# Pengenalan Concurrency

Dalam pemrograman, salah satu topik yang menantang adalah tentang *concurrency*. Concurrency sendiri berarti beberapa komputasi yang terjadi pada saat yang bersamaan[[4](http://web.mit.edu/6.005/www/fa14/classes/17-concurrency/" \t "_blank)]. Sejauh ini kita telah menuliskan kode secara *synchronous*. Lebih lanjut, pada modul ini kita akan mempelajari beberapa materi seperti:

* Bagaimana menjalankan program secara asynchronous
* Bagaimana menangani kode asynchronous

Sebelum membahas asynchronous lebih dalam, kita akan bahas dahulu apa perbedaan synchronous dan asynchronous. *Let’s go!*

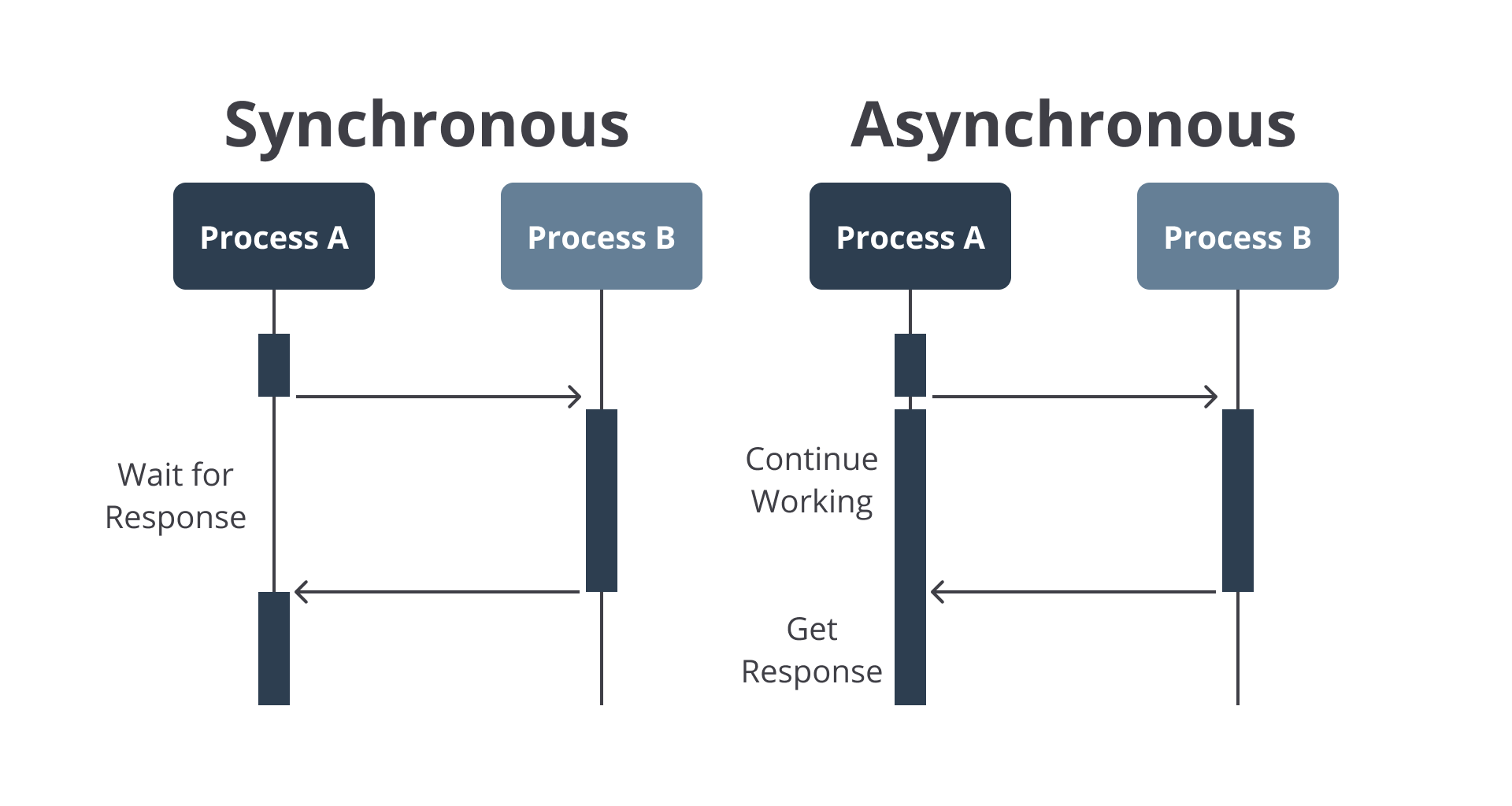
## Synchronous vs Asynchronous

Dalam synchronous program, kode 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. Kita bisa bayangkan ini dalam kehidupan nyata ketika mengantri membeli kopi di sebuah kedai kopi. 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 kita bisa membayangkan dengan memesan kopi, tetapi pemesanannya melalui pelayan. Sembari menunggu pesannya datang, kita dapat melakukan aktivitas lain seperti membuka laptop, menulis, hingga kopi itu datang dengan sendirinya.

Urutan di mana seseorang mendapatkan minumannya terlebih dahulu memiliki korelasi dengan kapan ia memesan makanannya. Namun bukan hanya itu, faktor ini juga dipengaruhi dengan minuman apa yang ia pesan. Contohnya jika kita memesan kopi espresso sedangkan teman kita hanya memesan air mineral. Walaupun kita memesannya terlebih dahulu, tiada jaminan kita akan mendapatkannya duluan. Membuat espresso tentu akan membutuhkan waktu lebih lama dibandingkan dengan menuangkan air mineral pada gelas, kan?

Dalam program yang dijalankan secara asynchronous pun demikian. Task yang kecil akan lebih dahulu selesai dibandingkan dengan task yang besar, meskipun task yang besar lebih dahulu dijalankan.

[](https://www.dicoding.com/academies/256/tutorials/14012?from=15805)

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.

## setTimeout

Fungsi setTimeout() merupakan cara yang paling mudah untuk membuat kode kita dijalankan secara asynchronous. Fungsi ini menerima dua buah parameter. Parameter pertama adalah fungsi yang akan dijalankan secara asynchronous. Kedua adalah nilai number dalam milisecond sebagai nilai tunggu sebelum fungsi dijalankan. Contoh penggunaannya adalah seperti ini:

1. console.log("Selamat datang!");
2. setTimeout(() => {
3. console.log("Terima kasih sudah mampir, silakan datang kembali!");
4. }, 3000);
5. console.log("Ada yang bisa dibantu?");

Jika hanya mengenal program secara synchronous, maka kita dapat membayangkan hasilnya memiliki urutan sebagai berikut:

* Mencetak -> Selamat datang!
* Menunggu selama tiga detik.
* Mencetak -> Terima kasih sudah mampir, silakan datang kembali!
* Mencetak -> Ada yang bisa dibantu?

Namun, nyatanya setTimeout() tidak akan menghentikan JavaScript untuk melakukan eksekusi kode pada baris berikutnya. Sehingga urutannya menjadi seperti berikut:

* Mencetak -> Selamat datang!
* Mencetak -> Ada yang bisa dibantu?
* Menunggu selama tiga detik
* Mencetak -> Terima kasih sudah mampir, silakan datang kembali!

Jika kode tersebut dijalankan, ia akan menampilkan output seperti berikut:

## Callback Function

Hal yang seringkali membingungkan ketika bekerja dengan program synchronous dan asynchronous adalah bagaimana menangani suatu nilai yang didapatkan secara asynchronous pada program yang berjalan secara synchronous. Contohnya seperti kode berikut:

1. const orderCoffee = () => {
2. let coffee = null;
3. console.log("Sedang membuat kopi, silakan tunggu...");
4. setTimeout(() => {
5. coffee = "Kopi sudah jadi!";
6. }, 3000);
7. return coffee;
8. }
10. const coffee = orderCoffee();
11. console.log(coffee);
13. /\* output
14. Sedang membuat kopi, silakan tunggu...
15. null
16. \*/

Jika kita melakukan hal seperti di atas untuk mencetak nilai coffee, maka hal tersebut tidak akan pernah terjadi. Seperti yang sudah kita ketahui, fungsi setTimeout() tidak akan menghentikan JavaScript untuk mengeksekusi kode yang ada selanjutnya. Jadi, fungsi orderCoffee() akan selalu mengembalikan nilai null, karena kode return coffee akan dieksekusi terlebih dulu dibandingkan dengan coffee = "Kopi sudah jadi!";. Kode asynchronous perlu disusun dengan cara yang berbeda dari synchronous. Cara paling dasar adalah dengan callback function.

Apa itu callback function? Mari kita bayangkan kembali ketika memesan kopi. Pada kasus ini mungkin terdapat dua aksi yang bisa kita lakukan:

* (Synchronous) Kita tetap menunggu di kasir sampai kopi datang.
* (Asynchronous) Kita menunggu di meja setelah memesan kopi. Selanjutnya kopi akan diantarkan oleh pelayan. Sehingga, kita tidak perlu menunggu di kasir dan dapat melakukan pekerjaan lain.

Nah, pada JavaScript, pelayan ini berperan layaknya callback function. Ia diperintahkan pada sebuah fungsi asynchronous kemudian akan dipanggil/digunakan ketika tugas itu selesai.

Bagaimana cara menerapkannya dalam kode? Pertama, kita tambahkan parameter dengan nama callback pada fungsi asynchronous.

1. const orderCoffee = callback => {
2. let coffee = null;
3. console.log("Sedang membuat kopi, silakan tunggu...");
4. setTimeout(() => {
5. coffee = "Kopi sudah jadi!";
6. }, 3000);
7. return coffee;
8. }

Kemudian kita panggil atau gunakan callback yang diisikan dengan data yang akan dibawa (coffee) ketika task selesai dilakukan.

1. setTimeout(() => {
2. coffee = "Kopi sudah jadi!";
3. **callback(coffee);**
4. }, 3000);

Setelah menggunakan callback, fungsi tidak perlu lagi mengembalikan nilai. Sehingga, kita bisa menghapus kode return coffee;. Keseluruhan fungsi akan tampak seperti ini:

1. const orderCoffee = callback => {
2. let coffee = null;
3. console.log("Sedang membuat kopi, silakan tunggu...");
4. setTimeout(() => {
5. coffee = "Kopi sudah jadi!";
6. callback(coffee);
7. }, 3000);
8. }

Kemudian untuk menggunakan fungsi orderCoffee, ubah kode dari:

1. const coffee = orderCoffee();
2. console.log(coffee);

Menjadi:

1. orderCoffee(coffee => {
2. console.log(coffee);
3. });

Sehingga ketika dijalankan akan sesuai dengan harapan kita.

1. const orderCoffee = callback => {
2. let coffee = null;
3. console.log("Sedang membuat kopi, silakan tunggu...");
4. setTimeout(() => {
5. coffee = "Kopi sudah jadi!";
6. callback(coffee);
7. }, 3000);
8. }

11. orderCoffee(coffee => {
12. console.log(coffee);
13. });

16. /\* output
17. Sedang membuat kopi, silakan tunggu...
18. ---- setelah 3 detik ----
19. Kopi sudah jadi!
20. \*/

### Callback Hell

Kita sudah mengetahui bahwa callback dibutuhkan untuk mendapatkan nilai yang berasal dari asynchronous function. Lantas bagaimana jika terdapat proses yang saling bergantung satu sama lain? Contohnya, untuk membuat kue tahapan yang perlu kita lakukan adalah:

1. Menyiapkan bahan
2. Membuat adonan
3. Memasukkan adonan ke cetakan
4. Memanggang adonan

Tahapan tersebut sangat bergantung satu sama lain. Kita tidak bisa mencetak adonan sebelum menyiapkan bahan dan membuat adonan. Jika seluruh tahapan tersebut berjalan secara synchronous, mungkin kita bisa melakukannya seperti ini:

1. function makeACake(...rawIngredients) {
2. const ingredients = collectIngredients(rawIngredients);
3. dough = makeTheDough(ingredients);
4. pouredDough = pourDough(dough);
5. cake = bakeACake(pouredDough);
6. console.log(cake);
7. }

Namun, jika fungsi-fungsi tersebut berjalan secara asynchronous, maka kita akan membuat yang namanya callback hell. Callback hell terjadi karena banyak sekali callback function yang bersarang karena saling membutuhkan satu sama lain. Sehingga, kode akan tampak seperti ini:

1. function makeACake(...rawIngredients) {
2. collectIngredients(rawIngredients, function(ingredients) {
3. makeTheDough(ingredients, function(dough) {
4. pourDough(dough, function(pouredDough) {
5. bakeACake(pourDough, function(cake) {
6. console.log(cake);
7. })
8. })
9. })
10. });
11. }

Melihat kode seperti ini saja kepala jadi pusing. Terbayang sulitnya memelihara kode ini di masa yang akan datang.

Lantas apa solusi agar kita dapat menghindari callback hell? Salah satunya adalah dengan menggunakan Promise.

1. function makeACake(...rawIngredients) {
2. collectIngredients(rawIngredients)
3. .then(makeTheDough)
4. .then(pourDough)
5. .then(bakeACake)
6. .then(console.log);
7. }

Dengan Promise, kita dapat meminimalisir callback hell dan mengubahnya menjadi kode yang sangat mudah dibaca. Bahkan dengan kode seperti itu, non-developer pun dapat mengerti apa maksud dari kode tersebut.

## Promise

Promise merupakan salah satu fitur penting dari ES6. Promise ini dapat menggantikan peran callback dengan menggunakan ciri khas fungsi .then-nya. Namun, mengapa fitur ini dinamakan dengan “Promise” alias “Janji”?

Fitur ini berfungsi seperti namanya, yaitu untuk membuat janji. Mari kita analogikan kembali dalam dunia nyata. Ketika kita memesan kopi kepada pelayan, maka secara tidak langsung pelayan tersebut berjanji kepada kita untuk membuatkan kopi dan menyajikannya pada kita. Namun janji bisa hanya tinggal janji. Dalam dunia nyata pun, janji juga bisa tidak terpenuhi, entah itu karena kopi pesanan kita sedang kosong, atau mesin pembuat kopi sedang rusak.

Nah, Promise memiliki perilaku yang sama dengan analogi di atas. Promise memiliki tiga kondisi, yaitu:

* **Pending** (Janji sedang dalam proses)
* **Fulfilled** (Janji terpenuhi)
* **Rejected** (Janji gagal terpenuhi)

Lantas bagaimana cara membuat janji (Promise) di JavaScript?

## Constructing Promise Object

Promise merupakan sebuah objek yang digunakan untuk membuat sebuah komputasi (kode) ditangguhkan dan berjalan secara asynchronous [[5](https://developer.mozilla.org/id/docs/Web/JavaScript/Reference/Global_Objects/Promise)]. Untuk membuat objek promise, kita gunakan keyword new diikuti dengan constructor dari Promise:

1. const coffee = new Promise();

Namun, jika kita jalankan kode tersebut akan mengakibatkan eror seperti ini:

1. TypeError: Promise resolver undefined is not a function

Di dalam constructor Promise, kita perlu menetapkan resolver function atau bisa disebut executor function. Fungsi tersebut akan dijalankan secara otomatis ketika constructor Promise dipanggil.

1. const executorFunction = (resolve, reject) => {
2. const isCoffeeMakerReady = true;
3. if (isCoffeeMakerReady) {
4. resolve("Kopi berhasil dibuat");
5. } else {
6. reject("Mesin kopi tidak bisa digunakan");
7. }
8. }

11. const makeCoffee = () => {
12. return new Promise(executorFunction);
13. }
14. const coffeePromise = makeCoffee();
15. console.log(coffeePromise);

18. /\* output
19. Promise { 'Kopi berhasil dibuat' }
20. \*/

Executor function memiliki dua parameter, yaitu resolve dan reject yang berupa fungsi. Berikut penjelasan detailnya:

* **resolve()** adalah parameter pertama pada executor function. Parameter ini merupakan fungsi yang dapat menerima satu parameter. Biasanya kita gunakan untuk mengirimkan data ketika promise berhasil dilakukan. Ketika fungsi ini terpanggil, kondisi Promise akan berubah dari **pending** menjadi **fulfilled**.
* **reject()** adalah parameter kedua pada executor function. Parameter ini merupakan fungsi yang dapat menerima satu parameter dan digunakan untuk memberikan alasan kenapa Promise tidak dapat terpenuhi. Ketika fungsi ini terpanggil, kondisi Promise akan berubah dari **pending** menjadi **rejected**.

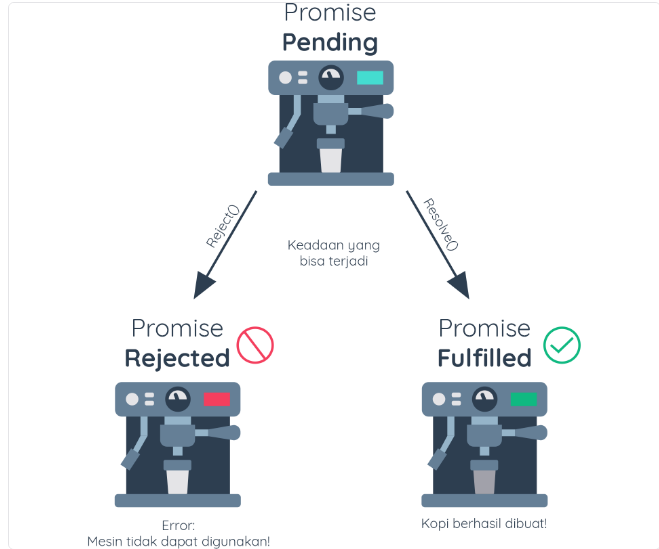
Executor function akan berjalan secara asynchronous hingga akhirnya kondisi Promise berubah dari **pending** menjadi **fulfilled/rejected**.

Pada contoh kode di atas, outputnya akan seperti ini:

1. /\* output
2. Promise { 'Kopi berhasil dibuat' }
3. \*/

Kenapa demikian? Executor function mengeksekusi resolve() dengan membawa data string “Kopi berhasil dibuat”. Jika kita ubah nilai dari variabel isCoffeeMakerReady menjadi false, maka executor function akan mengeksekusi reject() dengan membawa pesan penolakan “Mesin kopi tidak bisa digunakan”.

1. /\* output
2. Promise { <rejected> 'Mesin kopi tidak bisa digunakan' }
3. \*/

Dalam praktik aslinya, Promise digunakan untuk menjalankan proses asynchronous seperti mengambil data dari internet/API. Hasil permintaan data dapat terpenuhi atau mengalami kegagalan.

Output yang dihasilkan baik ketika fulfilled ataupun rejected masih berupa Promise, bukan nilai yang dibawa oleh fungsi resolve atau reject. Lantas bagaimana kita bisa mengakses nilai yang dibawa oleh fungsi-fungsi tersebut? Caranya adalah menggunakan method .then() yang tersedia pada objek Promise.

## Consuming Promises

Setelah mengetahui bagaimana membuat objek Promise, hal yang tentunya sangat penting adalah tahu bagaimana mengonsumsinya. Seperti yang telah kita bahas sebelumnya, status awal dari Promise adalah pending. Kemudian, akan ada dua kemungkinan yang terjadi, fulfilled atau rejected.

Untuk menangani hasil dari Promise, kita gunakan method .then(). Jika kita terjemahkan, “then” berarti “kemudian”, sehingga kurang lebih kita memerintahkan JavaScript seperti ini: “Jika janji saya sudah selesai, kemudian lakukan ...”. Jika dituliskan dalam bentuk kode akan seperti berikut:

1. const myPromise = new Promise(executorFunction);
2. myPromise.then(onFullfilled, onRejected);

.then() sendiri adalah sebuah higher-order function yang membutuhkan dua parameter. Keduanya adalah callback function yang juga dikenal sebagai handler. Handler pertama adalah fungsi yang akan dijalankan ketika Promise berstatus resolve. Sedangkan handler kedua adalah fungsi yang akan dijalankan ketika Promise berstatus reject.

Kembali ke kasus mesin kopi kita sebelumnya, mesin bisa gagal membuat kopi jika bahan-bahan tidak mencukupi. Sementara jika bahan cukup, mesin akan membuatkan satu gelas kopi. Di sinilah kita dapat memanfaatkan Promise sekaligus menangani dua kemungkinan promise yang terjadi.

Mari kita buat object untuk menyimpan stok dan fungsi yang mengembalikan objek Promise.

1. const stock = {
2. coffeeBeans: 250,
3. water: 1000,
4. }
6. const checkStock = () => {
7. return new Promise((resolve, reject) => {
8. if (stock.coffeeBeans >= 16 && stock.water >= 250) {
9. resolve("Stok cukup. Bisa membuat kopi");
10. } else {
11. reject("Stok tidak cukup");
12. }
13. });
14. };

Kemudian di bawahnya kita tambahkan dua fungsi untuk menangani masing-masing status resolve dan reject.

1. const handleSuccess = resolvedValue => {
2. console.log(resolvedValue);
3. }
5. const handleFailure = rejectionReason => {
6. console.log(rejectionReason);
7. }

Terakhir panggil method .then() pada checkStock() untuk menangani hasil yang dikembalikan dari promise.

1. checkStock().then(handleSuccess, handleFailure);

Sehingga, keseluruhan kode akan menjadi seperti ini:

1. const stock = {
2. coffeeBeans: 250,
3. water: 1000,
4. }
6. const checkStock = () => {
7. return new Promise((resolve, reject) => {
8. if (stock.coffeeBeans >= 16 && stock.water >= 250) {
9. resolve("Stok cukup. Bisa membuat kopi");
10. } else {
11. reject("Stok tidak cukup");
12. }
13. });
14. };
16. const handleSuccess = resolvedValue => {
17. console.log(resolvedValue);
18. }
20. const handleFailure = rejectionReason => {
21. console.log(rejectionReason);
22. }
24. checkStock().then(handleSuccess, handleFailure);

Mari kita bedah kode di atas:

* checkStock() merupakan fungsi yang mengembalikan promise dan akan menghasilkan resolve() dengan membawa nilai “Stok cukup. Bisa membuat kopi”.
* Lalu kita mendeklarasikan fungsi handleSuccess() dan handleFailure() yang mencetak nilai dari parameternya.
* Kemudian kita memanggil method .then() dari checkStock. Isi parameter then() dengan dua fungsi handler yang telah kita buat sebelumnya.
* Parameter pertama berisi fungsi handleSuccess untuk menangani kondisi ketika promise berstatus resolve. Parameter kedua berisi fungsi handleFailure yang menangani ketika promise berstatus reject.

Cobalah untuk mengubah nilai stock dan memastikan fungsi handleFailure telah dijalankan.

## onRejected with Catch Method

Salah satu cara menulis kode yang baik adalah mengikuti prinsip yang disebut separation of concerns atau pemisahan masalah. Pemisahan masalah berarti mengorganisasikan kode ke dalam bagian-bagian yang berbeda berdasarkan tugas tertentu. Hal ini akan memudahkan kita kelak mencari kode yang salah jika aplikasi tidak bekerja dengan baik.

Perlu diketahui bahwa method .then() akan mengembalikan nilai promise yang sama dengan ketika objek promise itu dipanggil. Melalui sifatnya ini, daripada kita menetapkan logika resolve dan reject pada satu method then(), kita dapat memisahkan kedua logika tersebut menggunakan masing-masing method then() seperti ini:

1. checkStock()
2. .then(handleSuccess)
3. .then(null, handleFailure);

Namun untuk menetapkan onRejected handler, kita perlu memberikan nilai null pada parameter pertama method .then(). Hal ini sedikit merepotkan bukan? Solusinya kita dapat menggunakan method lain, yakni .catch().

Method .catch() mirip seperti .then(). Namun, method ini hanya menerima satu parameter function yang digunakan untuk rejected handler. Method catch() ini akan terpanggil ketika objek promise memiliki kondisi onRejected. Berikut contoh penggunaan method .catch():

1. checkStock()
2. .then(handleSuccess)
3. .catch(handleFailure);

Dengan menggunakan method catch(), kita dapat menerapkan prinsip separation of concerns sekaligus membuat kodenya menjadi lebih rapi.

## Chaining Promises

Kita sudah tahu buruknya penulisan callback hell. Namun, kita tidak dapat menghindari keadaan di mana proses asynchronous saling bergantung satu sama lain. Untuk menghindari callback hell, salah satu solusinya adalah Promise.

Dengan promise kita dapat melakukan proses asynchronous secara berantai. Contohnya, ketika kita ingin membuat satu gelas kopi, akan ada beberapa tahapan yang dikerjakan oleh mesin pembuat kopi, seperti memastikan mesin sudah siap, memastikan stok biji kopi dan air cukup, membuat kopi, lalu menuangkannya ke dalam gelas. Tahapan tersebut harus dilakukan secara berurutan.

Untuk memastikan rangkaian promise berjalan dengan sesuai, kita perlu mengembalikan (return) promise selanjutnya. Contohnya adalah seperti ini:

1. function makeEspresso() {
2. checkAvailability()
3. .then((value) => {
4. console.log(value);
5. return checkStock();
6. })
7. .then((value) => {
8. console.log(value)
9. return brewCoffee();
10. })
11. .then((value) => {
12. console.log(value);
13. })
14. .catch((rejectedReason) => {
15. console.log(rejectedReason);
16. });
17. }
19. makeEspresso();

Mari kita bedah masing-masing fungsi promise di atas!

Pertama, mesin akan mengecek status ketersediaan. Jika mesin kopi tidak sibuk, maka promise akan mengembalikan status resolve(“Mesin kopi siap digunakan”). Namun, jika status mesin masih sibuk, maka yang dikembalikan adalah status reject(“Maaf, mesin sedang sibuk”).

Berikut adalah kode untuk fungsi checkAvailability():

1. const checkAvailability = () => {
2. return new Promise((resolve, reject) => {
3. setTimeout(() => {
4. if (!state.isCoffeeMachineBusy) {
5. resolve("Mesin kopi siap digunakan.");
6. } else {
7. reject("Maaf, mesin sedang sibuk.");
8. }
9. }, 1000);
10. });
11. };

Pada kode di atas, kita menggunakan fungsi setTimeout() untuk menyimulasikan proses asynchronous dan menunda proses selama 1 detik (1000 milisecond). Objek untuk menyimpan state dari mesin kopi adalah seperti ini:

1. const state = {
2. stock: {
3. coffeeBeans: 250,
4. water: 1000,
5. },
6. isCoffeeMachineBusy: false,
7. }

Kemudian, mesin kopi perlu memastikan bahwa stok biji kopi dan air cukup untuk membuat kopi. Di sini juga kita mengubah status mesin kopi menjadi sibuk.

1. const checkStock = () => {
2. return new Promise((resolve, reject) => {
3. state.isCoffeeMachineBusy = true;
4. setTimeout(() => {
5. if (state.stock.coffeeBeans >= 16 && state.stock.water >= 250) {
6. resolve("Stok cukup. Bisa membuat kopi.");
7. } else {
8. reject("Stok tidak cukup!");
9. }
10. }, 1500);
11. });
12. };

Lalu fungsi promise yang terakhir adalah fungsi untuk mencampurkan kopi dan air lalu menghidangkannya ke dalam gelas. Fungsi ini mengembalikan promise dengan status resolve yang membawa nilai “Kopi sudah siap!”.

1. const brewCoffee = () => {
2. console.log("Membuatkan kopi Anda...")
3. return new Promise((resolve, reject) => {
4. setTimeout(() => {
5. resolve("Kopi sudah siap!")
6. }, 2000);
7. });
8. };

Rangkaian proses di atas berjalan berurutan karena kita menggunakan method .then(). Jika kita baca kodenya kurang lebih akan seperti ini: “Untuk membuat espresso lakukan pengecekan ketersediaan mesin, **kemudian** periksa stok di dalam mesin, **kemudian** buat kopi.”

Apabila promise mengalami kegagalan (reject), ia akan ditangani oleh method catch() yang kita tuliskan di akhir. Entah itu disebabkan karena mesin kopi sedang sibuk atau stok bahannya habis.

Berikut ini adalah kode lengkap dari skenario di atas:

1. const state = {
2. stock: {
3. coffeeBeans: 250,
4. water: 1000,
5. },
6. isCoffeeMachineBusy: false,
7. }
9. const checkAvailability = () => {
10. return new Promise((resolve, reject) => {
11. setTimeout(() => {
12. if (!state.isCoffeeMachineBusy) {
13. resolve("Mesin kopi siap digunakan.");
14. } else {
15. reject("Maaf, mesin sedang sibuk.");
16. }
17. }, 1000);
18. });
19. };
21. const checkStock = () => {
22. return new Promise((resolve, reject) => {
23. state.isCoffeeMachineBusy = true;
24. setTimeout(() => {
25. if (state.stock.coffeeBeans >= 16 && state.stock.water >= 250) {
26. resolve("Stok cukup. Bisa membuat kopi.");
27. } else {
28. reject("Stok tidak cukup!");
29. }
30. }, 1500);
31. });
32. };
34. const brewCoffee = () => {
35. console.log("Membuatkan kopi Anda...")
36. return new Promise((resolve, reject) => {
37. setTimeout(() => {
38. resolve("Kopi sudah siap!")
39. }, 2000);
40. });
41. };
43. function makeEspresso() {
44. checkAvailability()
45. .then((value) => {
46. console.log(value);
47. return checkStock();
48. })
49. .then((value) => {
50. console.log(value)
51. return brewCoffee();
52. })
53. .then(value => {
54. console.log(value);
55. state.isCoffeeMachineBusy = false;
56. })
57. .catch(rejectedReason => {
58. console.log(rejectedReason);
59. state.isCoffeeMachineBusy = false;
60. });
61. }
63. makeEspresso();
65. /\* output
66. Mesin kopi siap digunakan.
67. Stok cukup. Bisa membuat kopi.
68. Membuatkan kopi Anda...
69. Kopi sudah siap!
70. \*/

## Promise All

Pada materi sebelumnya kita belajar bagaimana promise dapat menangani situasi di mana terdapat asynchronous process yang saling membutuhkan untuk melaksanakan tugasnya. Lalu bagaimana jika kita ingin menjalankan banyak promise sekaligus tanpa memedulikan urutan? Bukankah concurrency memungkinkan kita melakukan banyak proses bersamaan agar lebih efisien?

Ketika pergi ke sebuah kedai kopi bersama rekan kerja, kita biasanya memesan kopi secara bersamaan. Meskipun kopi yang kita pesan berbeda, tak jarang pelayanan mengantarkan pesanan bersamaan. Nah, pada kasus inilah pelayan menggunakan teknik Promise.all().

Method Promise.all() dapat menerima banyak promise dalam bentuk array pada parameternya. Kemudian method tersebut akan mengembalikan nilai seluruh hasil dari promise dalam bentuk array.

Contohnya seperti berikut:

1. const promises = [firstPromise(), secondPromise(), thirdPromise()];
3. Promise.all(promises)
4. .then(resolvedValue => {
5. console.log(resolvedValue);
6. });
8. /\* output
9. [ 'first promise', 'second promise', 'third promise' ]
10. \*/

Pada kasus mesin kopi, kita bisa menambahkan proses untuk memanaskan air dan menggiling biji kopi.

1. const boilWater = () => {
2. return new Promise((resolve, reject) => {
3. console.log("Memanaskan air...");
4. setTimeout(() => {
5. resolve("Air panas sudah siap!");
6. }, 2000);
7. })
8. }
10. const grindCoffeeBeans = () => {
11. return new Promise((resolve, reject) => {
12. console.log("Menggiling biji kopi...");
13. setTimeout(() => {
14. resolve("Kopi sudah siap!");
15. }, 1000);
16. })
17. }

Keduanya dapat berjalan bersamaan. Kita akan memanfaatkan Promise.all() untuk menjalankan kedua fungsi di atas sebelum fungsi brewCoffee(). Ubah kode fungsi makeEspresso() menjadi seperti ini:

1. function makeEspresso() {
2. checkAvailability()
3. .then((value) => {
4. console.log(value);
5. return checkStock();
6. })
7. **.then(value => {**
8. **console.log(value);**
9. **const promises = [boilWater(), grindCoffeeBeans()];**
10. **return Promise.all(promises);**
11. **})**
12. .then((value) => {
13. console.log(value)
14. return brewCoffee();
15. })
16. .then(value => {
17. console.log(value);
18. state.isCoffeeMachineBusy = false;
19. })
20. .catch(rejectedReason => {
21. console.log(rejectedReason);
22. state.isCoffeeMachineBusy = false;
23. });
24. }
26. makeEspresso();
28. /\* output
29. Mesin kopi siap digunakan.
30. Stok cukup. Bisa membuat kopi.
31. Memanaskan air...
32. Menggiling biji kopi...
33. [ 'Air panas sudah siap!', 'Kopi sudah siap!' ]
34. Membuatkan kopi Anda...
35. Kopi sudah siap!
36. \*/

Ketika kode di atas dieksekusi, kita perlu menunggu dua (2) detik untuk proses boilWater dan grindCoffeeBeans (durasi terlama dari promise yang dijalankan dari Promise.all()). Ini menunjukkan bahwa semua promise di dalam Promise.all() berjalan bersamaan dan menunggu sampai semua proses di dalamnya selesai dijalankan.

Yang perlu kita perhatikan, urutan nilai yang dihasilkan oleh method ini sesuai dengan promise yang kita tentukan pada parameternya.

1. /\* output
2. [ 'Air panas sudah siap!', 'Kopi sudah siap!' ]
3. \*/

Nilai dari boilWater akan tetap berada di posisi pertama, meskipun proses ini membutuhkan waktu lebih lama.

## Async-await

Pembahasan terakhir mengenai asynchronous process kali ini adalah penggunaan syntax async-await. Apa itu?

Seperti yang kita tahu, penulisan kode asynchronous sedikit berbeda dengan proses synchronous. Contohnya, untuk mendapatkan nilai coffee dari sebuah proses asynchronous, kita tidak dapat melakukannya dengan teknik seperti ini:

1. function makeCoffee() {
2. const coffee = getCoffee(); // async process menggunakan promise
3. console.log(coffee);
4. }
6. makeCoffee();

Melainkan harus seperti ini:

1. function makeCoffee() {
2. getCoffee().then(coffee => {
3. console.log(coffee);
4. });
5. }
7. makeCoffee();

Namun, sejak ES8 (ECMAScript 2017) kita dapat menuliskan asynchronous process layaknya synchronous process dengan bantuan keyword async dan await.

Fitur async/await sebenarnya hanya syntactic sugar. Itu berarti secara fungsionalitas bukanlah sebuah fitur baru dalam JavaScript. Namun, hanya gaya penulisan baru yang dikembangkan dari kombinasi penggunaan Promise dan generator (pembahasan mengenai generator bisa Anda pelajari [di sini](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Generator)). Sehingga, async/await ini tidak dapat digunakan jika tidak ada Promise.

Lantas bagaimana cara menggunakan async/await ini? Pada contoh kode sebelumnya, mari kita lihat juga fungsi getCoffee() dan bagaimana ia mengembalikan promise.

1. const getCoffee = () => {
2. return new Promise((resolve, reject) => {
3. const seeds = 100;
4. setTimeout(() => {
5. if (seeds >= 10) {
6. resolve("Kopi didapatkan!");
7. } else {
8. reject("Biji kopi habis!");
9. }
10. }, 1000);
11. })
12. }

Untuk mendapatkan nilai dari fungsi getCoffee() menggunakan .then(), maka kode kita akan seperti ini:

1. function makeCoffee() {
2. getCoffee().then(coffee => {
3. console.log(coffee);
4. });
5. }
7. makeCoffee();
9. /\* output
10. Kopi didapatkan!
11. \*/

Async-await memungkinkan kita menuliskan proses asynchronous layaknya proses synchronous. Kira-kira kode program kita akan seperti berikut:

1. function makeCoffee() {
2. const coffee = getCoffee();
3. console.log(coffee);
4. }
6. makeCoffee();
8. /\* output
9. Promise { <pending> }
10. \*/

Namun, ketika kode di atas dijalankan hasilnya tidak akan sesuai yang kita harapkan karena fungsi getCoffee() merupakan object Promise. Untuk menunggu fungsi getCoffee() yang berjalan secara asynchronous, tambahkan keyword await sebelum pemanggilan fungsi getCoffee().

1. const coffee = **await** getCoffee();

Kemudian, karena fungsi makeCoffee() sekarang menangani proses asynchronous, maka fungsi  tersebut juga menjadi fungsi asynchronous. Tambahkan async sebelum deklarasi fungsi untuk membuatnya menjadi asynchronous.

1. **async** function makeCoffee() { … }

Dengan perubahan di atas, kita telah berhasil menuliskan proses asynchronous dengan gaya synchronous.

1. async function makeCoffee() {
2. const coffee = await getCoffee();
3. console.log(coffee);
4. }
6. makeCoffee();
8. /\* output
9. Kopi didapatkan!
10. \*/

Keyword async digunakan untuk memberitahu JavaScript supaya menjalankan fungsi makeCoffee() secara asynchronous. Lalu, keyword await digunakan untuk menghentikan proses pembacaan kode selanjutnya sampai fungsi getCoffee() mengembalikan nilai promise resolve.

Walaupun await menghentikan proses pembacaan kode selanjutnya pada fungsi makeCoffee, tetapi ini tidak akan mengganggu proses runtime sesungguhnya pada JavaScript (global). Karena fungsi makeCoffee berjalan secara asynchronous, kita tidak dapat menggunakan await tanpa membuat function dalam scope-nya berjalan secara asynchronous.

### Handle onRejected using async-await

Perlu jadi catatan bahwa await hanya akan mengembalikan nilai jika promise berhasil dilakukan (onFulfilled). Lantas bagaimana jika promise gagal dilakukan (onRejected)? Caranya cukup sederhana. Kembali lagi kepada prinsip synchronous code. Kita dapat menangani sebuah eror atau tolakan dengan menggunakan try...catch.

Ketika menggunakan async/await, biasakan ketika mendapatkan resolved value dari sebuah promise, untuk menempatkannya di dalam blok try seperti ini:

1. async function makeCoffee() {
2. try {
3. const coffee = await getCoffee();
4. console.log(coffee);
5. }
6. }

Dengan begitu kita dapat menggunakan blok catch untuk menangani jika promise gagal dilakukan (onRejected).

1. async function makeCoffee() {
2. try {
3. const coffee = await getCoffee();
4. console.log(coffee);
5. } catch (rejectedReason) {
6. console.log(rejectedReason);
7. }
8. }
10. makeCoffee();
12. /\* output
13. Biji kopi habis!
14. \*/

### Chaining Promise using async-await

Pertanyaan selanjutnya adalah bagaimana melakukan promise berantai bila menggunakan async/await? Jawabannya adalah sama seperti ketika kita mendapatkan nilai dari function yang berjalan secara synchronous.

Dengan pendekatan async-await, maka kode mesin kopi kita akan menjadi seperti ini:

1. async function makeEspresso() {
2. try {
3. await checkAvailability();
4. await checkStock();
5. const coffee = await brewCoffee();
6. console.log(coffee);
7. } catch (rejectedReason) {
8. console.log(rejectedReason);
9. }
10. }
12. makeEspresso();
14. /\* output
15. Membuatkan kopi Anda...
16. Kopi sudah siap!
17. \*/

Terakhir untuk menjalankan beberapa promise sekaligus secara bersamaan dengan Promise.all, kita bisa menuliskannya seperti ini:

1. async function makeEspresso() {
2. try {
3. await checkAvailability();
4. await checkStock();
5. **await Promise.all([boilWater(), grindCoffeeBeans()]);**
6. const coffee = await brewCoffee();
7. console.log(coffee);
8. } catch (rejectedReason) {
9. console.log(rejectedReason);
10. }
11. }

Async/await ini menjadi fitur baru yang sangat berguna. Terlebih untuk kita yang lebih nyaman menangani proses asynchronous dengan menggunakan gaya synchronous.