Channels

코루틴을 채널을 통해서 보낼 수 있다.

1. Channel 이란 무엇인가, 어떻게 만들고 이를 통해 정보를 보내는가?
2. Channel Producer : 채널 만드는 옵션은 무엇인가?
3. Pipeline : 채널을 통해 만들 수 있다.
4. Fan-out, Fan-in : 여러개의 코루틴이 하나의 채널에서 정보를 받는가, 아니면 여러 채널에서 하나의 코루틴이 정보를 받는가?
5. Buffered Channels : 받는 쪽에서 정보를 더 빠르게 프로세싱 할 수 있는 것
6. Ticker Channels

Channel

데이터의 큐이다.

채널에 다른 프로세스가 데이터를 넣고 받은 데이터를 사용한다.

코루틴은 비동기적으로 .send(data) 할 수 있다.

블러킹 으로 .receive() 할 수 있다.

더 이상 요소가 없을때는 .close() 해서 채널을 닫을 수 있다.

- 채널을 초기화하고 데이터의 타입을 정한다.
- 코루틴이 채널에 데이터를 보낸다. `send(data)`
- 메인 프로세스에서 코루틴으로부터 데이터를 받는다. `receive()`

```
import kotlinx.coroutines.channels.Channel
import kotlinx.coroutines.launch
import kotlinx.coroutines.runBlocking

fun main() {

    runBlocking {
        val channel = Channel<Int>()
            launch {
```

```
                for(x in 1..5)
                    channel.send(x*x)
                channel.close()
            }
        for(i in channel){
            //channel 은 iterable 이므로 close 전까지
            //채널 안을 확인할 수 있다.
            println(i)
        }
//        for(i in 1..5)
//            println(channel.receive())
    }
}
```

Channel Producer

채널을 만드는 방법은?

채널 객체를 만들어서 거기에 데이터를 보내는 방식 대신, 데이터소스에게 채널을 반환하고 생성하는 기능을 부여한다.

- 코루틴 스코프 안에서 produce{ ... } 를 사용해서 데이터 만들고 send 한다. 다른 코드에서 채널을 통해 데이터를 받을 수 있음.

```
import kotlinx.coroutines.CoroutineScope
import kotlinx.coroutines.channels.consumeEach
import kotlinx.coroutines.channels.produce
import kotlinx.coroutines.runBlocking

fun main() {

    runBlocking {
        //Channel<T> 로 직접 만들지 않아도 된다.
//        val channel = produce {
//            for(x in 1..5){
//                send(x * x)
//            }
//        }

//        val channel = produceSquares()

        //더 간결하게 만들 수 있다.
        produceSquares().consumeEach {
            println(it)
        }

//        for(y in channel){
//            println(y)
//        }
    }
}
```

```
//extension function 을 활용할 수 있다.
fun CoroutineScope.produceSquares() = produce {
    for(x in 1..5){
        send(x * x)
    }
}
```

- `CoroutineScope.produceSquares()` **로 확장 함수를 만들고 produce{... } 를 사용해서 더 간단하게 채널 객체를 만들 수 있다.**
- `consumeEach()` **를 이용하면, 결과를 받아서 처리하는 과정을 더 줄일 수 있다.**

Pipelines

개발 패턴이다. 이론적인 방법.

하나의 채널의 아웃풋이 다른 채널의 인풋으로 사용되는 방식이다.

```
import kotlinx.coroutines.CoroutineScope
import kotlinx.coroutines.cancelChildren
import kotlinx.coroutines.channels.ReceiveChannel
import kotlinx.coroutines.channels.produce
import kotlinx.coroutines.runBlocking

fun main() {

    runBlocking {
        val numbers = produceNumbers()
        val squares = square(numbers) //위의 결과가 인풋으로 들어간다.

        for(i in 1..5){
            println(squares.receive())
        }
        println("Done!")
        coroutineContext.cancelChildren() //모든 코루틴을 취소한다.
    }
}

fun CoroutineScope.produceNumbers() =
    produce {
        var x = 1
        while(true){
            //x를 증가시켜서 보낸다.
            send(x++)
        }
    }

fun CoroutineScope.square(numbers : ReceiveChannel<Int>)
    = produce {
    for(x in numbers){
```

```
        send(x * x)
    }
}
```

Fan-out

만약 많은 코루틴이 다른 값들을 병렬적으로 만들어 내는 데 하나의 채널에 연결이 되어있다면?

- 하나의 채널에서 여러개의 코루틴이 값을 받으면, 코루틴 사이에서 값들이 (작업들이) 분배된다.

```
import kotlinx.coroutines.CoroutineScope
import kotlinx.coroutines.channels.ReceiveChannel
import kotlinx.coroutines.channels.produce
import kotlinx.coroutines.delay
import kotlinx.coroutines.launch
import kotlinx.coroutines.runBlocking

fun main() {
    runBlocking {
        val producer = produceNumbers2()
        repeat(5){
            //채널 하나, 코루틴 다섯 개 만든다.
            launchProcessor(it, producer)
        }
        delay(1500L)
        producer.cancel()
    }
}

fun CoroutineScope.produceNumbers2() = produce {
    var x = 1
    while(true){
        send(x++)
        delay(100L)
    }
}

fun CoroutineScope.launchProcessor(
    id : Int,
    channel : ReceiveChannel<Int>
) = launch {
    for(message in channel){
        println("Processor $id received $message")
    }
}
```

결과 :

```
Processor 0 received 1
Processor 0 received 2
Processor 1 received 3
Processor 2 received 4
Processor 3 received 5
Processor 4 received 6
Processor 0 received 7
Processor 1 received 8
Processor 2 received 9
Processor 3 received 10
Processor 4 received 11
Processor 0 received 12
Processor 1 received 13
Processor 2 received 14
Processor 3 received 15

Process finished with exit code 0
```

Fan-in

- 같은 채널에 여러개의 코루틴이 값을 보낸다.

```
import kotlinx.coroutines.cancelChildren
import kotlinx.coroutines.channels.Channel
import kotlinx.coroutines.channels.SendChannel
import kotlinx.coroutines.delay
import kotlinx.coroutines.launch
import kotlinx.coroutines.runBlocking

fun main() {

    runBlocking {
        val channel = Channel<String>()
        launch { sendString(channel, 200L, "message1") }
        launch { sendString(channel, 500L, "message2") }
        repeat(6){
            println(channel.receive())
        }
        coroutineContext.cancelChildren()
    }
}

suspend fun sendString(channel : SendChannel<String>, time : Long, message : String){

    while(true){
        delay(time)
```

```
        channel.send(message)
    }
}
```

- 큐처럼 먼저 들어간 것이 먼저 나온다.

결과:

```
message1
message1 //400밀리세컨이 지난 시점엔 메시지 1만이 있다.
message2
message1
message1
message2
```

Buffered Channels

많은 양의 데이터를 보낼 때 채널은 이를 제한하지 않는다. 이 때 채널에 사이즈를 부여해야 하는데, Buffered Channel 을 사용한다.

용량에 다다르면 sender 는 pause 된다.

용량이 남게 되면 새로운 값들이 send 될 수 있다.

```
import kotlinx.coroutines.channels.Channel
import kotlinx.coroutines.delay
import kotlinx.coroutines.launch
import kotlinx.coroutines.runBlocking

fun main() {

    runBlocking {
        val channel = Channel<Int>(4) //용량을 4개로 설정한다.

        val sender = launch {
            repeat(10){
                channel.send(it)
                println("Sent $it")
            }
        }
        repeat(3){
            delay(1000)
            println("Received ${channel.receive()}")
        }
```

```
        sender.cancel()
    }
}
```

결과 :

```
Sent 0
Sent 1
Sent 2
Sent 3
Received 0 //4개가 차면 받고, 그 후에 send 를 실행한다.
Sent 4
Received 1
Sent 5
Received 2
Sent 6
```

Ticker Channels

- 정기적으로 Unit 을 생성하는 것이다. 디레이가 있음.
- 초기 디레이는 선택적이다.

```
import kotlinx.coroutines.channels.consumeEach
import kotlinx.coroutines.channels.ticker
import kotlinx.coroutines.delay
import kotlinx.coroutines.launch
import kotlinx.coroutines.runBlocking

fun main() {

    runBlocking {
        val ticker = ticker(100)

        launch {
            val startTime = System.currentTimeMillis()
            ticker.consumeEach {
                val delta = System.currentTimeMillis() - startTime
                println("Received tick after $delta")
            }
        }
        delay(1000)
        println("Done!")
        ticker.cancel()
    }
}
```

결과:

```
Received tick after 107
Received tick after 189
Received tick after 288
Received tick after 389
Received tick after 489
Received tick after 587
Received tick after 689
Received tick after 789
Received tick after 889
Received tick after 987
Done!
```