**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 siku ([ ]). 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. Pastinya 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)

**Spread Operator**

Dart memiliki fitur menarik untuk menambahkan banyak nilai ke dalam List dengan cara yang singkat, yaitu spread operator.

Sesuai namanya “spread”, fitur ini digunakan untuk menyebarkan nilai di dalam collections menjadi beberapa elemen. Spread operator dituliskan dengan tiga titik (...). Mari kita lihat contoh kode berikut:

1. var favorites = ['Seafood', 'Salad', 'Nugget', 'Soup'];
2. var others = ['Cake', 'Pie', 'Donut'];
3. var allFavorites = [favorites, others];
4. print(allFavorites);
6. /\* output
7. [[Seafood, Salad, Nugget, Soup], [Cake, Pie, Donut]]
8. \*/

Nilai List tidak akan tergabung, Alih-alih menggabungkan nilainya, variabel allFavorites menjadi List yang menampung dua List di dalamnya. Nah, dengan spread operator kita dapat menggabungkan kedua List di atas:

1. var favorites = ['Seafood', 'Salad', 'Nugget', 'Soup'];
2. var others = ['Cake', 'Pie', 'Donut'];
3. var allFavorites = [...favorites, ...others];
4. print(allFavorites);
6. /\* output
7. [Seafood, Salad, Nugget, Soup, Cake, Pie, Donut]
8. \*/

Yup, dengan spread operator kita dapat memasukkan masing-masing elemen di dalam List ke dalam List lainnya.

Untuk mengatasi List yang bisa bernilai null, kita dapat menggunakan null-aware spread operator (...?) seperti berikut:

1. var list;
2. var list2 = [0, ...?list];
3. print(list2);
5. /\* output
6. [0]
7. \*/

## 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);
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};
3. numberSet.add(6);
4. numberSet.addAll({2, 2, 3});
5. numberSet.remove(3);
7. print(numberSet.elementAt(2));
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};
4. var union = setA.union(setB);
5. var intersection = setA.intersection(setB);
7. print(union);
8. print(intersection);
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']);
3. // Output: Indonesia

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

1. var mapKeys = capital.keys;
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;
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);
4. // Output: {Jakarta: Indonesia, London: England, Tokyo: Japan, New Delhi: India}

# **Paradigma 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;
6. Animal(this.name, this.age, this.weight);
8. void eat() {
9. print('$name is eating.');
10. weight = weight + 0.2;
11. }
13. void sleep() {
14. print('$name is sleeping.');
15. }

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. 4.300000000000001

**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 = 0;
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**](https://www.dicoding.com/academies/191/tutorials/7611?from=7604#tab1-code1)
* [Animal.dart](https://www.dicoding.com/academies/191/tutorials/7611?from=7604#tab1-code2)

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

*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 = 0;
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. }
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('$\_name is eating.');
3. \_weight = \_weight + 0.2;
4. }
6. void poop() {
7. print('$\_name is pooping.');
8. \_weight = \_weight - 0.1;
9. }

Sehingga, keseluruhan kode pada berkas Animal.dart akan menjadi seperti berikut.

1. class Animal {
2. **String \_name = '';**
3. **int \_age = 0;**
4. **double \_weight = 0;**
6. Animal(this.\_name, this.\_age, this.\_weight);
8. **// Setter**
9. **set name(String value) {**
10. **\_name = value;**
11. **}**
12. **// Getter**
13. **double get weight => \_weight;**

**Constructor**

Ketika suatu objek dibuat, semua properti pada kelas tersebut harus memiliki nilai. Kita dapat langsung menginisialisasi pada properti tertentu atau menginisialisasinya 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.

Pada materi sebelumnya, secara tidak langsung Anda telah belajar membuat *constructor*yang menerima beberapa argumen. Namun, secara default sebuah *constructor*pada kelas tidak menerima argumen apapun. Sebagai contoh pada kelas Animal akan menjadi seperti berikut:

1. class Animal {
2. String name = '';
3. int age = 0;
4. double weight = 0;
5. }

Untuk membuat objek baru dari kelas tersebut, Anda tidak perlu memberikan argumen apapun.

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 eror.

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. }

## Cascade Notation

Dart juga dilengkapi dengan cascade notation atau cascade operator. Operator ini memungkinkan kita untuk melakukan beberapa urutan operasi pada objek yang sama. Kita bisa mengakses property dari object dan menjalankan method di dalamnya bersamaan ketika kita menginstansiasi object. Cascade operator dituliskan dengan dua tanda titik (.. atau ?..).

Perhatikan contoh kode yang menggunakan cascade operator berikut berikut:

1. void main() {
2. var dicodingCat = Animal('', 2, 4.2)
3. ..name = 'Gray'
4. ..eat();
5. }

Apakah Anda mengerti maksud kode di atas? Kita menginstansiasi object Animal dengan constructor seperti biasa. Kemudian cascade operator yang mengikutinya akan melakukan operasi berdasarkan object yang dikembalikan oleh constructor. Contoh kode tersebut melakukan hal yang sama apabila kita menuliskan kode seperti ini:

1. var dicodingCat = Animal('', 2, 4.2);
2. dicodingCat.name = 'Gray';
3. dicodingCat.eat();

Cascade operator ini sering kali menghemat langkah dalam membuat variabel sementara, sehingga kode yang kita tulis menjadi lebih ringkas.

1. Animal('', 2, 4.2)
2. ..name = 'Gray'
3. ..eat();

6. /\* output
7. Gray is eating.
8. \*/

Cascade notation juga akan sering kita temui pada [builder pattern](https://dev.to/inakiarroyo/the-builder-pattern-in-dart-efg) seperti ini:

1. final addressBook = (AddressBookBuilder()
2. ..name = 'jenny'
3. ..email = 'jenny@example.com'
4. ..phone = '415-555-0100')
5. .build();

**Inheritance**

Beberapa objek bisa memiliki beberapa karakteristik atau perilaku yang sama, namun sebenarnya mereka bukanlah objek yang sama. Di sini hadirlah 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 {
3. }

Dengan begitu kita bisa membuat kelas Cat mewarisi kelas Animal.

* [**Cat.dart**](https://www.dicoding.com/academies/191/tutorials/7618?from=16298#tab1-code1)
* [Animal.dart](https://www.dicoding.com/academies/191/tutorials/7618?from=16298#tab1-code2)

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

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

1. import 'Cat.dart';
3. void main() {
4. var dicodingCat = Cat('Grayson', 2, 2.2, 'Gray');
5. dicodingCat.walk();
6. dicodingCat.eat();
7. print(dicodingCat.weight);
8. }
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;
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

## 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 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;
4. Bird(String name, int age, double weight, this.featherColor) : super(name, age, weight);
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;
4. Bird(String name, int age, double weight, this.featherColor) : super(name, age, weight);
6. @override
7. void fly() {
8. print('$name is flying');
9. }
11. }

Keyword atau anotasi @override menunjukkan fungsi tersebut mengesampingkan fungsi yang ada di interface atau kelas induknya, lalu menggunakan fungsi yang ada 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 Enumerated Type, sering disebut Enumerations atau Enums. **Enums**mewakili kumpulan konstan yang membuat kode kita lebih jelas dan mudah dibaca.

1. enum Rainbow {
2. red, orange, yellow, green, blue, indigo, violet
3. }
5. enum Weather {
6. sunny, cloudy, rain, storm;
7. }

 Enums pada Dart memiliki beberapa properti 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

Semenjak Dart versi 2.15, Enums memiliki properti baru untuk menampilkan nilai String setiap itemnya [[3](https://medium.com/dartlang/dart-2-15-7e7a598e508a)]. Anda dapat gunakan properti .name.

1. print(Rainbow.blue.name);   // blue

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

1. var weatherForecast = Weather.cloudy;
3. switch(weatherForecast) {
4. case Weather.sunny:
5. print("Today's weather forecast is sunny");
6. break;
7. case Weather.cloudy:
8. print("Today's weather forecast is cloudy");
9. break;
10. case Weather.rain:
11. print("Today's weather forecast is rain");
12. break;
13. case Weather.storm:
14. print("Today's weather forecast is storm");
15. break;
16. }

Dengan kecanggihan fitur yang disediakan oleh Enums, Dart versi 2.17 rilis dengan fitur baru [[4](https://medium.com/dartlang/dart-2-17-b216bfc80c5d)]. Enums mempunyai fitur layaknya kelas yang memiliki attribute dan behaviour. Dengan begitu, kita dapat memberikan variabel ataupun constructor di dalam Enums.

1. enum Weather {
2. sunny(15),
3. cloudy(34),
4. rain(69),
5. storm(83);
7. final int rainAmount;
9. const Weather(this.rainAmount);
10. }

Dengan kehadiran variabel di dalam Enums, kita dapat akses attribute rainAmount dengan cara seperti berikut.

1. print(Weather.rain.rainAmount);

Selain itu, kita dapat melakukan override method toString(). Dengan begitu, kita dapat mengonversi teks sesuai dengan apa yang diinginkan.

1. enum Weather {
2. …
4. **@override**
5. **String toString() => "Today's weather forecast is $name with a $rainAmount% chance of rain";**
6. }

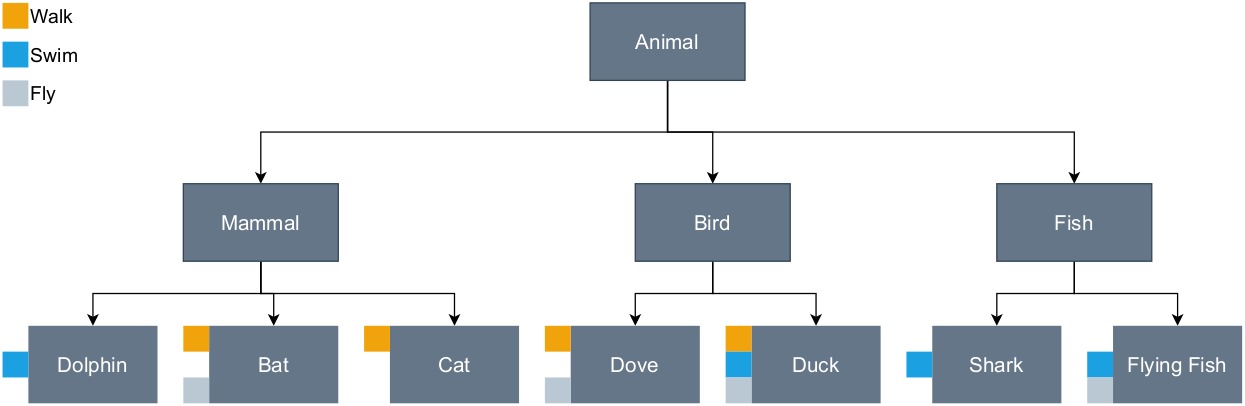
Untuk menjalankannya, Anda dapat jalankan perintah berikut.

1. print(Weather.cloudy.toString());
2. // atau
3. print(Weather.cloudy);

## 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 interface mengharuskan kita untuk meng-override method dan membuat implementasi fungsi di masing-masing kelas.

1. mixin Flyable {
2. void fly() {
3. print("I'm flying");
4. }
5. }
7. mixin Walkable {
8. void walk() {
9. print("I'm walking");
10. }
11. }
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 { }
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();
5. garfield.walk();
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](https://en.wikipedia.org/wiki/Multiple_inheritance#The_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. }
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. }
8. mixin Singer implements Performer {
9. @override
10. void perform() {
11. print('Singing');
12. }
13. }
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. } \*/
5. extension Sorting on List<int> {
6. List<int> sortAsc() {
7. var list = this;
8. var length = this.length;

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. }

20. int tmp = list[min];
21. list[min] = list[i];
22. list[i] = tmp;
23. }

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](https://www.tutorialspoint.com/data_structures_algorithms/selection_sort_algorithm.htm). 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);

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**](https://www.dicoding.com/academies/191/tutorials/7636?from=7633#tab1-code1)
* [extensions.dart](https://www.dicoding.com/academies/191/tutorials/7636?from=7633#tab1-code2)

1. import 'extension.dart';
3. void main() {
4. var unsortedNumbers = [2, 5, 3, 1, 4];
5. print(unsortedNumbers);
6. var sortedNumbers = unsortedNumbers.sortAsc();
7. print(sortedNumbers);
8. }

* [main.dart](https://www.dicoding.com/academies/191/tutorials/7636?from=7633#tab1-code1)
* [**extensions.dart**](https://www.dicoding.com/academies/191/tutorials/7636?from=7633#tab1-code2)

1. extension Sorting on List<int> {
2. List<int> sortAsc() {
3. var list = this;
4. var length = this.length;
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. }
14. int tmp = list[min];
15. list[min] = list[i];
16. list[i] = tmp;
17. }
19. return list;
20. }
21. }
23. extension NumberParsing on String {
24. int parseInt() {
25. return int.parse(this);
26. }
27. }

# Paradigma Functional Programming

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 state 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. };
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);

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. }
7. Function calculate(base) {
8. var count = 1;
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.

# Pengenalan 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.

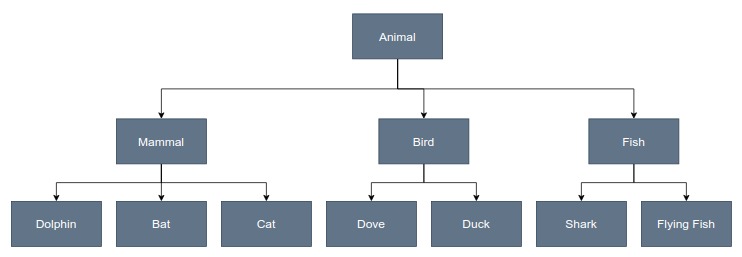
1. List<int> numberList = [1, 2, 3, 4, 5];
2. List<String> stringList = ['Dart', 'Flutter', 'Android', 'iOS'];
3. List dynamicList = [1, 2, 3, 'empat'];  // List<dynamic>

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();

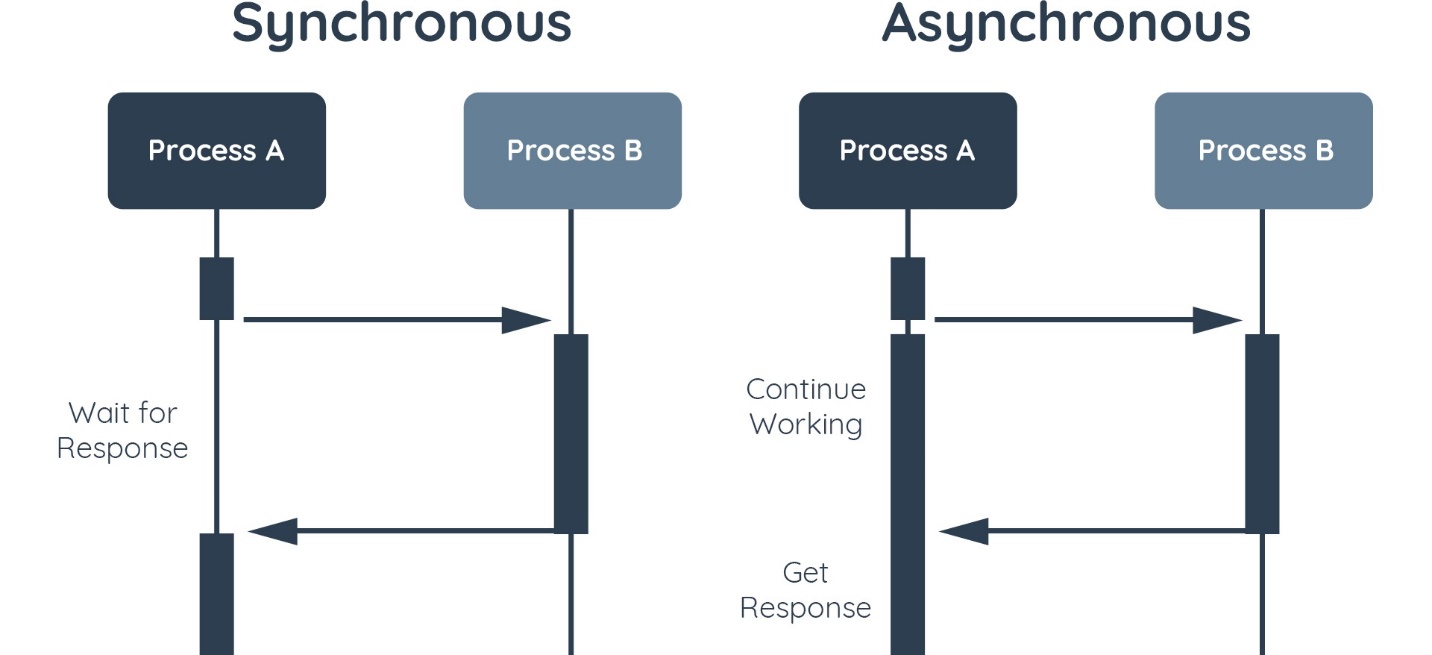
# **Pengenalan Dart Futures**

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.

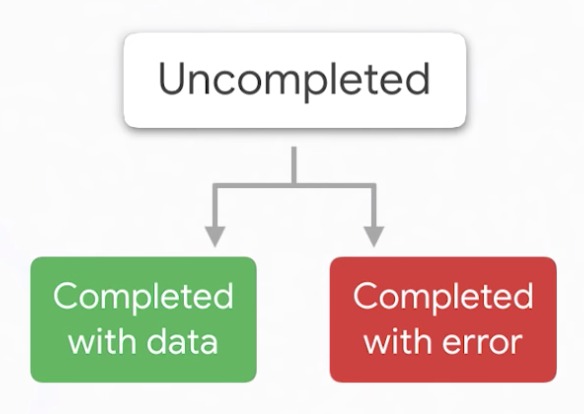
Dalam asynchronous program, jika kita menuliskan dua baris kode, kita dapat membuat baris kode kedua dieksekusi tanpa harus menunggu kode pada baris pertama selesai dieksekusi. Dalam dunia nyata bisa kita bayangkan dengan memesan kopi melalui pelayan, di mana sembari menunggu pesanan kita datang, kita 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.



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('Your ordered: $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 ordered: 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 ordered: $value');
4. })
5. **.catchError((error) {**
6. **print('Sorry. $error');**
7. **});**
8. print('Getting your order...');
9. }
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*:

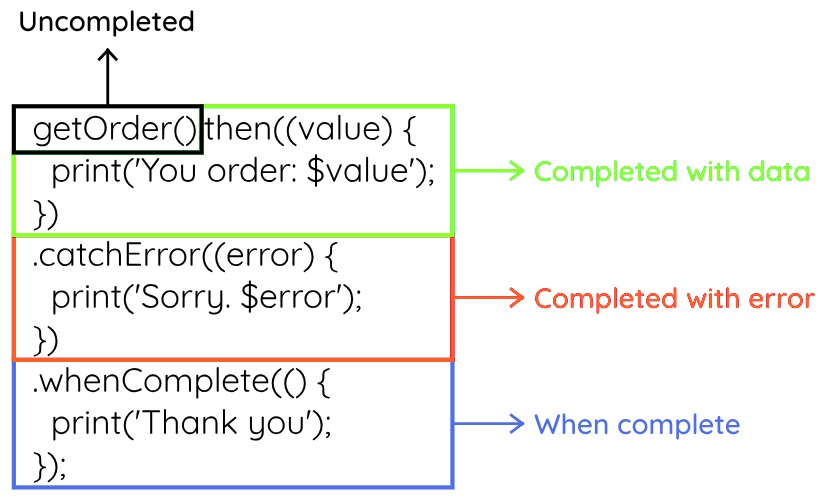
1. Fungsi getOrder() yang berisi Future yang masih ***uncompleted***.
2. Method then() yang menangani kondisi ***completed with data***.
3. Method catchError() yang menangani kondisi ***completed with error***.

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. void main() {
2. getOrder().then((value) {
3. print('You ordered: $value');
4. })
5. .catchError((error) {
6. print('Sorry. $error');
7. })
8. .whenComplete(() {
9. print('Thank you');
10. });
11. print('Getting your order...');
12. }

## Future dengan 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:



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.

|  |
| --- |
| Getting your order... 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.

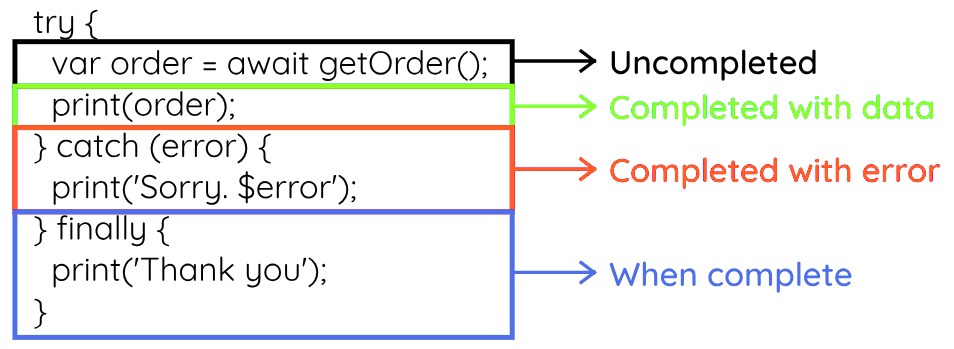
1. void main() **async** {
2. print('Getting your order...');
3. var order = **await** getOrder();
4. print('You order: $order');
5. }

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. } finally {
9. print('Thank you');
10. }
11. }

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



# **Pengenalan 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;
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. }

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)
4. // “hey garfield, eat!”
5. garfield.eat();
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!