**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.

class Animal {

  String name;

  int age;

  double weight;

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

  void **eat**() {

**print**('$name is eating.');

    weight = weight + 0.2;

  }

  void **sleep**() {

**print**('$name is sleeping.');

  }

  void **poop**() {

**print**('$name is pooping.');

    weight = weight - 0.1;

  }

}

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.

void **main**() {

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

  dicodingCat.**eat**();

  dicodingCat.**poop**();

**print**(dicodingCat.weight);

}

class Animal {

  String name;

  int age;

  double weight;

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

  void **eat**() {

**print**('$name is eating.');

    weight = weight + 0.2;

  }

  void **sleep**() {

**print**('$name is sleeping.');

  }

  void **poop**() {

**print**('$name is pooping.');

    weight = weight - 0.1;

  }

}

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:

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. }

## **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.

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

1. class Animal {
2. String name = '';
3. int age = 0;
4. double weight = 0;
6. **Animal(String name, int age, double weight) {**
7. **this.name = name;**
8. **this.age = age;**
9. **this.weight = weight;**
10. **}**
11. }

Keyword this di atas menunjuk pada **atribut** 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. String name = '';
3. int age = 0;
4. double weight = 0;
6. // Mendeklarasikan sebuah constructor
7. Animal(String name, int age, double weight) {
8. this.name = name;
9. this.age = age;
10. this.weight = weight;
11. }
13. // atau dengan cara berikut
14. // Animal(this.name, this.age, this.weight);
16. // Berikut beberapa contoh untuk mendeklarasikan
17. // Named Constructor.
18. **Animal.name(this.name);**
19. **Animal.age(this.age);**
20. **Animal.weight(this.weight);**
21. }

### Initializer list

Terkadang, kita ingin mendeklarasikan salah satu properti kelas dengan nilai default. Skenario tersebut dapat dilakukan dengan mendeklarasikan properti di dalam constructor body.

1. class Animal {
2. ...
4. Animal.cat(this.name, this.weight) **{**
5. **age = 2;**
6. }
7. }

Namun, cara tersebut kurang tepat karena variabel age seharusnya didefinisikan sebelum constructor body. Kita bisa memanfaatkan initializer list dengan mendeklarasikan properti kelas sebelum constructor body berjalan.

1. class Animal {
2. ...
4. Animal.cat(this.name, this.weight) **: age = 2** {
5. // write your code here.
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:

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.

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.

class Cat extends Animal {

  final String furColor;

  Cat(String name, int age, double weight, String furColor)

      : this.furColor = furColor,

        super(name, age, weight);

  void **walk**() {

**print**('$name is walking');

  }

}

class Animal {

  String name = '';

  int age = 0;

  double weight = 0;

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

  void **eat**() {

**print**('$name is eating.');

    weight = weight + 0.2;

  }

  void **sleep**() {

**print**('$name is sleeping.');

  }

}

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

### 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) : **this.furColor = furColor,** super(name, age, weight);

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'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 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.

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.

void **main**() {

**print**(Rainbow.values);

**print**(Rainbow.blue);

**print**(Rainbow.orange.index);

}

enum Rainbow { red, orange, yellow, green, blue, indigo, violet }

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.

void **main**() {

  var weatherForecast = Weather.cloudy;

  switch (weatherForecast) {

    case Weather.sunny:

**print**("Today's weather forecast is sunny");

      break;

    case Weather.cloudy:

**print**("Today's weather forecast is cloudy");

      break;

    case Weather.rain:

**print**("Today's weather forecast is rain");

      break;

    case Weather.storm:

**print**("Today's weather forecast is storm");

      break;

  }

}

enum Weather { sunny, cloudy, rain, storm }

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.

void **main**() {

  var weatherForecast = Weather.cloudy;

**print**(weatherForecast);

}

enum Weather {

**sunny**(15),

**cloudy**(34),

**rain**(69),

**storm**(83);

  final int rainAmount;

  const Weather(this.rainAmount);

  @override

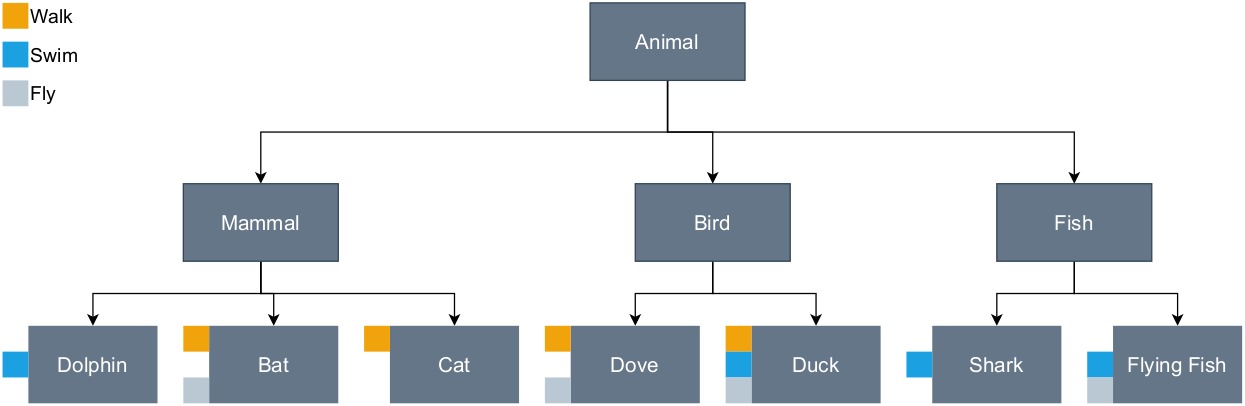
  String **toString**() =>

      "Today's weather forecast is $name with a $rainAmount% chance of rain";

## **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().

void **main**() {

  var arielNoah = Musician();

  arielNoah.**perform**();

}

abstract class Performer {

  void **perform**();

}

mixin Dancer implements Performer {

  @override

  void **perform**() {

**print**('Dancing');

  }

}

mixin Singer implements Performer {

  @override

  void **perform**() {

**print**('Singing');

  }

}

class Musician extends Performer with Dancer, Singer {

  void **showTime**() {

**perform**();

  }

}

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:

/\* extension <extension name> on <type> {

  (<member definition>)\*

} \*/

extension Sorting on List<int> {

  List<int> **sortAsc**() {

    var list = this;

    var length = this.length;

    for (int i = 0; i < length - 1; i++) {

      int min = i;

      for (int j = i + 1; j < length; j++) {

        if (list[j] < list[min]) {

          min = j;

        }

      }

      int tmp = list[min];

      list[min] = list[i];

      list[i] = tmp;

    }

    return list;

  }

}

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.

void **main**() {

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

**print**(unsortedNumbers);

  var sortedNumbers = unsortedNumbers.**sortAsc**();

**print**(sortedNumbers);

}

extension Sorting on List<int> {

  List<int> **sortAsc**() {

    var list = this;

    var length = this.length;

    for (int i = 0; i < length - 1; i++) {

      int min = i;

      for (int j = i + 1; j < length; j++) {

        if (list[j] < list[min]) {

          min = j;

        }

      }

      int tmp = list[min];

      list[min] = list[i];

      list[i] = tmp;

    }

    return list;

  }

}

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

## **Rangkuman Materi**

Kita telah berada di akhir dari modul Object Oriented Programming. Mari kita uraikan materi yang sudah Anda pelajari.

* OOP merupakan paradigma yang berdasarkan pada konsep objek yang memiliki atribut serta dapat melakukan operasi atau prosedur tertentu.
* Terdapat 4 (empat) pilar dalam object oriented programming, antara lain:
  + Encapsulation, kondisi di mana status atau kondisi di dalam class, dibungkus, dan bersifat privat. Artinya objek lain tidak bisa mengakses property secara langsung.
  + Abstraction, objek hanya menunjukkan operasinya secara high-level. Cukup tau method apa saja yang dibuat oleh class tersebut.
  + Inheritance, mewarisi properti kelas induk (parent class) ke kelas anak (child class).
  + Polymorphism, objek class dapat memiliki bentuk atau implementasi yang berbeda-beda pada satu metode yang sama.
* Class merupakan blueprint untuk mendefinisikan karakteristik dari sebuah objek.
* Di dalam class, variabel berlaku sebagai properti, sementara fungsi sebagai method.
* Abstract Class merupakan gambaran umum dari sebuah kelas. Abstract class tidak dapat direalisasikan ke dalam sebuah objek.
* Mixin digunakan untuk menggabungkan dan memanfaatkan property/method dari beberapa kelas lain.
* Extension methods digunakan untuk membuat method tambahan pada kelas yang sudah ada.