Vecka 8. Nästlade och generiska strukturer

Programmering, grundkurs (pgk)

Björn Regnell

Datavetenskap, LTH, Lunds universitet
https://lunduniversity.github.io/pgk

EDAA45, Lp1-2, HT2025

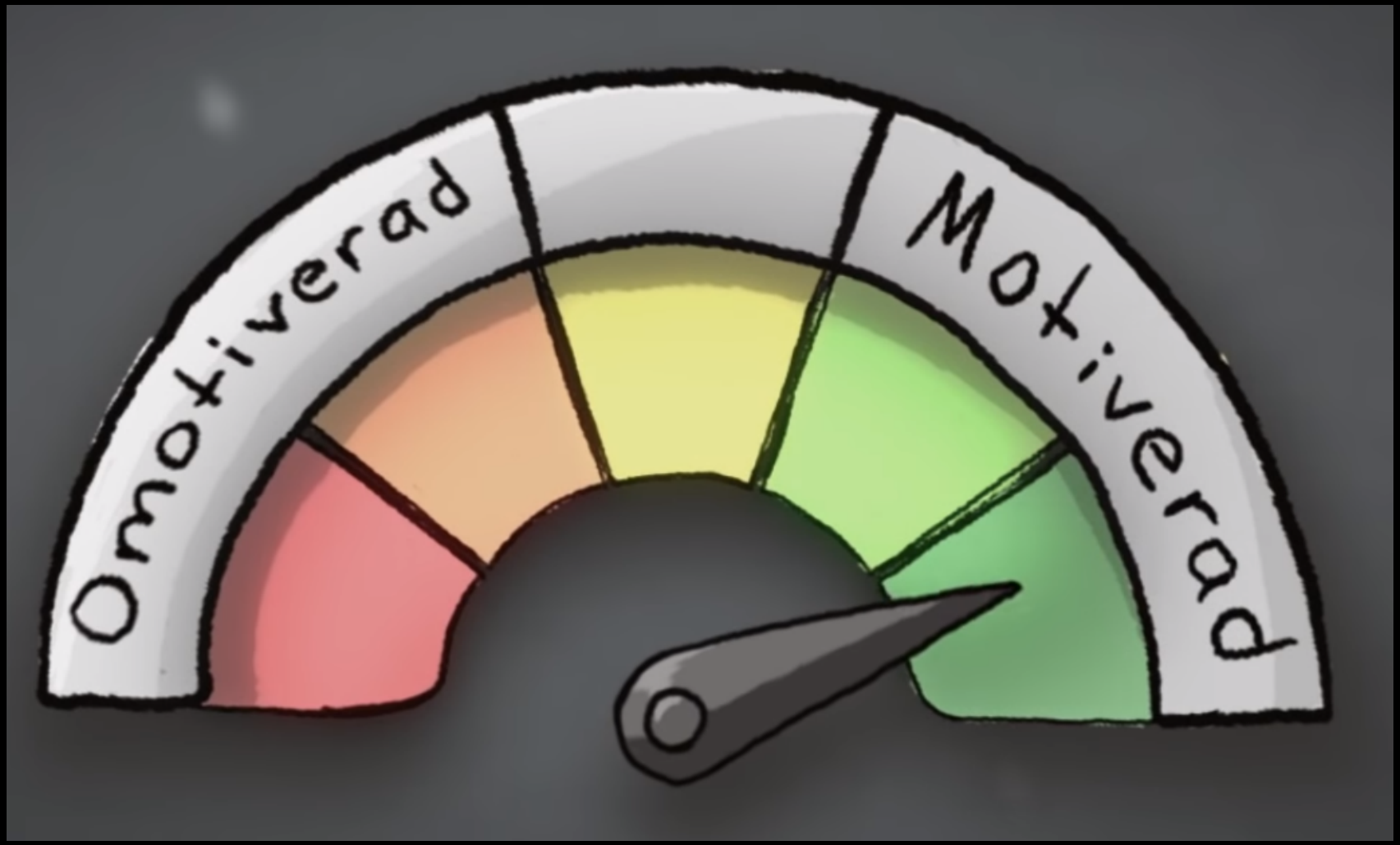
Kompilerad den 22 augusti 2025

8 Nästlade och generiska strukturer

- Veckans labb: `life`
- Matriser
- Typparametrar
- Upptäcka och åtgärda buggar

Självdiagnostik och planering

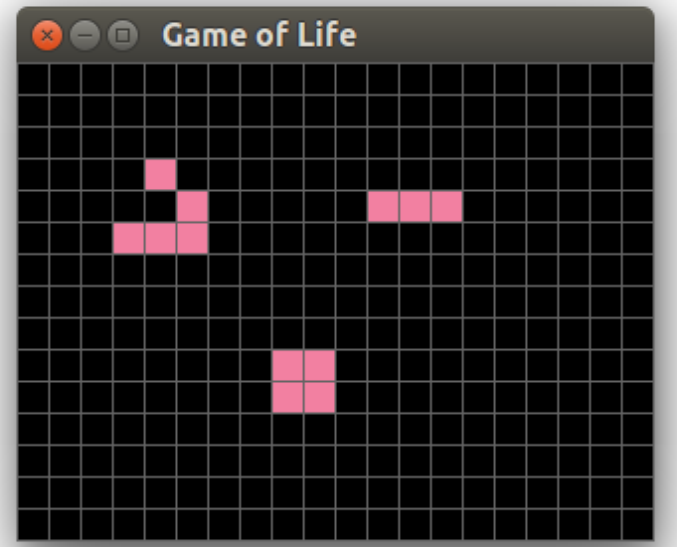
- Hur ska jag plugga?
  - Identifiera dina **behov** (trösklar, luckor, intressen...)
  - Gör ett **schema** dag för dag.
  - Dela upp varje pass i delar: teori, övningar, labbar.
  - Övningarnas syfte: att utöka din verktygslåda.
  - Identifiera övningar från lp1 om det du behöver träna på.
- Delta i all undervisning; dra nytta av höga lärartätheten!
- Alla får gå på extra resurtider i mån av plats!
  OBS! **Utöver** din **ordinarie** tid – hoppa inte över den!
  Prioritet om fullt i salen:
  1. De som deltar på sin ordinarie tid.
  2. De som varit sjuka och har labbar efter sig.
  3. De som behöver extra hjälp med svårigheter.
  4. Alla andra.


Omotiverad
Motiverad

Veckans labb: `life`

Veckans labb: `life`

- Universum är en binär matris av **celler** där **levande** celler representeras med **`true`** och **döda** med **`false`**.
- Följande regler gäller för **nästa generation** celler i universum:
  - **Fortlevnad**: en levande cell med 2 eller 3 grannar **lever vidare**
  - **Död**: en levande cell med färre än 2 eller fler än 3 grannar **dör**
  - **Födelse**: en död cell med exakt tre grannar föds
- Övning `matrices` uppgift 5: skapa en generisk **`case class`** `Matrix[T]`
- På labben: använd `Matrix[Boolean]`



- Du ska simulera *Game of Life* i ett `introprog.PixelWindow`
- Fördjupning: `https://en.wikipedia.org/wiki/Conway%27s_Game_of_Life`

Matriser

Vad är en matris?

- En **matris** inom **matematiken** innehåller **rader** och **kolumner**[1] med tal.
- I en **matematisk** matris har alla rader **lika många** element och
- även alla kolumner har **lika många** element.
- En matris av dimension 2×5 har $2 \cdot 5 = 10$ stycken element.
- Exempel på en matematisk matris av dimension 2×5:

$$M_{2,5} = \begin{pmatrix} 5 & 2 & 42 & 4 & 5 \\ 3 & 4 & 18 & 6 & 7 \end{pmatrix}$$

[1] även kallade *kolonner*

Indexering i en matris

- En matris av dimension $m \times n$ har $m \cdot n$ stycken element.
- En matris $A_{m,n}$ av dimension $m \times n$ ritas inom matematiken ofta så här:

$$A_{m,n} = \begin{pmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,n} \\ a_{2,1} & a_{2,2} & \cdots & a_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m,1} & a_{m,2} & \cdots & a_{m,n} \end{pmatrix}$$

- Matrisindexering inom matematiken sker ofta från 1, men ofta från 0 i datorprogram.
- Vad har talet 42 för index i matrisen $M_{2,5}$ nedan?
  - Inom matematiken?
  - I Scala och Java och många andra språk?

$$M_{2,5} = \begin{pmatrix} 5 & 2 & 42 & 4 & 5 \\ 3 & 4 & 18 & 6 & 7 \end{pmatrix}$$

Hur skapa matriser?

- Inom programmering används ordet **matris** ofta för att beteckna en **nästlad struktur** i två dimensioner. Exempel:
  - **Oföränderliga** sekvenser, t.ex. `Vector[Vector[Int]]`
    ```
    val xss = Vector(Vector(0, 0, 0), Vector(0, 0, 0))
    ```
    eller enklare:
    ```
    val xss = Vector.fill(2,3)(0)
    ```
  - **Föränderliga** sekvens, t.ex. `Array[Array[Int]]`
    ```
    val yss = Array(Array(0, 0, 0), Array(0, 0, 0))
    ```
    eller enklare:
    ```
    val yss = Array.fill(2,3)(0)
    ```

Hur indexera i matriser?

En matris med array av arrayer:

```
1 scala> val xss = Array(Array(5,2,42,4,5),Array(3,4,18,6,7))
2 xss: Array[Array[Int]] = Array(Array(5, 2, 42, 4, 5), Array(3, 4, 18, 6, 7))
```

Hur indexera i matriser?

En matris med array av arrayer:

```
scala> val xss = Array(Array(5,2,42,4,5),Array(3,4,18,6,7))
xss: Array[Array[Int]] = Array(Array(5, 2, 42, 4, 5), Array(3, 4, 18, 6, 7))
```

Man indexerar i en nästlad sekvens med upprepad `apply`:

```
scala> xss(0)(2)
res0: ???

scala> xss.apply(0).apply(2)
res1: ???

scala> xss(0)
res2: ???
```

Övning: Vad är typ och värde vid ??? ovan?

Hur indexera i matriser?

En matris med array av arrayer:

```
scala> val xss = Array(Array(5,2,42,4,5),Array(3,4,18,6,7))
xss: Array[Array[Int]] = Array(Array(5, 2, 42, 4, 5), Array(3, 4, 18, 6, 7))
```

Man indexerar i en nästlad sekvens med upprepad `apply`:

```
scala> xss(0)(2)
res0: Int = 42

scala> xss.apply(0).apply(2)
res1: Int = 42

scala> xss(0)
res2: Array[Int] = Array(5, 2, 42, 4, 5)
```

Övning: Rita en bild av minnet som referensen `xss` refererar till.

Uppdatering av en förändringsbar nästlad struktur

Man kan förändra en array av arrayer ”på plats” med tilldelning:

```
1  scala> val xss = Array(Array(5,2,42,4,5),Array(3,4,18,6,7))
2
3  scala> xss(0)(0) = 100
4
5  scala> xss
6  res0: ???
7
8  scala> xss(0)(2) = xss(0)(2) - 1
9
10 scala> xss
11 res1: ???
12
13 scala> xss(1) = Array.fill(5)(-1)
14
15 scala> xss
16 res2: ???
```

Uppdatering av en förändringsbar nästlad struktur

Man kan förändra en array av arrayer ”på plats” med tilldelning:

```
1  scala> val xss = Array(Array(5,2,42,4,5),Array(3,4,18,6,7))
2
3  scala> xss(0)(0) = 100
4
5  scala> xss
6  res0: Array[Array[Int]]=Array(Array(100, 2, 42, 4, 5), Array(3, 4, 18, 6, 7))
7
8  scala> xss(0)(2) = xss(0)(2) - 1
9
10 scala> xss
11 res1: Array[Array[Int]]=Array(Array(100, 2, 41, 4, 5), Array(3, 4, 18, 6, 7))
12
13 scala> xss(1) = Array.fill(5)(-1)
14
15 scala> xss
16 res2: Array[Array[Int]]=Array(Array(100, 2, 41, 4, 5), Array(-1,-1,-1,-1,-1))
```

Några olika sätt att skapa förändringsbara matriser

Det jobbiga, primitiva sättet:

```
scala> val xss = new Array[Array[Int]](2)
xss: Array[Array[Int]] = Array(null, null)

scala> for (i <- xss.indices) {xss(i) = new Array[Int](5)}

scala> xss
res0: Array[Array[Int]] = Array(Array(0, 0, 0, 0, 0), Array(0, 0, 0, 0, 0))

scala> println(xss)
[[I@196a99d0
```

Enklare sätt:

```
scala> val xss = Array.ofDim[Int](2,5)
xss: Array[Array[Int]] = Array(Array(0, 0, 0, 0, 0), Array(0, 0, 0, 0, 0))
```

Enklare och tydligare sätt, där initialvärdet anges explicit:

```
scala> val xss = Array.fill(2,5)(0)
xss: Array[Array[Int]] = Array(Array(0, 0, 0, 0, 0), Array(0, 0, 0, 0, 0))
```

Exempel på skapande av oföränderlig nästlad struktur

Om du kan beräkna initialvärde direkt, använd `Vector.fill`:

```
def fill[A](n1: Int, n2: Int)(elem: => A): Vector[Vector[A]]
```

```
scala> Vector.fill(2,5)(scala.util.Random.nextInt(6) + 1)
res0:
  typ???
  värde???
```

Om du kan beräkna initialvärde ur index, använd `Vector.tabulate`:

```
def tabulate[A](n1: Int, n2: Int)(f: (Int, Int) => A): Vector[Vector[A]]
```

```
scala> Vector.tabulate(5,2)((x,y) => x + y + 1)
res1:
  typ???
  värde???
```

Exempel på skapande av oföränderlig nästlad struktur

Om du kan beräkna initialvärde direkt, använd `Vector.fill`:

```
def fill[A](n1: Int, n2: Int)(elem: => A): Vector[Vector[A]]
```

```
scala> Vector.fill(2,5)(scala.util.Random.nextInt(6) + 1)
res0: Vector[Vector[Int]] =
  Vector(Vector(1, 2, 6, 2, 1), Vector(1, 4, 3, 3, 2))
```

Om du kan beräkna initialvärde ur index, använd `Vector.tabulate`:

```
def tabulate[A](n1: Int, n2: Int)(f: (Int, Int) => A): Vector[Vector[A]]
```

```
scala> Vector.tabulate(5,2)((x,y) => x + y + 1)
res1: Vector[Vector[Int]] =
  Vector(Vector(1,2), Vector(2,3), Vector(3,4), Vector(4,5), Vector(5,  6))
```

Uppdatering av en oföränderlig nästlad struktur

Uppdatering av endimensionell struktur med `xs.updated`:

```
def updated[A](index: Int, elem: A): Vector[A]
```

```
scala> var xs = Vector.tabulate(5)(x => x + 1)
xs: typ??? = värde???

scala> xs = xs.updated(1, 42)
xs: typ??? = värde???
```

Uppdatering av nästlad struktur i två dimensioner:

```
scala> var xss = Vector.tabulate(2, 5)((x,y) => x + y + 1)
xss:
  typ??? =
  värde???

scala> xss = xss.updated(0, xss(0).updated(1, 42))
xss:
  typ??? =
  värde???
```

Uppdatering av en oföränderlig nästlad struktur

Uppdatering av endimensionell struktur med `xs.updated`:

```
def updated[A](index: Int, elem: A): Vector[A]
```

```
scala> var xs = Vector.tabulate(5)(x => x + 1)
xs: Vector[Int] = Vector(1, 2, 3, 4, 5)

scala> xs = xs.updated(1, 42)
xs: Vector[Int] = Vector(1, 42, 3, 4, 5)
```

Uppdatering av nästlad struktur i två dimensioner:

```
scala> var xss = Vector.tabulate(2, 5)((x,y) => x + y + 1)
xss: Vector[Vector[Int]] =
  Vector(Vector(1, 2, 3, 4, 5), Vector(2, 3, 4, 5, 6))

scala> xss = xss.updated(0, xss(0).updated(1, 42))
xss:
  Vector[Vector[Int]] =
  Vector(Vector(1, 42, 3, 4, 5), Vector(2, 3, 4, 5, 6))
```

Iterera över nästlad struktur

Behandling av nästlade strukturer kräver ofta algoritmer med nästlad iterering.
Exempel: iterera med nästlad **`for`**-sats för utskrift av denna matris

```
val xss = Vector.tabulate(2,5)((x,y) => x + y + 1)
```

Iterera över nästlad struktur

Behandling av nästlade strukturer kräver ofta algoritmer med nästlad iterering.
Exempel: iterera med nästlad **`for`**-sats för utskrift av denna matris
`val xss = Vector.tabulate(2,5)((x,y) => x + y + 1)`

```
scala> for ??? do
         for ??? do
           print(xss(i)(j))
           print(" ")
         println

1 2 3 4 5
2 3 4 5 6
```

Övning:

Vad ska det stå vid `???` för att alla element ska skrivas ut?

Iterera över nästlad struktur

Behandling av nästlade strukturer kräver ofta algoritmer med nästlad iterering.
Exempel: iterera med nästlad **`for`**-sats för utskrift av denna matris
`val xss = Vector.tabulate(2,5)((x,y) => x + y + 1)`

```
scala> for xs <- xss do
         for x <- xs do
           print(x)
           print(" ")
         end for
         println()
       end for

1 2 3 4 5
2 3 4 5 6
```

Övning: skriv ut matrisen med nästlad `foreach`

Iterera över nästlad struktur

Behandling av nästlade strukturer kräver ofta algoritmer med nästlad iterering.
Exempel: iterera med nästlad **`for`**-sats för utskrift av denna matris
`val xss = Vector.tabulate(2,5)((x,y) => x + y + 1)`

```
scala> for xs <- xss do
         for x <- xs do
           print(x)
           print(" ")
         end for
         println()
       end for

1 2 3 4 5
2 3 4 5 6
```

Övning: skriv ut matrisen med nästlad `foreach`

```
xss.foreach { xs =>
  xs.foreach { x => print(x); print(" ") }
  println()
}
```

Övningsexempel: Yatzy

Skapa en funktion `roll` som ger utfallet av n st tärningskast:

```
scala> import scala.util.Random

scala> def roll(n: Int): Vector[Int] = ???
```

Skapa en funktion `isYatzy` som ger **`true`** om alla utfall är lika:

```
scala> def isYatzy(xs: Vector[Int]): Boolean = ???
```

Du kan anta att xs.length > 0
Tips: använd metoden xs.forall:

```
def forall[A](p: A => Boolean): Boolean
```

Övningsexempel: Yatzy

Skapa en funktion `roll` som ger utfallet av n st tärningskast:

```
1  scala> import scala.util.Random
2
3  scala> def roll(n: Int): Vector[Int] = Vector.fill(n)(Random.nextInt(6) + 1)
```

Skapa en funktion `isYatzy` som ger **`true`** om alla utfall är lika:

```
1  scala> def isYatzy(xs: Vector[Int]): Boolean = xs.forall(x => x == xs(0))
```

Du kan anta att xs.length > 0
Tips: använd metoden xs.forall:

```
def forall[A](p: A => Boolean): Boolean
```

Iterera över nästlad struktur: for-sats

Iterera med nästlad for-sats: (vad har xss för typ?)

```
scala> val xss = Vector.fill(100)(roll(5))

scala> for i <- ??? do
         for j <- ??? do
           print(s"($i)($j): ${xss(i)(j)} ")
         println(s" YATZY: ${isYatzy(xss(i))}")

(0)(0): 3 (0)(1): 6 (0)(2): 4 (0)(3): 4 (0)(4): 6  YATZY: false
(1)(0): 4 (1)(1): 1 (1)(2): 5 (1)(3): 2 (1)(4): 6  YATZY: false
(2)(0): 1 (2)(1): 3 (2)(2): 5 (2)(3): 6 (2)(4): 2  YATZY: false
(3)(0): 2 (3)(1): 1 (3)(2): 1 (3)(3): 5 (3)(4): 4  YATZY: false
(4)(0): 4 (4)(1): 4 (4)(2): 1 (4)(3): 6 (4)(4): 5  YATZY: false
(5)(0): 3 (5)(1): 3 (5)(2): 2 (5)(3): 3 (5)(4): 6  YATZY: false
(6)(0): 3 (6)(1): 6 (6)(2): 1 (6)(3): 1 (6)(4): 4  YATZY: false
(7)(0): 6 (7)(1): 2 (7)(2): 4 (7)(3): 4 (7)(4): 3  YATZY: false
(8)(0): 1 (8)(1): 5 (8)(2): 4 (8)(3): 2 (8)(4): 4  YATZY: false
(9)(0): 1 (9)(1): 1 (9)(2): 3 (9)(3): 6 (9)(4): 6  YATZY: false
(10)(0): 2 (10)(1): 5 (10)(2): 2 (10)(3): 4 (10)(4): 5  YATZY: false
(11)(0): 3 (11)(1): 4 (11)(2): 2 (11)(3): 5 (11)(4): 6  YATZY: false
...
```

Iterera över nästlad struktur: for-sats

Iterera med nästlad for-sats: (xss är en `Vector[Vector[Int]]`)

```
scala> val xss = Vector.fill(100)(roll(5))

scala> for i <- xss.indices do
         for j <- xss(i).indices do
           print(s"($i)($j): ${xss(i)(j)} ")
         println(s" YATZY: ${isYatzy(xss(i))}")

(0)(0): 3 (0)(1): 6 (0)(2): 4 (0)(3): 4 (0)(4): 6  YATZY: false
(1)(0): 4 (1)(1): 1 (1)(2): 5 (1)(3): 2 (1)(4): 6  YATZY: false
(2)(0): 1 (2)(1): 3 (2)(2): 5 (2)(3): 6 (2)(4): 2  YATZY: false
(3)(0): 2 (3)(1): 1 (3)(2): 1 (3)(3): 5 (3)(4): 4  YATZY: false
(4)(0): 4 (4)(1): 4 (4)(2): 1 (4)(3): 6 (4)(4): 5  YATZY: false
(5)(0): 3 (5)(1): 3 (5)(2): 2 (5)(3): 3 (5)(4): 6  YATZY: false
(6)(0): 3 (6)(1): 6 (6)(2): 1 (6)(3): 1 (6)(4): 4  YATZY: false
(7)(0): 6 (7)(1): 2 (7)(2): 4 (7)(3): 4 (7)(4): 3  YATZY: false
(8)(0): 1 (8)(1): 5 (8)(2): 4 (8)(3): 2 (8)(4): 4  YATZY: false
(9)(0): 1 (9)(1): 1 (9)(2): 3 (9)(3): 6 (9)(4): 6  YATZY: false
(10)(0): 2 (10)(1): 5 (10)(2): 2 (10)(3): 4 (10)(4): 5  YATZY: false
(11)(0): 3 (11)(1): 4 (11)(2): 2 (11)(3): 5 (11)(4): 6  YATZY: false
...
```

Nästlade for-uttryck

Iterera med **nästlad for-yield**:

```
1  scala> val xss = for i <- 1 to 2 yield
2                   for j <- 1 to 5 yield i + j + 1
3
4  val xss: IndexedSeq[IndexedSeq[Int]] =
5        ???
```

Nästlade for-uttryck

Iterera med **nästlad for-yield**:

```
scala> val xss = for i <- 1 to 2 yield
                   for j <- 1 to 5 yield i + j + 1

val xss: IndexedSeq[IndexedSeq[Int]] =
      ???
```

Om man skriver så här får man en endimensionell struktur:

```
scala> val xs = for { i <- 1 to 2; j <- 1 to 5 } yield i + j + 1
val xs: IndexedSeq[Int] =
    ???
```

Nästlade for-uttryck

Iterera med **nästlad for-yield**:

```
scala> val xss = for i <- 1 to 2 yield
                   for j <- 1 to 5 yield i + j + 1

val xss: IndexedSeq[IndexedSeq[Int]] =
    Vector(Vector(3, 4, 5, 6, 7), Vector(4, 5, 6, 7, 8))
```

Nästlade for-uttryck

Iterera med **nästlad for-yield**:

```
scala> val xss = for i <- 1 to 2 yield
                   for j <- 1 to 5 yield i + j + 1

val xss: IndexedSeq[IndexedSeq[Int]] =
    Vector(Vector(3, 4, 5, 6, 7), Vector(4, 5, 6, 7, 8))
```

Om man skriver så här får man en endimensionell struktur:

```
scala> val xs = for { i <- 1 to 2; j <- 1 to 5 } yield i + j + 1
val xs: IndexedSeq[Int] =
    Vector(3, 4, 5, 6, 7, 4, 5, 6, 7, 8)
```

Nästlade map-uttryck

Iterera med **nästlade map-uttryck**:

```
1 scala> val xss = (1 to 2).map(i => (1 to 5).map(j => i + j + 1))
2 xss: IndexedSeq[IndexedSeq[Int]] =
3       ???
```

Nästlade map-uttryck

Iterera med **nästlade map-uttryck**:

```
1 scala> val xss = (1 to 2).map(i => (1 to 5).map(j => i + j + 1))
2 xss: IndexedSeq[IndexedSeq[Int]] =
3       Vector(Vector(3, 4, 5, 6, 7), Vector(4, 5, 6, 7, 8))
```

Fallgrop: likhet av array

```
scala> Vector.fill(5, 2)(42) == Vector.fill(5, 2)(42)
val res0: ???

scala> Array.fill(5, 2)(42) == Array.fill(5, 2)(42)
val res1: ???
```

Fallgrop: likhet av array

```
scala> Vector.fill(5, 2)(42) == Vector.fill(5, 2)(42)
val res0: Boolean = true

scala> Array.fill(5, 2)(42) == Array.fill(5, 2)(42)
val res1: Boolean = false  // AAAARRGH!!! :(
```

Primitiva arrayer har en equals-metod som ger referenslikhet, **inte** innehållslikhet. Och det fungerar följaktligen ej heller på nästlade strukturer.

Kolla likhet mellan två heltalsmatriser (uppfinner hjulet)

```
def isEqual(xss: Array[Array[Int]], yss: Array[Array[Int]]) =
  if xss.length != yss.length then false else
    var i = 0
    var foundUnequal = false
    while ??? do
      if xss(i).length != yss(i).length then
        ???
      else
        var j = 0
        while ??? do
          if xss(i)(j) != yss(i)(j) then ???
          j += 1
        end while
      end if
      i += 1
    end while
    !foundUnequal
  end if
end isEqual
```

Kolla likhet mellan två heltalsmatriser (uppfinner hjulet)

```
def isEqual(xss: Array[Array[Int]], yss: Array[Array[Int]]) =
  if xss.length != yss.length then false else
    var i = 0
    var foundUnequal = false
    while i < xss.length && !foundUnequal do
      if xss(i).length != yss(i).length then
        foundUnequal = true
      else
        var j = 0
        while j < xss(i).length && !foundUnequal do
          if xss(i)(j) != yss(i)(j) then foundUnequal = true
          j += 1
        end while
      end if
      i += 1
    end while
    !foundUnequal
  end if
end isEqual
```

Använd INTE `sameElements` på nästlade arrayer

I Scala kan du använda metoden `sameElements` för att kolla innehållslikhet mellan två arrayer, men det funkar **INTE** på djupet i nästlade strukturer.

```
1  scala> val xs = Array(1,2,3)
2  xs: Array[Int] = Array(1, 2, 3)
3
4  scala> val ys = Array(1,2,3)
5  ys: Array[Int] = Array(1, 2, 3)
6
7  scala> xs.sameElements(ys)                          // xs, ys ej nästlade
8  res0: Boolean = true                                // innehåll lika!
9
10 scala> Array(Array(1)) sameElements Array(Array(1))
11 res1: Boolean = false                               //AAAARGH!
```

Använd i stället:
`java.util.Objects.deepEquals` eller `java.util.Arrays.deepEquals`

Den senare kräver typkonvertering av argumenten med:

`asInstanceOf[Array[Object]]`

Kontroll av innehållslikhet mellan nästlade arrayer

`java.util.Objects.deepEquals` fungerar **på djupet** för godtyckliga referenstyper:

```
scala> java.util.Objects.deepEquals(
          Array(Array("a", Array("b"), 42)),
          Array(Array("a", Array("b"), 42)))
val res0: Boolean = true

scala> java.util.Objects.deepEquals(
          Array(Array("a", Array("b"), 42)),
          Array(Array("a", Array("b"), 43)))
val res1: Boolean = false
```

`java.util.Objects.deepEquals` kontrollerar om argumenten är arrayer och anropar då i sin tur `java.util.Arrays.deepEquals` efter typkonvertering:

```
scala> java.util.Arrays.deepEquals(
          Array(Array("a", Array("b"), 42)).asInstanceOf[Array[Object]],
          Array(Array("a", Array("b"), 42)).asInstanceOf[Array[Object]])
val res3: Boolean = true
```

`https://stackoverflow.com/questions/63686721/best-replacement-of-deep-method-in-scala-2-13`

Om veckans övningar

- Träna på att iterera över nästlade strukurer
- Fortsätt jobba med Yatzy-exemplet
- träna på att skapa **imperativa** algoritmer:
  lös `isYatzy` med **`while`**-sats
- Extrauppgift där du ska bygga ett enkelt yatzy-spel i terminalen (kunde varit en tentauppgift...)

Typparametrar

Exempel: Icke-generisk case-klass med heltalsmatris

En *icke-generisk* datastruktur har inga obundna typparametrar; alla typer är **konkreta** (alltså specifika).

En icke-generisk case-class med en `Vector[Vector[Int]]`:

```
case class Matrix(data: Vector[Vector[Int]]):
  def apply(x: Int, y: Int): Int = data(x)(y)
```

```
1  scala> Matrix(Vector(Vector(5, 2, 42, 4, 5),Vector(3, 4, 18, 6, 7)))
2  res0: Matrix =
3    Matrix(Vector(Vector(5, 2, 42, 4, 5), Vector(3, 4, 18, 6, 7)))
4
```

Exempel: Generisk case-klass med generell matris

En *generisk* datastruktur har minst en **obunden typparameter** som vid användning ska bindas till ett **konkret typargument**.

```
case class Matrix[T](data: Vector[Vector[T]]):
  def apply(x: Int, y: Int): T = data(x)(y)
```

`Matrix` i exemplet ovan är en **generisk** case-class där `T` är obunden, eftersom `T` är en typparameter deklarerad inom `[]` **efter** klassens namn men **före** klassparameterlistan.

Användning där `T` binds till `Int` via kompilatorns typhärledning:

```
1  scala> Matrix(Vector(Vector(5, 2, 42, 4, 5),Vector(3, 4, 18, 6, 7)))
2  res1: Matrix[Int] =
3    Matrix(Vector(Vector(5, 2, 42, 4, 5), Vector(3, 4, 18, 6, 7)))
4
```

Vad är en typparameter?

- En **typparameter** gör det möjligt att ge ett **typargument**.
- Detta kallas **parametrisk polymorfism** (eng. *paramteric polymorphism*).
- Exempel: **generisk funktion**:

```
def tnirp[A](x: A):Unit = println(x.toString.reverse)
```

Vad är en typparameter?

- En **typparameter** gör det möjligt att ge ett **typargument**.
- Detta kallas **parametrisk polymorfism** (eng. *paramteric polymorphism*).
- Exempel: **generisk funktion**:

```
def tnirp[A](x: A):Unit = println(x.toString.reverse)
```

- En **fri** typparameter kan bindas till vilken typ som helst.
- Bindingen av typargument till typparametrar sker vid **kompileringstid**.
- En typparameter är **fri** om den **inte** fått något värde, annars **bunden**.

Vad är en typparameter?

- En **typparameter** gör det möjligt att ge ett **typargument**.
- Detta kallas **parametrisk polymorfism** (eng. *paramteric polymorphism*).
- Exempel: **generisk funktion**:

```
def tnirp[A](x: A):Unit = println(x.toString.reverse)
```

- En **fri** typparameter kan bindas till vilken typ som helst.
- Bindingen av typargument till typparametrar sker vid **kompileringstid**.
- En typparameter är **fri** om den **inte** fått något värde, annars **bunden**.
- Exempel: **generisk klass** med **generiska metoder**:

```
class Cell[A](   // A är här fri men måste bindas vid användning
    var value: A):              // A är bunden vid användning av Cell
  def update(a: A): Unit = value = a    // A är även här bunden vid anv.
  def replaced[B](b: B): Cell[B] = new Cell(b)  // första [B] är fri
```

Vad är en typparameter?

- En **typparameter** gör det möjligt att ge ett **typargument**.
- Detta kallas **parametrisk polymorfism** (eng. *paramteric polymorphism*).
- Exempel: **generisk funktion**:

```
def tnirp[A](x: A):Unit = println(x.toString.reverse)
```

- En **fri** typparameter kan bindas till vilken typ som helst.
- Bindingen av typargument till typparametrar sker vid **kompileringstid**.
- En typparameter är **fri** om den **inte** fått något värde, annars **bunden**.
- Exempel: **generisk klass** med **generiska metoder**:

```
class Cell[A](   // A är här fri men måste bindas vid användning
    var value: A):                // A är bunden vid användning av Cell
  def update(a: A): Unit = value = a    // A är även här bunden vid anv.
  def replaced[B](b: B): Cell[B] = new Cell(b)  // första [B] är fri
```

- **Skuggning kan förekomma**: Om `replaced` i `Cell` hade använt namnet A på sin typparameter hade den **skuggat** klassens typparameter och tolkats som en fri typparameter, alltså en godtycklig typ och **inte** klassens typparameter. (jämför namnöverskuggning vid **lokala** namn i nästlade block)

Exempel: Generisk funktion

Vad händer här?

```
1
2  scala> def skrikBaklänges(x: T): String = x.toString.toUpperCase.reverse
3  ???
4
5
6
7  scala> def skrikBaklänges[T](x: T): String = x.toString.toUpperCase.reverse
8
9  scala> skrikBaklänges("gurka är gott")
10 val res0: ???
```

Exempel: Generisk funktion

Vad händer här?

```
1
2  scala> def skrikBaklänges(x: T): String = x.toString.toUpperCase.reverse
3  1 |def skrikBaklänges(x: T): String = x.toString.toUpperCase.reverse
4    |                      ^
5    |                      Not found: type T
6                               ^
7
8  scala> def skrikBaklänges[T](x: T): String = x.toString.toUpperCase.reverse
9
10 scala> skrikBaklänges("gurka är gott")
11 val res0: ???
```

Exempel: Generisk funktion

Vad händer här?

```
1
2  scala> def skrikBaklänges(x: T): String = x.toString.toUpperCase.reverse
3  1 |def skrikBaklänges(x: T): String = x.toString.toUpperCase.reverse
4    |                      ^
5    |                      Not found: type T
6    |                          ^
7
8  scala> def skrikBaklänges[T](x: T): String = x.toString.toUpperCase.reverse
9
10 scala> skrikBaklänges("gurka är gott")
11 val res0: String = TTOG RÄ AKRUG
```

Om ingen typparameter deklareras inom hakparenteser efter funktionens namns så vet inte kompilatorn vad T är för en typ. Men med en typparameter `[T]` efter funktionsnamnet tolkar kompilatorn funktionen som **generisk** och typen T bestäms av argumentets typ **vid anrop** och T kan bindas till godtycklig typ.

Exempel: Generisk case-klass

- En generisk klass har en eller flera typparametrar efter klassnamnet:

```
case class Box[A](value: A)
```

- Kompilatorn härleder typparameterarnas typ utifrån givna värden.

```
scala> Box("gurka")
val res1: Box[String] = Box(gurka)
```

- Du kan också ge typpparametern en typ explicit:

```
scala> Box[Int](42)
val res2: Box[Int] = Box(42)
```

- Om typen inte stämmer får du hjälp av kompilatorn att hitta felet:

```
scala> Box[String](42)
-- Error:
1 |Box[String](42)
  |            ^^ Found:    (42 : Int)  Required: String
```

- En generisk klass, här `Box`, kallas också **typkonstruktor** (eng. *type constructor*) då den ”färdiga” typen `Box[Int]` ”konstrueras” på platsen där den används.

Fallgrop: Typradering (eng. *type erasure*)

Informationen om typerna i typparametrar raderas innan kodgenerering för JVM av prestandaskäl och **typparametrar saknas vid runtime** i bytekoden.

```
scala> def isIntVector[T](xs: Vector[T]) = xs.isInstanceOf[Vector[Int]]
-- Warning:
1 |def isIntVector[T](xs: Vector[T]) = xs.isInstanceOf[Vector[Int]]
  |                                     ^^^^^^^^^^^^^^^^^^^^^^^^^^
  |                the type test for Vector[Int] cannot be checked at runtime
def isIntVector[T](xs: Vector[T]): Boolean

scala> isIntVector(Vector("hej"))
res42: Boolean = true  // AAAARGHH!! :(
```

Måste ”packa upp” samlingen och typtesta alla element:

```
scala> def isIntVector[T](xs: Vector[T]) = xs.forall(_.isInstanceOf[Int])

scala> isIntVector(Vector("hej"))
res43: Boolean = false  // FUNKAR :)
```

Typkontroll vid körtid görs oftast hellre med **`match`**.

Upptäcka och åtgärda buggar

Testning och avlusning

- Läs om testning och avlusning (eng. *debugging*) i Appendix D: ”Fixa buggar”
- Träna på println-debugging
- Prova debuggern i VS code