SCALA TAIWAN

WE USE SCALA IN THE REAL WORLD

AGENDA

- Problems we are facing
- Separate Data and Behavior
- What' ADT(Algebraic data type)?
- How to use Typeclass
- How to use Free Monad
- How to use FreeK
- Conclusions

WHO AM I

- Used to be a Java developer
- My Scala adventure begins at 2016
- Work for HTC, VIVEPORT Account team
- Use Twitter solutions to construct micro services

PROBLEMS WE ARE FACING

- Loads of jargon / Be afraid to learn new things
- OOP mindset
- Finatra mixes OOP and FP techniques
- Hard to change context
- Misuse Implicit Conversion / Implicit Class

OBJECT ORIENTED PROGRAMMING

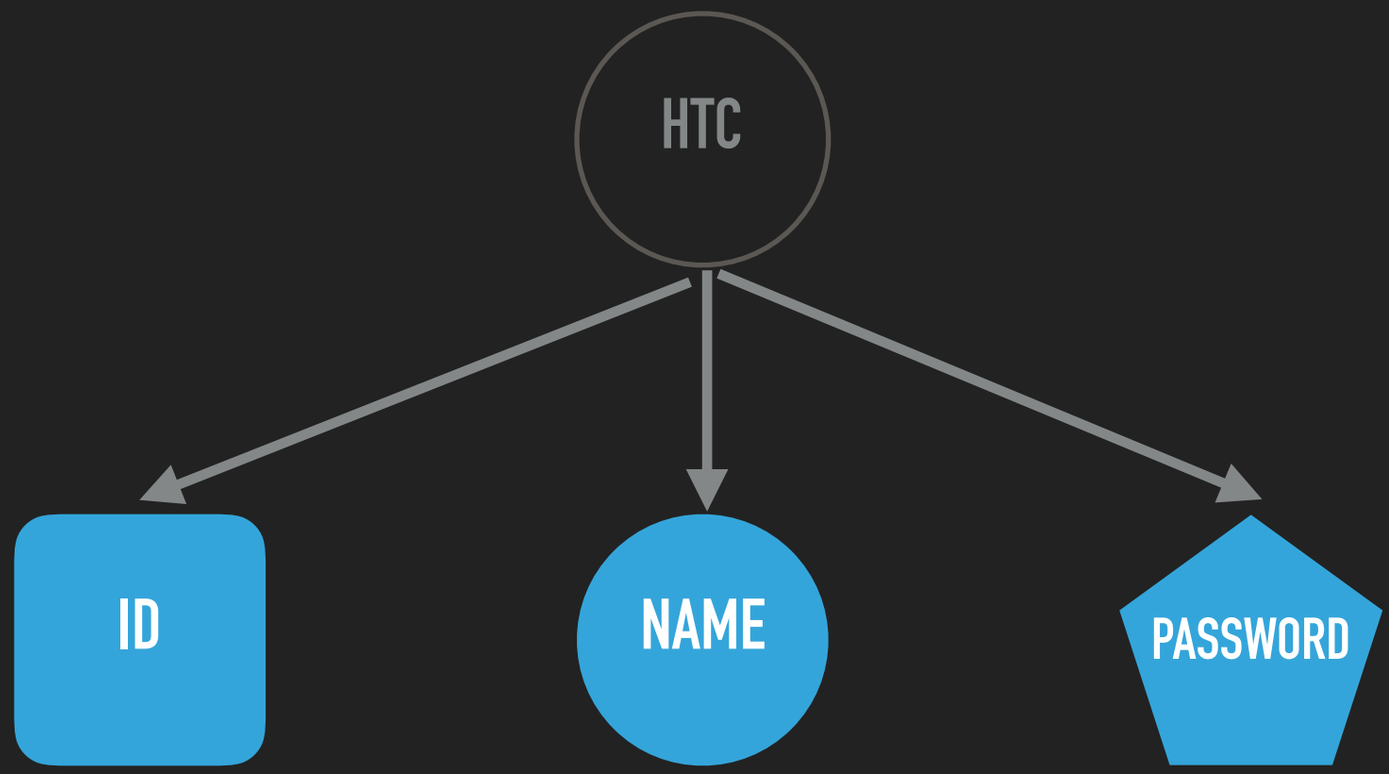
```
class User {
  // Data
  var id: String = _
  var name: String = _
  var password: String = _

  // Getter and setter
  def getId = id
  def setId(id: String) = this.id = id

  // Behaviors
  def signUp(name: String, password: String) = ???

  def signIn(id: String, password: String) = ???
}
```

SEPARATE DATA AND BEHAVIORS IN OOP

- Using “Design Patterns” to separate Data and Behavior
- Data
  - Creational patterns
- Behaviors
  - Structural patterns
  - Behavioral patterns

SEPARATE DATA AND BEHAVIORS IN FP

- Put data/function into “Context”

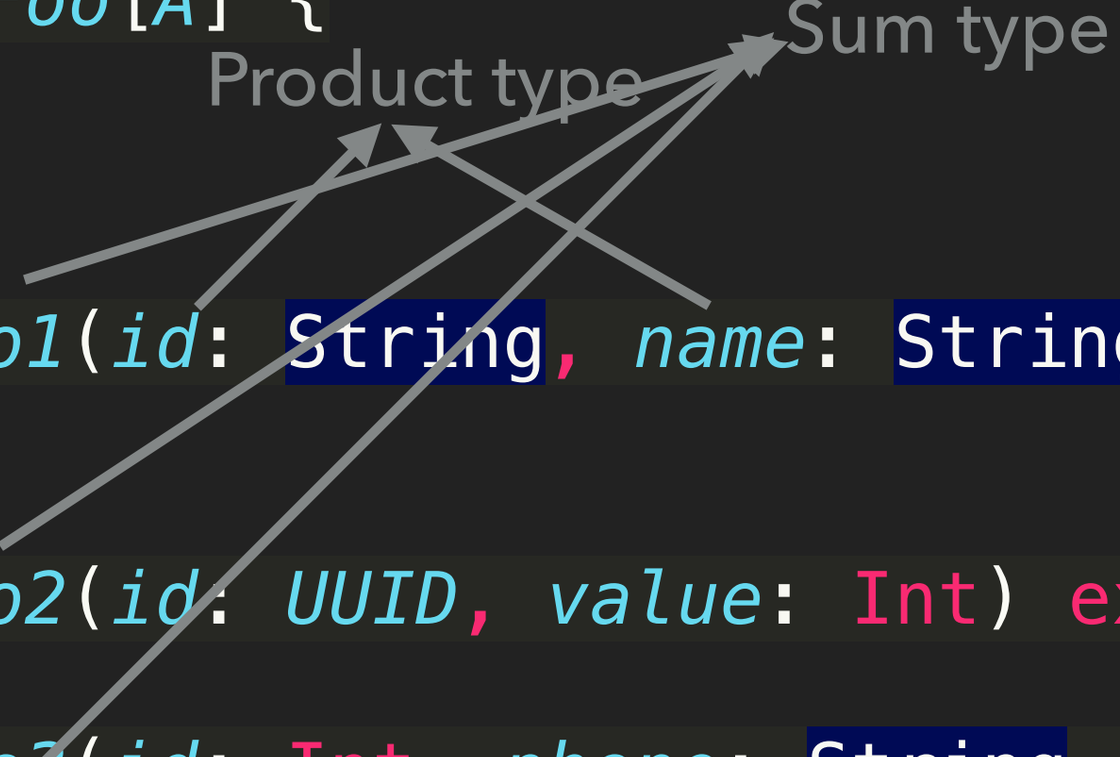


DATA

```scala
sealed trait Id[A] {
  val id: A
}

case class UserId(id: String) extends Id[String]
case class ProfileId(id: UUID) extends Id[UUID]
case class ProductId(id: Int) extends Id[Int]
```

ALGEBRAIC DATA TYPE

```scala
sealed trait Foo[A] {
  val id: A
}

case class Foo1(id: String, name: String) extends
Foo[String]

case class Foo2(id: UUID, value: Int) extends Foo[UUID]

case class Foo3(id: Int, phone: String, address:
String) extends Foo[Int]
```



DUPLICATED CODES

```
class EmailSignUpController @Inject()(emailSignUpService:
EmailSignUpService)
{
request: SignUpWithEmailRequest =>
  // Verify captcha
  // Check if email is duplicated
  emailSignUpService.signUpWithEmail(…)
  // Create profile
  // Send activation email
}
```

```
class EmailSignUpService @Inject()(repo: emailAccountRepo){
  def signUpWithEmail : TwitterFuture[Either[Errors, Unit]]
= ???
}
```

DUPLICATED CODES

```
class PhoneSignUpController @Inject()(phoneSignUpService:
PhoneSignUpService)
{
request: SignUpWithPhoneRequest =>
  // Verify captcha
  // Check if phone is duplicated
  phoneSignUpService.signUp(…)
  // Create profile
  // Send activation SMS
}
```

```
class PhoneSignUpService @Inject()(repo: phoneAccountRepo){
  def signUp : TwitterFuture[Either[Errors, Unit]] = ???
}
```

ADT

```
case class EA(email: String, name: String, address: String)
case class PA(phone: String, countryCode: String, name:
String, address: String, age: Int)

sealed trait Account[A]
case class EmailAccount(value: EA) extends Account[EA]
case class PhoneAccount(value: PA) extends Account[PA]
```

TYPECLASS

```scala
@typeclass
trait SignUp[A] {
  def signUp(a: => A): TwitterFuture[Either[Errors, A]]
}

object SignUp{
  implicit val signUpEmailAccount = new SignUp[EmailAccount] {
    override def signUp(a: => EmailAccount)(implicit repo:
EmailAccountRepo
): TwitterFuture[Either[Errors, EmailAccount]] = ???
  }

  implicit val signUpPhoneAccount = new SignUp[PhoneAccount] {
    override def signUp(a: => PhoneAccount)(implicit repo:
PhoneAccountRepo
): TwitterFuture[Either[Errors, PhoneAccount]] = ???
  }
}
```

HOW TO USE IT

```scala
import SignUp.ops._

val phoneAccount = PhoneAccount(PA("0911222333", "886",
"Stark", "Taiwan", 23))
val emailAccount = EmailAccount(EA("abc@abc.com", "Candy",
"Japan"))

// style 1
SignUp[EmailAccount].signUp(emailAccount)
SignUp[PhoneAccount].signUp(phoneAccount)


// style 2
emailAccount.signUp
phoneAccount.signUp
```

AFTER REFACTORING

```
class EmailSignUpController @Inject()(emailSignUpService:
EmailSignUpService)
{
request: SignUpWithEmailRequest =>
  // Verify captcha
  // Check if email is duplicated
  emailSignUpService.signUpWithEmail(…)
  SignUp[EmailAccount].signUp(…)
  // Create profile
  // Send activation email
}
```

```
class PhoneSignUpController @Inject()(phoneSignUpService:
PhoneSignUpService)
{
request: SignUpWithPhoneRequest =>
  // Verify captcha
  // Check if phone is duplicated
  phoneSignUpService.signUp(…)
  SignUp[PhoneAccount].signUp(…)
  // Create profile
  // Send activation SMS
}
```

USE TYPECLASS TO SOLVE FIXING CONTEXT

```scala
sealed trait Error
case class SomethingWrong(e: Throwable) extends Error

@typeclass
trait Capture[F[_]] {
  def capture[A](a: => A): F[A]
}

object Capture {
  implicit def futureCapture(implicit ec: ExecutionContext) = new Capture[Future] {
    override def capture[A](a: => A): Future[A] = Future(a)(ec)
  }

  implicit val tryCapture = new Capture[Try] {
    override def capture[A](a: => A): Try[A] = Try(a)
  }

  implicit val optionCapture = new Capture[Option] {
    override def capture[A](a: => A): Option[A] = Option(a)
  }

  implicit val eitherCapture = new Capture[Either[Error, ?]] {
    override def capture[A](a: => A): Either[Error, A] = {
      val fatal: Either[Throwable, A] = Either.catchNonFatal(a)
      val either: Either[Error, A]    = fatal.leftMap(SomethingWrong(_))
      either
    }
  }

  implicit def fgCapture[F[_]: Capture, G[_]: Capture] = new Capture[λ[α => F[G[α]]]] {
    override def capture[A](a: => A): F[G[A]] = {
      implicitly[Capture[F]].capture(implicitly[Capture[G]].capture(a))
    }
  }
}
```

USE TYPECLASS TO SOLVE FIXING CONTEXT PROBLEM

```scala
def getUser[F[_]: Capture](id: String): F[String] = {
  Capture[F].capture(s"Get a user id: ${id}")
}

private val user1: Try[String]                    = getUser[Try]("123")
private val user2: Option[String]                 = getUser[Option]("123")
private val user3: Either[Error, String]          = getUser[Either[Error, ?]]("123")
private val user4: Future[String]                 = getUser[Future]("123")
private val user5: Future[Try[String]]            = getUser[λ[α => Future[Try[α]]]]("123")
private val user6: Future[Option[String]]         = getUser[λ[α => Future[Option[α]]]]("123")
private val user7: Future[Either[Error, String]] = getUser[λ[α => Future[Either[Error, α]]]]
("123")
```

Gist: https://gist.github.com/pandaforme/2b913d5fedde6861e30b39fe476873ce

Scasite: https://scastie.scala-lang.org/pandaforme/SUqLTpyeRmiwQc8S0fB7FQ

PROS AND CONS

- Pros:
  - Compile-time type checking
  - Don’t need to have any mock framework
  - Don’t need to have any dependency injection framework
- Cons:
  - Some limitation of Simulacrum
  - Write lots of boilerplate codes without Simulacrum
  - Implicit Hell

ABUSIVE IMPLICIT CONVERSION / CLASS

```scala
object Implicits {

  implicit class ErrorsUtils(errors: Errors) {
    def toTwitterResponse(…) = ???
    def to(…)                = ???
  }

  implicit class ErrorUtils(error: Error) {
    def toHttpResponseError: HttpResponseError = ???
    def toErrors: Errors                       = ???
  }

  implicit class HttpClientUtils(httpClient: HttpClient) extends Logging {
    def safeExecute(…): Future[Maybe[Response]] = ???
    def safeExecuteJson[T: Manifest](…): Future[Maybe[T]] = ???
  }

  implicit class SwaggerUtils(operation: Operation) {
    def requestBodyWith[T: TypeTag](…): Operation = ???
    def responseBodyWith[T: TypeTag](…): Operation = ???
  }
}
```

GUIDELINE

- Implicit Design Patterns in Scala: http://www.lihaoyi.com/post/ImplicitDesignPatternsinScala.html
- Effective Scala: http://twitter.github.io/effectivescala/

HOW TO USE FREE MONAD

- Define DSL (Domain Specific Language)
- Lift to Free
- Build a program
- Implement interpreters
- Compose interpreters (order matters) and run it

DEFINE DSL (DOMAIN SPECIFIC LANGUAGE)

```
sealed trait CaptchaOps[A]
final case class Validate(captcha: Captcha) extends CaptchaOps[Unit]


sealed trait AccountOps[A]
final case class Create(account: Account) extends AccountOps[Account]
final case class Get(id: Id)              extends
AccountOps[Option[Account]]
final case class Exist(id: Id)            extends AccountOps[Unit]


sealed trait NotifyOps[A]
final case class Send(account: Account) extends NotifyOps[Unit]
```

LIFT TO FREE

```
class CaptchaOpsI[F[_]](implicit I: Inject[CaptchaOps, F]) {
  def validate(captcha: Captcha): Free[F, Unit] = Free.inject[CaptchaOps, F](Validate(captcha))
}

object CaptchaOpsI {
  implicit def captchaOpsI[F[_]](implicit I: Inject[CaptchaOps, F]): CaptchaOpsI[F] =
    new CaptchaOpsI[F]
}

class  AccountOpsI[F[_]](implicit I: Inject[AccountOps, F]) {
  def create(account: Account): Free[F, Account] = Free.inject[AccountOps, F](Create(account))
  def get(id: Id): Free[F, Option[Account]]      = Free.inject[AccountOps, F](Get(id))
  def exist(id: Id): Free[F, Unit]               = Free.inject[AccountOps, F](Exist(id))
}

object AccountOpsI {
  implicit def accountOps[F[_]](implicit I: Inject[AccountOps, F]): AccountOpsI[F] = new AccountOpsI[F]
}

class NotifyOpsI[F[_]](implicit I: Inject[NotifyOps, F]) {
  def send(account: Account): Free[F, Unit] = Free.inject[NotifyOps, F](Send(account))
}

object NotifyOpsI {
  implicit def notifyOps[F[_]](implicit I: Inject[NotifyOpsI, F]): NotifyOpsI[F] = new NotifyOpsI[F]
}
```

BUILD A PROGRAM

```
type CA[A] = Coproduct[CaptchaOps, AccountOps, A]
type ALL[A] = Coproduct[CA, NotifyOps, A]

def program(implicit C: CaptchaOpsI[ALL], A: AccountOpsI[ALL], N:
NotifyOpsI[ALL]): Free[ALL, Account] = {
  import C._
  import A._
  import N._

  for {
    _ <- validate(Captcha("abcdef"))
    _ <- exist(Id("Cher Wang"))
    a <- create(Account("Cher Wang"))
    _ <- send(a)
  } yield a
}
```

IMPLEMENT INTERPRETERS

```
val captchaOpsInterpreter = new (CaptchaOps ~> Future) {
  def apply[A](fa: CaptchaOps[A]): Future[A] = ???
}

val accountOpsInterpreter = new (AccountOps ~> Future) {
  def apply[A](fa: AccountOps[A]): Future[A] = ???
}

val notifyOpsInterpreter = new (NotifyOps ~> Future) {
  def apply[A](fa: NotifyOps[A]): Future[A] = ???
}
```

RUN IT

```
val interpreters: ALL ~> Future = NotifyOpsInterpreter or
(CaptchaOpsInterpreter or AccountOpsInterpreter)

program.foldMap(interpreters)
```

Scasite: https://scastie.scala-lang.org/pandaforme/YnwJc3UvSTarkiAQhKr80A/3

BUT . . .

- Lots of boilerplate codes
- It’s pretty awful that you want to combine multiple DSLs
- **Order matters !!!**
- Need a Monad transformer if uses multiple context
- Let’s use FreeK to combine your DSL seamlessly

LET'S USE FREEK

- Define DSL (Domain Specific Language)
- ~~Lift to Free~~
- Build a program
- Implement interpreters
- Compose interpreters ~~(order matters)~~ and run it

BEFORE AND AFTER

```
type TMP[A] = Coproduct[CaptchaOps, AccountOps, A]
type ALL[A] = Coproduct[TMP, NotifyOps, A]

def program(implicit C: CaptchaOpsI[ALL], A: AccountOpsI[ALL], N: NotifyOpsI[ALL]): Free[ALL,
Account] = {
  import C._
  import A._
  import N._

  for {
    _ <- validate(Captcha("abcdef"))
    _ <- exist(Id("Cher Wang"))
    a <- create(Account("Cher Wang"))
    _ <- send(a)
  } yield a
}
```

```
type ALL = AccountOps :|: CaptchaOps :|: NotifyOps :|: NilDSL
val program = for {
  _ <- CaptchaOps.Validate(Captcha("abcdef")).freek[ALL]
  _ <- AccountOps.Exist(Id("Cher Wang")).freek[ALL]
  a <- AccountOps.Create(Account("Cher Wang")).freek[ALL]
  _ <- NotifyOps.Send(a).freek[ALL]
} yield {
  a
}
```

BEFORE AND AFTER

```scala
val interpreters: ALL ~> Future = NotifyOpsInterpreter or
(CaptchaOpsInterpreter or AccountOpsInterpreter)

program.foldMap(interpreters)
```

```scala
val interpreters = captchaOpsInterpreter :&:
notifyOpsInterpreter :&: accountOpsInterpreter

program.interpret(interpreters)
```

Scasite: https://scastie.scala-lang.org/pandaforme/5zdHrziMRGyxSJihdZtHeA

HOW TO HANDLE MONAD TRANSFORMER

```
val captchaOpsInterpreter = new (CaptchaOps ~> Lambda[A => Future[Either[Error, A]]]) {
  override def apply[A](fa: CaptchaOps[A]): Future[Either[Error, A]] = ???
}

val accountOpsInterpreter = new (AccountOps ~> Lambda[A => Future[Either[Error, A]]]) {
  override def apply[A](fa: AccountOps[A]): Future[Either[Error, A]] = ???
}

val notifyOpsInterpreter = new (NotifyOps ~> Lambda[A => Future[Either[Error, A]]]) {
  override def apply[A](fa: NotifyOps[A]): Future[Either[Error, A]] = ???
}
```

```
type ALL = AccountOps :|: CaptchaOps :|: NotifyOps :|: NilDSL
type O   = Either[Error, ?] :&: Bulb

val program = for {
  _ <- CaptchaOps.Validate(Captcha("abcdef")).freeko[ALL, O]
  _ <- AccountOps.Exist(Id("Cher Wang")).freeko[ALL, O]
  a <- AccountOps.Create(Account("Cher Wang")).freeko[ALL, O]
  _ <- NotifyOps.Send(a).freeko[ALL, O]
} yield {
  a
}

val interpreters = captchaOpsInterpreter :&: notifyOpsInterpreter :&:
accountOpsInterpreter

program.interpret(interpreters)
```

COMPILE ERROR

```
[error] value :&: is not a member of cats.~>[Main.AccountOps,
[A]scala.concurrent.Future[scala.util.Either[Main.Error,A]]]
```

LET'S FIX IT

```scala
type FE[A] = Future[Either[Error, A]]

val captchaOpsInterpreter = new (CaptchaOps ~> FE) {
  override def apply[A](fa: CaptchaOps[A]): FE[A] = ???
}
```

```scala
val accountOpsInterpreter = new (AccountOps ~> FE) {
  override def apply[A](fa: AccountOps[A]): FE[A] = ???
}

val notifyOpsInterpreter = new (NotifyOps ~> FE) {
  override def apply[A](fa: NotifyOps[A]): FE[A] = ???
}
```

Issue: https://github.com/ProjectSeptemberInc/freek/issues/17

COMPILE ERROR AGAIN

```
[error] could not find implicit value for evidence parameter of
type cats.Monad[Main.FE]
```

LET'S FIX IT AGAIN

```scala
implicit val feMonad = new Monad[FE] {
  def flatMap[A, B](fa: FE[A])(f: (A) => FE[B]): FE[B] = ???

  def tailRecM[A, B](a: A)(f: (A) => FE[Either[A, B]]): FE[B]
= ???

  def pure[A](x: A): FE[A] = ???
}
```

Scasite: https://scastie.scala-lang.org/pandaforme/uKJ3u3NTSDK6frMPIJg5xg/1

MY THOUGHTS ABOUT FREE MONAD / FREEK

- Free Monad
  - Make sure contexts of all interpreters are same
  - Is it feasible to add extra context into DSL ?
- FreeK
  - Make your IntelliJ IDE crazy and slow
  - Need tips to solve it if interpreters are in 2 layer context
  - Not implement MonadFilter

CONCLUSIONS

- Open your mind
- Don’t think how to implement it at first
- Apply what you learn into simple project
- Learn Category Theory to dispel your doubts and fears

WE WANT YOU!
APPLY NOW

MOOD[UNHAPPY] FLATMAP SCALA

DEF SCALA: _ => MOOD[HAPPY]