Advanced Functions

1. Recursive Function
   Fungsi yang **memanggil dirinya sendiri** sampai kondisi tertentu (base case) terpenuhi. Biasanya dipakai untuk masalah yang bisa dipecah menjadi sub-masalah yang mirip.

   Contoh:

   ```
   function factorial(n) {
     if (n === 0) return 1; // base case
     return n * factorial(n - 1); // recursive call
   }
   console.log(factorial(5)); // 120
   ```

   Penjelasan: Setiap pemanggilan factorial akan memanggil dirinya dengan nilai n-1 sampai n sama dengan 0. Proses akan berhenti saat base case terpenuhi.

2. Higher-Order Functions
   Fungsi yang **menerima fungsi lain sebagai argumen** atau **mengembalikan fungsi lain.** Bermanfaat untuk membuat kode lebih fleksibel dan reusable.

   Contoh:

   ```
   function operate(a, b, operation) {
       return operation(a, b);
   }

   function add(x, y) {
       return x + y;
   }

   let resultAdd = operate(3, 4, add);
   console.log(resultAdd); // 7
   ```

   Penjelasan: operate disebut Higher-Order Function karena menerima fungsi lain (add) sebagai argumen dan mengeksekusinya (add(3,4)) untuk menghasilkan 7, HOF membuat kode fleksibel: kita bisa ganti operation dengan fungsi lain (misal pengurangan) tanpa mengubah operate.

3. Closures
   Closure terjadi ketika **fungsi mengingat variabel di lingkup (scope) luar** meskipun scope luar sudah selesai dieksekusi.

   Contoh:

```
function createCounter() {
  let count = 0;
  return function () {
    count++;
    return count;
  };
}
const counter = createCounter();
console.log(counter()); // 1
console.log(counter()); // 2
```

   Penjelasan: Fungsi dalam masih bisa mengakses variabel count walau createCounter sudah selesai dipanggil. Itulah closure.

4. IIFE (Immediately Invoked Function Expression)
   Fungsi yang **langsung dijalankan begitu didefinisikan**, biasanya untuk membuat scope agar variabel tidak bentrok.

   Contoh:

```
(function () {
  console.log("IIFE dieksekusi!");
})();
```

   Penjelasan: Fungsi dibungkus dengan () dan langsung dipanggil dengan (). Sangat berguna untuk isolasi variabel.

5. Module Pattern

   Cara membuat **modul privat dan publik** dengan memanfaatkan closure dan IIFE. Membantu mengatur kode agar tidak semua variabel dan fungsi terekspos ke global.

   Contoh:

```
const myModule = (function () {
  let privateVar = "rahasia";
  function privateFunc() {
    return privateVar;
  }
  return {
    publicFunc: function () {
      return privateFunc();
    },
  };
})();

console.log(myModule.publicFunc()); // "rahasia"
```

   Penjelasan: Variabel dan fungsi yang tidak direturn tetap privat. Hanya publicFunc yang bisa diakses luar.

6. Rest Parameters

   Sintaks ... pada parameter fungsi untuk **mengumpulkan sisa argumen menjadi array.**

   Contoh:

```
function sum(...numbers) {
  return numbers.reduce((a, b) => a + b, 0);
}
console.log(sum(1, 2, 3, 4)); // 10
```

   Penjelasan: Semua argumen setelah parameter lain akan dikumpulkan menjadi array numbers.

7. Generator Functions
   Fungsi spesial yang bisa **pause dan resume eksekusi** menggunakan yield. Ditandai dengan tanda * setelah kata function.

   Contoh:

```
function* simpleGen() {
  yield 1;
  yield 2;
}
const gen = simpleGen();
console.log(gen.next().value); // 1
console.log(gen.next().value); // 2
```

   Penjelasan: yield menghentikan sementara fungsi. next() memulai kembali dari titik terakhir.

8. Function Bind, Call, Apply
   Tiga cara untuk mengatur **this** dan argumen saat memanggil fungsi.
   - **call**: jalankan fungsi langsung, argumen dipisah koma.
   - **apply**: sama seperti call tapi argumen berupa array.
   - **bind**: mengembalikan fungsi baru dengan **this** tetap.

   Contoh:

```
const person = { name: "Rifa" };
function greet(greeting) {
  console.log(`${greeting}, ${this.name}`);
}
greet.call(person, "Halo"); // Halo, Rifa
greet.apply(person, ["Hi"]); // Hi, Rifa
const bound = greet.bind(person);
bound("Hello"); // Hello, Rifa
```

   Penjelasan: Di dalam fungsi greet, **this** adalah **objek yang menjadi konteks eksekusi fungsi saat itu**. Dengan call, apply, dan bind, kita **secara eksplisit menetapkan** bahwa konteks tersebut adalah **objek person**, sehingga:

```
this.name  // sama dengan person.name
```

   Jadi ketika greet dijalankan melalui call/apply/bind, this merujuk ke **person**, bukan ke window atau undefined.

9. Composition

   Teknik untuk **menggabungkan beberapa fungsi kecil menjadi fungsi besar**. Membuat kode lebih modular dan mudah dirawat.

   Contoh:

```
const add = (x) => x + 1;
const double = (x) => x * 2;

const compose = (f, g) => (x) => f(g(x));
const addThenDouble = compose(double, add);

console.log(addThenDouble(3)); // 8
```

   Penjelasan: compose(double, add) artinya jalankan add dulu, lalu hasilnya masuk ke double.

10. Arrow Functions

    Cara ringkas menulis function expression. Tidak memiliki this sendiri, jadi this akan mengacu ke lingkup luar (lexical this).

    Contoh:

```
const square = (x) => x * x;
let result = square(4);
console.log(result); // 16
```

    Penjelasan: (x) => x * x adalah arrow function yang menerima satu parameter dan langsung mengembalikan hasil kuadratnya. Tidak perlu menulis function dan return.

11. Function Expressions

    Fungsi yang didefinisikan sebagai ekspresi dan disimpan di variabel.

    Contoh:

```
const square = function (x) {
  return x * x; };
let result = square(3);
console.log(result); // 9
```

    Penjelasan: Fungsi dibuat tanpa nama khusus (bisa juga dinamai) dan disimpan ke variabel square. Dipanggil seperti variabel.

12. Anonymous Functions

    Function expression yang tidak memiliki nama. Umum dipakai sebagai argumen fungsi lain.

    Contoh:

```
let greet = function (name) {
  console.log("Hello, " + name + "!");
};
greet("Alice"); // Hello, Alice!
```

    Penjelasan: Fungsi tidak diberi nama, hanya disimpan ke variabel greet dan langsung dipanggil.

Asynchronous functions

1. Synchronous & Asynchronous Functions.
   Synchronous
   - **Urut & blokir**: Baris kode dijalankan satu per satu.
   - Baris berikutnya **menunggu** baris sebelumnya selesai.
   - Contoh: let x = 5 + 3; console.log(x);
     Hasil langsung keluar tanpa jeda.

   Asynchronous
   - Kode tidak menunggu baris sebelumnya selesai.
   - Program bisa lanjut ke baris lain sementara proses itu berlangsung.
   - Saat ada proses yang memakan waktu (misal: request ke server, baca file), JavaScript akan melanjutkan baris lain dulu.
   - Contoh: setTimeout, request API.

2. Callbacks
   Callback adalah fungsi yang diteruskan sebagai argumen ke fungsi lain. Callback umumnya digunakan dalam pemrograman asinkron untuk mengeksekusi kode setelah operasi tertentu selesai.

   Contoh:

```
setTimeout(() => {
  console.log("Selesai menunggu 2 detik");
}, 2000);
```

   Di sini setTimeout akan memanggil fungsi di dalamnya setelah 2 detik. Kelemahan cara ini: jika banyak proses berantai, callback bisa bersarang dan sulit dibaca (sering disebut callback hell).

3. Promises
   Promise adalah cara yang lebih mudah untuk menangani operasi asinkron. Promise merepresentasikan nilai yang mungkin tersedia sekarang, di masa mendatang, atau bahkan tidak akan pernah tersedia.

Promise adalah objek yang mewakili status proses async:

- pending: masih berjalan
- fulfilled: selesai sukses resolve
- rejected: gagal reject

Cara pakai:

```
const janji = new Promise((resolve, reject) => {
  // proses async
  if (berhasil) resolve("OK");
  else reject("Error");
});
```

Penanganan hasil:

```
janji
  .then(result => console.log(result))  // saat resolve
  .catch(err => console.error(err))     // saat reject
  .finally(() => console.log("Selesai")); // selalu dijalankan
```

Kelebihan: alur lebih rapi, tidak bersarang seperti callback.

4. fetch API

fetch() adalah fungsi bawaan JavaScript untuk melakukan HTTPrequest dan otomatis mengembalikan Promise. Langkah kerja:

- Panggil fetch(url).
- Promise dari fetch berisi respons HTTP.
- Gunakan .json() untuk mengubah body respons menjadi data JavaScript.

Contoh:

```
fetch('https://jsonplaceholder.typicode.com/users')
    .then(response => response.json())
    .then(data => console.log(data))
    .catch(error => console.log("Error:", error));
```

Penjelasan: fetch meminta data ke URL. response.json() mengubah data menjadi objek JavaScript. Hasilnya diproses di .then(data => ...). Jika gagal, .catch() menangkap error.

5. Async – Await

Async function

- Menandakan fungsi selalu mengembalikan Promise, walau kamu hanya menulis return data;.
- Di dalam fungsi async, kamu bisa menggunakan await.

Async function

- Berfungsi "menunggu" Promise selesai.
- Hanya bisa dipakai di dalam async function.
- Jika Promise resolve, await mengembalikan hasilnya.
- Jika Promise reject, akan melempar error yang bisa ditangkap dengan try...catch.

Keuntungan dibanding `.then().catch()`

- Kode lebih rapi, alurnya seperti sinkron.
- Penanganan error cukup pakai try...catch, tidak perlu rantai .then() yang panjang.

Contoh:

```
async function getUsers() {
  try {
    const response = await
fetch("https://jsonplaceholder.typicode.com/users");
    const data = await response.json();
    console.log(data);
  } catch (error) {
    console.error("Terjadi error:", error);
  }
}

getUsers();
```

Penjelasan:

- **async function getUsers()** → Mendeklarasikan fungsi getUsers sebagai *asynchronous* sehingga bisa memakai await.
- **await fetch("https://jsonplaceholder.typicode.com/users")** → fetch() mengembalikan Promise yang isinya *HTTP response*. → await membuat JavaScript "menunggu" sampai *request* selesai, lalu menyimpan hasilnya di response.

- **await response.json()** → response.json() juga mengembalikan Promise (proses parsing JSON). → await menunggu proses parsing selesai dan hasilnya disimpan ke data.
- **console.log(data)** → Menampilkan array user dari API ke console.
- **try { ... } catch (error) { ... }** → Jika ada masalah saat fetch (misal koneksi internet putus), eksekusi langsung masuk ke catch dan mencetak pesan error.
- **getUsers();** → Memanggil fungsi untuk menjalankan proses pengambilan data