# Funktionale Datenstrukturen

## Was macht funktionale Datenstrukturen aus?

- unveränderbar
- basieren auf algebraischen Strukturen
- Operationen auf Datenstrukturen erzeugen neue Datenstrukturen
- haben ähnliche Eigenschaften wie Wertetypen
- Effiziente Konzepte für persistent Collections (hashtables, finger trees, ...)

## Wie kann man ADTs in Scala implementieren?

- ADTs sind algebraische Datentypen
- ADTs können in Scala durch `sealed trait` und `case class` implementiert werden

```
sealed trait Shape // nur in der Datei definierte Klassen können Shape
erweitern
case class Rect(pos: Point, w: Int, h: Int) extends Shape
case class Circle(pos: Point, radius: Int) extends Shape
```

## Was ist besonders an case classes?

- unveränderbar (public final)
- haben automatisch `equals`, `hashCode`, `toString` und `copy` Methoden
- erlauben Pattern Matching

## Was kann man noch verwenden um einfache ADTs zu implementieren?

- Enumerationen mit enum classes

```
enum Shape :
    case Rect(pos: Point, w: Int, h: Int) extends Shape
    case Circle(pos: Point, radius: Int) extends Shape
```

## Wie funktioniert Pattern Matching in Scala?

- überprüft Typ und reagiert darauf
- arbeitet mit Mustern, die mit den Werten im Code verglichen werden

```
x match {
  case pattern1 => result1
  case pattern2 => result2
```

```
  case _ => defaultResult  // catch-all Pattern für alle anderen Fälle
}
```

- x ist der Wert, der überprüft wird
- case pattern => result ist ein Pattern, das mit x verglichen wird und

```
val x = 5
x match {
  case 1 => println("Eins")
  case 2 => println("Zwei")
  case 3 => println("Drei")
  case _ => println("Andere Zahl")
}
```

## Was ist Option in Scala?

- Option ist ein ADT, der entweder einen Wert Some oder None enthält

```
val optPrime : Option[Int] = list123.find(x => isPrime(x)) // erstes Element,
das die Bedingung erfüllt oder None

optPrime match {
  case Some(x) => println(x)
  case None => println("Keine Primzahl gefunden")
}
```

## Implementiere eine Funktionale Liste in Scala

```
sealed trait FList[+A] :
    val isEmpty : Boolean
    val size : Int

    def find(pred : A => Boolean) : Option[A] =
        this match {
            case FNil => None
            case FCons(head, tail) =>
                if pred(head) then Some(head)
                else tail.find(pred)
        }

    def filter(pred : A => Boolean) : FList[A] =
        this match {
            case FNil => FNil
            case FCons(head, tail) =>
                if pred(head) then FCons(head, tail.filter(pred))
```

```
                else tail.filter(pred)
        }

    def map[B](f : A => B) : FList[B] =
        this match {
            case FNil => FNil
            case FCons(head, tail) => FCons(f(head), tail.map(f))
        }

    def all(pred : A => Boolean) : Boolean =
        this match {
            case FNil => true
            case FCons(head, tail) => pred(head) && tail.all(pred)
        }

    def any(pred : A => Boolean) : Boolean =
        this match {
            case FNil => false
            case FCons(head, tail) => pred(head) || tail.any(pred)
        }

case object FNil extends FList[Nothing] :
    val isEmpty = true
    val size = 0

case class FCons[+A](head: A, tail: FList[A]) extends FList[A] :
    val isEmpty = false
    val size = 1 + tail.size
```

## Was sind persistente Datenstrukturen?

Prinzip: Änderungen an einer Datenstruktur erzeugen eine neue Datenstruktur, die die Änderung enthält. Die alte Datenstruktur bleibt unverändert, minimale Kopieroperationen, maximale Wiederverwendung von unveränderten Teilen.

## Was sind Vor- und Nachteile von persistente Datenstrukturen?

Vorteile:

- einfach zu verwenden
- zuverlässig
- thread-safe

Nachteile:

- Overhead bei Speicher und Laufzeit

## Wie funktioniert partition(pred: T => Boolean) und welches Ergebnis liefert es?

- teilt Sammlung in zwei Teile auf, die durch die Bedingung pred getrennt sind

```
val (even, odd) = List(1, 2, 3, 4, 5).partition(x => x % 2 == 0)
// even = List(2, 4), odd = List(1, 3, 5)
```

## Wie unterscheiden sich map und foreach?

- map gibt eine neue Liste zurück, die durch die Funktion verändert wurde
- foreach führt die Funktion auf jedem Element aus, gibt aber keine Liste zurück

## Was ist der Unterschied zwischen get(k) und apply(k) bei Map?

- get gibt ein Option zurück
- apply wirft eine Exception, wenn der Key nicht existiert

## Wie erstellt man eine neue Liste mit einer for-Schleife?

- mit yield

```
val list = for i <- 1 to 10 yield i * 2
// list = List(2, 4, 6, 8, 10, 12, 14, 16, 18, 20)
```