

# Type Level Programming in scala

알아두면 쓸모있을 만한 스칼라 타입에 대한 이야기

@liam.m

# Requirements

- Scala? 문법에 조금은 익숙하면 좋습니다.
- 졸려도 참을 수 있는 체력!
- 욕하지 않고 견딜수 있는 인내심!
- 남들이 안가는 길을 가는 용기!
- 유닛 테스트를 최대한 만들지 않겠다는 패기!

# 버그는 어떻게 예방하나요? 💊

- 테스트
- 방어 프로그래밍
- 테스트
- 코드 리뷰
- 테스트
- 테스트
- 테스트

# 코드 그 자체로 안전할 수는 없을까요?

# Once your code compiles it usually works.<sup>2</sup>

컴파일되면 엔간하면 동작한다.

<sup>2</sup> [https://wiki.haskell.org/Why\\_Haskell\\_just\\_works](https://wiki.haskell.org/Why_Haskell_just_works)

# TDD

TDD - Type driven development<sup>3</sup>

컴파일이 되면 코드가 잘 동작하도록  
로직이 이상하면 컴파일이 안되도록

<sup>3</sup> <https://www.manning.com/books/type-driven-development-with-idris>

# 이야기 순서

1. Type safe equality - ===
2. Builder pattern using Phantom type
3. Type class pattern - implicit
4. Literal type - 42.type
5. Dependent type - a.B

에 관한 **Type** 이야기를 해보겠습니다. 😎

# Equality?

```
val items = List(  
    Item(1, Some("ON_SALE")), Item(2, Some("SOLDOUT")),  
    Item(3, Some("ON_SALE")), Item(4, None))  
  
items.filter(item => item.status == "ON_SALE")
```

결과값은?

# Equality?

```
val items = List(  
    Item(1, Some("ON_SALE")), Item(2, Some("SOLDOUT")),  
    Item(3, Some("ON_SALE")), Item(4, None))
```

```
items.filter(item => item.status == "ON_SALE")
```

결과값은 Nil 아무것도 없다.

```
item.status: Option[String] != "ON_SALE": String
```

유닛 테스트로 오류를 잡을수 있죠. 😜

```
test("item status filter") {  
    val expected = List(  
        Item(1, Some("ON_SALE")),  
        Item(3, Some("ON_SALE"))  
    )  
  
    assert(items == expected) // 테스트 실패!  
}
```

유닛 테스트로 오류를 잡을수 있죠. 😜

```
test("item status filter") {  
    val expected = List(  
        Item(1, Some("ON_SALE")),  
        Item(3, Some("ON_SALE"))  
    )  
  
    assert(items == expected) // 테스트 실패!  
}
```

하지만 우리가 원하는건?

유닛 테스트로 오류를 잡을수 있죠. 😜

```
test("item status filter") {  
    val expected = List(  
        Item(1, Some("ON_SALE")),  
        Item(3, Some("ON_SALE"))  
    )  
  
    assert(items == expected) // 테스트 실패!  
}
```

하지만 우리가 원하는건?

Once your code compiles it usually works.

# Type safe equality

```
implicit class StrictEq[A](a: A) {  
    def ===(b: A) = a == b  
}
```

# Type safe equality

```
implicit class StrictEq[A](a: A) {  
    def ===(b: A) = a == b  
}
```

implicit class는 특정 타입에 함수를 추가할수 있다.

A 타입에 === 함수를 추가

이제 **COMPILE TIME**에 문제를 해결할수 있습니다.

```
val x = Some("ON_SALE")
val y = Some("SOLDOUT")
val z = "ON_SALE"
```

```
x === y      // false
x === z      // doesn't compile, 서로 다른 타입 비교안됨
```

이제 보다 안전한 값 비교. 

==== 를 사용하여 전체 코드를 변경하는건 귀찮다.

어느 세월에...



다음 생에는 돌로 태어나  
아무것도 안 했으면 좋겠다

```
scalacOptions += "-Xfatal-warnings"
```

```
case class UserId(n: Int)  
val john = UserId(7)
```

```
john == 7  
// <console>:11: warning: comparing values of  
// types UserId and Int using `=='  
// will always yield false john == 7
```

**warning -> error**

로 변경해서 다른 타입 비교는 컴파일 막자 !

# 쉬어가는 코너

Hey! Have you heard about new cool features in Java?



Now we can use lambdas, streams, functional interfaces, optional values...



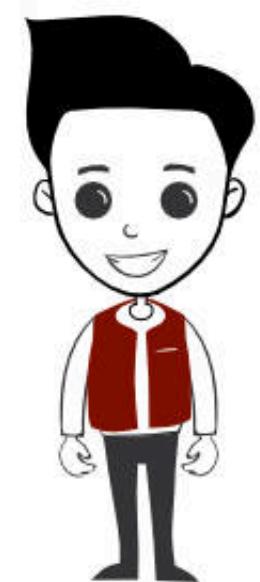
And what is awesome in Java 9:  
JShell is introduced...



Man, stop enumeration of Scala features



Hey! Have you heard about new cool features in Java?



Now we can use lambdas, streams, functional interfaces, optional values...



And what is awesome in Java 9: JShell is introduced...



Man, stop enumeration of Scala features



# Builder pattern

"The builder pattern is a good choice when designing classes whose constructors or static factories would have more than a handful of parameters."

Joshua Bloch, Effective Java

빌더 패턴은 좋다 

# Builder pattern

"The builder pattern is a good choice when designing classes whose constructors or static factories would have more than a handful of parameters."

Joshua Bloch, Effective Java

빌더 패턴은 좋다  필수값은 빌더 패턴에서 예외이다 

```
// 선물하기에서 상품을 표현하는 테이블, 필수값이 많다.
public class Item {
    private Integer id;
    private Integer brandId;
    private Integer catalogId;
    private Integer supplyChannelId;           // optional
    private String supplyChannelItemCode;
    private String supplyChannelCategoryCode;
    private String name;
    private String displayName;
    private Integer itemType;
    private Integer basicPrice;
    private Integer sellingPrice;
    private Integer discountRate;
    private Integer feeRate;
    private Byte periodType;
    private Integer validPeriod;
    private Byte pinIssueType;
    private String pinIssueCsInfo;             // optional
    private Boolean isCancelable;
    private String imageUrl;                  // optional
    private Integer status;
    private Boolean displayYn;
    private Short distType;                  // optional
    private String detailInfo;               // optional
    private String noticeInfo;               // optional
    private Instant couponExpiredAt;         // optional
    private Instant releasedAt;
    private Instant expiredAt;
    ... // 이보다 더 있음 총 51개의 필드
}
```

필드가 많으니 빌더 패턴이 좋겠죠?

# 객체의 필수값은 빌더의 생성자로 넘기라 - Joshua Bloch

```
public class Item {  
    public static class Builder {  
        // 필수값들은 Builder의 생성자에  
        public Builder(Integer id, Integer brandId, Integer catalogId, String supplyChannelItemCode,  
                      String supplyChannelCategoryCode, String name, String displayName, Integer itemType,  
                      Integer basicPrice, Integer sellingPrice, Integer discountRate, Integer feeRate,  
                      Byte periodType, Integer validPeriod, String pinIssueCsInfo, Byte pinIssueType,  
                      Boolean isCancelable, Integer status, Boolean displayYn, Instant releasedAt, Instant expiredAt) {  
            this.id = id; this.brandId = brandId; this.catalogId = catalogId;  
            this.supplyChannelItemCode = supplyChannelItemCode;  
            this.supplyChannelCategoryCode = supplyChannelCategoryCode;  
            this.name = name; this.displayName = displayName; this.itemType = itemType;  
            this.basicPrice = basicPrice; this.sellingPrice = sellingPrice;  
            this.discountRate = discountRate; this.feeRate = feeRate;  
            this.periodType = periodType; this.validPeriod = validPeriod;  
            this.pinIssueType = pinIssueType; this.pinIssueCsInfo = pinIssueCsInfo;  
            this.isCancelable = isCancelable; this.status = status;  
            this.displayYn = displayYn; this.releasedAt = releasedAt; this.expiredAt = expiredAt;  
        }  
        // 선택값들은 보조 함수로 편리하게  
        public Builder withSupplyChannelId(Integer val) {  
            this.supplyChannelId = val;  
            return this;  
        }  
        public Builder withPinIssueCsInfo(String val) {  
            this.pinIssueCsInfo = val;  
            return this;  
        }  
        ...  
    }  
}
```

# 근데 Builder같지 않는 builder가 만들어졌다.

```
Item item = new Item.Builder(  
    10,  
    1000,  
    2000,  
    "I123",  
    "C123",  
    "(주)맥도날드",  
    "맥도날드",  
    101,  
    5000,  
    3000,  
    40,  
    5,  
    (byte)2,  
    365,  
    "CS",  
    (byte) 3,  
    true,  
    201,  
    true,  
    Instant.now(),  
    Instant.MAX  
).withNoticeInfo("공지").withImageUrl("http://builder.jpg")  
    .build();
```

## 미션 : **Builder** 같은 **builder** 만들기

- Builder의 withXXX 함수만 사용해서 필수값 제약하기

```
/* 스칼라 버전 Item.scala */
case class Item(
    id: Int,
    brandId: Int,
    catalogId: Int,
    supplyChannelId: Option[Int], // 옵션 타입 말고는
    supplyChannelItemCode: String, // 필수값으로 받고 싶다.
    supplyChannelCategoryCode: String,
    name: String,
    displayName: String,
    itemType: Int,
    basicPrice: Int,
    sellingPrice: Int,
    discountRate: Int,
    feeRate: Int,
    periodType: Byte,
    validPeriod: Int,
    pinIssueType: Byte,
    pinIssueCsInfo: Option[String],
    isCancelable: Boolean,
    imageUrl: Option[String],
    status: Int,
    displayYn: Boolean,
    distType: Option[Short],
    detailInfo: Option[String],
    noticeInfo: Option[String],
    couponExpiredAt: Option[Instant],
    releasedAt: Instant,
    expiredAt: Instant
)
```

이제 필드의 빌드 상태를 표현하는 type을 만들자.

이제 필드의 빌드 상태를 표현하는 type을 만들자.

Why?

이제 필드의 빌드 상태를 표현하는 type을 만들자.

Why? Compiler는 type만 아는 type 바보

이제 필드의 빌드 상태를 표현하는 type을 만들자.

Why? Compiler는 type만 아는 type 바보

// 타입 제약에만 쓰인다.

```
sealed trait BuildState {
```

```
type Id <: Bool // True이면 가격은 추가됨
```

```
type Name <: Bool // True이면 이름은 추가됨
```

```
}
```

# 값을

```
sealed trait Bool
```

```
val True = new Bool {}
val False = new Bool {}
```

```
val hasOption: Bool = True
```

# 값을 타입으로 바꾼다

```
sealed trait Bool
```

```
sealed trait True extends Bool
```

```
sealed trait False extends Bool
```

```
type HasOption = True
```

# 값을 타입으로 바꾸는 규칙들

- ADT Values : val → trait

val True = new Bool {} → trait True extends Bool

## 값을 타입으로 바꾸는 규칙들

- ADT Values : val → trait

val True = new Bool {} → trait True extends Bool

- members : val → type X

val hasOption: Bool = True → type HasOption = True

# 참과 거짓을 타입으로 표현했다

```
sealed trait Bool
```

// 값을

```
val True = new Bool {}
val False = new Bool {}
```

```
val hasOption: Bool = True
```

// 타입으로 바꾼다

```
sealed trait True extends Bool
sealed trait False extends Bool
```

```
type HasOption = True
```

Name, DisplayName에 값이 생성되면 True 타입을 할당

```
class Builder[B <: BuildState] { self =>
    private var name: Option[String] = None
    private var displayName: Option[String] = None

    def newBuilder[C <: BuildState] = this.asInstanceOf[Builder[C]]

    def withName(name: String) = {
        self.name = Some(name)
        newBuilder[B {type Name = True}]
    }

    def withDisplayName(displayName: String) = {
        self.displayName = Some(displayName)
        newBuilder[B {type DisplayName = True}]
    }
}
```

# 값이 생성되었는지는 타입에 True 타입이 할당 되었는지로 판단

```
class Builder[B <: BuildState] { self =>
    def build(implicit
        ev1: B#Id =:= True,    // True 타입이 할당되어야만 컴파일됨
        ev2: B#BrandId =:= True,
        ev3: B#CatalogId =:= True,
        ev4: B#SupplyChannelItemCode =:= True,
        ev5: B#SupplyChannelCategoryCode =:= True,
        ev6: B#Name =:= True,
        ev7: B#DisplayName =:= True,
        ev8: B#ItemType =:= True,
        ev9: B#BasicPrice =:= True,
        ev10: B#SellingPrice =:= True,
        ev11: B#DiscountRate =:= True,
        ev12: B#FeeRate =:= True,
        ...
    ): Item =
        Item(id.get, brandId.get, catalogId.get, supplyChannelId, supplyChannelItemCode.get,
            supplyChannelCategoryCode.get, name.get, displayName.get, itemType.get, basicPrice.get,
            sellingPrice.get, discountRate.get, feeRate.get, periodType.get, validPeriod.get,
            pinIssueType.get, pinIssueCsInfo, isCancelable.get, imageUrl, status.get, displayYn.get,
            distType, detailInfo, noticeInfo, couponExpiredAt, releasedAt.get, expiredAt.get)
}
```

선택값의 경우는 BuildState의 타입을 바꿀필요가 없다.

```
class Builder[B <: BuildState] { self =>
    private var name: Option[String] = None
    private var detailInfo: Option[String] = None // optional

    def withName(name: String) = { // 필수값
        self.name = Some(name)
        newBuilder[B {type Name = True}]
    }

    def withDetailInfo(detailInfo: String) = { // 선택값
        self.detailInfo = Some(detailInfo)
        newBuilder[B]
    }
}
```

# 잘동작하는지 확인해보자

```
object Builder {  
    def apply() = new Builder[BuildState {}]  
}
```

// 필수값이 설정되지 않았기 때문에 컴파일 되지 않는다  
Builder().build // doesn't compile, 오 커파일 안된다. 나이스

Builder()  
.withBasicPrice(5000).build // doesn't compile, 오오 이것도 커파일 안된다.

// 컴파일 된다. withXXX 함수만을 이용해 필수값을 제약. 아주 맘에 듭니다. 🤜

```
val item = Builder()  
    .withId(10)  
    .withBrandId(1000)  
    .withCatalogId(2000)  
    .withSupplyChannelItemCode("I123")  
    .withSupplyChannelCategoryCode("C123")  
    .withName("(주)맥도날드")  
    .withDisplayName("맥도날드")  
    .withItemType(101)  
    .withBasicPrice(5000)  
    .withSellingPrice(3000)  
    .withDiscountRate(40)  
    .withFeeRate(5)  
    .withPeriodType(2)  
    .withValidPeriod(365)  
    .withPinIssueType(3)  
    .withIsCancelable(true)  
    .withStatus(201)  
    .withDisplayYn(true)  
    .withReleasedAt(now)  
    .withExpiredAt(Instant.MAX)  
    .build
```

// named argument를 이용한 객체 생성... 그냥 이게 나을려나? 뺄짓했나? 💀

```
val item = Item(  
    id = 10, brandId = 1000, catalogId = 2000,  
    supplyChannelId = None, supplyChannelItemCode = "I123",  
    supplyChannelCategoryCode = "C123", name = "(주)맥도날드",  
    displayName = "맥도날드", itemType = 101,  
    basicPrice = 5000, sellingPrice = 3000,  
    discountRate = 40, feeRate = 5,  
    periodType = 2, validPeriod = 365,  
    pinIssueType = 3, pinIssueCsInfo = None,  
    isCancelable = true, imageUrl = None,  
    status = 201, displayYn = true,  
    distType = None, detailInfo = None,  
    noticeInfo = None, couponExpiredAt = None,  
    releasedAt = now, expiredAt = Instant.MAX  
)
```

**그래도 괜찮지 않았나요?**

너무 갑자기 훅 치고 들어왔유?

이  
게  
뭐  
야

!!

벌써 졸릴각?



# Type class Pattern

다양한 타입에 대한 합을 구하는 함수 `sum`을 구현하고 싶다.

```
case class Point(x: Int, y: Int) // 2차원 좌표
```

```
sum(List(1, 2, 3)) => 6
```

```
sum(List(Point(1,10), Point(5, 5))) => Point(6, 15)
```

```
trait Adder[A] {  
    def zero: A  
    def add(x: A, y: A): A  
}
```

```
trait Adder[A] {  
    def zero: A  
    def add(x: A, y: A): A  
}  
  
def sum[A](xs: List[A])(adder: Adder[A]): A =  
    xs.foldLeft(adder.zero)(adder.add)
```

```
val intAdder = new Adder[Int] {  
    def zero: Int = 0  
    def add(x: Int, y: Int) = x + y  
}
```

```
val pointAdder = new Adder[Point] {  
    def zero = Point(0, 0)  
    def add(a: Point, b: Point) = Point(a.x + b.x, a.y + b.y)  
}
```

잘된다.

```
sum(List(1, 2, 3))(intAdder)           // 7
sum(List(Point(1, 10), Point(5, 5)))(pointAdder) // Point(6, 15)
```

# 잘된다. 그러나 뭔가 아름답지 못하다.

```
sum(List(1, 2, 3))(intAdder)           // <= 달고 다니기 귀찮다.  
sum(List(Point(1, 10), Point(5, 5)))(pointAdder) // <= 여기도
```

# Type class pattern 이란?

```
def sum[A](xs: List[A])(implicit adder: Adder[A]): A
```

Ad-hoc polymorphism - type parameter를 이용해서  
implicit instance 주입 받아 polymorphism하는 기법

DI - compiler가 dependency injection

Spring, Guice는 runtime에 DI가 실행

# implicit은 마법사 ★

```
// Adder를 implicit하게 주입
def sum[A](xs: List[A])(implicit adder: Adder[A]): A =
  xs.foldLeft(adder.zero)(adder.add)

// type class instance 도 implicit 하게
implicit val intAdder = new Adder[Int] {
  def zero = 0
  def add(x: Int, y: Int) = x + y
}

implicit val pointAdder = new Adder[Point] {
  def zero = Point(0, 0)
  def add(a: Point, b: Point) = Point(a.x + b.x, a.y + b.y)
}
```

```
sum(List(1, 2, 3))(intAdder)  
sum(List(Point(1, 10), Point(5, 5)))(pointAdder)
```

```
sum(List(1, 2, 3))
```

```
sum(List(Point(1, 10), Point(5, 5)))
```

```
sum(List(1, 2, 3))
sum(List(Point(1, 10), Point(5, 5)))
```

// String은 합칠 방법을 모르기 때문에 compile이 안된다.

```
sum(List("Hello", "World")) // doesn't compile
```

// 깔끔 깔끔 

# String도 더하고 싶다면?

```
// string instance만 추가하면 된다.  
implicit val stringAdder = new Adder[String] {  
    def zero = ""  
    def add(x: String, y: String) = x + y  
}  
  
sum(List("Hello", "World")) // => "HelloWorld"
```

회사에서 뾰큐라니 최송합니다. π·π

이 차트 그래프가 젤 괜찮아서...

정보를 전달하고 싶어서 양해하고 바주

세요 π·π

핫 라인에 신고하지 말아 주세요 π·π  
그냥 손가락 부상 당한 주먹이라 생각해  
주세요. π·π

— productivity

time

Social learning curve

hmm,  
I'm fine.  
too much...

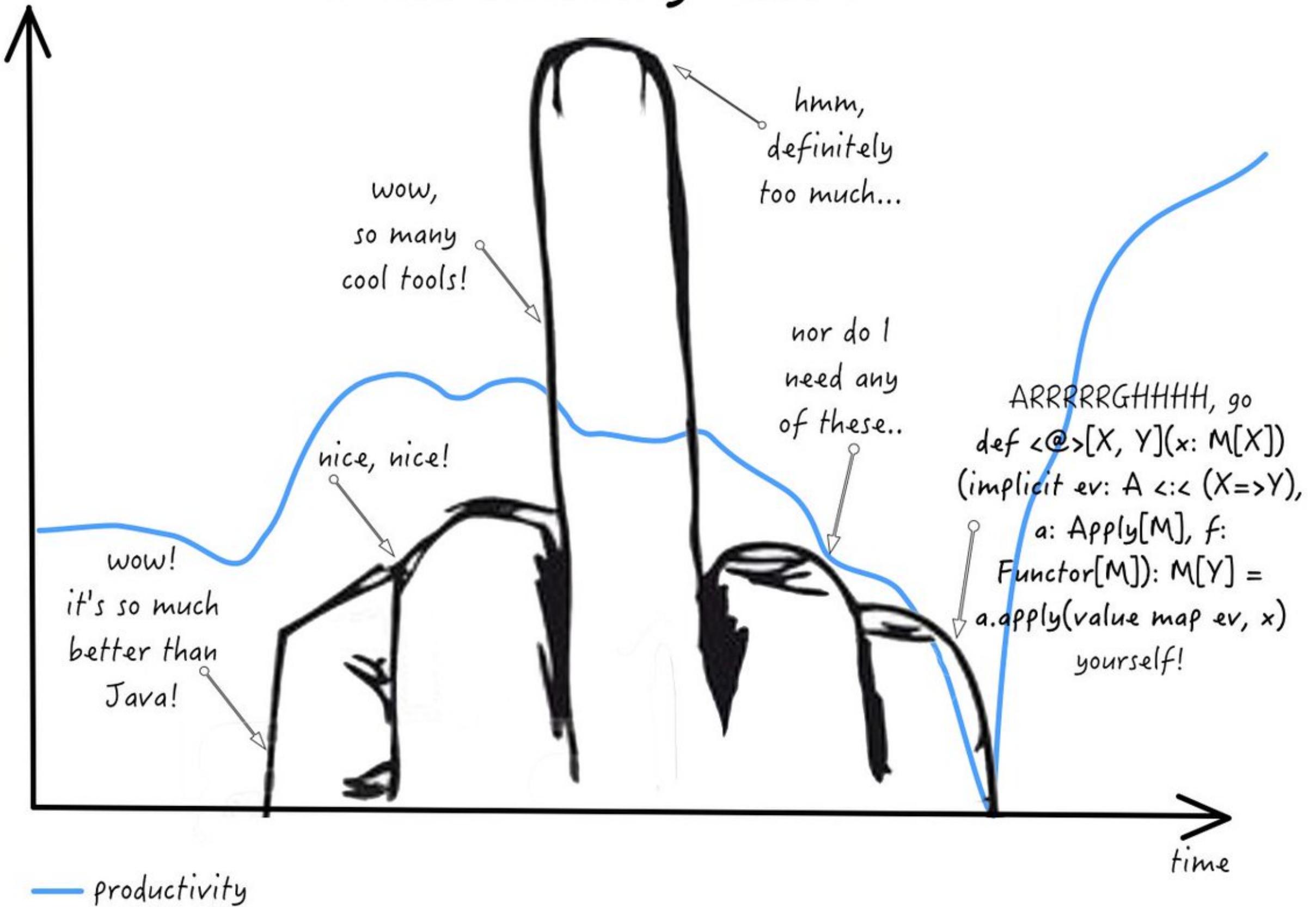
wow,  
so many  
cool tools!

how do I

need any  
of these..

ARRRRRGHHHH, go  
def <@>[X, Y](x: M[X])  
(implicit ev: A <:< (X=>Y),  
a: Apply[M]. f:  
Function[M]: M[•] =  
a.apply(value map ev, x)  
yourself

# Scala learning curve



# **Literal type - 42.type**

값은 타입이 될수 없을까?

# Literal type - 42.type

값은 타입이 될수 없을까? 됩니다. 🤝

```
val t: 42 = 42  
val x: "Jedi" = "Jedi"
```

# Literal type - 42.type

값은 타입이 될수 없을까? 됩니다. 

```
val t: 42 = 42  
val x: "Jedi" = "Jedi"
```

함수의 반환타입은?

# Literal type - 42.type

값은 타입이 될수 없을까? 됩니다. 

```
val t: 42 = 42  
val x: "Jedi" = "Jedi"
```

함수의 반환타입은? 됩니다. 

```
def f(t: Double): t.type = t  
val a: 1.2 = f(1.2)
```

# Literal type in scala

Scala에는 곧 추가될 예정 <http://docs.scala-lang.org/sips/pending/42.type.html>

Typelevel Scala에는 반영됨 <https://typelevel.org/scala/>

Dotty도 이미 구현됨 <http://dotty.epfl.ch/docs/reference/singleton-types.html>

미리 준비하고 익숙해지자.

Literal singleton type은 표준이 될것이고 이를 활용하는 library들은 점점더 많아질것이다.

# true.type을 이용한 if 없는 condition

```
trait Cond[T] { type V ; val value: V }

implicit val condTrue = new Cond[true] { type V = String ; val value = "foo" }

implicit val condFalse = new Cond[false] { type V = Int ; val value = 23 }

def cond[T](implicit cond: Cond[T]): cond.V = cond.value

// true is type! 😬
cond[true]          // "foo"

// flase is type! 😮
cond[false]         // 23
```

# Path Dependent type - 경로 의존적인 제약

```
class A {  
    class B  
    var b: Option[B] = None  
}
```

# Path Dependent type - 경로 의존적인 제약

```
class A {  
    class B  
    var b: Option[B] = None  
}  
val a1: A = new A  
val a2: A = new A  
  
val b1: a1.B = new a1.B // a1.B는 타입이다.  
val b2: a2.B = new a2.B // a1.B와 a2.B는 다른 타입이다.  
b1 === b2 // 컴파일 에러
```

# Path Dependent type - 경로 의존적인 제약

```
val b1: a1.B = new a1.B // a1.B는 타입이다.
```

```
val b2: a2.B = new a2.B // a1.B와 a2.B는 다른 타입이다.
```

```
a1.b = Some(b1)
```

```
a2.b = Some(b1) // does not compile
```

Dependent Type Programming

# Map - 탑 정보를 정확하게 유지한다.

```
val strIntMap: Map[String, Int] = Map("width" -> 120)
```

## Map - 탑 정보를 정확하게 유지한다.

```
val strIntMap: Map[String, Int] = Map("width" -> 120)
```

String이 Map에 Value 타입으로 들어온다면?

## Map - 탑 정보를 정확하게 유지하지 못한다.

```
val strIntMap: Map[String, Int] = Map("width" -> 120)
```

String이 Map에 Value 타입으로 들어온다면? Any

```
val strAnyMap: Map[String, Any] = strIntMap + ("sort" -> "time")
```

# Map - 탑 정보를 정확하게 유지하지 못한다.

```
val strIntMap: Map[String, Int] = Map("width" -> 120)
```

String이 Map에 Value 타입으로 들어온다면? Any

```
val strAnyMap: Map[String, Any] = strIntMap + ("sort" -> "time")
```

// 탑 정보는 어디로 갔나?

```
val width: Option[Any] = map2.get("width")
val sort: Option[Any] = map2.get("sort")
```

# HMap - Heterogenous Map<sup>1</sup>

Dependent type을 이용해 타입 정보를 보존해보자!

<sup>1</sup> Dotty : <https://www.slideshare.net/Odersky/from-dot-to-dotty>

# HMap - Heterogenous Map<sup>1</sup>

Dependent type을 이용해 타입 정보를 보존해보자!

```
trait Key { type Value }
trait HMap {
  def get(key: Key): Option[key.Value] // key의 Value, dependent type!
  def add(key: Key)(value: key.Value): HMap
}
```

<sup>1</sup> Dotty : <https://www.slideshare.net/Odersky/from-dot-to-dotty>

```
val sort = new Key { type Value = String }
val width = new Key { type Value = Int }
```

```
val sort = new Key { type Value = String }
val width = new Key { type Value = Int }
```

저장할때 Key와 연관된 Value 타입만 저장가능

```
val hmap: HMap = HMap.empty
  .add(width)(120)
  .add(sort)("time")
  .add(width)(true) // doesn't compile, width는 Int Value를 가진다.
```

```
val sort = new Key { type Value = String }
val width = new Key { type Value = Int }
```

저장할때 Key와 연관된 Value 타입만 저장가능

```
val hmap: HMap = HMap.empty
  .add(width)(120)
  .add(sort)("time")
  .add(width)(true) // doesn't compile, width는 Int Value를 가진다.
```

값을 가져올때 Value 타입이 온전히 유지된다.

```
val optionInt: Option[Int] = hmap.get(width)
val optionString: Option[String] = hmap.get(sort)
```

HMap 구현은?

Martin Ordersky 발표자료에 안나와있다. 😢

# HMap 뜯! 9줄

```
trait HMap { self =>
    val underlying: Map[Any, Any]

    def get(key: Key): Option[key.Value] =
        underlying.get(key).map(_.asInstanceOf[key.Value])

    def add(key: Key)(value: key.Value): HMap =
        new HMap {
            val underlying = self.underlying + (key -> value)
        }
}
```

# Believing that: Life is Study!<sup>4</sup>



<sup>4</sup> <https://twitter.com/ktosopl>



Once your code compiles it usually works.

# Back up side

시간이 남았다면?

# Back up side

시간이 남았다면?

우리는 서비스를 하는데 너무 코드 이야기만 했나요?

# Back up side

시간이 남았다면?

우리는 서비스를 하는데 너무 코드 이야기만 했나요?

조금더 실용적인 이야기를 해볼까요?

# Back up side

시간이 남았다면?

우리는 서비스를 하는데 너무 코드 이야기만 했나요?

조금더 실용적인 이야기를 해볼까요?

# **Scala in Real World.**

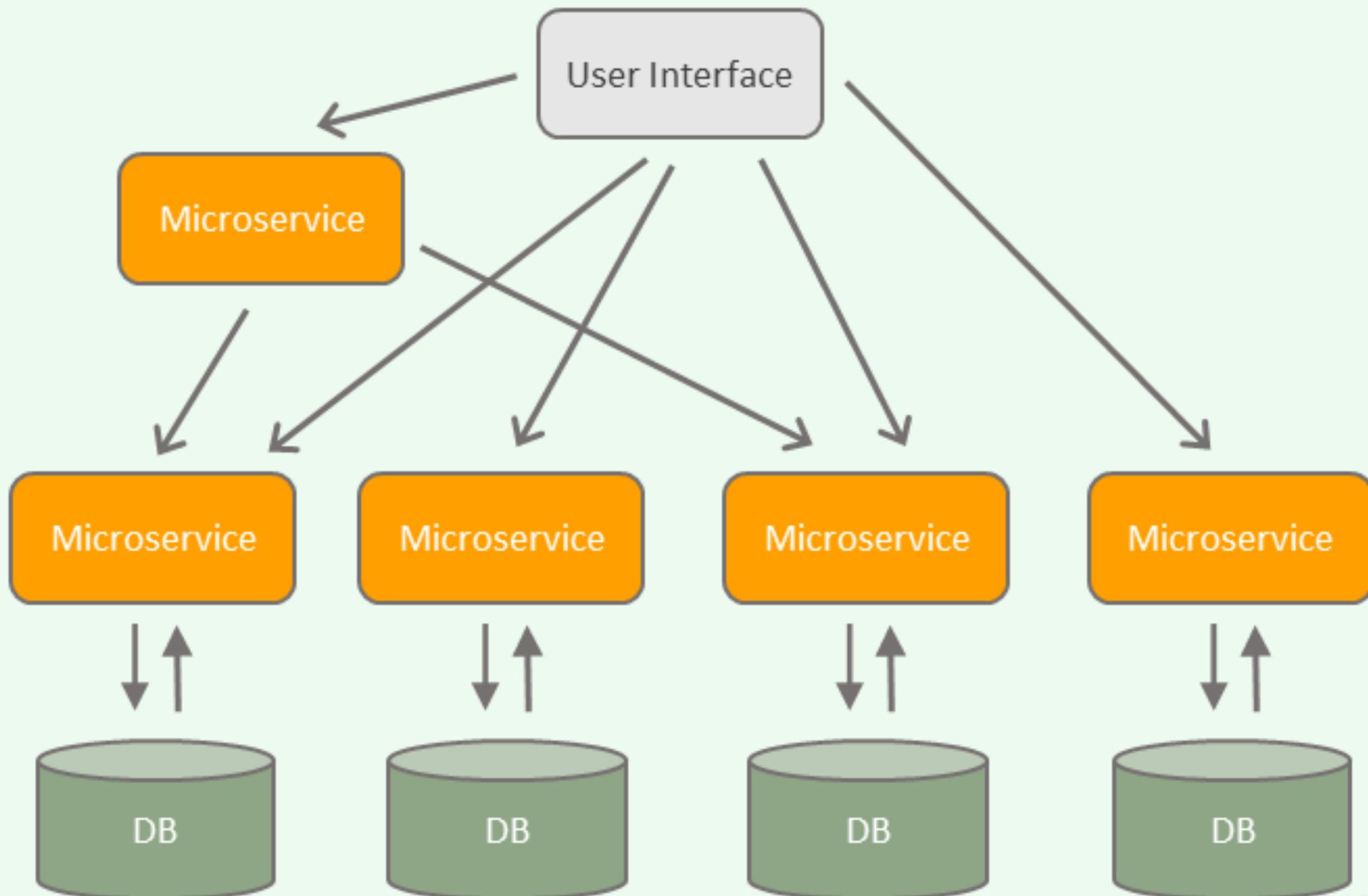
Parallel Programming in Micro Service Architecture.<sup>5</sup>

<sup>5</sup> <http://tech.kakao.com/2017/09/02/parallel-programming-and-applicative-in-scala/>

## MONOLITHIC ARCHITECTURE



## MICROSERVICES ARCHITECTURE



# 모노로틱 아키텍쳐에서 개발하기

대부분의 데이터는 한곳에 저장되어 있다.

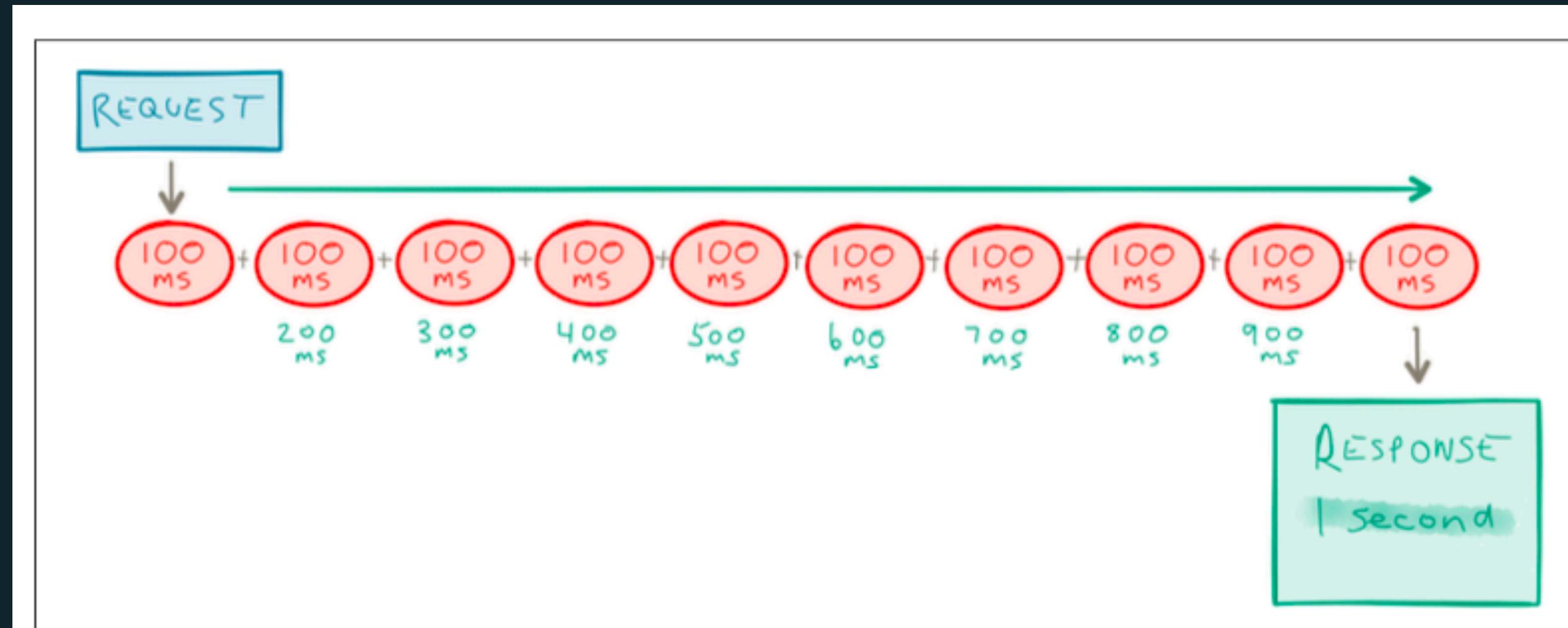
```
SELECT *
FROM
    items, catalogs, wishes, categories, details, certifications
WHERE
    items.id = ?
    AND items.id = catalogs.itemId
    AND items.id = wishes.itemId
    ...
```

# マイクロ 서비스에서 개발하기

데이터는 분산되어 있습니다.

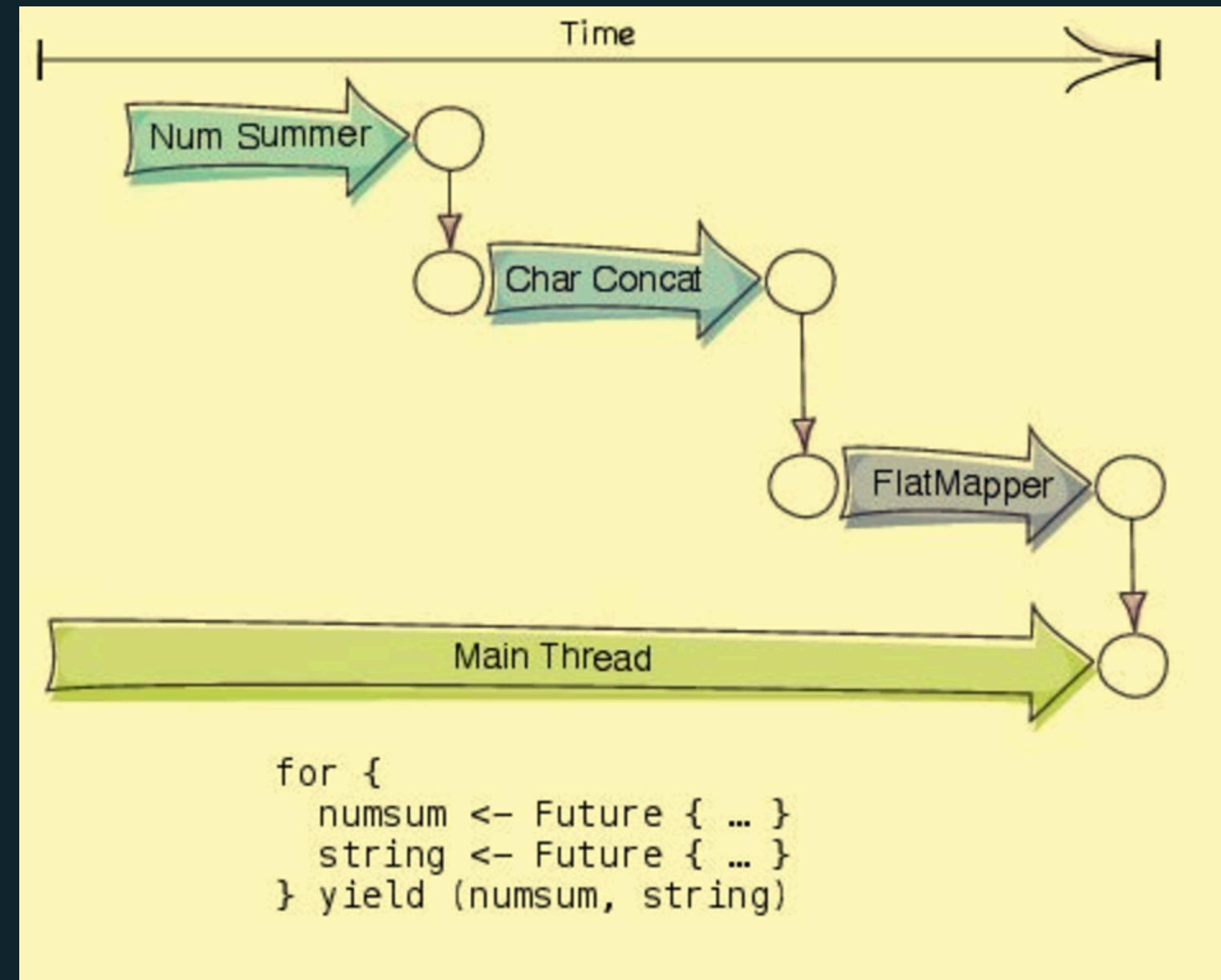
```
val product: Future[ProductDto] = for {
    item <- itemRepository.findById(itemId)                                // 분리
    catalog <- catalogRepository.findById(item.catalogId)                  // 되어
    brand <- brandRepository.findById(item.brandId)                        // 있으니
    wish <- itemWishCountRepository.findByItemId(item.id)                  // 독립적으로
    category <- categoryRepository.findOneByBrandId(item.brandId)          // 데이터를
    detail <- itemDetailRepository.findById(item.id)                        // 가져
    cert <- itemCertificationRepository.findById(item.id)                  // 온다
} yield ProductFactory.of(item, brand, catalog, wish, category, detail)
```

# 순서는? 넘나 질서 정렬한것

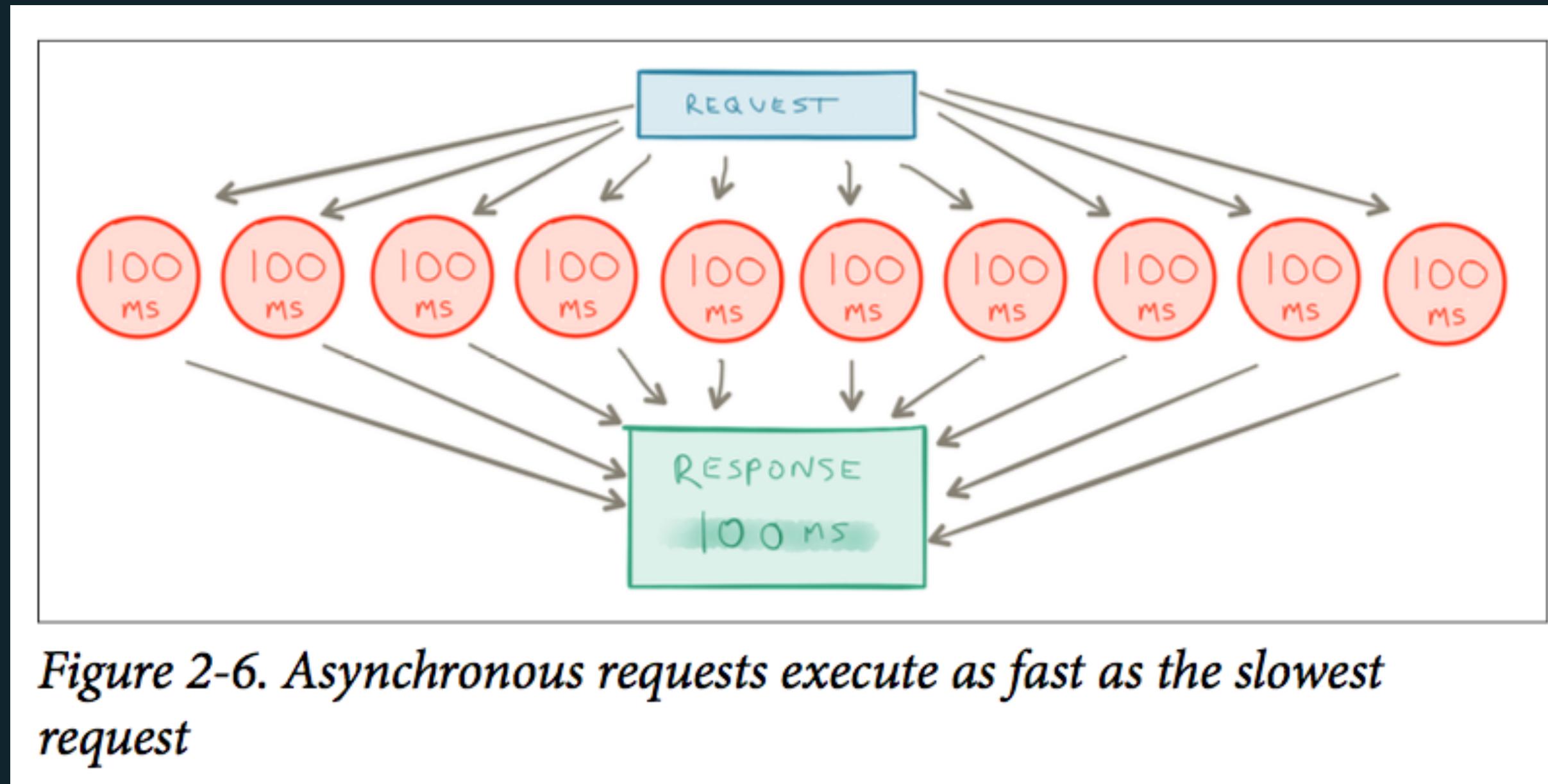


*Figure 2-5. Synchronous requests increase latency*

# 비동기로 동작하기만 병렬은 아니다.



# 병렬 프로그래밍!!! 🚀



# 어떻게 병렬 프로그래밍을 할수 있을까요?

방법은 다양하지만 Applicative를 이용한 방법을 추천합니다.

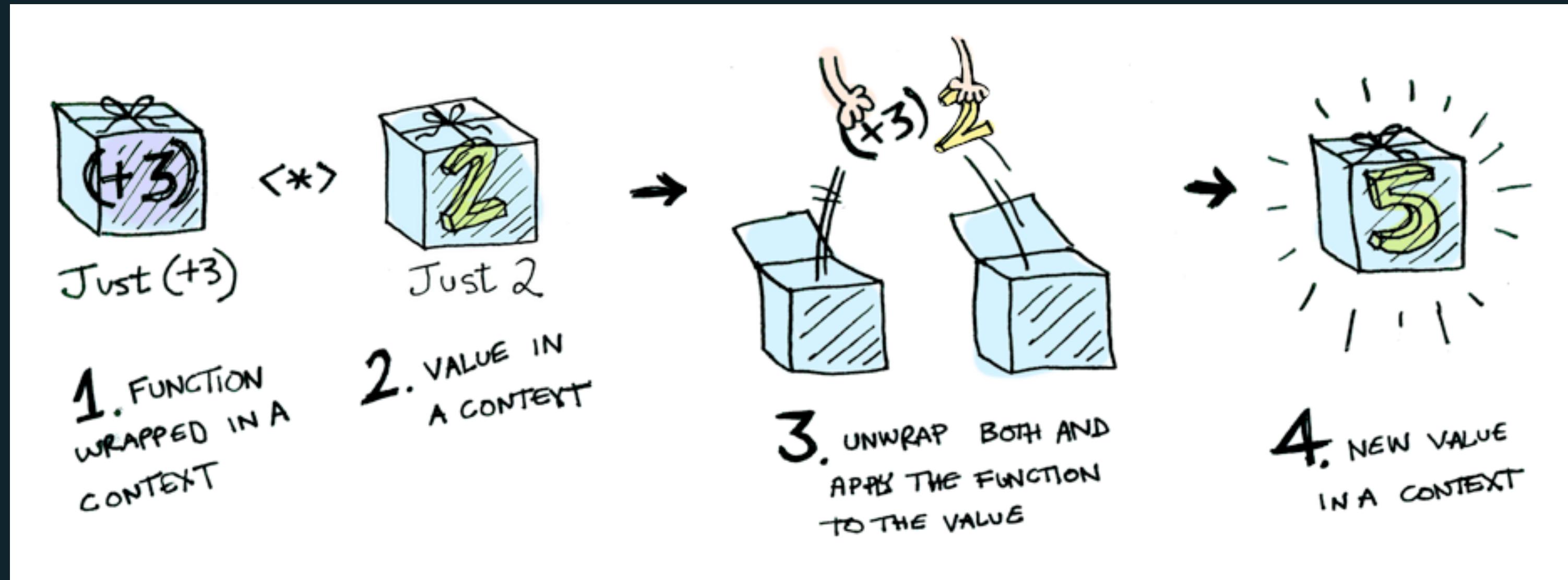
# Applicative?



뭐래는거야.

뭐라고 뭐라고  
자들끼리 쟁알대.

# Applicative - 그림 버전



# Applicative - 코드 버전

```
trait Applicative[F[_]] {
    def pure[A](a: A): F[A]

    def ap[A, B](fa: F[A])(ff: F[A => B]): F[B]
}
```

# Applicative - ap함수 구현하기

```
import scala.concurrent.Future

val futureAp = new Applicative[Future] {
    def pure[A](a: A) = Future.successful(a)

    def ap[A, B](fa: Future[A])(ff: Future[A => B]): Future[B] =
        ff.zip(fa).map { case (f, a) => f(a)}
}
```

# Applicative can be parallel

```
itemRepository.findById(item.id).flatMap { item =>
  (
    catalogRepository.findById(item.catalogId) |@|
    brandRepository.findById(item.brandId) |@|
    itemWishCountRepository.findById(item.id) |@|
    categoryRepository.findOneByBrandId(item.brandId) |@|
    itemDetailRepository.findById(item.id) |@|
    itemCertificationRepository.findById(item.id)
  ).map { case (catalog, brand, wish, category, detail, cert) =>
    List(brand, catalog, wish, category, detail, cert)
  }
}
```





# Background - Phantom type 😈

Phantom(유령) type은 Compile time에만 존재하는 타입

Runtime에는 사라짐

```
trait Phantom          // 빈 트레이트를 만듬
type MyInt = Int with Phantom // Phantom type을 mixin한 타입을 만듬
```

## 타입에 관한 문제점

Int has **no meaning**

A Int can contains **anything**

Int는 42억 9496만 7296개의 숫자로 이루어져 있습니다.

## 타입에 관한 문제점

```
def findItemsBy(brandId: Int, itemId: Int,  
                status: Int, page: Int, size: Int = 20) =  
  items.filter(_.brandId == brandId).drop(page * size).take(size)  
  
// ItemService.scala  
// 아무 문제없이 잘 동작하는 코드이다.  
findItemsByBrandId(brandId, itemId, status, page, size)
```

# 코드가 변하면?

```
// catalogId가 추가된다, 보기 이쁘게 Id끼리 묶어놔야지
def findItemsBy(brandId: Int, itemId: Int, catalogId: Int,
                status: Int, page: Int, size: Int = 20) =
  items.filter(_.brandId == brandId).drop(page * size).take(size)
```

```
// BrandService.scala
// API를 변경한 사람은 신경써서 넣었다.
findItemsByBrandId(brandId, itemId, catalogId, status, page, size)
```

# 하지만 과거의 코드는?

```
// catalogId가 추가된다, 보기 이쁘게 Id끼리 묶어놔야지
def findItemsBy(brandId: Int, itemId: Int, catalogId: Int,
                status: Int, page: Int, size: Int = 20) =
  items.filter(_.brandId == brandId).drop(page * size).take(size)

// BrandService.scala
// API를 변경한 사람은 신경써서 넣었다.
findItemsByBrandId(brandId, itemId, catalogId, status, page, size)

// ItemService.scala
// 그리고 기존에 호출하던 코드는 컴파일이 잘된다?
findItemsByBrandId(brandId, itemId, status, page, size)
```

물론 유닛 테스트 열심히 짜면 다 해결할수 있습니다.

하지만 컴파일러도 뭔가 해줄수 있는게 있지 않을까요?

# User defined type

Compiler가 도움 줄수 있게 각각의 인자에 대한 타입을 만들자.

```
case class ItemId(value: Int) extends AnyVal  
case class CatalogId(value: Int) extends AnyVal  
case class BrandId(value: Int) extends AnyVal  
case class Status(value: Int) extends AnyVal  
case class Page(value: Int) extends AnyVal  
case class Size(value: Int) extends AnyVal
```

# 만든 타입을 이용해서 API바꾸자

```
def findItemsByBrandId(  
    brandId: BrandId, itemId: ItemId, catalogId: CatalogId,  
    status: Status, page: Page, size: Size = Size(20)  
): List[Item] =  
    items.filter(_.brandId == brandId.value)  
    .drop(page.value * size.value)  
    .take(size.value)  
  
findItemsByBrandId(brandId, itemId, status, page, size) // doesn't compile, 실수 방지 😊
```

# 그러나? 내부 구현이 다 바뀌었다. 🤔

```
// before
def findItemsBy(
    brandId: Int, itemId: Int, catalogId: Int,
    status: Int, page: Int, size: Int = 20): List[Item] =
  items
    .filter(_.brandId == brandId)
    .drop(page * size)
    .take(size)

// after
def findItemsByBrandId(
    brandId: BrandId, itemId: ItemId, catalogId: CatalogId,
    status: Status, page: Page, size: Size = Size(20)): List[Item] =
  items.filter(_.brandId == brandId.value)
    .drop(page.value * size.value) // 여기
    .take(size.value) // 여기도
```

좋은건가?  나쁜건가? 

## Page type의 단점

값을 Int 하나만 가지고 있는 타입이지만 Int와 호환되지 않는다.

```
case class Page(value: Int) extends AnyVal
```

만약 Page가 Int의 하위 타입이 된다면? Page extends Int

Page는 Int로 상위 타입으로 변환이 될수 있다.

```
val page: Int = Page(1) // 상위 타입변환이 될것이다.
```

# **shapeless** 도움 살짝만 받겠습니다.

```
import shapeless._  
val shapely =  
  "Dependent type programming"  
::: "Generic type programming" ::: HNil
```

<https://github.com/milessabin/shapeless>

이번이 처음이자 마지막으로 도움입니다.

# Tagged type 🧟

Phantom(유령) type은 Compile time에만 존재하는 타입(복습)  
TypeTag @@은 Phantom type 타입 생성자

```
import shapeless.tag.@@  
type Page = Int @@ PageTag // Page는 Int와 PageTag 상속 받은 타입  
  
val page: Page = tag[PageTag][Int](1) // 생성하는건 좀 구리지만, 완전 나쁘진 않다
```

# Tagged type 🧟

Phantom(유령) type은 Compile time에만 존재하는 타입  
TypeTag @@은 Phantom type 타입 생성자

```
import shapeless.tag.@@  
type Page = Int @@ PageTag // Page는 Int와 PageTag 상속 받은 타입  
  
val page: Page = tag[PageTag][Int](1) // 생성하는건 좀 구리지만, 완전 나쁘진 않다  
  
// 실행시에는 PageTag는 사라지고 Int만 남음  
page.getClass // => class java.lang.Integer  
  
// 그리고 Int로 변형된다.  
val int : Int = page
```

# Phantom type을 이용하면

- 특정 타입의 하위 타입을 손쉽게 만들수 있습니다.
- 상속이 불가능한 타입(final)의 하위 타입도 만들수 있습니다.
- API나 로직의 변경 없이 호환되는 코드를 작성 할수 있습니다.
- 런타임에는 사라지기 때문에 성능에 대한 추가 이슈도 없습니다.

```
import shapeless.tag.@@
trait BrandIdTag; trait ItemIdTag
trait BrandTag; trait ItemTag
trait PageTag; trait SizeTag
object Pagable {
    type Page = Int @@ PageTag // Page는 Int의 하위 타입
    type Size = Int @@ SizeTag // Size도 Int의 하위 타입
}

def findItemsByBrandId(brandId: Int, page: Page, size: Size): List[Item] =
    items.filter(_.brandId == brandId)
        .drop(page * size) // Page => Int 로 바로 변환이 된다! 😎
        .take(size) // 바꾸지 않음

val page: Page = tag[PageTag][Int](1)
val size: Size = tag[SizeTag][Int](20)

findItemsByBrandId(1, page, size) // works
// 이제 page와 size의 순서가 바뀔일은 없다.
findItemsByBrandId(1, size, page) // doesn't compile
```