

# EDAA45 Programmering, grundkurs

## Läsvecka 5: Sekvensalgoritmer

Björn Regnell

Datavetenskap, LTH

Lp1-2, HT 2016

## 5 Sekvensalgoritmer

- Vad är en sekvensalgoritm?
- SEQ-COPY
- For-satser och arrayer i Java
- Exempel: PolygonWindow
- SEQ-INSERT/REMOVE-COPY
- Variabelt antal argument, "varargs"
- SEQ-APPEND/INSERT/COPY i förändringsbar polygon
- SEQ-APPEND/INSERT/COPY med oföränderlig Polygon
- Förändringsbar eller oföränderlig?
- Registrering
- Att välja sekvenssamling

# Vad är en sekvensalgoritm?

# Vad är en sekvensalgoritm?

- En algoritm är en stegvis beskrivning av hur man löser ett problem.
- En sekvensalgoritm är en algoritm där dataelement i sekvens utgör en viktig del av problembeskrivningen och/eller lösningen.
- Exempel: sortera en sekvens av personer efter deras ålder.
- Två olika principer:
  - Skapa **ny sekvens** utan att förändra indatasekvensen
  - Ändra **på plats** (eng. *in place*) i den **förändringsbara** indatasekvensen

# Skapa ny sekvenssamling eller ändra på plats?

- Ofta är det **lättast att skapa ny samling** och kopiera över elementen medan man loopar.
- Om man har mycket stora samlingar kan man behöva ändra på plats för att spara tid/minne.
- Det är bra att själv kunna implementera sekvensalgoritmer även om många av dem finns färdiga, för att bättre förstå vad som händer "under huven", och för att i enstaka fall kunna optimera om det verkligen behövs.
- Vi illustrerar därför hur man kan implementera några sekvensalgoritmer med primitiva arrayer även om man sällan gör så i praktiken (i Scala).

# SEQ-COPY

# Algoritm: SEQ-COPY

**Pseudokod** för algoritmen SEQ-COPY som kopierar en sekvens, här en Array med heltal:

---

**Indata** : Heltalsarray  $xs$

**Resultat**: En ny heltalsarray som är en kopia av  $xs$ .

$result \leftarrow$  en ny array med plats för  $xs.length$  element

$i \leftarrow 0$

**while**  $i < xs.length$  **do**

$result(i) \leftarrow xs(i)$

$i \leftarrow i + 1$

**end**

**return**  $result$

---

# Implementation av SEQ-COPY med while

```
1  object seqCopy {
2
3      def arrayCopy(xs: Array[Int]): Array[Int] = {
4          val result = new Array[Int](xs.length)
5          var i = 0
6          while (i < xs.length) {
7              result(i) = xs(i)
8              i += 1
9          }
10         result
11     }
12
13     def test: String = {
14         val xs = Array(1,2,3,4,42)
15         val ys = arrayCopy(xs)
16         if (xs sameElements ys) "OK!" else "ERROR!"
17     }
18
19     def main(args: Array[String]): Unit = println(test)
20 }
```



# Implementation av SEQ-COPY med for

```
1  object seqCopyFor {  
2  
3    def arrayCopy(xs: Array[Int]): Array[Int] = {  
4      val result = new Array[Int](xs.length)  
5      for (i <- xs.indices) {  
6        result(i) = xs(i)  
7      }  
8      result  
9    }  
10  
11    def test: String = {  
12      val xs = Array(1,2,3,4,42)  
13      val ys = arrayCopy(xs)  
14      if (xs sameElements ys) "OK!" else "ERROR!"  
15    }  
16  
17    def main(args: Array[String]): Unit = println(test)  
18  }
```

# Implementation av SEQ-COPY med for-yield

```
1  object seqCopyForYield {  
2  
3      def arrayCopy(xs: Array[Int]): Array[Int] = {  
4          val result = for (i <- xs.indices) yield xs(i)  
5          result.toArray  
6      }  
7  
8      def test: String = {  
9          val xs = Array(1,2,3,4,42)  
10         val ys = arrayCopy(xs)  
11         if (xs sameElements ys) "OK!" else "ERROR!"  
12     }  
13  
14     def main(args: Array[String]): Unit = println(test)  
15 }
```

# For-satser och arrayer i Java

# For-satser och arrayer i Java

En for-sats i Java har följande struktur:

```
for (initialisering; slutvillkor; inkrementering) {  
    sats1;  
    sats2;  
    ...  
}
```

En primitiv heltals-array deklareras så här i Java:

```
int[] xs = new int[42]; // rymler 42 st heltal, init 0:or  
int[] ys = {10, 42, -1}; // initiera med 3 st heltal
```

Exempel på for-sats: fyll en array med 1:or

```
for (int i = 0; i < xs.length; i = i + 1){ // vanligare: i++  
    xs[i] = 1;                             // indexera med [i]  
}
```

# Implementation av SEQ-COPY i Java med for-sats

```

1  public class SeqCopyForJava {
2
3      public static int[] arrayCopy(int[] xs){
4          int[] result = new int[xs.length];
5          for (int i = 0; i < xs.length; i++){
6              result[i] = xs[i];
7          }
8          return result;
9      }
10
11     public static String test(){
12         int[] xs = {1, 2, 3, 4, 42};
13         int[] ys = arrayCopy(xs);
14         for (int i = 0; i < xs.length; i++){
15             if (xs[i] != ys[i]) {
16                 return "FAILED!";
17             }
18         }
19         return "OK!";
20     }
21
22     public static void main(String[] args) {
23         System.out.println(test());
24     }
25 }

```

Lite syntax och semantik för Java:

- En Java-klass med enbart statiska medlemmar motsvarar ett singelobjekt i Scala.
- Typen kommer **före** namnet.
- Man **måste** skriva **return**.
- Man **måste** ha semikolon efter varje sats.
- Metodnamn **måste** följas av parenteser; om inga parametrar finns används ( )
- En array i Java är inget vanligt objekt, men har ett "attribut" length som ger antal element.
- **Övning:** skriv om med **while**-sats i stället; har samma syntax i Scala & Java.

# Exempel: PolygonWindow

## Exempel: PolygonWindow

- En polygon kan representeras som en sekvens av punkter, där varje punkt är en 2-tupel: `Seq[(Int, Int)]`
- `PolygonWindow` nedan är ett fönster som kan rita en polygon.

```
1 class PolygonWindow(width: Int, height: Int) {  
2   val w = new cslib.window.SimpleWindow(width, height, "PolyWin")  
3  
4   def draw(pts: Seq[(Int, Int)]): Unit = if (pts.size > 0) {  
5     w.moveTo(pts(0)._1, pts(0)._2)  
6     for (i <- 1 until pts.length) w.lineTo(pts(i)._1, pts(i)._2)  
7     w.lineTo(pts(0)._1, pts(0)._2)  
8   }  
9 }
```

# Exempel: PolygonWindow

- En polygon kan representeras som en sekvens av punkter, där varje punkt är en 2-tupel: `Seq[(Int, Int)]`
- `PolygonWindow` nedan är ett fönster som kan rita en polygon.

```
1 class PolygonWindow(width: Int, height: Int) {  
2   val w = new cslib.window.SimpleWindow(width, height, "PolyWin")  
3  
4   def draw(pts: Seq[(Int, Int)]): Unit = if (pts.size > 0) {  
5     w.moveTo(pts(0)._1, pts(0)._2)  
6     for (i <- 1 until pts.length) w.lineTo(pts(i)._1, pts(i)._2)  
7     w.lineTo(pts(0)._1, pts(0)._2)  
8   }  
9 }
```

```
1 object polygonTest1 {  
2   def main(args: Array[String]): Unit = {  
3     val pw = new PolygonWindow(200,200)  
4     val pts = Array((50,50), (100,100), (50,100), (30,50))  
5     pw.draw(pts)  
6   }  
7 }
```



# Typ-alias för att abstrahera typnamn

Med hjälp av nyckelordet **type** kan man deklarerar ett **typ-alias** för att ge ett **alternativt** namn till en viss typ. Exempel:

```
1 scala> type Pt = (Int, Int)
2
3 scala> def distToOrigo(pt: Pt): Int = math.hypot(pt._1, pt._2)
4
5 scala> type Pts = Vector[Pt]
6
7 scala> def firstPt(pts: Pts): Pt = pts.head
8
9 scala> val xs: Pts = Vector((1,1),(2,2),(3,3))
10
11 scala> firstPt(xs)
12 res0: Pt = (1,1)
```

Detta är bra om:

- man har en lång och krånglig typ och vill använda ett kortare namn,
- om man vill abstrahera en typ och öppna för möjligheten att byta implementation senare (t.ex. till en egen klass), medan man ändå kan fortsätta att använda befintligt namn.

# SEQ-INSERT/REMOVE-COPY

# Exempel: SEQ-INSERT/REMOVE-COPY

Nu ska vi "uppfinna hjulet" och som träning implementera **insättning** och **borttagning** till en **ny** sekvens utan användning av sekvenssamlingsmetoder (förutom `length` och `apply`):

```
object pointSeqUtils {  
  type Pt = (Int, Int) // a type alias to make the code more concise  
  
  def primitiveInsertCopy(pts: Array[Pt], pos: Int, pt: Pt): Array[Pt] = ???  
  
  def primitiveRemoveCopy(pts: Array[Pt], pos: Int): Array[Pt] = ???  
}
```

# Pseudo-kod för SEQ-INSERT-COPY

**Indata** : *pts*: Array[Pt],  
          *pt*: Pt,  
          *pos*: Int

**Resultat**: En ny sekvens av typen Array[Pt] som är en kopia av *pts* men där *pt* är infogat på plats *pos*

---

```
result ← en ny Array[Pt] med plats för pts.length + 1 element
for i ← 0 to pos - 1 do
  | result(i) ← pts(i)
end
result(pos) ← pt
for i ← pos + 1 to xs.length do
  | result(i) ← xs(i - 1)
end
return result
```

---

# Pseudo-kod för SEQ-INSERT-COPY

**Indata** : *pts*: Array[Pt],  
          *pt*: Pt,  
          *pos*: Int

**Resultat**: En ny sekvens av typen Array[Pt] som är en kopia av *pts* men där *pt* är infogat på plats *pos*

---

```
result ← en ny Array[Pt] med plats för pts.length + 1 element
for i ← 0 to pos - 1 do
  | result(i) ← pts(i)
end
result(pos) ← pt
for i ← pos + 1 to xs.length do
  | result(i) ← xs(i - 1)
end
return result
```

---

**Övning**: Skriv pseudo-kod för SEQ-REMOVE-COPY

# Insättning/borttagning i kopia av primitiv Array

```
1  object pointSeqUtils {
2      type Pt = (Int, Int) // a type alias to make the code more concise
3
4      def primitiveInsertCopy(pts: Array[Pt], pos: Int, pt: Pt): Array[Pt] = {
5          val result = new Array[Pt](pts.length + 1) // initialized with null
6          for (i <- 0 until pos) result(i) = pts(i)
7          result(pos) = pt
8          for (i <- pos + 1 to pts.length) result(i) = pts(i - 1)
9          result
10     }
11
12     def primitiveRemoveCopy(pts: Array[Pt], pos: Int): Array[Pt] =
13         if (pts.length > 0) {
14             val result = new Array[Pt](pts.length - 1) // initialized with null
15             for (i <- 0 until pos) result(i) = pts(i)
16             for (i <- pos + 1 until pts.length) result(i - 1) = pts(i)
17             result
18         } else Array.empty
19
20     // above methods implemented using the powerful Scala collection method patch:
21
22     def insertCopy(pts: Array[Pt], pos: Int, pt: Pt) = pts.patch(pos, Array(pt), 0)
23
24     def removeCopy(pts: Array[Pt], pos: Int) = pts.patch(pos, Array.empty[Pt], 1)
25 }
```

# Insättning/borttagning i kopia av primitiv Array

```
1  object pointSeqUtils {
2      type Pt = (Int, Int) // a type alias to make the code more concise
3
4      def primitiveInsertCopy(pts: Array[Pt], pos: Int, pt: Pt): Array[Pt] = {
5          val result = new Array[Pt](pts.length + 1) // initialized with null
6          for (i <- 0 until pos) result(i) = pts(i)
7          result(pos) = pt
8          for (i <- pos + 1 to pts.length) result(i) = pts(i - 1)
9          result
10     }
11
12     def primitiveRemoveCopy(pts: Array[Pt], pos: Int): Array[Pt] =
13         if (pts.length > 0) {
14             val result = new Array[Pt](pts.length - 1) // initialized with null
15             for (i <- 0 until pos) result(i) = pts(i)
16             for (i <- pos + 1 until pts.length) result(i - 1) = pts(i)
17             result
18         } else Array.empty
19
20     // above methods implemented using the powerful Scala collection method patch:
21
22     def insertCopy(pts: Array[Pt], pos: Int, pt: Pt) = pts.patch(pos, Array(pt), 0)
23
24     def removeCopy(pts: Array[Pt], pos: Int) = pts.patch(pos, Array.empty[Pt], 1)
25 }
```

Man gör **mycket lätt fel** på gränser/specialfall: +-1, to/until, tom sekvens etc.

# Exempel: Test av SEQ-INSERT/REMOVE-COPY

```
1  object polygonTest2 {  
2      def main(args: Array[String]): Unit = {  
3          val pw = new PolygonWindow(200,200)  
4          val pts = Array((50,50), (100,100), (50,100), (30,50))  
5          pw.draw(pts)  
6  
7          val morePts = pointSeqUtils.primitiveInsertCopy(pts, 2, (90,130))  
8          //val morePts = pointSeqUtils.insertCopy(pts, 2, (90,130))  
9          pw.draw(morePts)  
10  
11         val lessPts = pointSeqUtils.primitiveRemoveCopy(morePts, morePts.length - 1)  
12         //val lessPts = pointSeqUtils.removeCopy(morePts, morePts.length - 1)  
13         pw.draw(lessPts)  
14     }  
15 }  
16 }
```



## Exempel: Göra insättning med take/drop

Om du inte vill "uppfinna hjulet" och inte använda patch kan du göra så här:

Använd take och drop tillsammans med :+ och ++

Du kan också göra insättningen generiskt användbar för alla sekvenser:

```
scala> val xs = Vector(1,2,3)
xs: scala.collection.immutable.Vector[Int] =
  Vector(1, 2, 3)

scala> val ys = (xs.take(2) :+ 42) ++ xs.drop(2)
ys: scala.collection.immutable.Vector[Int] =
  Vector(1, 2, 42, 3)

scala> def insertCopy[T](xs: Seq[T], elem: T, pos: Int) =
  (xs.take(pos) :+ elem) ++ xs.drop(pos)

scala> insertCopy(xs, 42, 2)
res0: Seq[Int] = Vector(1, 2, 42, 3)
```

**Övning:** Implementera insertCopy[T] med patch istället.

# Variabelt antal argument, "varargs"

## Parameter med variabelt antal argument, "varargs"

Med en asterisk efter parametertypen kan antalet argument variera:

```
def sumSizes(xs: String*): Int = xs.map(_.size).sum
```

```
scala> sumSizes("Zaphod")
```

```
res0: Int = 6
```

```
scala> sumSizes("Zaphod","Beeblebrox")
```

```
res1: Int = 16
```

```
scala> sumSizes("Zaphod","Beeblebrox","Ford","Prefect")
```

```
res3: Int = 27
```

```
scala> sumSizes()
```

```
res4: Int = 0
```

Typen på `xs` blir en `Seq[String]`, egentligen en `WrappedArray[String]` som kapslar in en array så den beter sig mer som en "vanlig" Scala-samling.

# Sekvenssamling som argument till varargs-parameter

```
def sumSizes(xs: String*): Int = xs.map(_.size).sum  
  
val veg = Vector("gurka", "tomat")
```

Om du *redan har* en sekvenssamling så kan du applicera den på en parameter som accepterar variabelt antal argument med typannoteringen

: \_\*

direkt **efter** sekvenssamlingen.

```
scala> sumSizes(veg: _*)  
res5: Int = 10
```

# SEQ-APPEND/INSERT/COPY i förändringsbar polygon

# Implementera Polygon

- En polygon kan representeras som en sekvens av punkter.
- Vi vill kunna lägga till punkter, samt ta bort punkter.
- En polygon kan implementeras på många olika sätt:

# Implementera Polygon

- En polygon kan representeras som en sekvens av punkter.
- Vi vill kunna lägga till punkter, samt ta bort punkter.
- En polygon kan implementeras på många olika sätt:
  - **Förändringsbar** (eng. *mutable*)
    - Med punkterna i en **Array**
    - Med punkterna i en **ArrayBuffer**
    - Med punkterna i en **ListBuffer**
    - Med punkterna i en **Vector**
    - Med punkterna i en **List**
  - **Oföränderlig** (eng. *immutable*)
    - Som en case-klass med en oföränderlig **Vector** som returnerar nytt objekt vid uppdatering. Vi kan låta datastrukturen vara **publik** eftersom allt är oföränderligt.
    - Som en "vanlig" klass med någon lämplig **privat** datastruktur där vi **inte** möjliggör förändring av efter initialisering och där vi returnerar nytt objekt vid uppdatering.

# Implementera Polygon

- En polygon kan representeras som en sekvens av punkter.
- Vi vill kunna lägga till punkter, samt ta bort punkter.
- En polygon kan implementeras på många olika sätt:
  - **Förändringsbar** (eng. *mutable*)
    - Med punkterna i en **Array**
    - Med punkterna i en **ArrayBuffer**
    - Med punkterna i en **ListBuffer**
    - Med punkterna i en **Vector**
    - Med punkterna i en **List**
  - **Oföränderlig** (eng. *immutable*)
    - Som en case-klass med en oföränderlig **Vector** som returnerar nytt objekt vid uppdatering. Vi kan låta datastrukturen vara **publik** eftersom allt är oföränderligt.
    - Som en "vanlig" klass med någon lämplig **privat** datastruktur där vi **inte** möjliggör förändring av efter initialisering och där vi returnerar nytt objekt vid uppdatering.

Val av implementation **beror på** sammanhang & användning!



# Exempel: PolygonArray, ändring på plats

```
1  class PolygonArray(val maxSize: Int) {
2      type Pt = (Int, Int)
3      private val points = new Array[Pt](maxSize) // initialized with null
4      private var n = 0
5      def size = n
6
7      def draw(w: PolygonWindow): Unit = w.draw(points.take(n))
8
9      def append(pts: Pt*): Unit = {
10         for (i <- pts.indices) points(n + i) = pts(i)
11         n += pts.length
12     }
13
14     def insert(pos: Int, pt: Pt): Unit = { // exercise: change pt to varargs pts
15         for (i <- n until pos by -1) points(i) = points(i - 1)
16         points(pos) = pt
17         n += 1
18     }
19
20     def remove(pos: Int): Unit = { // exercise: change pos to fromPos, replaced
21         for (i <- pos until n) points(i) = points(i + 1)
22         n -= 1
23     }
24
25     override def toString = points.mkString("PrimitivePolygon(", ",", ",")
26 }
```

# Test av PolygonArray, ändring på plats

```
1  object polygonTest3 {  
2      def main(args: Array[String]): Unit = {  
3          val pw = new PolygonWindow(200,200)  
4          val poly = new PolygonArray(100)  
5  
6          poly.append((50,50), (100,100), (50,100), (30,50))  
7          println(poly)  
8          poly.draw(pw)  
9  
10         poly.insert(2, (100,150))  
11         println(poly)  
12         poly.draw(pw)  
13  
14         poly.remove(0)  
15         println(poly)  
16         poly.draw(pw)  
17     }  
18 }
```

# Exempel: PolygonVector, variabel referens till oföränderlig datastruktur

```
1  class PolygonVector {
2      type Pt = (Int, Int)
3      private var points = Vector.empty[Pt] // note var declaration to allow mutation
4      def size = points.size
5
6      def draw(w: PolygonWindow): Unit = w.draw(points.take(size))
7
8      def append(pts: Pt*): Unit = {
9          points += pts.toVector
10     }
11
12     def insert(pos: Int, pt: Pt): Unit = { // exercise: change pt to varargs pts
13         points = points.patch(pos, Vector(pt), 0)
14     }
15
16     def remove(pos: Int): Unit = { // exercise: change pos to fromPos, replaced
17         points = points.patch(pos, Vector(), 1)
18     }
19
20     override def toString = points.mkString("PrimitivePolygon(", ",", ")")
21 }
```

# Test av PolygonVector, variabel referens till oföränderlig datastruktur

```
1  object polygonTest4 {  
2    def main(args: Array[String]): Unit = {  
3      val pw = new PolygonWindow(200,200)  
4      val poly = new PolygonVector  
5  
6      poly.append((50,50), (100,100), (50,100), (30,50))  
7      println(poly)  
8      poly.draw(pw)  
9  
10     poly.insert(2, (100,150))  
11     println(poly)  
12     poly.draw(pw)  
13  
14     poly.remove(0)  
15     println(poly)  
16     poly.draw(pw)  
17   }  
18 }
```

# SEQ-APPEND/INSERT/COPY med oföränderlig Polygon

# Exempel: Polygon som oföränderlig case class

```

1  object Polygon {
2      type Pt = (Int, Int)
3      type Pts = Vector[Pt]
4      def apply() = new Polygon(Vector())
5  }
6
7  import Polygon.{Pt, Pts}
8
9  case class Polygon(points: Pts) {
10     def size = points.size // for convenience but not strictly necessary (why?)
11
12     def append(pts: Pt*) = copy(points ++ pts.toVector)
13
14     def insert(pos: Int, pts: Pt*) = copy(points.patch(pos, pts, 0))
15
16     def remove(pos: Int, replaced: Int = 1) = copy(points.patch(pos, Seq(), replaced))
17 }

```

- Nu är attributet points en publik **val** som vi kan dela med oss av eftersom datastrukturen Vector är oföränderlig.
- Vi behöver inte införa ett beroende till PolygonWindow här då vi ger tillgång till sekvensen av punkter som kan användas vid anrop av PolygonWindow.draw
- Att ändra implementationen till något annat än Vector blir lätt om klientkoden använder typ-alias Polygon.Pts i stället för Vector[(Int, Int)].

# Test av Polygon som oföränderlig case class

```
1  object polygonTest5 {  
2    def main(args: Array[String]): Unit = {  
3      val pw = new PolygonWindow(200,200)  
4      var poly = Polygon()  
5  
6      poly = poly.append((50,50), (100,100), (50,100), (30,50))  
7      println(poly)  
8      pw.draw(poly.points)  
9  
10     poly = poly.insert(2, (100,150))  
11     println(poly)  
12     pw.draw(poly.points)  
13  
14     poly = poly.remove(0)  
15     println(poly)  
16     pw.draw(poly.points)  
17   }  
18 }
```

# Förändringsbar eller oföränderlig?



# Förändringsbar eller oföränderlig?

- Om den underligande **oföränderliga** datastrukturen är **smart** implementerad så att den **återanvänder redan allokerade objekt** – vilket ju är ofarligt eftersom de aldrig kommer att ändras – så är oföränderlighet **minst lika snabbt** som förändring på plats.
- Det är först när man gör **väldigt många** upprepade ändringar på, fördatastrukturen ogynsam plats, som det blir långsamt.
- Hur många är "väldigt många"?

# Förändringsbar eller oföränderlig?

- Om den underligande **oföränderliga** datastrukturen är **smart** implementerad så att den **återanvänder redan allokerade objekt** – vilket ju är ofarligt eftersom de aldrig kommer att ändras – så är oföränderlighet **minst lika snabbt** som förändring på plats.
- Det är först när man gör **väldigt många** upprepade ändringar på, fördatastrukturen ogynsam plats, som det blir långsamt.
- Hur många är "väldigt många"?  
→ Det ska vi undersöka nu.

# String eller StringBuilder?

- Strängar i JVM är **oföränderliga**.
- Implementationen av sekvensdatastrukturen `java.lang.String` är **mycket effektivt** implementerad, där **redan allokerade objekt återanvänds** på smarta sätt.
- **MEN** väldigt många tillägg på slutet blir långsamt. Därför finns den föränderliga `StringBuilder` med den effektivt implementerade metoden `append` som **ändrar på plats**.

# String eller StringBuilder?

- Strängar i JVM är **oföränderliga**.
- Implementationen av sekvensdatastrukturen `java.lang.String` är **mycket effektivt** implementerad, där **redan allokerade objekt återanvänds** på smarta sätt.
- **MEN** väldigt många tillägg på slutet blir långsamt. Därför finns den föränderliga `StringBuilder` med den effektivt implementerade metoden `append` som **ändrar på plats**.
- Undersök dokumentationen för `StringBuilder` här:  
<https://docs.oracle.com/javase/8/docs/api/java/lang/StringBuilder.html>

# String eller StringBuilder?

- Strängar i JVM är **oföränderliga**.
- Implementationen av sekvensdatastrukturen `java.lang.String` är **mycket effektivt** implementerad, där **redan allokerade objekt återanvänds** på smarta sätt.
- **MEN** väldigt många tillägg på slutet blir långsamt. Därför finns den föränderliga `StringBuilder` med den effektivt implementerade metoden `append` som **ändrar på plats**.
- Undersök dokumentationen för `StringBuilder` här:  
<https://docs.oracle.com/javase/8/docs/api/java/lang/StringBuilder.html>
- För vilka teckensekvensalgoritmer är det lönt att använda `StringBuilder`?

# String eller StringBuilder?

- Strängar i JVM är **oföränderliga**.
- Implementationen av sekvensdatastrukturen `java.lang.String` är **mycket effektivt** implementerad, där **redan allokerade objekt återanvänds** på smarta sätt.
- **MEN** väldigt många tillägg på slutet blir långsamt. Därför finns den föränderliga `StringBuilder` med den effektivt implementerade metoden `append` som **ändrar på plats**.
- Undersök dokumentationen för `StringBuilder` här:  
<https://docs.oracle.com/javase/8/docs/api/java/lang/StringBuilder.html>
- För vilka teckensekvensalgoritmer är det lönt att använda `StringBuilder`?  
→ Det ska vi undersöka nu.

# Timer

- `System.currentTimeMillis` ger tiden i millisekunder sedan januari 1970.
- Med `Timer.measure{ xxx }` nedan kan man mäta tiden det tar för `xxx`.
- Ett par (`elapsedMillis`, `result`) returneras som innehåller tiden det tar att köra blocket, samt resultatet av blocket.

```
1 object Timer {  
2   private var startTime: Long = System.currentTimeMillis  
3  
4   def elapsedMillis: Long = System.currentTimeMillis - startTime  
5  
6   def reset: Unit = { startTime = System.currentTimeMillis }  
7  
8   def measure[T](block: => T): (Long, T) = {  
9     reset  
10    val result = block  
11    (elapsedMillis, result)  
12  }  
13 }
```

# NanananananananaNanananananananaBatman

```
1  object NanananananananananaNanananananananananaBatman {
2    import Timer.measure
3
4    val na = "NanananananananananaNanananananananananaBatman"
5
6    def batmanImmutableTime(n: Int): (Long, String) = measure {
7      var result: String = na // Strings are immutable
8      for (i <- 2 to n) {
9        result = result + na // Creates a new String
10     }
11     result // return da String
12   }
13
14   def batmanMutableTime(n: Int): (Long, String) = measure {
15     var sb = new StringBuilder(na) // StringBuilder is mutable
16     for (i <- 2 to n) {
17       sb.append(na) // append ***mutates*** StringBuilder
18     }
19     sb.toString // convert to immutable String and return
20   }
21
22   def main(args: Array[String]): Unit = {
23     val warmupJVM = (batmanMutableTime(100), batmanImmutableTime(100))
24     println("Warmup:\n" + warmupJVM._1 + "\n" + warmupJVM._2 + "\n\n")
25   }
```



# Registrering

# Att välja sekvenssamling

# Oföränderlig eller förändringsbar?

- **Oföränderlig**: Kan ej ändra elementreferenserna, men effektiv på att skapa kopia som är (delvis) förändrad (vanliga i Scala, men inte i Java): **Vector** eller **List**
- **Förändringsbar**: kan ändra elementreferenserna
  - Kan **ej ändra storlek** efter allokering:  
Scala+Java: **Array**: indexera och uppdatera varsomhelst
  - Kan ändra storlek efter allokering:  
Scala: **ArrayBuffer** eller **ListBuffer**  
Java: **ArrayList** eller **LinkedList**

# Egenskaper hos några sekvenssamlingar

## ■ Vector

- **Oföränderlig**. Snabb på att skapa kopior med små förändringar.
- Allsidig prestanda: **bra till det mesta**.

## ■ List

- **Oföränderlig**. Snabb på att skapa kopior med små förändringar.
- Snabb vid bearbetning **i början**.
- Smidig & snabb vid **rekursiva** algoritmer.
- Långsam vid upprepad **indexering** på godtyckliga ställen.

## ■ Array

- **Föränderlig**: **snabb indexering & uppdatering**.
- Kan **ej ändra storlek**; storlek anges vid allokering.
- Har särställning i JVM: ger snabbaste minnesaccessen.

## ■ ArrayBuffer

- **Föränderlig**: **snabb indexering & uppdatering**.
- Kan **ändra storlek** efter allokering. Snabb att indexera överallt.

## ■ ListBuffer

- **Föränderlig**: snabb indexering & uppdatering **i början**.
- Snabb om du bygger upp sekvens genom många tillägg i början.

# Vilken sekvenssamling ska jag välja?

## ■ Vector

- Om du vill ha oföränderlighet: **val** `xs = Vector[Int](1,2,3)`
- Om du behöver ändra (men ej prestandakritiskt):  
**var** `xs = Vector.empty[Int]`
- Om du ännu inte vet vilken sekvenssamling som är bäst; du kan alltid ändra efter att du mätt prestanda och kollat flaskhalsar.

## ■ List

- Om du har en rekursiv sekvensalgoritm och/eller bara lägger till i början.

## ■ Array

- Om det behövs av prestandaskäl och du **vet** storlek vid allokering:  
**val** `xs = Array.fill(initSize)(initValue)`

## ■ ArrayBuffer

- Om det behövs av prestandaskäl och du **inte** vet storlek vid allokering:  
**val** `xs = scala.collection.mutable.empty[Int]`

## ■ ListBuffer

- om det behövs av prestandaskäl och du bara behöver lägga till i början:  
**val** `xs = scala.collection.mutable.ListBuffer.empty[Int]`

# Lämna det öppet: använd Seq[T]

```
def varannanBaklänges[T](xs: Seq[T]): Seq[T] =  
  for (i <- xs.indices.reverse by -2) yield xs(i)
```

Fungerar med alla sekvenssamlingar:

```
scala> varannanBaklänges(Vector(1,2,3,4,5))  
res0: Seq[Int] = Vector(5, 3, 1)
```

```
scala> varannanBaklänges(List(1,2,3,4,5))  
res1: Seq[Int] = List(5, 3, 1)
```

```
scala> varannanBaklänges(collection.mutable.ListBuffer(1,2))  
res2: Seq[Int] = Vector(2)
```

Scalas standardbibliotek returnerar ofta lämpligaste specifika sekvenssamlingen som är subtyp till Seq[T].

# Denna veckas övning: sequences

- Kunna implementera funktioner som tar argumentsekvenser av godtycklig längd.
- Kunna tolka enkla sekvensalgoritmer i pseudokod och implementera dem i programkod, t.ex. tillägg i slutet, insättning, borttagning, omvändning, etc., både genom kopiering till ny sekvens och genom förändring på plats i befintlig sekvens.
- Kunna använda föränderliga och oföränderliga sekvenser.
- Förstå skillnaden mellan om sekvenser är föränderliga och om innehållet i sekvenser är föränderligt.
- Kunna välja när det är lämpligt att använda `Vector`, `Array` och `ArrayBuffer`.
- Känna till att klassen `Array` har färdiga metoder för kopiering.
- Kunna implementera algoritmer som registrerar antalet förekomster av objekt i en sekvens som indexeras med antalet förekomster.
- Kunna generera sekvenser av pseudoslumptal med specificerat slumptalsfrö.
- Kunna implementera sekvensalgoritmer i Java med **for**-sats och primitiva arrayer.
- Kunna beskriva skillnaden i syntax mellan arrayer i Scala och Java.
- Kunna använda klassen `java.util.Scanner` i Scala och Java för att läsa in heltalssekvenser från `System.in`.

# Denna veckas laboration: shuffle

- Kunna skapa och använda sekvenssamlingar.
- Kunna använda sekvensalgoritmen SHUFFLE för blandning på plats av innehållet i en array.
- Kunna registrera antalet förekomster av olika värden i en sekvens.