalan.

```
1. var x = 5;  
2. x = x + 1; // Contoh variable yang tidak immutable
```

## Functions are first-class citizen and can be higher-order

Maksud dari *function* merupakan *first-class citizen* adalah bahwa *function* berlaku sama seperti komponen pemrograman yang lain. Sebuah fungsi bisa dimasukkan ke variabel menjadi parameter dalam suatu fungsi dan juga menjadi nilai kembalian pada fungsi. *Higher order functions* adalah fungsi yang mengambil fungsi lain sebagai argumen dan juga dapat mengembalikan fungsi.

Pada modul ini kita akan mempelajari bagaimana penulisan gaya *functional* dengan bahasa Dart.

# Anonymous Functions

Masih ingatkah Anda dengan materi *function* dan cara membuatnya? Seperti yang kita tahu, untuk mendeklarasikan sebuah fungsi kita perlu mendefinisikan nilai kembalian dan juga nama fungsinya.

```
1. int sum(int num1, int num2) {  
2.     return num1 + num2;  
3. }
```

Kebanyakan fungsi pada Dart memiliki nama seperti `sum()`, `main()`, atau `print()`. Pada Dart kita bisa membuat fungsi yang tidak bernama alias *nameless* atau *anonymous*. *Anonymous function* ini juga dikenal dengan nama **lambda**.

Untuk membuat *lambda* atau *anonymous function* kita cukup menuliskan tanda kurung untuk menerima parameter dan *body function*-nya.

```
1. void main() {  
2.     (int num1, int num2) {  
3.         return num1 + num2;  
4.     };  
5. }
```

Lalu bagaimana kita bisa menggunakan fungsi tersebut? Seperti yang telah dijelaskan sebelumnya bahwa *function* adalah *first-class citizen*, maka fungsi juga merupakan sebuah objek yang bisa disimpan ke dalam variabel. Kita bisa menggunakan keyword `var` atau secara eksplisit menggunakan tipe data `Function`.

```
1. void main() {  
2.     var sum = (int num1, int num2) {  
3.         return num1 + num2;  
4.     };  
5.  
6.     Function printLambda = () {  
7.         print('This is lambda function');  
8.     };  
9. }
```

Untuk memanggilnya kita bisa langsung memanggil nama variabelnya seperti berikut:

```
1. printLambda();  
2. print(sum(3, 4));
```

Selain itu *lambda* juga mendukung *function expression* untuk membuat kode fungsi menjadi lebih ringkas dengan memanfaatkan *fat arrow* (`=>`).

```
1. var sum = (int num1, int num2) => num1 + num2;  
2. Function printLambda = () => print('This is lambda function');
```

## Higher-Order Functions

Setelah mempelajari modul sebelumnya, Anda mungkin bertanya apa yang bisa dilakukan dengan *lambda* atau *anonymous function*?

Kita bisa memanfaatkan *lambda* untuk membuat *higher-order function*. *Higher order function* adalah fungsi yang menggunakan fungsi lainnya sebagai parameter, menjadi tipe kembalian, atau keduanya.

Coba perhatikan fungsi berikut:

```
1. void myHigherOrderFunction(String message, Function myFunction) {  
2.     print(message);  
3.     print(myFunction(3, 4));  
4. }
```

Fungsi di atas merupakan *higher order function* karena menerima parameter berupa fungsi lain. Untuk memanggil fungsi di atas, kita bisa langsung memasukkan *lambda* sebagai parameter maupun variabel yang berisi nilai berupa fungsi.

```
1. // Opsi 1  
2. Function sum = (int num1, int num2) => num1 + num2;  
3. myHigherOrderFunction('Hello', sum);  
4.  
5.  
6. // Opsi 2  
7. myHigherOrderFunction('Hello', (num1, num2) => num1 + num2);
```

Jika disimulasikan fungsi `myHigherOrderFunction` akan memanggil fungsi `sum` yang dijadikan parameter.

```
1. void myHigherOrderFunction(String message, Function myFunction) {  
2.     print(message);  
3.     print(myFunction(3, 4)); // sum(3, 4)    // return 3 + 4  
4. }
```

Namun deklarasi *higher order function* ini bisa menjadi sedikit *tricky*. Misalnya kode di bawah ini tidak akan terdeteksi eror namun ketika dijalankan, aplikasi Anda akan mengalami *crash*. Tahukah kenapa?

```
1. void myHigherOrderFunction(String message, Function myFunction) {  
2.     print(message);  
3.     print(myFunction(4));  
4. }
```

Karena kita tidak menentukan spesifikasi dari fungsi seperti jumlah parameter atau nilai kembaliannya, maka semua jenis fungsi akan bisa dijalankan termasuk pemanggilan `myFunction` seperti di atas. Untuk mengatasinya kita bisa lebih spesifik menentukan seperti apa fungsi yang *valid* untuk menjadi parameter.

```
1. void myHigherOrderFunction(String message, int Function(int num1, int num2) myFunction) { }
```

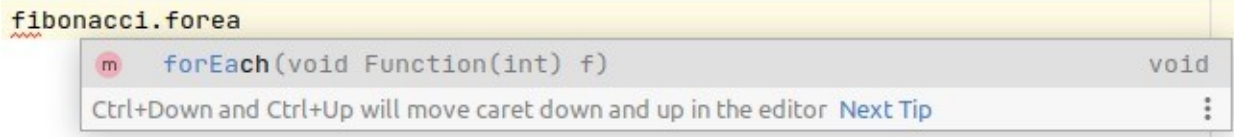
Pada fungsi di atas kita perlu memasukkan fungsi dengan dua parameter dan nilai kembali berupa `int` sebagai parameter.



Pada materi *collection* sebenarnya kita telah menggunakan satu fungsi yang merupakan *higher order function* yaitu fungsi `forEach()`. Sebagai contoh kita punya daftar bilangan *fibonacci* yang disimpan ke sebuah variabel.

```
1. var fibonacci = [0, 1, 1, 2, 3, 5, 8, 13];
```

IntelliJ IDEA akan menunjukkan *suggestion* apa saja yang perlu menjadi parameter. Kita bisa melihat bahwa `forEach` membutuhkan satu parameter berupa fungsi.



Sehingga ketika memanggil fungsi ini kita bisa melakukan operasi pada masing-masing item misalnya mencetak ke konsol.

```
1. fibonacci.forEach((item) {  
2.     print(item);  
3. });
```

## Closures

Suatu fungsi dapat dibuat dalam lingkup global atau di dalam fungsi lain. Suatu fungsi yang dapat mengakses variabel di dalam *lexical scope*-nya disebut dengan ***closure***. *Lexical scope* berarti bahwa pada sebuah fungsi bersarang (*nested functions*), fungsi yang berada di dalam memiliki akses ke variabel di lingkup induknya.

Berikut ini adalah contoh kode implementasi *closure*:

```
1. void main() {  
2.     var closureExample = calculate(2);  
3.     closureExample();  
4.     closureExample();  
5. }  
6.  
7. Function calculate(base) {  
8.     var count = 1;  
9.  
10.    return () => print("Value is ${base + count++}");  
11. }
```

Ketika kode di atas dijalankan, konsol akan tampil seperti berikut:

```
1. Value is 3  
2. Value is 4
```

Di dalam fungsi `calculate()` terdapat variabel `count` dan mengembalikan nilai berupa fungsi. Fungsi *lambda* di dalamnya memiliki akses ke variabel `count` karena berada pada lingkup yang sama. Karena variabel `count` berada pada *scope* `calculate`, maka umumnya variabel tersebut akan hilang atau dihapus ketika fungsinya selesai dijalankan. Namun pada kasus di atas fungsi *lambda* atau `closureExample` masih memiliki referensi atau akses ke variabel `count` sehingga bisa diubah. Variabel pada mekanisme di atas telah tertutup (*close covered*), yang berarti variabel tersebut berada di dalam *closure*.

## Dart Type System

Dalam bahasa pemrograman, *type system* adalah sistem logis yang terdiri dari seperangkat aturan yang menetapkan properti atau tipe ke berbagai konstruksi program komputer, seperti variabel, *expression*, fungsi, atau modul. *Type system* ini memformalkan atau memberikan standar kategori tersirat yang digunakan *programmer* untuk tipe data, struktur data, atau komponen lainnya.

Dart menyebut *type system*-nya sebagai ***sound type system***. *Soundness* ini berarti program Anda tidak akan pernah bisa memasuki keadaan di mana sebuah ekspresi mengevaluasi nilai yang tidak cocok dengan jenis tipenya.

*Sound type system* pada Dart ini sama dengan *type system* pada Java atau C#. Di mana kondisi *soundness* ini dicapai dengan menggunakan kombinasi pemeriksaan statis (*compile-time error*) dan pemeriksaan saat *runtime*. Sebagai contoh, menetapkan `String` ke variabel `int` adalah kesalahan *compile-time*. Casting `Object` ke `String` dengan `as String` akan gagal ketika *runtime* jika objek tersebut bukan `String`.

Manfaat dari *sound type system* ini, antara lain:

- **Mengungkap *bug* terkait tipe pada saat *compile time*.**  
*Sound type system* memaksa kode untuk tidak ambigu tentang tipenya, sehingga *bug* terkait tipe yang mungkin sulit ditemukan saat *runtime*, bisa ditemukan pada waktu kompilasi.
- **Kode lebih mudah dibaca.**  
Kode menjadi lebih mudah dibaca karena Anda dapat mengandalkan nilai yang benar-benar memiliki tipe yang ditentukan. Tipe pada Dart tidak bisa berbohong.
- **Kode lebih mudah dikelola.**  
Ketika Anda mengubah satu bagian kode, *type system* dapat memperingatkan Anda tentang bagian kode mana yang baru saja rusak.
- **Kompilasi *ahead of time* (AOT) yang lebih baik.**  
Kode yang dihasilkan saat kompilasi AOT menjadi jauh lebih efisien.

## Generic

Jika Anda perhatikan pada dokumentasi *collection* seperti `List`, sebenarnya tipe dari *List* tersebut adalah `List<E>`. Tanda `<...>` ini menunjukkan bahwa *List* adalah tipe *generic*, tipe yang memiliki tipe

parameter. Menurut *coding convention* dari Dart, tipe parameter dilambangkan dengan satu huruf kapital seperti **E**, **T**, **K**, atau **V**.

Secara umum *generic* merupakan konsep yang digunakan untuk menentukan tipe data yang akan kita gunakan. Kita bisa mengganti tipe parameter *generic* pada Dart dengan tipe yang lebih spesifik dengan menentukan *instance* dari tipe tersebut.

Sebagai contoh, perhatikan **List** yang menyimpan beberapa nilai berikut:

```
1. List<int> numberList = [1, 2, 3, 4, 5];
```

Tipe parameter yang digunakan pada variabel list di atas adalah **int**, maka nilai yang bisa kita masukkan adalah nilai dengan tipe **int**. Begitu juga jika kita menentukan tipe parameter **String**, maka tipe yang bisa kita masukkan ke dalam *list* hanya berupa **String**.

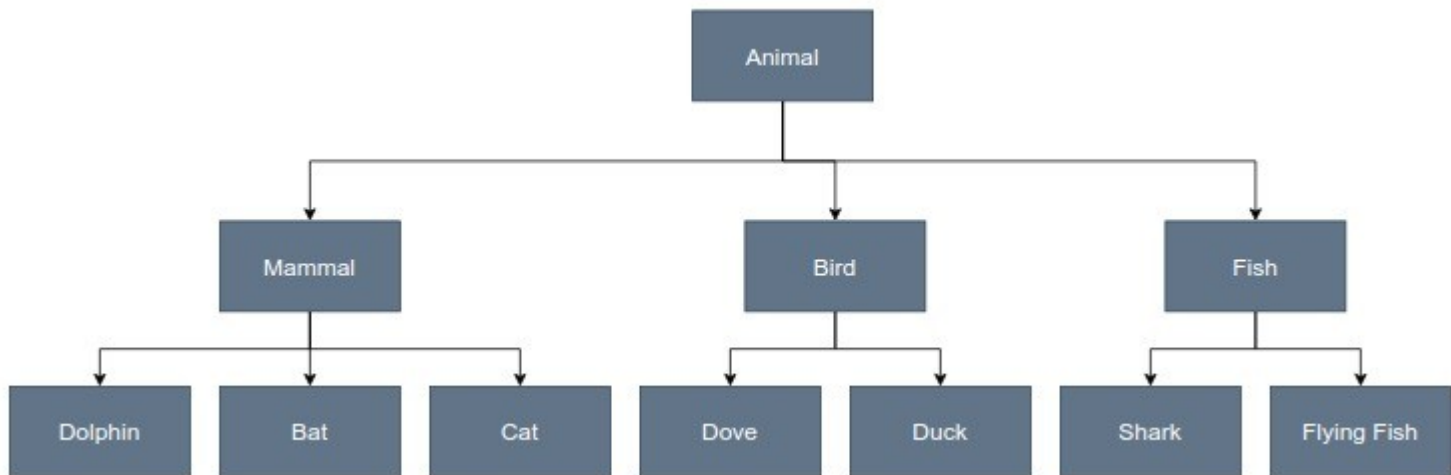
```
1. List<int> numberList = [1, 2, 3, 4, 5];
2. List<String> stringList = ['Dart', 'Flutter', 'Android', 'iOS'];
3. List dynamicList = [1, 2, 3, 'empat']; // List<dynamic>
```

Berbeda jika kita tidak menentukan tipe parameter dari *list*. *List* tersebut tidak memiliki tipe yang menjadi acuan bagi kompiler sehingga semua tipe bisa disimpan ke dalam list. Variabel **dynamicList** di atas sebenarnya masih menerapkan *generic* dengan tipe **dynamic** sehingga tipenya menjadi **List<dynamic>**.

Dari kasus di atas kita bisa simpulkan bahwa Dart membantu kita menghasilkan kode yang *type safe* dengan membatasi tipe yang bisa digunakan ke dalam suatu objek dan menghindari *bug*. Selain itu *generic* juga bermanfaat mengurangi duplikasi kode. Misalnya ketika Anda perlu untuk menyimpan objek **cache** bertipe **String** dan **int**. Alih-alih membuat dua objek **StringCache** dan **IntCache**, Anda bisa membuat satu objek saja dengan memanfaatkan tipe parameter dari *generic*.

```
1. abstract class Cache<T> {
2.   T getByKey(String key);
3.   void setByKey(String key, T value);
4. }
```

Dengan Dart *type system* kita bisa mengganti tipe parameter yang digunakan sesuai dengan susunan hierarkinya. Perhatikan hierarki objek Animal berikut:



Dengan hierarki di atas, jika kita memiliki objek `List<Bird>` maka objek apa saja yang bisa kita masukkan ke *list* tersebut?

```
1. List<Bird> birdList = [Bird(), Dove(), Duck()];
```

Seluruh objek `Bird` atau objek turunannya bisa masuk ke dalam `birdList`. Namun, ketika menambahkan objek dari `Animal`, terjadi *compile error* karena objek `Animal` belum tentu merupakan objek `Bird`.

```
1. List<Bird> birdList = [Bird(), Dove(), Duck(), Animal()]; // Error
```

Berbeda jika kita mengisi `List<Bird>` dengan `List<Animal>` seperti berikut:

```
1. List<Bird> myBird = List<Animal>();
```

Kompiler tidak akan menunjukkan eror namun ketika kode dijalankan akan terjadi *runtime error* karena `List<Animal>` bukanlah subtype dari `List<Bird>`.

```
1. Unhandled exception:
2. type 'List<Animal>' is not a subtype of type 'List<Bird>'
```

## Type Inference

Seperti yang kita tahu Dart mendukung *type inference*. Dart memiliki *analyzer* yang dapat menentukan menyimpulkan tipe untuk *field*, *method*, variabel lokal, dan beberapa tipe argumen *generic*. Ketika *analyzer* tidak memiliki informasi yang cukup untuk menyimpulkan tipe tertentu, maka tipe `dynamic` akan digunakan.

Misalnya berikut ini adalah contoh penulisan variabel *map* dengan tipe yang eksplisit:

```
1. Map<String, dynamic> company = {'name': 'Dicoding', 'yearsFounded': 2015};
```

Atau, Anda dapat menggunakan `var` dan Dart akan menentukan tipenya.

```
1. var company = {'name': 'Dicoding', 'yearsFounded': 2015}; // Map<String, Object>
```

Type inference menentukan tipe dari entri kemudian menentukan tipe dari variabelnya. Pada contoh di atas, kedua *key* dari *map* adalah `String`, namun nilainya memiliki tipe yang berbeda, yaitu `String` dan `int`, di mana keduanya merupakan turunan dari `Object`. Sehingga variabel `company` akan memiliki tipe `Map<String, Object>`.

Saat menetapkan nilai objek ke dalam objek lain, kita bisa mengganti tipenya dengan tipe yang berbeda tergantung pada apakah objek tersebut adalah *consumer* atau *producer*. Perhatikan *assignment* berikut:

```
1. Fish fish = Fish();
```

`Fish fish` adalah *consumer* dan `Fish()` adalah *producer*. Pada posisi *consumer*, aman untuk mengganti *consumer* bertipe yang spesifik dengan tipe yang lebih umum. Jadi, aman untuk mengganti `Fish fish` dengan `Animal fish` karena `Animal` adalah *supertype* dari `Fish`.

```
1. Animal fish = Fish();
```

Namun mengganti `Fish fish` dengan `Shark fish` melanggar *type safety* karena bisa saja `Fish` memiliki *subtype* lain dengan perilaku berbeda, misalnya `FlyingFish`.

```
1. Shark fish = Fish(); // Error
```

Pada posisi *producer*, aman untuk mengganti tipe yang umum (*supertype*) dengan tipe yang lebih spesifik (*subtype*).

```
1. Fish fish = Shark();
```

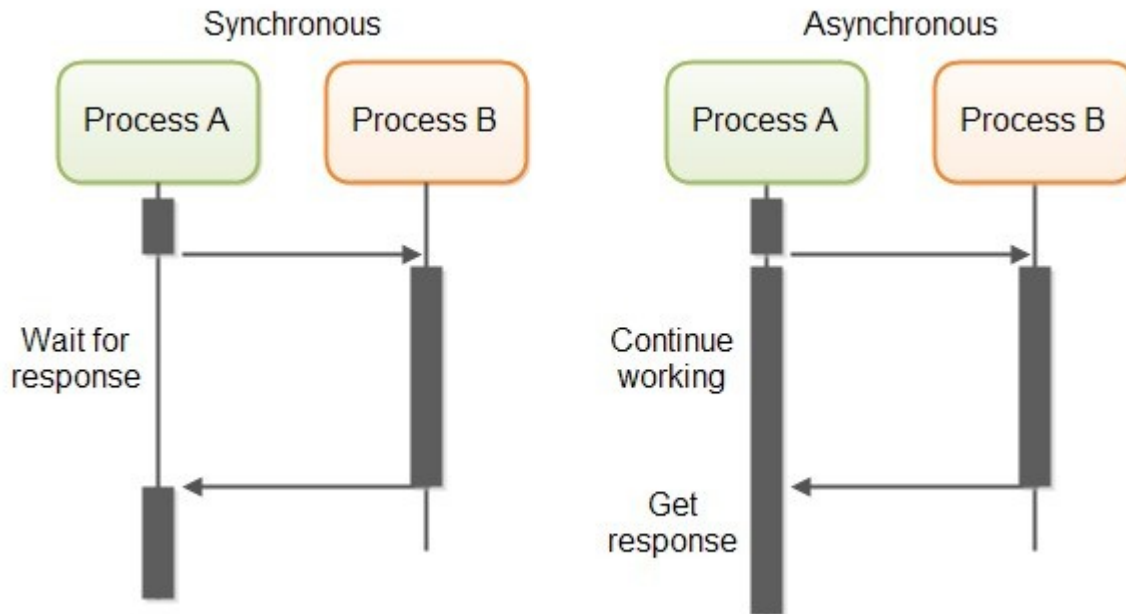
## Dart Futures

Sejauh ini kita telah menuliskan kode secara *synchronous*. Lebih lanjut, pada modul ini kita akan mempelajari kode yang bekerja secara *asynchronous*. Sebelum membahas *asynchronous* lebih dalam, kita akan bahas dahulu apa perbedaan *synchronous* dan *asynchronous*.

### Synchronous vs Asynchronous

Dalam *synchronous program*, kode program dijalankan secara berurutan dari atas ke bawah. Artinya jika kita menuliskan dua baris kode maka baris kode kedua tidak bisa dieksekusi sebelum kode baris pertama selesai dieksekusi. Kita bisa bayangkan ketika sedang berada dalam antrian kasir. Kita tidak akan dilayani sebelum semua antrian di depan kita selesai dilayani, begitu pula orang di belakang kita pun harus menunggu gilirannya.

Dalam *asynchronous program*, jika kita menuliskan dua baris kode, kita dapat membuat baris kode kedua dieksekusi tanpa harus menunggu kode pada baris pertama selesai dieksekusi. Dalam dunia nyata bisa kita bayangkan dengan memesan kopi melalui pelayan, di mana sembari menunggu pesanan kita datang, kita dapat melakukan aktivitas lain seperti membuka laptop atau menulis.



Program *asynchronous* memungkinkan suatu operasi bisa berjalan sembari menunggu operasi lainnya selesai. Umumnya kita memanfaatkan *asynchronous* pada operasi yang besar dan membutuhkan waktu lama, seperti mengambil data dari internet atau API, menyimpan data ke *database*, dan membaca data dari sebuah berkas.

## Apa itu Future?

Untuk melakukan pemrograman secara *asynchronous* dengan Dart, kita menggunakan **Future**. *Future* adalah sebuah objek yang mewakili nilai potensial atau kesalahan yang akan tersedia pada suatu waktu di masa mendatang. Anda dapat membayangkan *future* sebagai sebuah kotak paket. Saat Anda menerima paket tersebut, akan ada tiga kondisi yang mungkin terjadi, antara lain paket masih tertutup (**uncompleted**), paket dibuka lalu berisi barang sesuai pesanan (**completed with data**), dan paket dibuka namun terjadi kesalahan atau tidak sesuai (**completed with error**). Penerima dari *future* dapat menentukan *callbacks* yang akan menangani nilai atau kesalahan tersebut.

# Uncompleted

```
graph TD; A[Uncompleted] --> B[Completed with data]; A --> C[Completed with error];
```

Completed  
with data

Completed  
with error

Sekarang coba perhatikan kode berikut:

```
1. void main() {  
2.   print('Creating the future');  
3.   print('main() done');  
4. }
```

Tentunya Anda sudah tahu urutan program dan apa yang akan ditampilkan pada konsol. Lalu bagaimana jika perintah **print** yang pertama kita pindahkan ke dalam objek *future*.

```
1. void main() {  
2.   final myFuture = Future(() {  
3.     print('Creating the future');  
4.     return 12;  
5.   });  
6.   print('main() done');  
7. }
```

Jika kode di atas dijalankan, seluruh fungsi **main** akan dieksekusi sebelum fungsi yang ada di dalam **Future()**. Ini disebabkan karena *future* masih berstatus *uncompleted*. Sehingga ketika program dijalankan, konsol akan tampil seperti berikut:

```
1. main() done  
2. Creating the future
```

Lantas bagaimana caranya kita membuat dan memanfaatkan *future* ini?

### Uncompleted

Mari kita buat sebuah fungsi yang mengembalikan nilai *Future*.

```
1. Future<String> getOrder() {  
2.     return Future.delayed(Duration(seconds: 3), () {  
3.         return 'Coffee Boba';  
4.     });  
5. }
```

Pada *Future* kita bisa menambahkan *method* *delayed* untuk menunda eksekusi kode di dalam *Future*. Di mana parameter pertama berisi durasi penundaan dan parameter kedua adalah blok kode atau fungsi yang akan dijalankan. Pada kasus ini kita menggunakan *delayed* untuk menunda waktu eksekusi seolah kita sedang mengambil data dari internet. Karena nilai yang dikembalikan pada *Future* berupa *String*, kita bisa secara eksplisit menentukan tipe parameter *generic* *Future<String>*.

### Completed with data

Kemudian setelah *Future* dijalankan, kita memerlukan *handler* untuk menangani status *completed with data*. Caranya gunakan *method* *.then()* dari objek *Future*.

```
1. void main() {  
2.     getOrder().then((value) {  
3.         print('You order: $value');  
4.     });  
5.     print('Getting your order...');  
6. }
```

Fungsi *getOrder()* akan dijalankan secara *asynchronous* hingga setelah 3 detik kode *Future* akan dijalankan dan mengembalikan nilai.

```
1. Getting your order...  
2. You order: Coffee Boba // Muncul setelah 3 detik
```

Bagaimana jika objek *Future* menghasilkan kondisi “*completed with error*?” Bagaimana menanganinya?

### Completed with error

Kita dapat menambahkan *method* *.catchError()* setelah *then*. Sehingga ketika terjadi eror atau *exception* di dalam *Future*, blok kode ini akan dijalankan. Mari kita ubah sedikit kode di atas untuk mendukung skenario *completed with error*.

```
1. void main() {  
2.     getOrder().then((value) {  
3.         print('You order: $value');  
4.     })  
5.     .catchError((error) {
```



```

6.     print('Sorry. $error');
7.   });
8.   print('Getting your order...');
9. }
10.
11. Future<String> getOrder() {
12.   return Future.delayed(Duration(seconds: 3), () {
13.     var isStockAvailable = false;
14.     if (isStockAvailable) {
15.       return 'Coffee Boba';
16.     } else {
17.       throw 'Our stock is not enough.';
18.     }
19.   });
20. }

```

Menurut Anda apa yang akan ditampilkan di konsol? Coba jalankan aplikasinya untuk membuktikan.

Sampai sini harusnya Anda sudah paham dengan ketiga *state* yang ada pada *Future* serta bagaimana menuliskan kode untuk menanganinya. Seperti pada fungsi `main()` ada tiga blok kode yang mewakili *state Future*:

1. Fungsi `getOrder()` yang berisi *Future* yang masih ***uncompleted***.
2. Method `then()` yang menangani kondisi ***completed with data***.
3. Method `catchError()` yang menangani kondisi ***completed with error***.

Ada satu *method* lagi yang bisa kita gunakan yaitu `whenComplete()`. Method ini akan dijalankan ketika suatu fungsi *Future* selesai dijalankan, tak peduli apakah menghasilkan nilai atau eror. Ini seperti blok ***finally*** pada *try-catch-finally*.

```

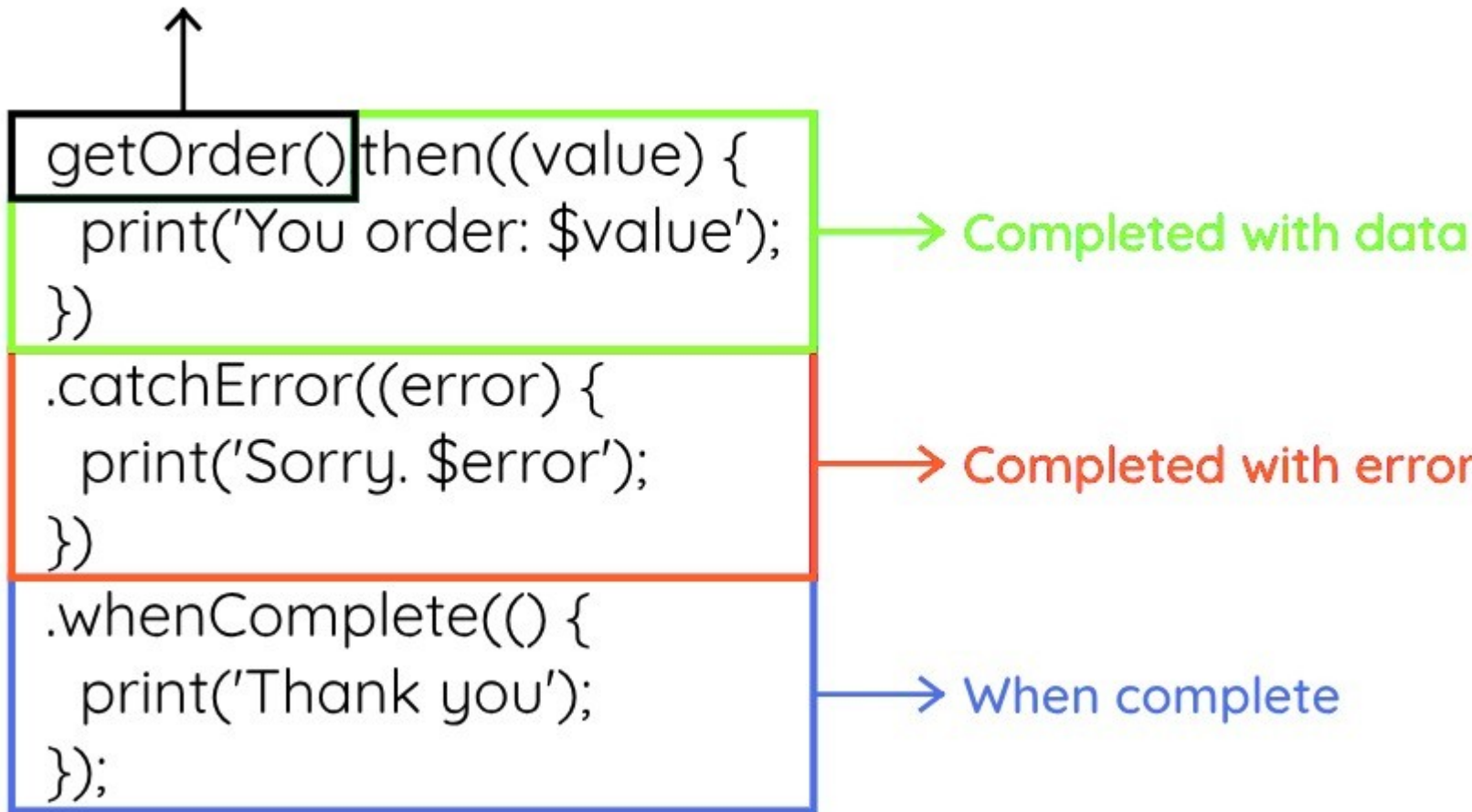
1. membatalkan main () {
2.   getOrder (). then ((value) {
3.     print ('Anda memesan: $ value');
4.   })
5.   .catchError ((kesalahan) {
6.     cetak ('Maaf. $ kesalahan');
7.   })
8.   .whenComplete (() {
9.     print ('Terima kasih');
10.  });
11. print ('Menerima pesanan Anda ...');
12. }

```

## Future with async-await

Pada materi sebelumnya kita telah mempelajari *Future* untuk berurusan dengan proses *asynchronous*. Seperti yang kita tahu, penulisan kode *asynchronous* akan sedikit berbeda dengan proses *synchronous*. Contohnya program pesan kopi kita sebelumnya jika dituliskan secara *asynchronous* akan seperti berikut:

Uncompleted



Dart memiliki keyword `async` dan `await` yang merupakan sebuah alternatif supaya kita bisa menuliskan proses *asynchronous* layaknya proses *synchronous*. Bagaimana caranya?

Dengan gaya penulisan *synchronous*, kira-kira kode program pesan kopi kita akan seperti berikut:

```
1. void main() {  
2.   print('Getting your order...');  
3.   var order = getOrder();  
4.   print('You order: $order');  
5. }
```

Namun ketika kode di atas dijalankan hasilnya tidak akan sesuai yang kita harapkan karena fungsi `getOrder()` merupakan objek `Future`.

```
1. Getting your order...  
2. You order: Instance of 'Future<String>'
```

Output ini disebabkan karena fungsi `main()` masih merupakan fungsi *synchronous*. Untuk mengubahnya menjadi fungsi *asynchronous* tambahkan keyword `async` sebelum *function body*.

```
1. void main() async { ... }
```

Kemudian tambahkan keyword `await` sebelum fungsi yang mengembalikan nilai `Future`.

```
1. var order = await getOrder();
```

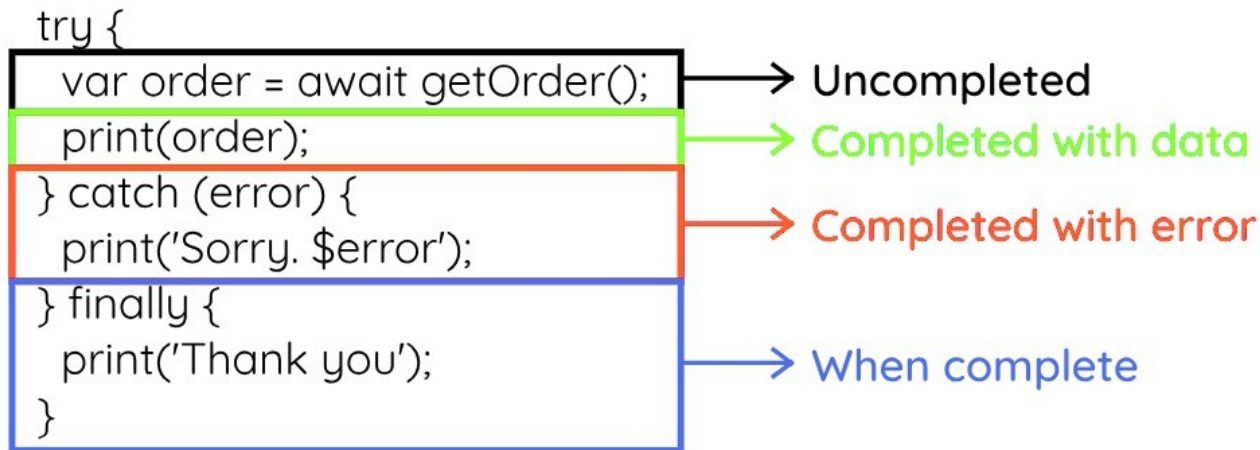
Dari perubahan di atas yang memanfaatkan *async-await* kita telah berhasil menuliskan proses *asynchronous* dengan gaya *synchronous*.

```
1. void main() async {
2.   print('Getting your order...');
3.   var order = await getOrder();
4.   print('You order: $order');
5. }
6.
7.
8. /*
9.   Output :
10.  Getting your order...
11.  You order: Coffee Boba
12. */
```

Lalu bagaimana menangani ketika terjadi eror? Caranya cukup sederhana yaitu dengan memanfaatkan *try-catch*:

```
1. void main() async {
2.   print('Getting your order...');
3.   try {
4.     var order = await getOrder();
5.     print('You order: $order');
6.   } catch (error) {
7.     print('Sorry. $error');
8.   }
9. }
```

Begitu juga seperti yang telah disebutkan, method `whenComplete()` bisa digantikan dengan blok `finally`. Sehingga keseluruhan kode akan menjadi seperti berikut:



## Effective Dart

Akhirnya kita telah sampai di modul terakhir di kelas Memulai Pemrograman dengan Dart. Pada dasarnya sebagai seorang programmer, khususnya yang bekerja dalam sebuah tim, mayoritas waktu kita akan digunakan untuk membaca kode daripada menulisnya. Untuk itulah hadir *coding convention*. *Coding convention* adalah panduan untuk bahasa pemrograman tertentu yang merekomendasikan gaya pemrograman, praktik, dan metode untuk setiap aspek program yang ditulis dalam bahasa tersebut. Konvensi ini biasanya meliputi indentasi, komentar, deklarasi, penamaan, arsitektur, dll.

*Code convention* sangat penting bagi programmer karena beberapa alasan berikut:

- 40% - 80% biaya dari sebuah perangkat lunak digunakan untuk pemeliharaan (*maintenance*).
- Sangat jarang suatu perangkat lunak dipelihara seterusnya oleh penulis aslinya.
- *Code convention* meningkatkan keterbacaan kode, memungkinkan programmer untuk memahami kode baru dengan lebih cepat dan menyeluruh.
- Source code lebih tertata rapi dan bersih sebagai sebuah produk.

Dart juga telah mengembangkan *coding convention* yang bertujuan supaya kita bisa menulis kode yang konsisten, kuat, dan cepat. Convention Dart ini dikenal dengan Effective Dart. Intinya Effective Dart dibuat untuk mewujudkan dua hal:

1. *Be consistent* (Konsisten). Ketika membahas tentang *formatting* akan banyak argumen subjektif tentang mana yang lebih baik. Namun konsistensi tentunya akan sangat membantu secara objektif.
2. *Be brief* (Ringkas). Dart dirancang supaya familiar dengan bahasa pemrograman lain seperti C, Java, JavaScript, dll. Namun Dart ditambah dengan fitur-fitur lain yang belum ditawarkan bahasa

lain. Jika ada banyak cara untuk mengungkapkan sesuatu, tentu Anda akan menggunakan cara yang paling ringkas bukan?

## Panduan Effective Dart

*Guidelines* dari Effective Dart dibagi menjadi empat bagian sesuai dengan fungsinya.

- *Style guide*. Mendefinisikan aturan untuk meletakkan dan mengatur kode. Panduan ini juga menentukan bagaimana format penamaan sebuah identifier, apakah menggunakan camelCase, `_underscore`, dll.
- *Documentation guide*. Panduan ini terkait tentang apa yang boleh dan tidak ada di dalam komentar. Baik itu komentar dokumentasi atau komentar biasa.
- *Usage guide*. Panduan ini mengajarkan bagaimana memanfaatkan fitur bahasa secara terbaik untuk menerapkan perilaku. Penggunaan statement atau expression akan dibahas di sini.
- *Design guide*. Ini adalah panduan dengan cakupan terluas namun paling tidak mengikat. Panduan ini mencakup bagaimana mendesain API library yang konsisten dan bisa digunakan.

Effective Dart memiliki banyak *rules*. Masing-masing aturan akan diawali dengan kata kunci untuk menentukan bagaimana sifat dari aturan tersebut. Lima kata kunci tersebut, antara lain:

- **DO**. Ketika aturan diawali dengan DO maka praktik tersebut harus selalu diikuti.
- **DON'T**. Sebaliknya, aturan yang diawali dengan DON'T bukan merupakan hal yang baik untuk diterapkan.
- **PREFER**. Ini adalah praktik yang harus diikuti. Namun, mungkin ada keadaan di mana lebih masuk akal untuk melakukan sebaliknya. Pastikan Anda memahami konsekuensi ketika Anda mengabaikan aturan ini.
- **AVOID**. Ini adalah kebalikan dari PREFER. Panduan ini menjelaskan hal-hal yang tidak boleh dilakukan, namun kemungkinan ada alasan bagus untuk melakukannya pada beberapa kejadian.
- **CONSIDER**. Panduan ini adalah praktik yang bisa Anda ikuti atau tidak Anda ikuti, tergantung pada keadaan dan preferensi Anda sendiri.

Meskipun ada banyak sekali aturan dan pedoman yang perlu diikuti dalam Effective Dart, kita tidak perlu khawatir dengan peraturan yang sangat ketat. Karena sebagian besar panduan yang ada bisa dibilang merupakan *common sense* dalam membuat program, bahkan jika tidak tertulis sekalipun. Selain itu konvensi bertujuan supaya kode kita menjadi lebih bagus, mudah dibaca, dan tentunya dipelihara.

## Ringkasan Aturan Effective Dart

Setelah mempelajari *code convention* dan Effective Dart, pada modul ini kita akan membahas tentang beberapa contoh aturan yang ada di dalam Effective Dart.

DO name types using UpperCamelCase.

Class, enum, typedef, dan type parameter harus menggunakan huruf kapital pada huruf pertama dari setiap kata termasuk kata pertama.

```
1. abstract class Animal {}
2. abstract class Mammal extends Animal {}
3. mixin Flyable {}
4. class Cat extends Mammal with Walkable {}
```

DON'T use prefix letters.

Karena Dart dapat memberitahu Anda tipe, cakupan, dan properti lain pada sebuah deklarasi, maka tidak ada alasan untuk menambahkan prefix pada sebuah identifier.

```
1. var instance; // good
2. var mInstance; // bad
```

PREFER starting function or method comments with third-person verbs.

Sebuah komentar dokumentasi harus fokus menjelaskan apa yang dilakukan kode tersebut.

Menambahkan kata kerja dari sudut pandang orang ketiga di awal komentar adalah salah satu cara melakukannya.

```
1. // Returns `true` if [username] and [password] inputs are valid.
2. bool isValid(String username, String password) { }
```

PREFER a noun phrase for a non-boolean property or variable.

Seorang developer yang membaca kode kita akan fokus pada *apa* yang ada pada property. Jika mereka lebih peduli tentang *bagaimana* suatu property ditentukan, maka lebih baik dibuat menjadi method dengan nama menggunakan kata kerja.

```
1. // Good
2. cat.furColor;
3. calculator.firstNumber;
4. list.length;
5.
6. // Bad
7. list.deleteItems;
```

Sementara untuk variabel atau property booleans PREFER gunakan kata kerja *non-imperative*, seperti:

```
1. list.isEmpty
2. dialog.isOpen
```

DO use ?? to convert null to a boolean value.

Aturan ini berlaku ketika sebuah expression dapat mengevaluasi nilai true, false, atau null dan Anda perlu meneruskan hasilnya ke sesuatu yang tidak menerima nilai null.

```
1. if(stock?.isEnough) {
2.   print('Making you a cup of coffee...');
3. }
```

Kode di atas akan menghasilkan exception ketika stock-nya null. Untuk mengatasinya kita perlu memberikan nilai default ketika nilai awalnya null. Sehingga kodenya akan menjadi seperti berikut:

```
1. stock?.isEnough ?? false;
```

AVOID using curly braces in interpolation when not needed.

Seperti yang kita tahu, Dart dilengkapi dengan fitur *string interpolation* untuk menggabungkan nilai string dengan nilai lain secara lebih mudah.

```
1. print('Hi, ${name}, You are ${thisYear - birthYear} years old.');
```

Namun jika Anda menginterpolasi identifier sederhana, maka curly braces ({} ) tidak perlu ditulis.

```
1. print('Hi, $name, You are ${thisYear - birthYear} years old.');
```

PREFER async/await over using raw futures.

Kode *asynchronous* bisa jadi sangat sulit untuk dibaca dan di-*debug*, bahkan ketika menggunakan abstraksi yang bagus seperti Future. *Sintaks* async-await memungkinkan Anda menuliskan kode *asynchronous* dengan gaya *synchronous* sehingga memudahkan membaca kode.

```
1. // raw future
2. void main() {
3.   getOrder().then((value) {
4.     print('You order: $value');
5.   })
6.   .catchError((error) {
7.     print('Sorry. $error');
8.   });
9.   print('Getting your order...');
```

```

10. }
11.
12.
13. // async-await
14. void main() async {
15.   print('Getting your order...');
16.   try {
17.     var order = await getOrder();
18.     print('You order: $order');
19.   } catch (error) {
20.     print('Sorry. $error');
21.   }
22. }

```

CONSIDER making the code read like a sentence.

Penamaan dalam kode baik itu nama variabel, fungsi, maupun lainnya adalah hal yang sangat penting namun juga tidak mudah. Sebagai solusi kita bisa membayangkannya seolah sedang membuat kalimat.

```

1. // "If store is open ..."
2. If (store.isOpen)
3.
4. // "hey garfield, eat!"
5. garfield.eat();
6.
7. // Bad example
8. // Ambigu apakah memerintahkan toko untuk tutup atau mendapatkan status dari toko
9. If (store.close)

```

CONSIDER using function type syntax for parameters.

Dart memiliki *sintaks* khusus untuk mendefinisikan parameter yang tipenya adalah fungsi. Caranya yaitu dengan menuliskan tipe kembalian sebelum nama parameter kemudian parameter dari fungsi setelahnya.

```

1. List filter(bool predicate(item)) { }

```

Sejak Dart versi 2, terdapat notasi umum untuk tipe fungsi sehingga bisa digunakan untuk parameter berupa fungsi.

```

1. List filter(Function predicate) { } // function type syntax

```

Sebenarnya beberapa aturan di atas hanyalah sebagian dari seluruh aturan yang ada dalam Effective Dart. Selengkapnya Anda dapat mempelajari panduan dan aturan Effective Dart ini pada tautan berikut: <https://dart.dev/guides/language/effective-dart>.



Terakhir, maksimalkan panduan dan convention yang telah dibuat ini supaya kode Anda menjadi lebih berkualitas ya. *Do your best!*