Coroutine

- 코루틴을 구분하는 기준을 알아보자

코루틴을 구분하는 기준을 알아보자. 이를 통해 구분하는 기준(symmetric/asymmetric, first-class or not, stackful/stackless) 정도를 따져보면 이해하는데 큰 도움이 될 듯하다.

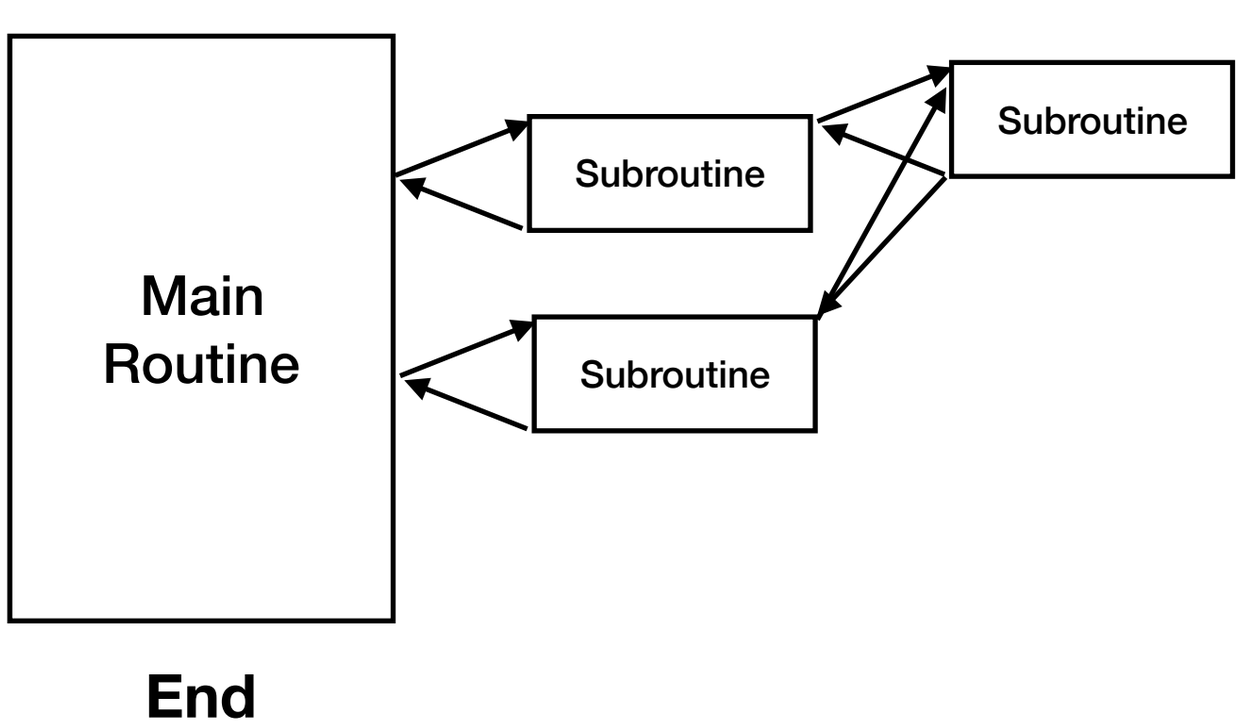
앞으로 설명한 코루틴을 구분하는 기준은

다음 **Revisiting Coroutines 2004** 페이퍼에

설명되어 있는 기준을 말한다.

물론 다른 기준이 있을 수도 있다.

- 함수 : call / return



- 코루틴 : call / return / suspend / resume

루틴의 로컬 상태를 유지하면서 제어를 반환했다가 (suspend),

제어권을 다시 획득했을 때 (resume) 흐름을 이어갈 수 있다.

코루틴 동작의 다양성을 다음 3가지 기준으로 나눈다

- 제어권 전달 방식 (symmetric / asymmetric)
- 일급 객체 (first-class or not)
- 콜스택이 쌓인 경우도 지원하는지 (stackful / stackless)

2, 3번을 만족한다 -> **완전 코루틴**

ex) Lua 코루틴 - 비대칭(Asymmetric) 완전 코루틴

대칭 (Symmetric) / 비대칭 (Asymmetric)

- 대칭 : 제어권을 넘길때 다른 코루틴을 지정
- 비대칭 : 제어권을 Caller 에게 넘김

ex) 제너레이터 - 비대칭 코루틴

비대칭 코루틴을 이용하면
대칭 코루틴을 쉽게 만들 수 있다

- 비대칭 코루틴 + dispatch loop

대칭 코루틴 예제

```
var q:= new queue

coroutine producer
   loop
      while q is not full
         create some new items
         add the items to q
      yield to consumer

coroutine consumer
   loop
      while q is not empty
         remove some items from q
         use the items
      yield to producer
```

제너레티어를 이용하여 대칭 코루틴을 흉내낸 것

```
var q:= new queue

generator produce
   loop
      while q is not full
         create some new items
         add the items to q
      yield consume
```

```
generator consumer
   loop
      while q is not empty
         remove some items from q
         use the items
      yield produce
```

```
subroutine dispatcher
   var d := new dictionary(generator -> iterator)
   d[produce] := start produce
   d[consume] := start consume
   var current := produce
   loop
      current := next d[current]
```

콜스택 지원 여부

콜스택 지원 여부는 현재 진행 중인 코루틴을 nested call 에서도 멈출 수 있느냐로 결정

```
function *g() {
  nested()
}

function nested() {
  yield;     // not allowed!
}
```

```
function *g() {
  yield* nested()
}

function *nested() {
  yield;
}
```

그래서 stackless 코루틴만 지원되는 경우 대개는 코루틴 중첩이 필요함

Reference

- 코루틴을 구분해보자