Async Programming

- Synchronous Programming (sync.dart)
- Asynchronous Programming
  - async1.dart
  - async2.dart
  - async3.dart
  - async4.dart
  - async_then1.dart
  - async_then2.dart
- Comparison with the JavaScript async
  - JavaScript Promise
- Dart vs JavaScript Async Programming

- Key Question: How to use computer resources effectively?

1. Don't let CPU sit idle: Use async programming to avoid blocking
2. Parallelize independent tasks
   Run multiple operations simultaneously.

Synchronous Programming (sync.dart)

- This blocks the entire program for 3 seconds.

```dart
void fetchUserData() {
  print("1. Starting request...");
  sleep(Duration(seconds: 3));
  print("2. User data received!");
  print("3. Continue with other tasks");
}
```

- Application freezes during network calls, file operations, or database queries.

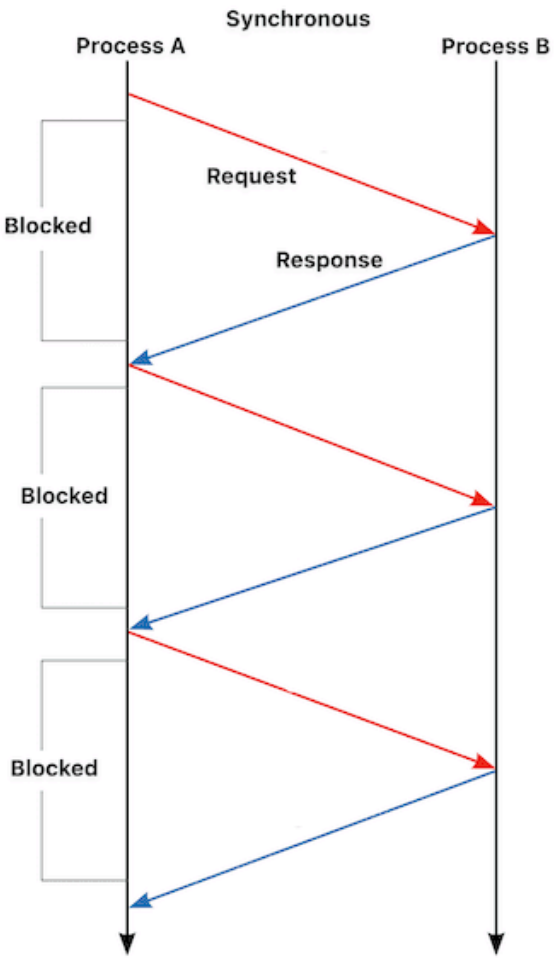
```
void main() {
  fetchUserData(); // wait for 3 seconds
  print("4. This waits until above is done");
}
```

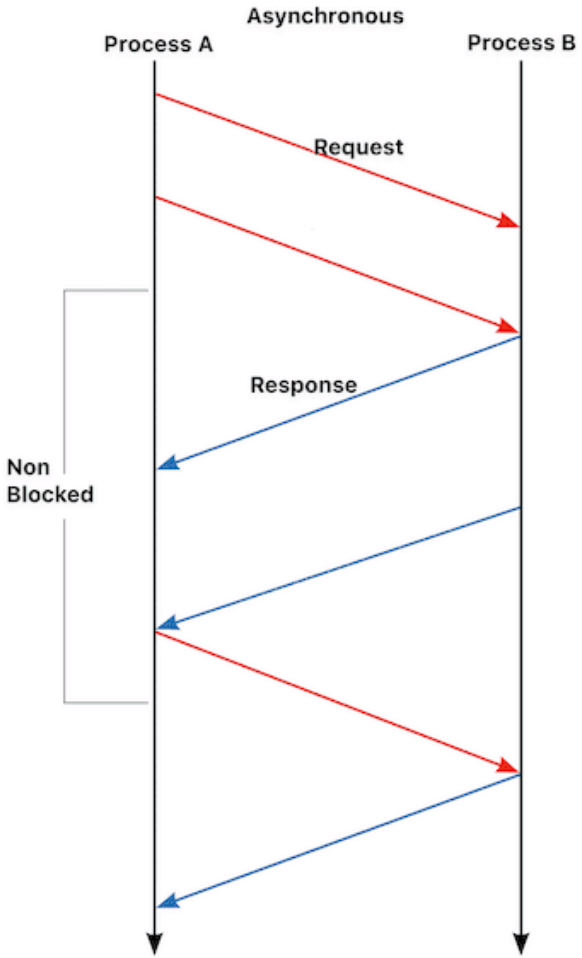
```
1. Starting request...
2. User data received!
3. Continue with other tasks
4. This waits until the above is done
```

- In this case, we need to wait for `fetchUserData()` to use the results of the function.
- However, in most cases, we do not have to wait for the `fetchUserData()` to end.
- This can cause a serious waste of computing resources.

Asynchronous Programming

- To address this issue, we use Non-blocking Operations: async programming.
- Advantages of async programming:
  - Responsive UI
  - Better performance
  - Multiple operations simultaneously


Synchronous
Process A
Process B
Request
Response
Blocked
Blocked
Blocked


Asynchronous
Process A
Process B
Request
Response
Non
Blocked

Three keywords: await, async, and Future

```dart
import 'dart:async';

Future<void> fetchUserData() async {
  print("1. Starting request...");
  // This doesn't block! Other code can run
  await Future.delayed(Duration(seconds: 3));
  print("2. User data received!");
}
```

await

- We specify that any code after the `wait` should wait for the computation to finish.

```
await Future.delayed(Duration(seconds: 3));
print("2. User data received!");
```

async and Future

- When a function has `await` in it, the function should be decorated as `async`.
- The return value should be `Future<T>`.

```
Future<void> fetchUserData() async {
  ...
  await Future.delayed(Duration(seconds: 3));
  ...
}
```

async1.dart

- In this example, print("3 ...") does not wait for the "fetchUserData()" to finish.

```dart
Future<void> fetchUserData() async {
  print("1. Starting request...");
  await Future.delayed(Duration(seconds: 3));
  print("2. User data received!");
}
void main() {
  fetchUserData();
  print("3. This waits until above is done");
}
```

How does this work

```
await Future.delayed(Duration(seconds: 3));
print("2. User data received!");
```

- The async function "Future.delayed" is moved to an event queue.
- The next code (print("2 ...)) waits for the "Future.delayed" to return, as it should `await`.

```
fetchUserData(); // async
print("3. This waits until above is done");
```

- The `async` function "fetchUserData()" returns a Future.
- There is no `await`, so the next code (print("3...)) is run.

```
> dart async1.dart
1. Starting request...
3. This waits until the above is done
2. User data received!
```

- The sequence is "1 -> 3 -> 2" as the "2" should wait for "Future.delayed" to return, but "3" runs asynchronously.

async2.dart

- If print("3 ...") must wait for the `fetchUserData` to finish, we need `async` / `await` again.

```dart
Future<void> fetchUserData() async {
  print("1. Starting request...");
  await Future.delayed(Duration(seconds: 3));
  print("2. User data received!");
}
void main() async {
  await fetchUserData();
  print("3. This waits until above is done");
}
```

```
> dart async2.dart
1. Starting request...
2. User data received!
3. This waits until the above is done
```

async3.dart

- With async/await, we can return a value.
- We should be careful to use the return value of async function.

```dart
Future<int> getNumber() {
  return Future.delayed(Duration(seconds: 1), () {
    return 42;
  });
}
void main() {
  var res = getNumber(); // Future<int>
  // ERROR!
  print(res);
}
```

- "Instance of 'Future<int>'" is printed, as when the "print(res)" is executed, the Future<int> value is not set yet.

async4.dart

- To resolve this issue, we should add `await` and `async` to wait for the "getNumber()" to return a value.

```
void main() await {
  var res = async getNumber();
  print(res);
}
```

Callback using then

- We can use the `then` function to use the callback function.
- Using `then`, we can remove the `await/async`.

```
v = await function(); f(v)

function().then((v) => f(v))
```

async_then1.dart

```dart
Future<void> fetchUserData() async {
  print("1. Starting request...");
  await Future.delayed(Duration(seconds: 3));
  print("2. User data received!");
}

void main() {
  fetchUserData().then((_)
    => print("3. This waits until awaitbove is done"));
}
```

- The (_) means there is no argument passed.

async_then2.dart

```
Future<int> getNumber() {
  return Future.delayed(Duration(seconds: 1), () {
    return 42;
  });
}

void main() {
  getNumber().then((res) => print(res));
}
```

- We can pass the argument.

Comparison with the JavaScript async

- JavaScript async is almost the same as Dart async
  - `async` and `await` makes the function async.
  - The `Promise` object is returned.

JavaScript Promise

- An object representing the eventual completion of an async operation.
- A Container for Future Values.
- The `resolve` function is used for returning a value, and the `reject` function is used for failure operation.

```
function fetchUserData(userId) {
  return new Promise((resolve, reject) => {
    if (userId > 0) {
      // Success case
      setTimeout(() => {
        // Promise succeeds with this value
        resolve("John Doe");
      }, 1000);
    } else {
      // Failure case
      // Promise fails with this error
      reject(new Error("Invalid user ID"));
    }
  });
}
```

- It uses *resolve* to return value, and *reject* to throw an error.

Dart:

```dart
Future<String> fetchUserData(int userId) async {
  if (userId > 0) {
    // Success case
    await Future.delayed(Duration(seconds: 1));
    // Future completes with this value
    return "John Doe";
  } else {
    // Failure case
    // Future completes with an error
    throw Exception("Invalid user ID");
  }
}
```

- It uses normal if/else to return a value or throw an exception.

JavaScript await

- We use `await` to wait for the `Promise` to return a value (using resolve) or reject with an error.

```
// Using it:
async function main() {
  try {
    const user = await fetchUserData(123); //   Gets "John Doe"
    console.log(user);
  } catch (error) {
    console.log(error.message); // Handles any   rejection
  }
}
```

Dart await

```dart
void main() async {
  try {
    final user = await fetchUserData(123); //   Gets "John Doe"
    print(user);
  } catch (error) {
    print(error.toString()); // Handles any   exception
  }
}
```

- The Dart code is almost identical.
- For Dart and JavaScript, any function that has `async` should have `await`.

- Without the `await`, the function returns immediately without any results.
- Just like Dart, this may cause an issue.

```
// Without await - Promise object returned immediately
const promise = fetchUserData();
console.log(promise); // Promise { <pending> }
```

Dart vs JavaScript Async Programming

- Same concept, different syntax.
- Promise -> Futures
- async function -> async { ... }
- resolve/reject vs if/else