What is Monad?

모나드를 수학적으로 이해 x

프로그래밍적으로 이해를 돕고자합니다.

Agenda

- map() 과 flatMap()

- 모나드는 어떤 경우에 필요한가?

- Functor ?

- 그럼 Monad 는 대체 무엇인가?

- 결론 - Monad 의 정의

map() 과 flatMap()

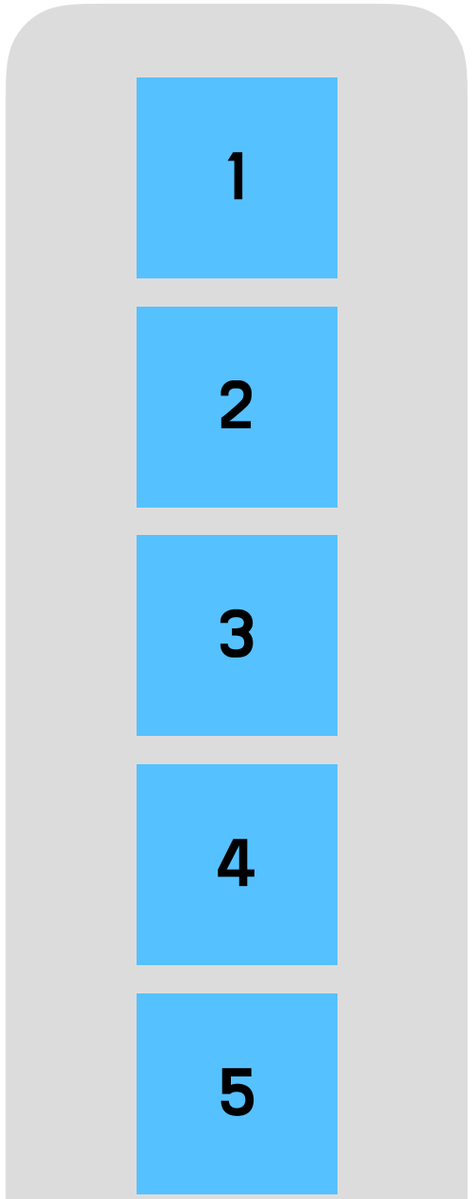
map()

```
List<String> alpha = Arrays.asList("a", "b", "c", “d");

List<String> collect = alpha.stream()
    .map(String::toUpperCase)
    .collect(Collectors.toList());
```

map()

intList



newList

map()

intList

1

2

3

4

5

.map(x => x + 1)

1 => 1 + 1

newList

2

map()

intList

1

2

3

4

5

.map(x => x + 1)

newList

2

3

4

5

6

아! for 문을 옆으로 쓰는 거구나!
일종의 이터레이터구나!

flatMap()

intList

intList1

1

2

3

intList2

4

5

.flatMap(Collection::stream)

newList

1

2

3

4

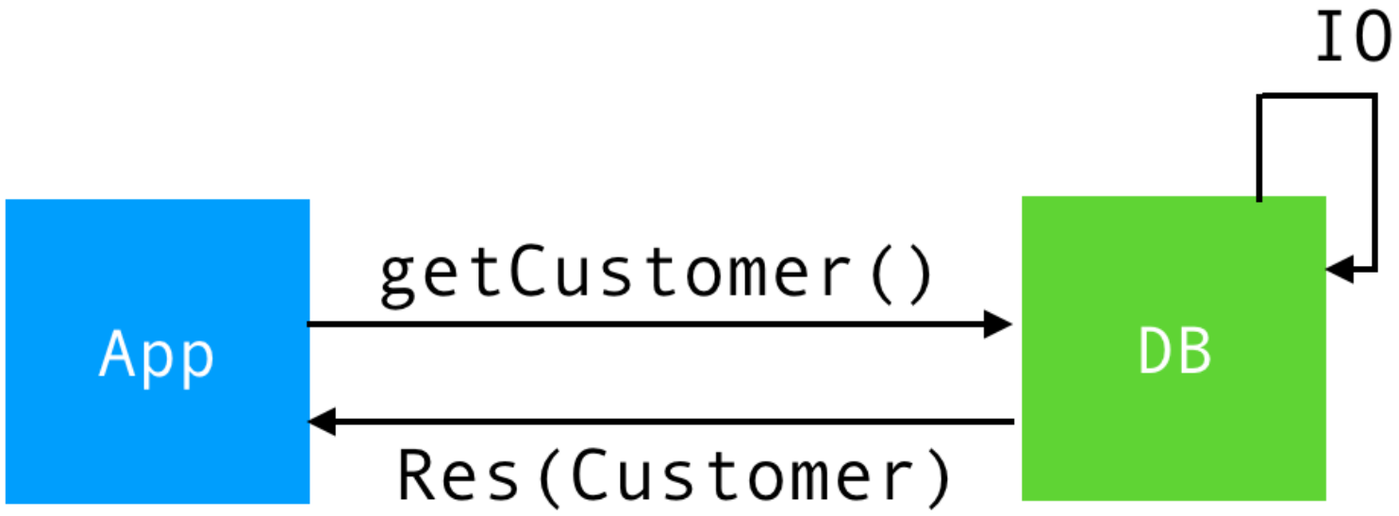
5

아! map 인데 이름처럼 평평하게 만들어주네

이렇게 이해하는 순간 Monad 를 이해하기 어려워진다 ;;

모나드는 어떤 경우에 필요한가?

비동기 연산 처리



Thread 소진
비동기 방식으로 처리

Promise 가 Monad 라고 볼 수 있다

Null 처리

```
Cart cart = response.getCart();
if (cart != null) {
    Product product = cart.getProduct();
    if (product != null) {
        System.out.println(product.getName());
    }
}
```

Null 처리

자바 코드

```
Optional.ofNullable(response.getCart()).ifPresent(c -> {
   Optional
     .ofNullable(c.getProduct())
     .ifPresent(p -> System.out.println(p.getName()));
});
```

스칼라 코드

```
response.getCart().map{ c => {
   c.getProduct().map(p => System.out.println(p.getName()))
}}
```

Null 처리

모나드를 이용하여 null 체크를 따로 하지않아도 간결하게 코드를 작성

`Optional` 클래스 자체도 일종의 모나드라고 할 수 있다.

모나드는 어떤 경우에 필요한가?

일반적으로 우리가 할 수 없는 비동기 연산

즉 값이 미래에 준비되거나 값이 null 인 경우,

이런 것들을 모델링할 때 모나드를 사용한다.

모나드 정의

- 모나드는 값을 담는 컨테이너의 일종
- Functor 를 기반으로 구현되었음
- flatMap() 메소드를 제공함
- Monad Laws 를 만족시키는 구현체를 말함

Functor ?

Functor 의 정의

```
import java.util.function.Function;

interface Functor<T> {
   <R> Functor<R> map(Function<T,R> f);
}
```

1. 함수를 인자로 받는 map 메소드만 가짐.
2. 타입 인자 <T> 를 가진다.
3. 전달인자인 함수 f 는 <T> 타입 값을 받아 <R> 타입 값을 반환하는 함수
4. Functor는 map 함수를 거쳐 <R> 타입의 Functor를 반환

Functor 의 정의

Functor

Int stringToInt(String args)

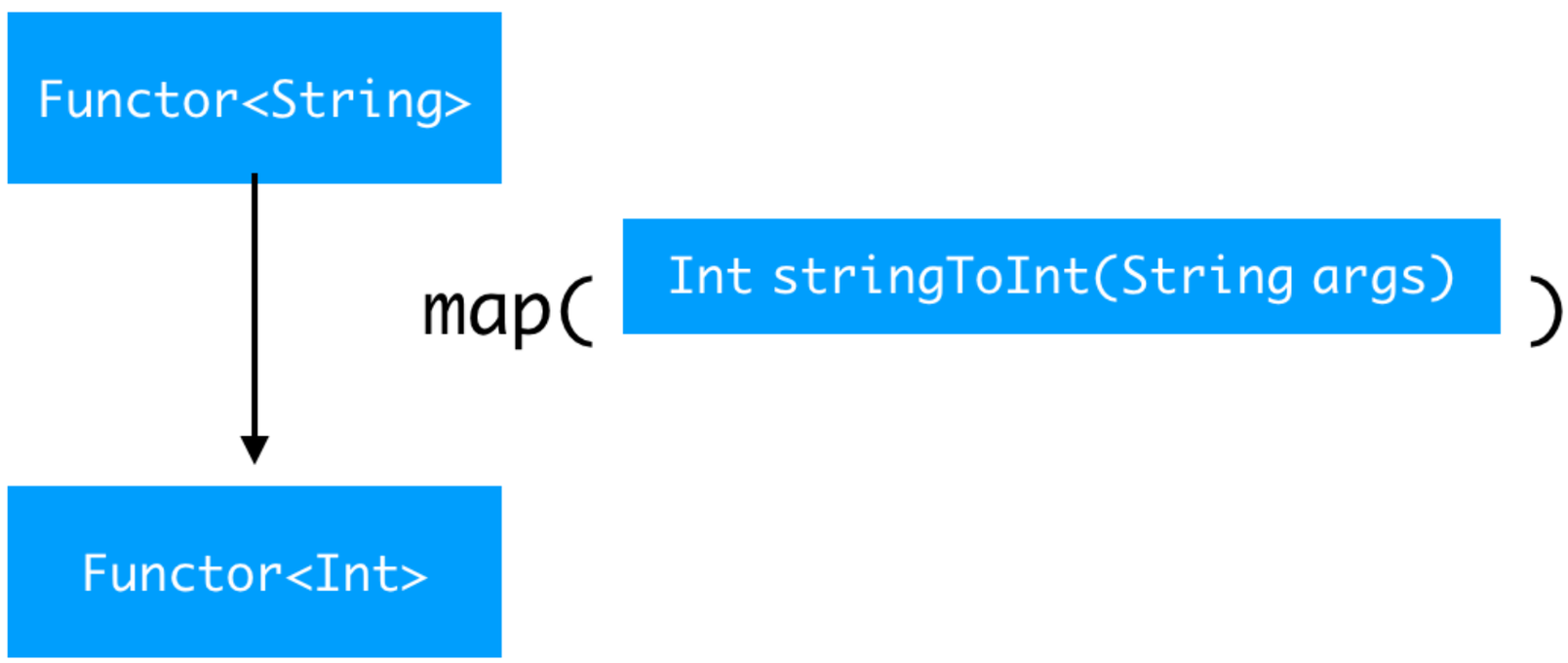
Functor

Function<String, Int> f

Functor 에 String 이 있고 stringToInt 함수를 전달

이 함수는 Function<String, Int> f 와 같은 형태가 됨

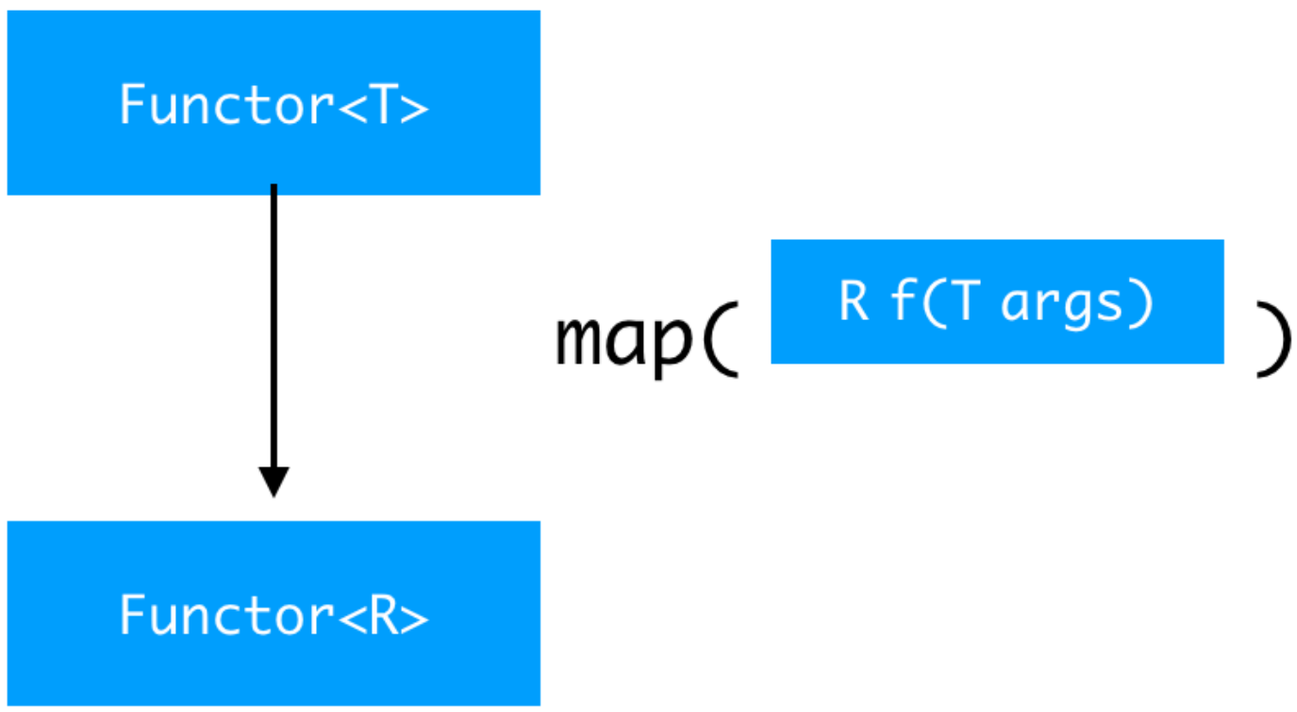
Functor 의 정의



이 함수를 map 에 전달

Functor<String> 이 Functor<Int> 로 바뀜

map() 의 진정한 의미



컬렉션의 원소를 순회하는 방법 X

<T> 타입의 Functor 를 <R> 타입의 Functor 로 바꾸는 기능

Functor

Q. 값을 꺼낼 수도 없고, 고작 map() 메소드로 값을 변경하는 것 뿐인데 왜 Functor 를 사용하는 건가요?

1. 함수를 쉽게 합성
2. 일반적으로 모델링 할 수 없는 상황을 모델링 가능

    Ex1) 값이 없는 케이스

    Ex2) 값이 미래에 준비될 것으로 예상되는 케이스

값이 없는 케이스

```
class FOptional<T> implements Functor<T, FOptional<?>> {

   private final T valueOrNull; // T 값을 하나 가짐
   private FOptional(T valueOrNull) {
      this.valueOrNull = valueOrNull;
   }
   public <R> FOptional<R> map(Function<T,R> f) {
      if (valueOrNull == null) {
        // 값이 비어있으면, empty() 호출, f 함수는 호출하지 않음
        return empty();
      } else {
         return of(f.apply(valueOrNull);
      }
   }
   public static <T> FOptional<T> of(T a) {
      return new FOptional<T>(a);
   }
   public static <T> FOptional<T> empty() {
      // 값이 비어있는 경우 null을 값으로 가지는 Functor를 반환
      return new FOptional<T>(null);
   }
}
```

값이 없는 케이스

다음과 같이 사용할 수 있다

```
FOptional<String> optionStr = FOptional(null);
FOptional<Integer> optionInt = optionStr.map(Integer::parseInt);
```

```
FOptional<String> optionStr = FOptional("1");
FOptional<Integer> optionInt = optionStr.map(Integer::parseInt);
```

사용하는 쪽에서 null check 필요 X

null 인 경우, 그냥 로직 실행 X

타입안정성을 유지하면서 null을 인코딩

값이 미래에 준비되는 케이스

```
Promise<Customer> customer = // ... DB에서 customer 정보를 들고왔다.
Promise<byte[]> bytes = customer.map(Customer::getAddress) // return Promise<Address>
          .map(Address::street)                            // return Promise<String>
          .map((String a) -> a.substring(0, 3))            // return Promise<String>
          .map(String::toLowerCase)                        // return Promise<String>
          .map(String::getBytes);                          // return Promise<byte[]>
```

customer 값을 아직 가지고 있지 않지만

FOptional 과 동일하게 마치 값이 있는 것처럼 map 메소드를 적용 가능.

비동기 연산들의 합성이 가능

그럼 List 는 무엇인가?

List 도 일종의 Functor. 단지 값이 하나가 아닌 리스트

```
class FList<T> implements Functor<T, FList<?>> {
   private final ImmutableList<T> list; // 단순히 Functor 가 담고 있는 값이 List 임
   FList(Iterable<T> value) {
      this.list = ImmutableList.copyOf(value);
   }

   @Override
   public <R> FList<?> map (Function<T,R> f) {
      ArrayList<R> result = new ArrayList<R>(list.size());
      for (T t : list) {
         result.add(f.apply(t)); // List의 모든 원소에 함수 f 를 적용
      }
      return new FList<>(result);
   }
}
```

그럼 List 는 무엇인가?

```
List<String> num = Arrays.asList("1","2","3","4","5")
List<Integer> collect1 = num.stream()
        .map(Integer::parseInt)
        .collect(Collectors.toList());
```

그래서 리스트에 대해 parseInt 메소드를 각 원소에 적용한다.

그럼 List 는 무엇인가?

실제로 스칼라 컬렉션에 리스트를 보면 이미 map 함수가 있는 것을 볼 수 있다

```scala
final override def map[B, That](f: A => B)(implicit bf: CanBuildFrom[List[A], B, That]): That = {
  if (bf eq List.ReusableCBF) {
    if (this eq Nil) Nil.asInstanceOf[That] else {
      val h = new ::[B](f(head), Nil)
      var t: ::[B] = h
      var rest = tail
      while (rest ne Nil) {
        val nx = new ::(f(rest.head), Nil)
        t.tl = nx
        t = nx
        rest = rest.tail
      }
      h.asInstanceOf[That]
    }
  }
  else super.map(f)
}
```

앞에 보았던 map 함수보다 먼가 더 어려워 보이지만::

안에 있는 모든 원소들에 대해 전달된 f 함수를 적용하고 있다

맵과 시퀀스

맵이나 시퀀스 같은 녀석들도 map 함수들을 가지게 된다

TraversableLike.scala

```
230  def map[B, That](f: A => B)(implicit bf: CanBuildFrom[Repr, B, That]): That = {
231    def builder : mutable.Builder[B, That] = { // extracted to keep method size under 35 bytes, so that it can be JIT-inlined
232      val b = bf(repr)
233      b.sizeHint(this)
234      b
235    }
236    val b = builder
237    for (x <- this) b += f(x)
238    b.result
239  }
```

IterableLike.scala

```
53  trait IterableLike[+A, +Repr] extends Any with Equals with TraversableLike[A, Repr] with GenIterableLike[A, Repr] {
54    self =>
55
56    override protected[this] def thisCollection: Iterable[A] = this.asInstanceOf[Iterable[A]]
57    override protected[this] def toCollection(repr: Repr): Iterable[A] = repr.asInstanceOf[Iterable[A]]
```

맵과 시퀀스

IterableLike.scala

```
53  trait IterableLike[+A, +Repr] extends Any with Equals with TraversableLike[A, Repr] with GenIterableLike[A, Repr] {
54  self =>
55
56    override protected[this] def thisCollection: Iterable[A] = this.asInstanceOf[Iterable[A]]
57    override protected[this] def toCollection(repr: Repr): Iterable[A] = repr.asInstanceOf[Iterable[A]]
```

MapLike.scala

```
60  trait MapLike[K, +V, +This <: MapLike[K, V, This] with Map[K, V]]
61    extends PartialFunction[K, V]
62    with IterableLike[(K, V), This]
63    with GenMapLike[K, V, This]
64    with Subtractable[K, This]
65    with Parallelizable[(K, V), ParMap[K, V]]
66  {
67  self =>
```

SeqLike.scala

```
64  trait SeqLike[+A, +Repr] extends Any with IterableLike[A, Repr]
65
66    override protected[this] def thisCollection: Seq[A] = this.as
67    override protected[this] def toCollection(repr: Repr): Seq[A]
68
```

IterableLike 를 MapLike 와 SeqLike 가 상속하고 있다

맵과 시퀀스

MapLike.scala

```
60  trait MapLike[K, +V, +This <: MapLike[K, V, This] with Map[K, V]]
61    extends PartialFunction[K, V]
62      with IterableLike[(K, V), This]
63      with GenMapLike[K, V, This]
64      with Subtractable[K, This]
65      with Parallelizable[(K, V), ParMap[K, V]]
66  {
67  self =>
```

Map.scala

```
34  trait Map[K, +V] extends Iterable[(K, V)] with GenMap[K, V] with MapLike[K, V, Map[K, V]] {
35    def empty: Map[K, V] = Map.empty
36
37    override def seq: Map[K, V] = this
38  }
```

최종적으로 MapLike 를 Map 이 상속하여 앞에서 본 map 함수를 가지게 되는 것이다

Future 와 Option

Option.scala

```
162    @inline final def map[B](f: A => B): Option[B] =
163      if (isEmpty) None else Some(f(this.get))
```

Future.scala

```
292    def map[S](f: T => S)(implicit executor: ExecutionContext): Future[S] = transform(_ map f)
```

스칼라 대부분 컬렉션에 map 함수를 가지고 있다

Option 이나 Future 에도 map 이 들어 있다

이런 것들이 다 Functor 로 구현

그럼 Monad 는 대체 무엇인가 ?

그럼 Monad 는 대체 무엇인가?

Monad 는 Functor 에 flatMap() 을 추가한 것

Functor 에 문제점이 있어 나온 것이라고 생각 할 수 있다

Functor 문제점

다음과 같이 tryParse 라는 함수가 있다고 하자

```
FOptional<Integer> tryParse(String s) {
   try {
      final int i = Integer.parseInt(s);
      return FOptional.of(i); // 여기서 이미 Functor 를 반환
   } catch (NumberFormatException e) {
      return FOptional.empty();
   }
}
```

tryParse 함수는 String 을 받아서 parseInt 해서 Integer 로 바꾸면

그 값을 반환하고 만약 문제가 있으면 empty() 로 반환

Functor 에다가 tryParse 함수를 전달하는 순간 문제 발생

Functor 문제점

다음과 같이 tryParse 라는 함수가 있다고 하자

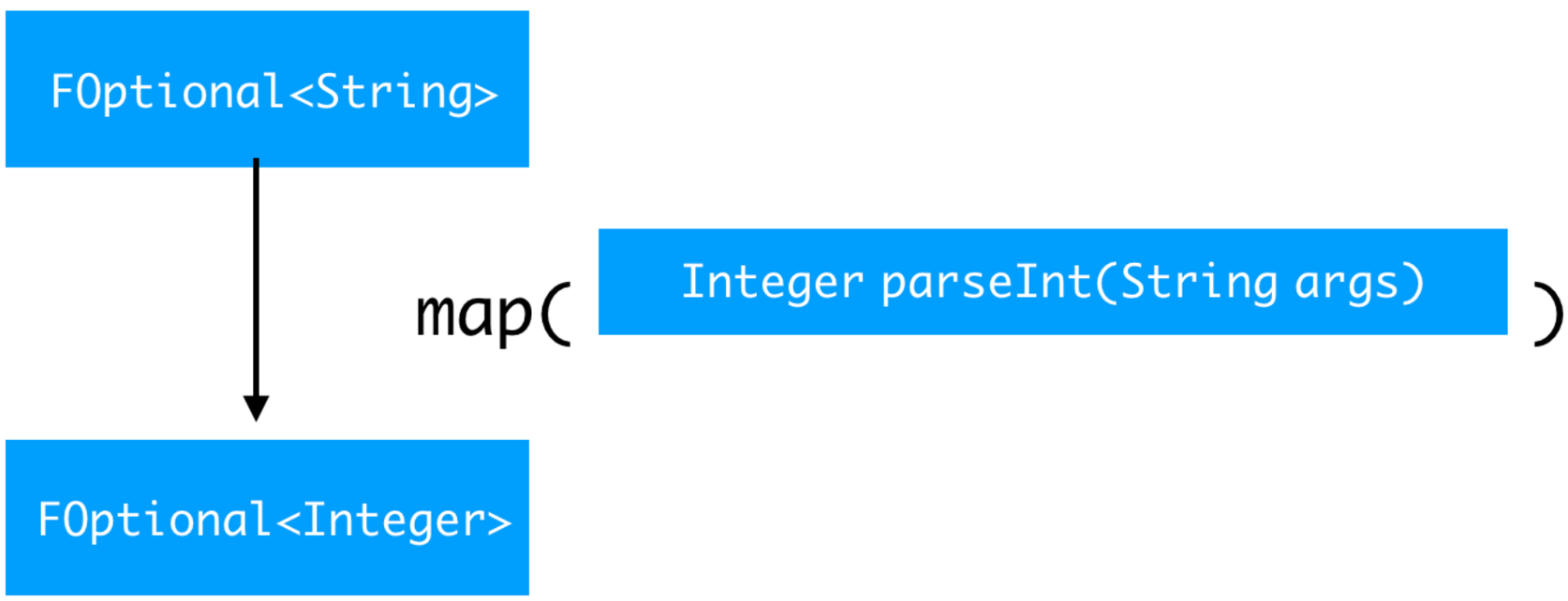
```
FOptional<Integer> tryParse(String s) {
   try {
      final int i = Integer.parseInt(s);
      return FOptional.of(i); // 여기서 이미 Functor 를 반환
   } catch (NumberFormatException e) {
      return FOptional.empty();
   }
}
```

tryParse 함수는 String 을 받아서 parseInt 해서 Integer 로 바꾸면

그 값을 반환하고 만약 문제가 있으면 empty() 로 반환

Functor 에다가 tryParse 함수를 전달하는 순간 문제 발생

Functor 문제점



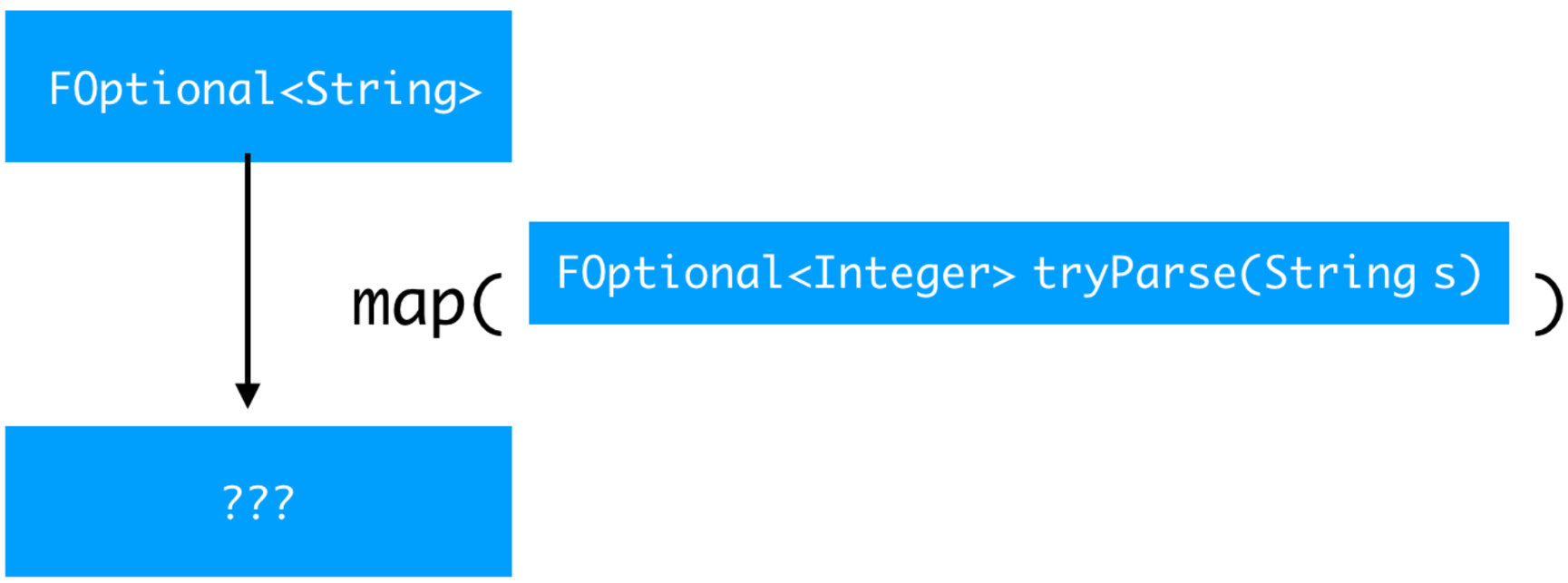
일반적으로 앞에서 본 예제를 보면

FOptional<String> 이었는데 parseInt 라는 메소드를 전달해서

반환값이 Integer 이기 때문에

FOptional<Integer> 가 된다

Functor 문제점



tryParse 를 전달하면 tryParse 는 이미 FOptional<Integer> 를 반환하고 있다

따라서 FOptional<FOptional<Integer>> 형태로 되어버림

함수의 합성과 체이닝이 어려워진다

Functor 문제점

```
FOptional<Integer> num1 = //...
FOptional<FOptional<Integer>> num2 = // ...

FOptional<Date> date1 = num1.map(t -> new Date(t))
FOptional<Date> date2 = num2.map(t -> new Date(t)) // 컴파일 안됨
```

마지막 줄 t 가 이미 FOptional<Integer> 이기 때문에 Date 에 전달을 할 수 없다

Functor 가 두번 감싸져, Functor 가 제기능을 하지 못함

하지만 tryParse 처럼 Functor 를 반환하는 함수는 매우 일반적인 케이스

Functor 의 map 함수는 어떤 형태든 받을 수 있는 구조인데...

Functor 를 반환한다고 처리가 안되면 안됨...

그래서 나온 것이 **flatMap** 이다

flatMap 의 정의

```
interface Monad<T, M extends Monad<?,?>> extends Functor<T, M> {
  M flatMap(Function<T,M> f); // 변형함수 f의 타입인자인 M을 반환
}
```

flatMap 도 결국엔 함수를 전달인자로 받음

Function 은 T 를 받아 M 을 반환하고

flatMap 의 반환값도 M 이다

flatMap 의 정의

```
interface Monad<T, M extends Monad<?,?>> extends Functor<T, M> {
  M flatMap(Function<T,M> f); // 변형함수 f의 타입인자인 M을 반환
}
```

```
interface Functor<T> {
  <R> Functor<R> map(Function<T,R> f);
}
```

map 과 비교를 해보자

- map : f 함수가 R 타입을 반환하니깐 Functor<R> 로 반환
- flatMap : f 가 M 을 반환하면 그냥 M 을 반환

flatMap 적용

```
FOptional<string> num = FOptional.of("42");
// tryParse 반환값: FOptional<Integer>
FOptional<Integer> answer = num.flatMap(this::tryParse);
FOptional<Date> date = answer.map(t -> new Date(t)); // 합성 가능
```

tryParse 를 flatMap 에 전달

tryParse 반환값이 FOptional<Integer> 이기 때문에

그냥 값으로 바뀌게 되어 answer.map(t -> new Date(t)) 처럼 함수를 합성할 수 있다

flatMap 적용

```
num.flatMap(this::tryParse)
   .map(t -> new Date(t)) // 합성 가능
   .
   .
   .
```

flatMap 을 하고 난 다음 map 을 할 수 있고 그 다음
또 다른 flatMap 을 할 수 있고 ...

이 처럼 합성을 할 수 있다는게 모나드의 강점

결론 - Monad 의 정의

결론 - Monad 정의

- 모나드는 값을 담는 컨테이너의 일종
- Functor 를 기반으로 구현되었음
- flatMap() 메소드를 제공함
- Monad Laws 를 만족시키는 구현체를 말함

Monad Laws 를 만족하지 않아도 스칼라에서는 모나딕이라고 하면서 똑같이 사용가능하다고 함

결론 - Monad 정의

값이 없는 상황이나, 값이 미래에 이용가능한 상황

일반적으로 할 수 없는 **여러 상황을 모델링** 할 수 있다

비동기 로직을 구현하 때 마치 동기 로직을 구현하는 것과 동일한 형태로 구현하면서도

함수의 합성 및 완전한 non-blocking pipeline 을 구현할 수 있다

Reference

Naver D2 - Monad 란 무엇인가?