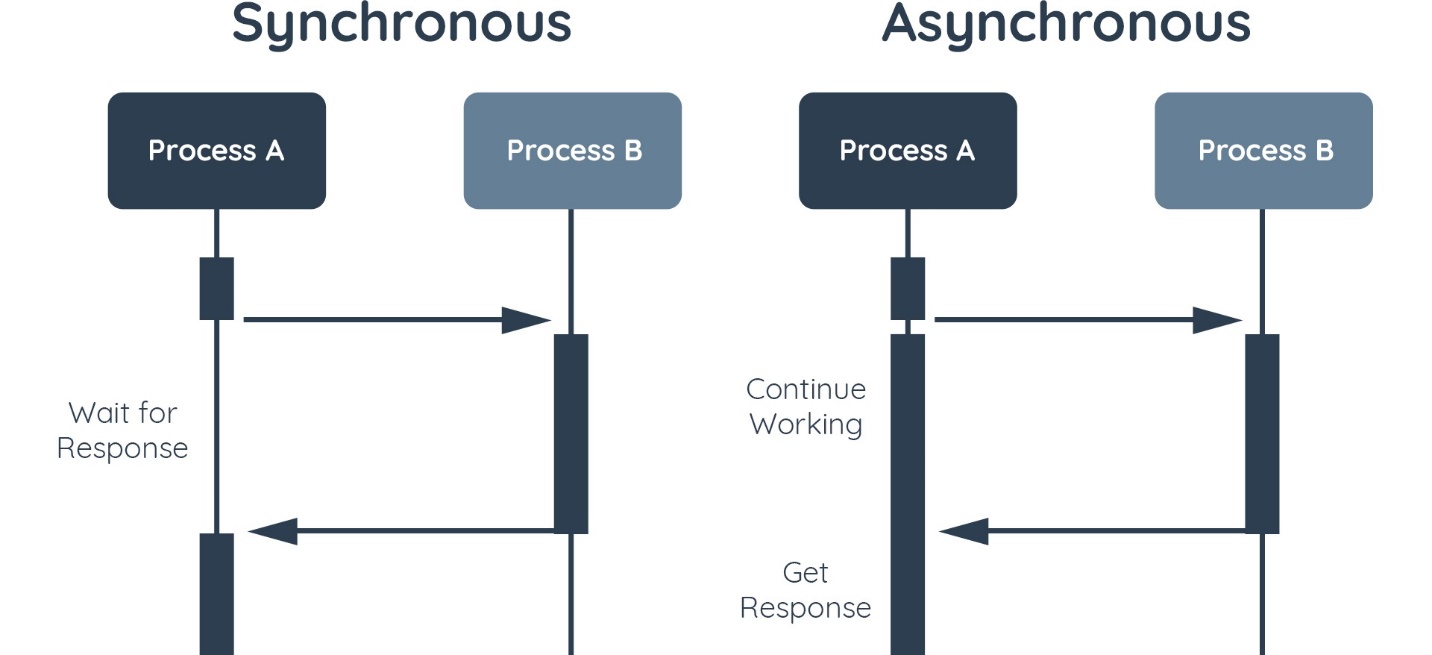
**Pengenalan Dart Futures**

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.

void **main**() {

  final myFuture = Future(() {

**print**('Creating the future');

    return 12;

  });

**print**('main() done');

}

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.

void **main**() {

**getOrder**().**then**((value) {

**print**('Your ordered: $value');

  });

**print**('Getting your order...');

}

Future<String> **getOrder**() {

  return Future.**delayed**(Duration(seconds: 3), () {

    return 'Coffee Boba';

  });

}

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

1. Getting your order...
2. Your 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.

void **main**() {

**getOrder**().**then**((value) {

**print**('Your ordered: $value');

  }).**catchError**((error) {

**print**('Sorry. $error');

  });

**print**('Getting your order...');

}

Future<String> **getOrder**() {

  return Future.**delayed**(Duration(seconds: 3), () {

    var isStockAvailable = false;

    if (isStockAvailable) {

      return 'Coffee Boba';

    } else {

      throw 'Our stock is not enough.';

    }

  });

}

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*.

void **main**() {

**getOrder**().**then**((value) {

**print**('Your ordered: $value');

  }).**catchError**((error) {

**print**('Sorry. $error');

  }).**whenComplete**(() {

**print**('Thank you');

  });

**print**('Getting your order...');

}

Future<String> **getOrder**() {

  return Future.**delayed**(Duration(seconds: 3), () {

    var isStockAvailable = false;

    if (isStockAvailable) {

      return 'Coffee Boba';

    } else {

      throw 'Our stock is not enough.';

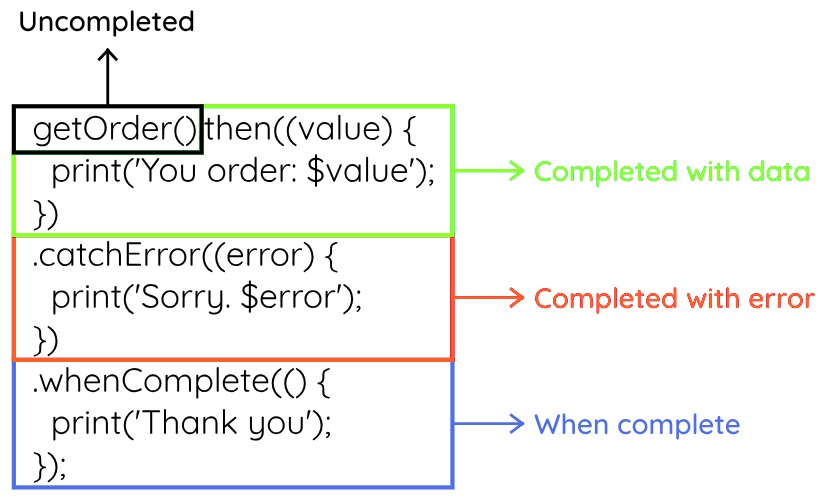
    }

  });

}

## 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.

void **main**() async {

**print**('Getting your order...');

  var order = await **getOrder**();

**print**('You order: $order');

}

Future<String> **getOrder**() {

  return Future.**delayed**(Duration(seconds: 3), () {

    return 'Coffee Boba';

  });

}

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

void **main**() async {

**print**('Getting your order...');

  try {

    var order = await **getOrder**();

**print**('You order: $order');

  } catch (error) {

**print**('Sorry. $error');

  } finally {

**print**('Thank you');

  }

}

Future<String> **getOrder**() {

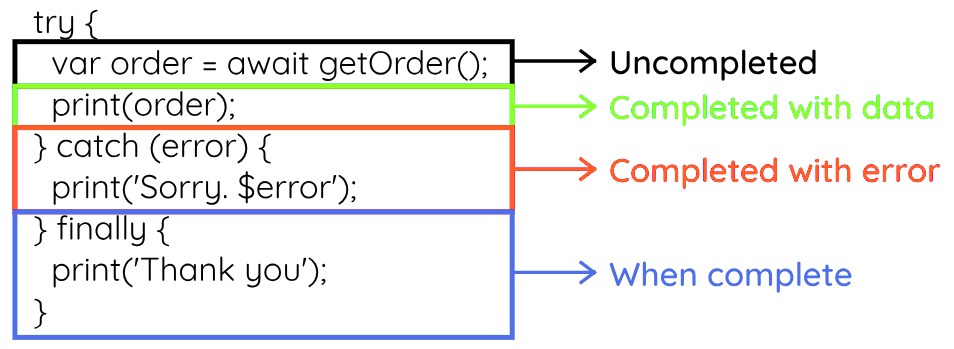
  return Future.**delayed**(Duration(seconds: 3), () {

    return 'Coffee Boba';

  });

}

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



## **Rangkuman Materi**

Pada modul ini kita telah mempelajari konsep asynchronous process pada Dart. Prinsip umumnya adalah beberapa kode atau proses yang bisa berjalan bersamaan. Pada modul ini juga kita mengenal Future dan implementasinya untuk menangani proses asynchronous.

Beberapa hal yang telah kita pelajari, antara lain:

* Dalam synchronous program, kode program akan berjalan secara berurutan dari atas ke bawah.
* Dalam asynchronous program memungkinkan suatu operasi bisa berjalan sembari menunggu operasi lainnya selesai.
* Future adalah sebuah objek yang mewakili nilai potensial atau kesalahan yang akan tersedia pada waktu mendatang.
* Terdapat tiga kondisi yang memungkinkan saat menerapkan objek Future, antara lain:
  + Uncompleted, ketika operasi asynchronous tertunda pada durasi waktu tertentu sehingga operasi tersebut mengembalikan nilai Future.
  + Completed with data, ketika operasi asynchronous berhasil berjalan, objek Future akan mengembalikan nilai dengan tipe data tertentu.
  + Completed with error, ketika operasi asynchronous berjalan dan mengalami kegagalan, objek Future akan mengembalikan nilai eror.
* Metode delayed pada objek Future berfungsi untuk menunda jalannya operasi kode bergantung dengan durasi yang ditentukan.
* var coffee = Future.delayed(Duration(seconds: 3), () {
* return 'Coffee Boba';

});

* Objek Future memiliki beberapa fungsi/method untuk menangani state, antara lain:
  + then(), fungsi yang menangani kondisi completed with data.
  + catchError(), fungsi yang menangani kondisi completed with error.
  + whenComplete(), fungsi yang berjalan saat proses asynchronous dan state lainnya berakhir.
* Keyword async dan await mendukung proses asynchronous program, sehingga memungkinkan untuk menulis kode asynchronous yang terlihat mirip layaknya kode synchronous.