Functional puzzlers

Faggruppemøte Scala & JVM

Funksjonell programmering

‹‹In computer science, functional programming is a programming paradigm, a style of building the structure and elements of computer programs, that treats computation as the evaluation of mathematical functions and avoids state and mutable data. It is a declarative programming paradigm, which means programming is done with expressions››

Pure functions

- Samme input = samme output
- Ingen observerbare side-effects
- Referential transparent

Hva er galt med koden?

```
class Cafe {
  def buyCoffee(cc: CreditCard, p: Payments): Coffee = {
    val cup = new Coffee()
    p.charge(cc, cup.price)
    cup
  }
}
```

En bedre løsning

```
class Cafe {
  def buyCoffee(cc: CreditCard): (Coffee, Charge) = {
    val cup = new Coffee()
    (cup, Charge(cc, cup.price))
  }
}
```

Oppgave: SideEffects

Higher order functions

- Funksjoner som tar inn og/eller returnerer funksjoner
- Essensiell del av FP
- map, filter, flatmap, find, etc...

Oppgave: Mapping

Rekursjon

- Den funksjonelle måten å loope på

```
def sum1(xs: List[Int]): Int = xs match {
  case Nil => 0
  case head :: tail => head + sum1(tail)
}
```

Tail recursion

- Kan optimaliseres til en loop av kompilator

```
def sum2(xs: List[Int]): Int = {
  def loop(acc: Int, xs: List[Int]): Int = xs match {
    case Nil => acc
    case head :: tail => loop(acc + head, tail)
  }
  loop(0, xs)
}
```

Fold

- Hånterer rekursjonen for oss
- Brukes ofte til å redusere lister til ett resultat

```
def sum1(xs: List[Int]) = xs.foldLeft(0)(_ + _)
List(1, 2, 3).foldLeft(0)(_ + _) == (((0 + 1) + 2) + 3)

def sum2(xs: List[Int]) = xs.foldRight(0)(_ + _)
List(1, 2, 3).foldRight(0)(_ + _) == (1 + (2 + (3 + 0)))
```

Oppgaver: Recursion & Folds

Option

- Bli kvitt *null* og *NullPointerException* for godt

```
sealed trait Option[+A]
case class Some[+A](get: A) extends Option[A]
case object None extends Option[Nothing]
```

Option

- *Option* kan bli sett på som en collection, men kan holde på maksimalt ett element.
  - *map*, *flatMap*, *filter*, *exists*, *foreach*, *head*, *tail*, *isEmpty*, osv...

Option

- Den mest idiomatiske måten å benytte *Option* på er å behandle den som en collection eller monade, ved å bruke *map*, *flatMap*, *filter*, eller bruke den i en *for comprehension*.

```
def avdeling(navn: String): String =
    ansattelisten.find(_.navn == navn)
        .map(_.avdeling)
        .filter(_ != "BMC")
        .getOrElse("BEKK")
```

Option

- Støtter også eksplisitt pattern matching

```
def avdeling(navn: String): String =
    ansattelisten.find(navn).map(_.avdeling)  match {
        case Some("BMC") => "Bekk Management Consulting"
        case Some(x) => x
        case None => "BEKK"
    }
```

Try

- Fungerer på samme måte som en *Option*. *Success* inneholder en verdi, mens *Failure* inneholder en exception.

```
sealed trait Try[+T]
case class Success[+T](value: T) extends Try[T]
case class Failure[T <: Nothing](exception: Throwable) extends Try[T]
```

Either

- Either representerer på samme måte som *Option* en av to muligheter, men denne gangen er begge en verdi.

```
sealed trait Either[+E, +A]
case class Left[+E](value: E) extends Either[E, Nothing]
case class Right[+A](value: A) extends Either[Nothing, A]
```

- Av konvensjon er *Left* forbeholdt feilsituasjoner, mens *Right* brukes til suksessverdien.

Oppgave: OptionalValues

Eksempel: DependencyInjection