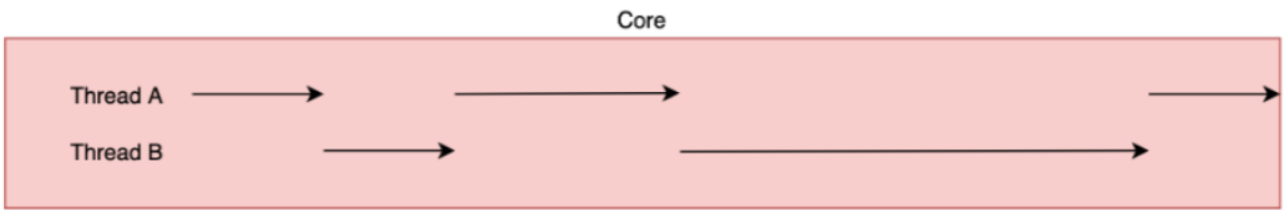
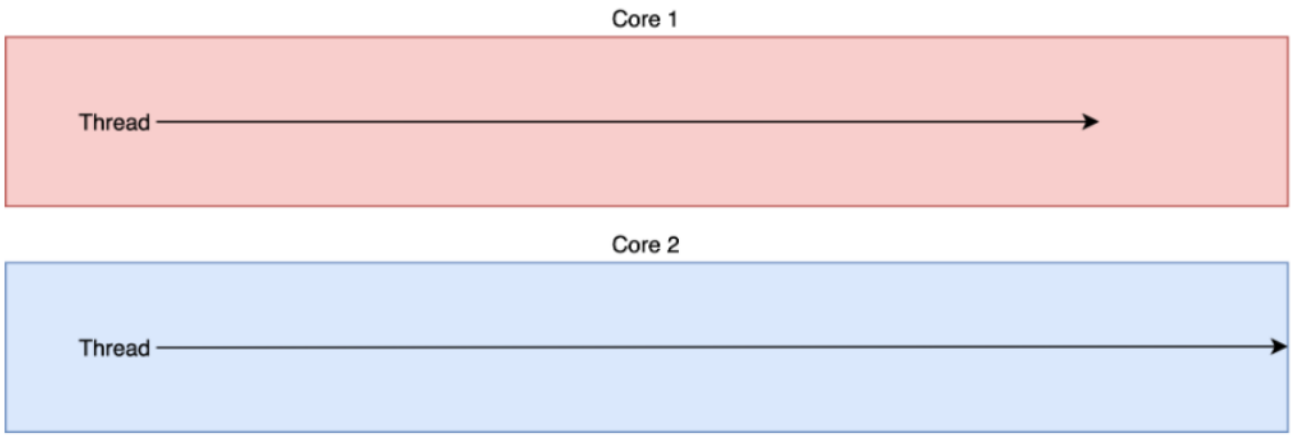
비동기프로그래밍

- 동기(Synchronous)
  - 작업에 대한 요청과 그 결과가 동시에 일어나다는 약속으로 요청받은 작업이 끝날 때까지 다음 작업을 수행할 수 없다.
- 비동기(Asynchronous)
  - 작업에 대한 요청과 결과가 동시에 일어나지 않을 거라는 약속으로 하나의 요청에 따른 응답이 돌아오기 전에 다른 요청을 처리할 수 있다.

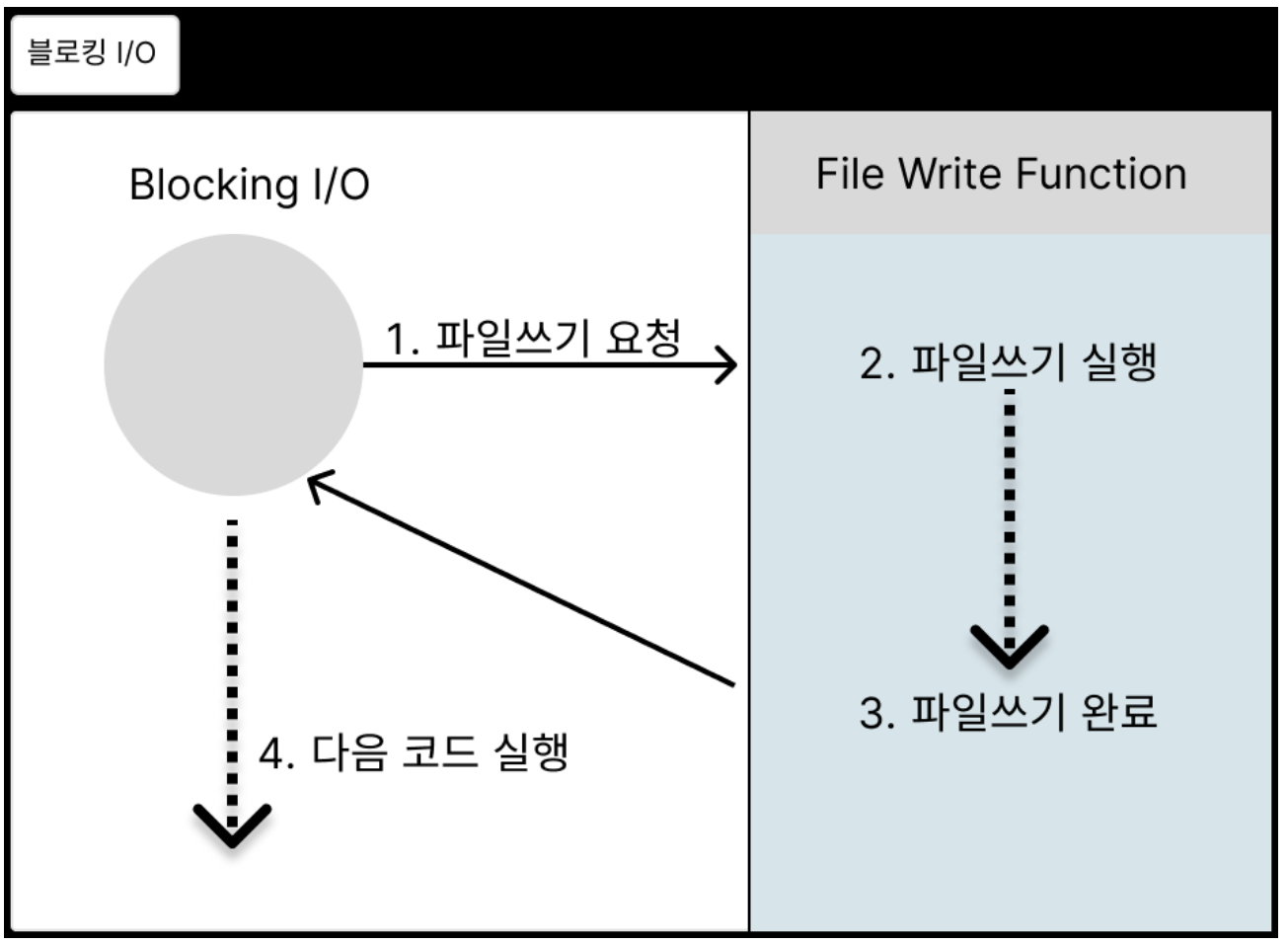
> 동시성 VS 병렬성
>
> - 동시성(Concurrency)
>   - 논리적 용어로 하나의 시스템이 여러 작업을 동시에 처리하는 것처럼 보이게 하는 것
>   - 실질적으로 한 번에 하나의 작업만을 처리
>   - 싱글 코어에서 멀티 쓰레드(Multi thread)를 동작



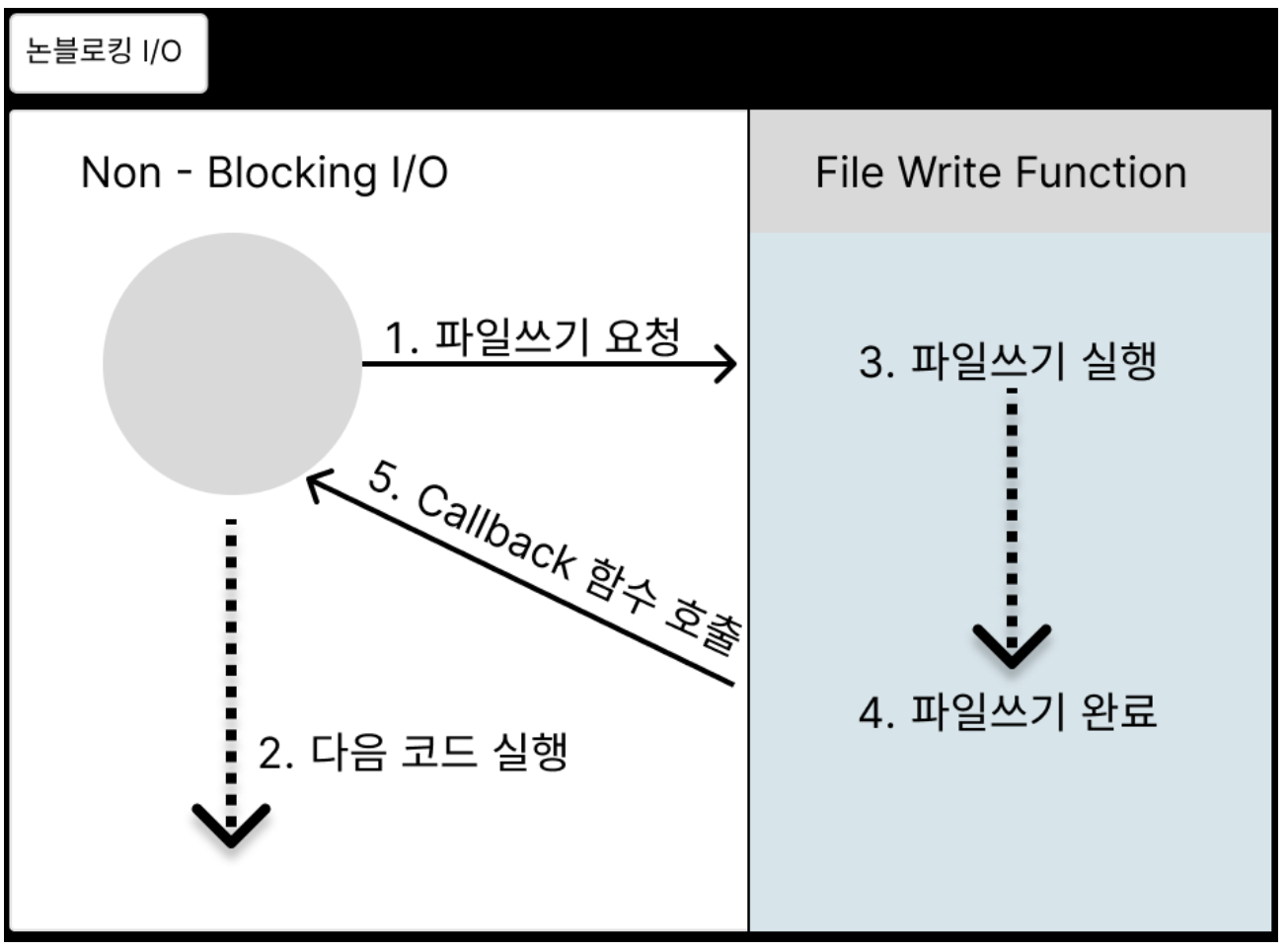
- 병렬성(Parallelism)
  - 물리적 영어로 여러 작업을 실제로 동시에 처리하는 것
  - 멀티 코어에서 멀티 쓰레드(Multi thread)를 동작



> 블로킹 VS 논블로킹
>
> - 블로킹(blocking)
>   - 새로운 작업을 요청할 경우 현재 진행 중인 작업을 중단시키고 요청한 작업이 끝난 후 다시 중단된 작업을 진행



- 논블로킹(non-blocking)
  - 새로운 작업을 요청해서 작업을 진행하는 동안 현재 진행 중인 작업을 중단시키지 않고 그대로 진행



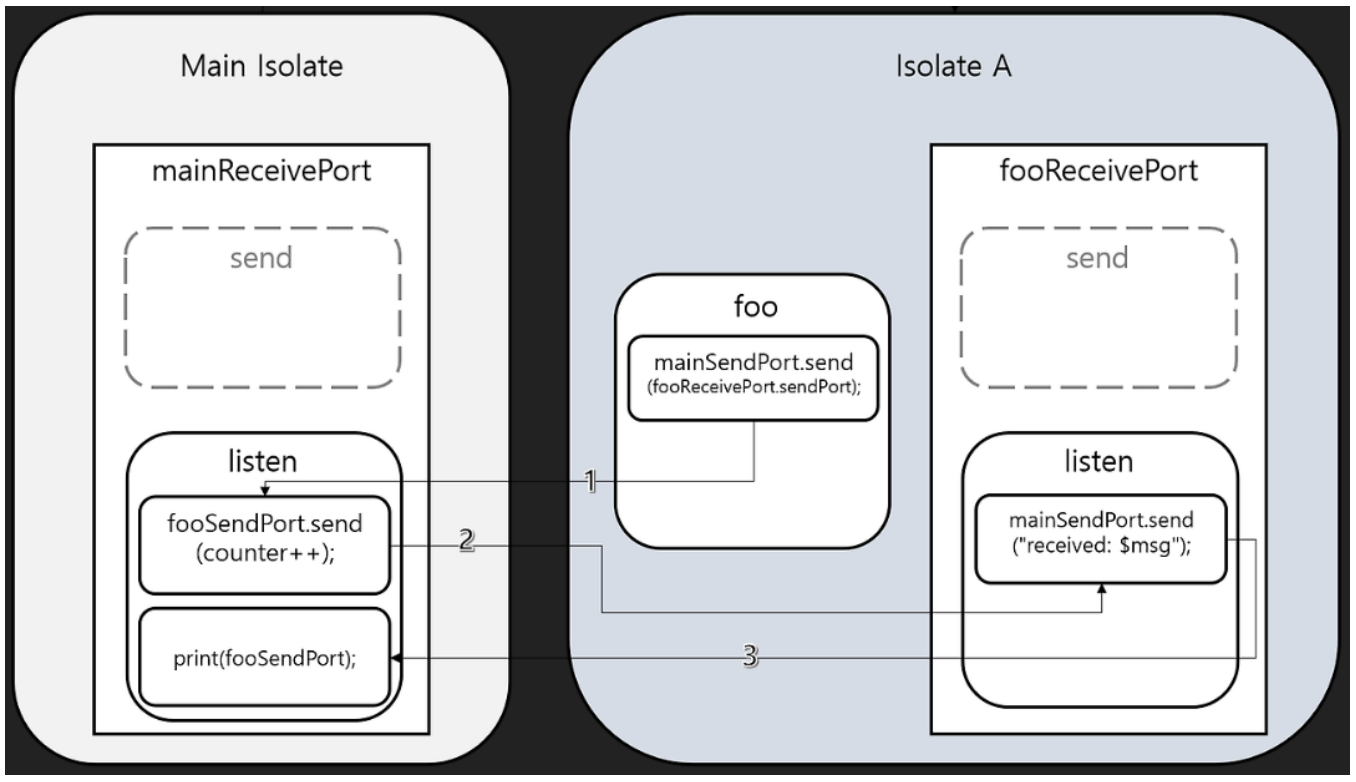
1. isolate
   a. Dart(다트)의 모든 코드가 실행되는 공간으로 스레드(thread)와 달리 메모리를 공유하지 않음
   b. 싱글 스레드를 가지고 있고 이벤트 루프를 통해 작업을 처리
      - 이벤트 루프를 통해 비동기 작업을 지원하므로 동시에 다른 작업을 할 수 있도록 해 효율을 높임
   c. 기본 isolate인 main isolate가 런타임에 실행되며 필요에 따라 추가 isolate를 실행할 수 있음

```dart
import 'dart:isolate';

void main() {
  // 새로운 isolate 생성
  Isolate.spawn(isolateTest, 1);
```

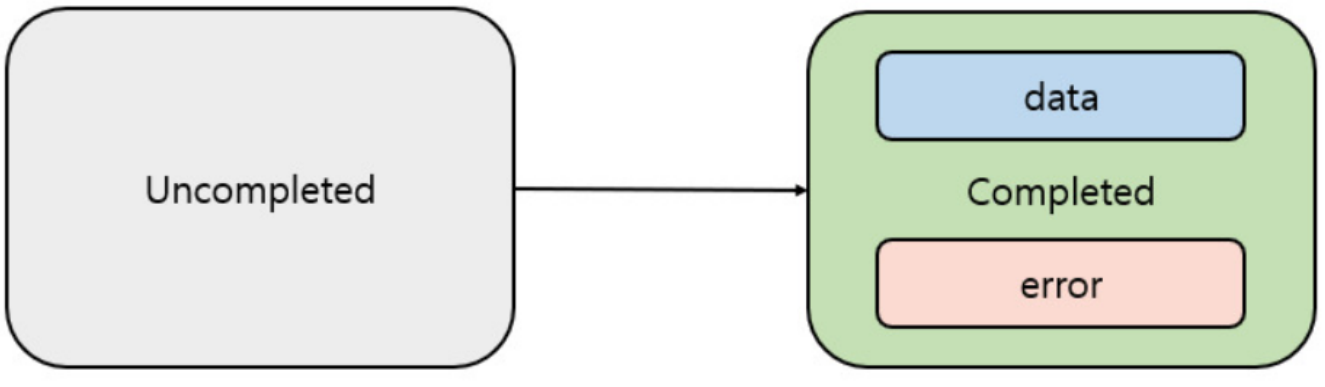
```
  Isolate.spawn(isolateTest, 2);
  Isolate.spawn(isolateTest, 3);
}

isolateTest(var m) {
  print('isolate no.$m');
}
```

d. 서로 다른 isolate는 메모리를 공유하지 않으므로 함께 작업하는 경우 message를 주고 받음

```
import 'dart:isolate';

void main() {
  int counter = 0;

  // isolate 간 메세지를 주고받기 위해 ReceiveProt 객체 생성
  ReceivePort mainReceivePort = new ReceivePort();

  // main isolate의 ReceiveProt 객체에 메세지가 전달된 경우 처리
  mainReceivePort.listen((fooSendPort){
    // SendPort : ReceivePort 객체에 메세지를 보내기 위한 클래스
    if(fooSendPort is SendPort){
      // #2
      // 해당 SendPort 객체를 생성한 ReceivePort에 메세지를 보냄
      fooSendPort.send(counter++);
    }else{
        // #4
      print(fooSendPort);
    }
  });

    for(int i = 0; i < 5; i++){
      // 새로운 isolate 생성
      Isolate.spawn(foo, mainReceivePort.sendPort);
    }

}

foo(SendPort mainSendPort){
  // 새로운 isolate 마다 개별 ReceivePort 객체 생성
  ReceivePort fooReceivePort = new ReceivePort();
  // #1
  // main isolate의 ReceivePort 객체에
  // 새로운 isolate의 ReceivePort 객체의 SendPort 객체 전달
  mainSendPort.send(fooReceivePort.sendPort);

  // #3
  // 새로운 isolate의 ReceiveProt 객체에 메세지가 전달된 경우 처리
  fooReceivePort.listen((msg){
    mainSendPort.send('receive: $msg');
  });
}
```



2. future
    a. 어떤 작업의 결과 값을 나중에 받기로 약속한 것으로 요청한 작업의 결과를 기다리지 않고 다음 작업으로 바로 진행
    ⇒ 작업이 완료되면 결과를 받는 방식으로 비동기로 작업을 처리
    b. 상태



1. Uncompleted(미완료)
    - Future 객체를 만들어서 작업을 요청한 상태
2. Completed(완료)
    - 요청한 작업이 완료된 상태
    - 결과
        1. data : 정상적으로 작업을 수행해 결과값을 반환하고 작업완료
        2. error : 작업 중 문제 발생 시 에러와 함께 작업 완료

c. 진행
    i. Future 객체를 생성한다.
    ii. 작업을 시작하면 Uncompleted Future가 Event Queue에 들어간다.
    iii. 해당 작업이 완료되기 전까지는 다른 작업들이 Event Quere에 들어가고 Event Loop에 의해 꺼내져 처리된다.
    iv. Future 내부에 등록된 작업이 완료되면 Completed Future가 Event Quere에 들어간다.
    v. Event Loop에 의해 Completed Future가 선택되면 해당 Future가 가진 결과값이나 에러를 처리한다.

```
main() {
  print('start');

  // Future 객체 생성 : Uncompleted
  Future<String> myFuture = new Future(() {
```

```
    for (int i = 0; i < 10000000000; i++) {
      // 10,000,000,000
      // Ten billion(십억) times. My PC takes about four seonds.
    }
    // 해당 작업이 정상적으로 완료되면 반환
    return 'I got lots of Data!';
    // 작업 중 에러가 발생할 경우 가정
    //return throw Exception('Failed : data is too many');
  });

  // Future 객체 내부에 등록된 작업이 완료될 경우 처리
  // future.then(data 일 경우, error 일 경우);

  // 람다식
  myFuture.then((data) => print(data), onError: (e) => print(e));

  /* 일반
  myFuture.then((data) {
    // 정상적으로 완료된 경우 처리 함수
    print(data);
  }, onError: (e) {
    // 에러가 난 경우 처리 함수
    print(e);
  });
  */

  print('do sometiong');
}
```

- data

```
start
do sometiong
I got lots of Data!
```

- error

```
start
do sometiong
Exception: Failed : data is too many
```

```
main() {
  print('start');

  Future<String> myFuture = new Future(() {
    for (int i = 0; i < 10000000000; i++) {
      // 10,000,000,000
      // Ten billion(십억) times. My PC takes about four seonds.
    }
    return 'I got lots of Data!';
```

```
  });

  // onError 사용
  //myFuture.then((data) => printFutureResult(data), onError: (e) => print(e));

  // onError 대신 catchError 메소드 사용
  // Future 객체 내부의 에러 뿐만 아니라 then 메서드에서 발생한 에러 포함

  myFuture.then((data) {
    printFutureResult(data);
  }).catchError((e) => print(e));


  print('do sometiong');
}

printFutureResult(result) {
  print(result);
  return throw Exception('Process Error');
}
```

- onError

```
start
do sometiong
I got lots of Data!
DartPad caught unhandled _Exception:
Exception: Process Error
https://storage.googleapis.com/nnbd_artifacts/3.5.0/dart_sdk.js 12006:11  Object.throw_
blob:null/1aa48c6e-bf44-4e02-a3a0-26f066a5fae8 390:22                     Object.printFutureResult
blob:null/1aa48c6e-bf44-4e02-a3a0-26f066a5fae8 385:54                     <fn>
blob:null/1aa48c6e-bf44-4e02-a3a0-26f066a5fae8 1224:98                     <fn>
blob:null/1aa48c6e-bf44-4e02-a3a0-26f066a5fae8 1276:16                     [_run]
blob:null/1aa48c6e-bf44-4e02-a3a0-26f066a5fae8 1224:80                     <fn>
https://storage.googleapis.com/nnbd_artifacts/3.5.0/dart_sdk.js 48518:46  _rootRunUnary
https://storage.googleapis.com/nnbd_artifacts/3.5.0/dart_sdk.js 47565:14
async._CustomZone.new.runUnary
https://storage.googleapis.com/nnbd_artifacts/3.5.0/dart_sdk.js 43291:29
_FutureListener.then.handleValue
https://storage.googleapis.com/nnbd_artifacts/3.5.0/dart_sdk.js 43903:49
handleValueCallback
https://storage.googleapis.com/nnbd_artifacts/3.5.0/dart_sdk.js 43941:17
_Future._propagateToListeners
```

- catchError

```
start
do sometiong
I got lots of Data!
Exception: Process Error
```

d. async 와 await

```
main() {
  // 각 함수는 비동기로 처리
  futureTest(1);
  futureTest(2);
```

```
}

futureTest(no) async {
  // 함수 내부 작업은 순서대로 작업하도록 보장하나
  // async 가 선언된 함수 자체는 비동기 작업
  print('$no start');

  var myFuture = await getData(); // 비동기 작업을 동기로 처리보장
  print('$no result : $myFuture');

  print('$no do someting');
}

Future<String> getData() {
  return new Future(() {
    for (int i = 0; i < 10000000000; i++) {
      // 10,000,000,000
      // Ten billion(십억) times. My PC takes about four seonds.
    }
    return 'I got lots of Data!';
  });
}
```

```
1 start
2 start
1 result : I got lots of Data!
1 do someting
2 result : I got lots of Data!
2 do someting
```

```
main() async {
    // 각 함수는 순서대로 처리
  await futureTest(1);
  await futureTest(2);
}

futureTest(no) async {
  // 함수 내부 작업은 순서대로 작업하도록 보장하나
  // async 가 선언된 함수 자체는 비동기 작업
  print('$no start');

  var myFuture = await getData(); // 비동기 작업을 동기로 처리보장
  print('$no result : $myFuture');

  print('$no do someting');
}

Future<String> getData() {
  return new Future(() {
    for (int i = 0; i < 10000000000; i++) {
      // 10,000,000,000
```

```
      // Ten billion(십억) times. My PC takes about four seonds.
    }
    return 'I got lots of Data!';
  });
}
```

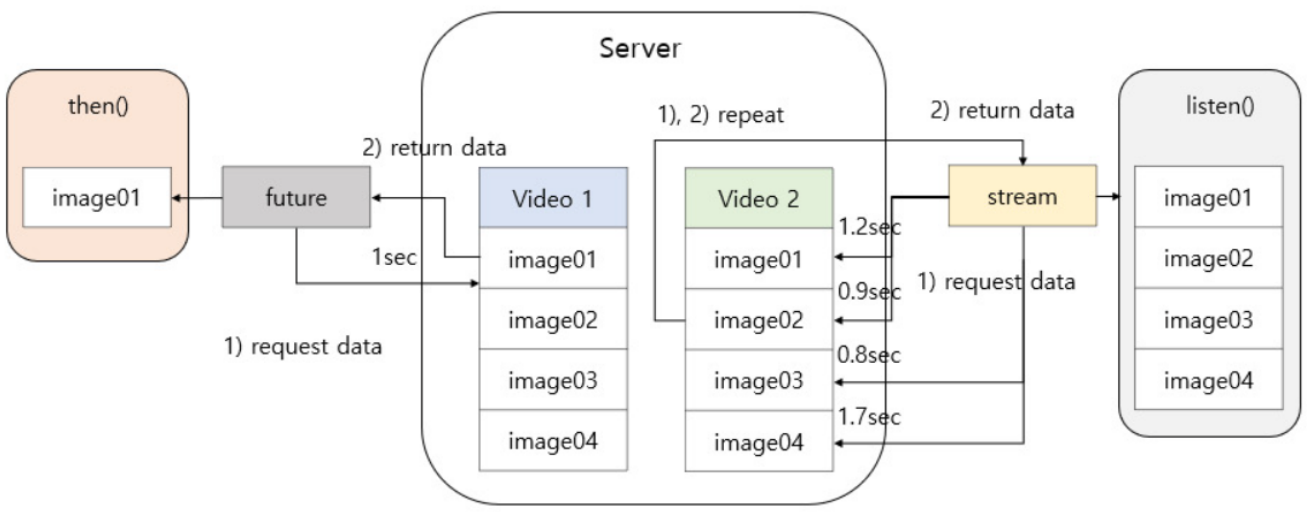
```
1 start
1 result : I got lots of Data!
1 do someting
2 start
2 result : I got lots of Data!
2 do someting
```

3. stream

a. 어떤 작업의 결과 값을 나중에 받기로 약속한 것으로 요청한 작업의 결과를 기다리지 않고 다음 작업으로 바로 진행
⇒ 작업이 완료되면 결과를 받는 방식으로 비동기로 작업을 처리하며 future와 달리 연속된 작업들의 결과를 처리할 수 있음

- 동작방식

> 가상의 동영상 스트리밍 환경을 가정
>
> - 실시간 스트리밍을 위한 동영상 파일은 여러장의 이미지 파일로 구성됨
> - 해당 파일은 서버에 존재하며 서버는 한 번에 이미지 파일 한 장씩 전송
> - 클라이언트(future or stream)에서는 한 번에 이미지 파일 한 장씩 수신
> - 각 이미 파일은 최소 2초 이내에 가져와야 원할한 재생이 가능

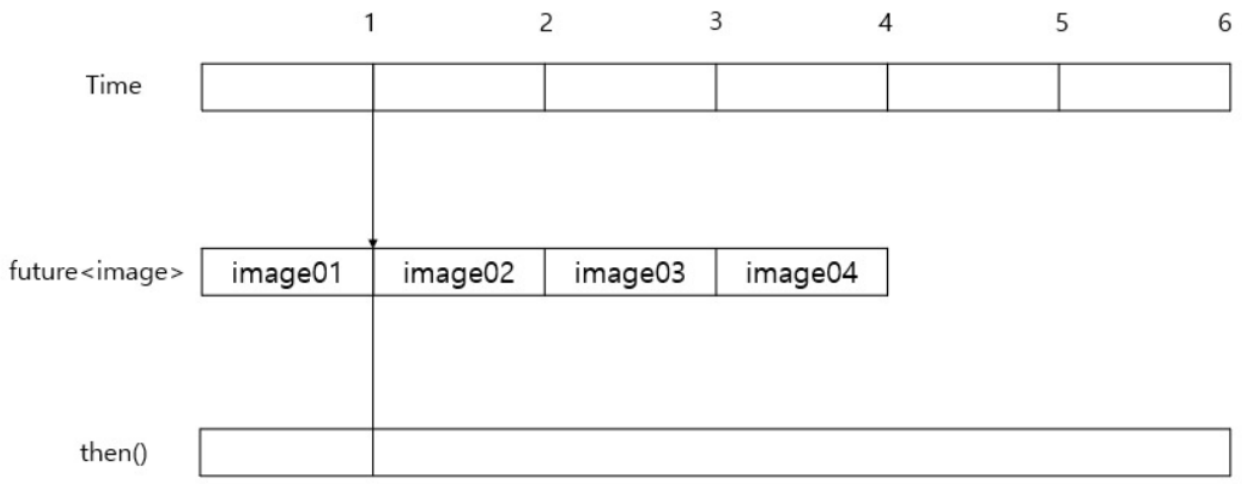


- future

```
import 'dart:io';

void main() {
    // 서버가 가지고 있는 이미지로 가정
  List<String> messages = [
    'Hello !! ',
    'Welcome ',
    'to ',
    'Flutter World!'
  ];
  print('start');
```

```
  // 서버에서 순차적으로 이미지를 가져오는 stream 객체 생성
  var future = Future((){
    for(int i = 0; i < messages.length; i++){
      return messages[i];
    }
  });

  // 서버에서 이미지를 가져왔을 때 처리하는 부분
  future.then((x) => stdout.write(x), onError: (e)=>print(e));

  print('do someting');
}
```

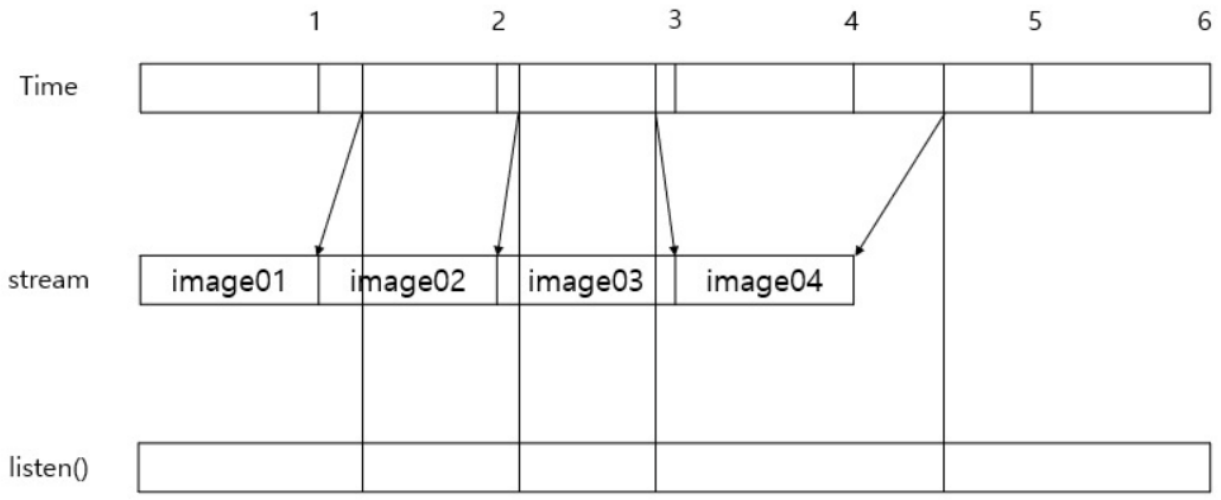
```
start
do someting
Hello !!
Exited.
```



- stream

```
import 'dart:io';

void main(List<String> arguments) {
  // 서버가 가지고 있는 이미지로 가정
  List<String> messages = [
    'Hello !! ',
    'Welcome ',
    'to ',
    'Flutter World!'
  ];
  print('start');

  // 서버에서 순차적으로 이미지를 가져오는 stream 객체 생성
  var stream = Stream.fromIterable(messages);

  // 서버에서 이미지를 가져왔을 때 처리하는 부분
  stream.listen((x) => stdout.write(x));

  print('do someting');
}
```

```
start
do someting
Hello !! Welcome to Flutter World!
Exited.
```



- 기본

```
print('start');

// 가장 단순한 형태
var stream = Stream.value(100).listen((x) => print('getData : $x'));

print('do someting');
```

```
start
do someting
getData : 100
```

```
print('start');

// 특정 주기로 반복적으로 이벤트를 발생
// stream 객체 생성
var stream =
    Stream.periodic(const Duration(seconds: 1), (i) => i * i).take(5);

// 작업 요청 : 객체 생성과 요청을 분리
stream.listen((x) => print('print : $x'));

print('do someting');
```

```
start
do someting
print : 0
print : 1
print : 4
print : 9
print : 16
```

```
void main() {
  print('start');

    // 컬렉션을 기준으로
  Stream.fromIterable(['one', '2.5', 'three', 4, 5])
      .listen((dynamic x) => print('fromIterable : $x'));

    // Future를 기준으로
  Stream.fromFuture(getData()).listen((x) => print('fromFuture : $x'));

  print('do something');
}

Future<String> getData() async {
  return Future.delayed(Duration(seconds: 3), () => 'after 3 seconds');
}
```

```
start
do something
fromIterable : one
fromIterable : 2.5
fromIterable : three
fromIterable : 4
fromIterable : 5
fromFuture : after 3 seconds
```

- StreamController
  - 비동기 함수에 의해 전달되는 형태가 아니라 Stream에 이벤트를 직접 지정할 경우 사용

```
import 'dart:async';

void main() {
  print('start');

    // StreamController 객체 생성, 멤버 변수로 Stream 객체를 포함.
  StreamController streamCtrl = StreamController();
  // 멤버 변수로 등록된 stream 객체에 등록할 listen을 등록, 기본적으로 1개만 등록가능
  streamCtrl.stream.listen((x)=>print(x));

  // StreamController 객체의 add()를 통해 이벤트 등록
  streamCtrl.add(100);
  streamCtrl.add('test');
```

```
  streamCtrl.add(200);
  streamCtrl.add(300);
  streamCtrl.close();


  print('do something');
}
```

```
start
do something
100
test
200
300
```

- Broadcast
  - StreamController 객체에 2개 이상의 listen을 등록할 경우 사용

```
import 'dart:async';

void main() {
  print('start');

    // StreamController.broadcast()를 통해 StreamController 객체 생성
  StreamController streamCtrl = StreamController.broadcast();
  // 멤버 변수로 등록된 stream 객체에 등록할 listen을 등록
  streamCtrl.stream.listen((x)=>print('listen 1 : $x'));
  streamCtrl.stream.listen((x)=>print('listen 2 : $x'));

  // StreamController 객체의 add()를 통해 이벤트 등록
  streamCtrl.add(100);
  streamCtrl.add('test');
  streamCtrl.add(200);
  streamCtrl.add(300);
  streamCtrl.close();


  print('do something');
}
```

```
start
do something
listen 1 : 100
listen 2 : 100
listen 1 : test
listen 2 : test
listen 1 : 200
listen 2 : 200
listen 1 : 300
listen 2 : 300
```

- 제너레이터(Generator)
  - 제너레이터(Generator) 함수는 반복 가능한 함수
  - return 문 대신 yield를 사용하고 함수명 뒤에 async*라는 키워드를 붙이며 리턴 타입은 Stream이다.
    ⇒ Stream을 생성하기 위한 함수

```
import 'dart:async';

void main() {
  print('start');
  // 제너레이터 함수의 결과 Stream 객체 생성
  var stream = getData();

  // 해당 객체의 listen 등록
  stream
  .listen((x) => print('generator : $x'));

  // 기본적인 Stream 객체 선언 방식
  Stream.fromIterable([0,1,2,3,4])
        .listen((x) => print('fromIterable : $x'));

  print('do something');
}

// 제너레이터 함수 정의
Stream<int> getData() async* {
  // 반복문의 결과 : 컬렉션을 기준으로 한 Stream 객체 생성
  for (int i = 0; i < 5; i++) {
    yield i;
  }
}
```

```
start
do something
generator : 0
fromIterable : 0
generator : 1
fromIterable : 1
generator : 2
fromIterable : 2
generator : 3
fromIterable : 3
generator : 4
fromIterable : 4
```