

# EDAA45 Programmering, grundkurs

## Läsvecka 4: Datastrukturer

Björn Regnell

Datavetenskap, LTH

Lp1-2, HT 2016

## 4 Datastrukturer

- Vad är en datastruktur?
- Tupler
- Klasser
- Case-klasser
- Samlingar
- Integrerad utvecklingsmiljö (IDE)

# Denna vecka: Datastrukturer och IDE

- Datastrukturer med tupler, klasser och färdiga samlingar
  - Mer om klasser senare:
    - w06 Klasser
    - w07 Arv
    - w09 Typparametrar
  - Mer om samlingar senare:
    - w05 Sekvensalgoritmer
    - w09 Matriser
    - w10 Sökning, Sortering
- Övning data: prova tupler, klasser och samlingsmetoder
- Laboration pirates: prata som pirater och prova IDE
- Börja använda en integrerad utvecklingsmiljö (IDE), labbförberedelse bl.a.: läs appendix D och få igång en IDE

# Vad är en datastruktur?

# Vad är en datastruktur?

- En datastruktur är en struktur för organisering av data som...
  - kan innehålla många element,
  - kan refereras till som en helhet, och
  - ger möjlighet att komma åt enskilda element.
- En **samling** (eng. *collection*) är en datastruktur som kan innehålla många element av **samma typ**.
- Exempel på **färdiga samlingar** i Scalas standardbibliotek där elementen är organiserade på olika vis så att samlingen får olika egenskaper som passar **olika användningsområden**:
  - `scala.collection.immutable.Vector`, snabb access **överallt**.
  - `scala.collection.immutable.List`, snabb access **i början**.
  - `scala.collection.immutable.Set`, `scala.collection.mutable.Set`, unika element, snabb innehållstest.
  - `scala.collection.immutable.Map` `scala.collection.mutable.Map`, nyckel-värde-par, snabb access via nyckel.
  - `scala.collection.mutable.ArrayBuffer`, kan ändra storlek.
  - `scala.Array`, motsvarar primitiva, föränderliga Java-arrayer, fix storlek.

# Olika sätt att skapa datastrukturer

## ■ Tupler

- samla  $n$  st datavärden i element **\_1**, **\_2**, ... **\_n**
- elementen kan vara av **olika** typ

## ■ Klasser

- samlar data i **attribut** med (väl valda!) namn
- attributen kan vara av **olika** typ
- definierar även **metoder** som använder attributen  
(kallas även **operationer** på data)

## ■ Färdiga samlingar

- speciella klasser som samlar data i element av **samma** typ
- exempel: `scala.collection.immutable.Vector`
- har ofta *många* färdiga **bra-att-ha-metoder**,  
se snabbreferensen <http://cs.lth.se/pgk/quickref>

## ■ Egenimplementerade samlingar

- → fördjupningskurs

# Tupler

# Vad är en tupel?

- En tupel samlar  $n$  st objekt i en enkel struktur, med koncis syntax.
- Elementen kan vara av **olika** typ.
- `("hej", 42, math.Pi)` är en **3-tupel** av typen:  
`(String, Int, Double)`
- Du kan komma åt det enskilda elementen med **\_1**, **\_2**, ... **\_n**

```
1 scala> val t = ("hej", 42, math.Pi)
2 t: (String, Int, Double) = (hej,42,3.141592653589793)
3
4 scala> t._1
5 res0: String = hej
6
7 scala> t._2
8 res1: Int = 42
```

- Tupler är praktiska när man inte vill ta det lite större arbetet att skapa en egen klass. (Men med klasser kan man göra mycket mer än med tupler.)
- I Scala kan du skapa tupler upp till en storlek av 22 element.  
(Behöver du fler element, använd i stället en samling, t.ex. `Vector`.)



# Tupler som parametrar och returvärde.

- Tupler är smidiga när man på ett enkelt och typsäkert sätt vill låta en funktion **returnera mer än ett värde**.

```
1 scala> def längd(p: (Double, Double)) = math.hypot(p._1, p._2)
2
3 scala> def vinkel(p: (Double, Double)) = math.atan2(p._1, p._2)
4
5 scala> def polär(p: (Double, Double)) = (längd(p), vinkel(p))
6
7 scala> polär((3,4))
8 res2: (Double, Double) = (5.0,0.6435011087932844)
```

- Om typerna passar kan man skippa dubbla parenteser vid **ensamt tupel-argument**:

```
1 scala> polär(3,4)
2 res3: (Double, Double) = (5.0,0.6435011087932844)
```

[https://sv.wikipedia.org/wiki/Polära\\_koordinater](https://sv.wikipedia.org/wiki/Polära_koordinater)

# Ett smidigt sätt att skapa 2-tupler med metoden ->

Det finns en metod vid namn `->` som kan användas på objekt av **godtycklig** typ för att **skapa par**:

```
1 scala> ("Ålder", 42)
2 res0: (String, Int) = (Ålder,42)
3
4 scala> "Ålder".->(42)
5 res1: (String, Int) = (Ålder,42)
6
7 scala> "Ålder" -> 42
8 res2: (String, Int) = (Ålder,42)
9
10 scala> Vector("Ålder" -> 42, "Längd" -> 178, "Vikt" -> 65)
11 res3: scala.collection.immutable.Vector[(String, Int)] =
12     Vector((Ålder,42), (Längd,178), (Vikt, 65))
```

# Klasser

# Vad är en klass?

Vi har tidigare deklarerat **singelobjekt** som bara finns i **en instans**:

```
scala> object Björn { var ålder = 49; val längd = 178 }
```

Med en **klass** kan man skapa **godtyckligt många instanser av klassen** med hjälp av nyckelordet **new** följt av klassens namn:

```
scala> class Person { var ålder = 0; var längd = 0 }
```

```
scala> val björn = new Person  
björn: Person = Person@7ae75ba6
```

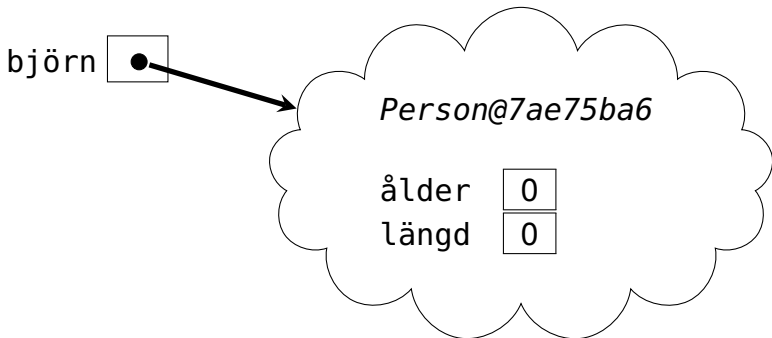
```
scala> björn.ålder = 49
```

```
scala> björn.längd = 178
```

- En klass kan ha **medlemmar** (i likhet med singelobjekt).
- Funktioner som är medlemmar kallas **metoder**.
- Variabler som är medlemmar kallas **attribut**.

## Vid new allokeras plats i minnet för objektet

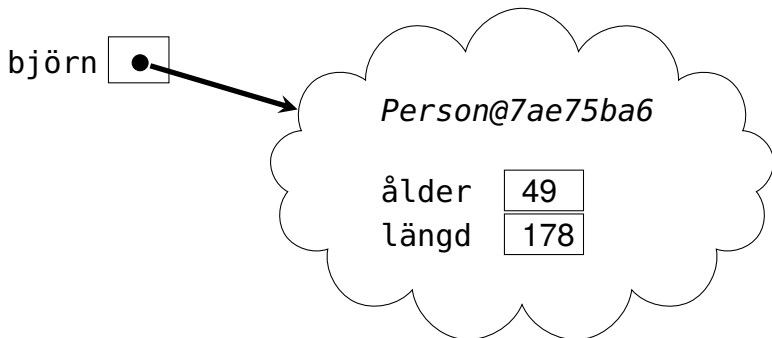
```
scala> class Person { var ålder = 0; var längd = 0 }  
  
scala> val björn = new Person  
björn: Person = Person@7ae75ba6
```



`Person@7ae75ba6` är en unik idenfierare för instansen, så att JVM hittar den i heapen.

Med punktnotation kan förändringsbara variabler tilldelas nya värden och objektets tillstånd uppdateras.

```
scala> björn.ålder = 49  
scala> björn.längd = 178
```



# En klass kan ha parametrar som initialiserar attribut

- Med en parameterlista efter klassnamnet får man en så kallad **primärkonstruktor** för initialisering av attribut.
- Argumenten för initialiseringen ges vid **new**.

```
scala> class Person(var ålder: Int, var längd: Int)
```

```
scala> val björn = new Person(49, 178)
```

```
björn: Person = Person@354baab2
```

```
scala> println(s"Björn är ${björn.ålder} år gammal.")
```

```
Björn är 49 år gammal.
```

```
scala> björn.ålder = 18
```

```
scala> println(s"Björn är ${björn.ålder} år gammal.")
```

```
Björn är 18 år gammal.
```

# En klass kan ha privata medlemmar

Med **private** blir en medlem **privat**: access utifrån **medges ej**.

```
1 scala> class Person(private var minÅlder: Int, private var minLängd: Int){  
2     def ålder = minÅlder  
3     }  
4  
5 scala> val björn = new Person(42, 178)  
6 björn: Person = Person@4b682e71  
7  
8 scala> println(s"Björn är ${björn.ålder} år gammal.")  
9 Björn är 42 år gammal.  
10  
11 scala> björn.minÅlder = 21  
12 error: variable minÅlder in class Person cannot be accessed in Person  
13  
14 scala> björn.längd  
15 error: value längd is not a member of Person
```

Med **private** kan man förhindra tokiga förändringar.



# Privata förändringsbara attribut och publika metoder

```
class Människa(val födelseLängd: Double, val födelseVikt: Double){  
    private var minLängd = födelseLängd  
    private var minVikt   = födelseVikt  
    private var ålder     = 0  
  
    def längd = minLängd // en sådan här metod kallas "getter"  
    def vikt   = minVikt  // vi förhindrar attributändring "utanför" klassen  
  
    val slutaVäxaÅlder      = 18  
    val tillväxtfaktorLängd = 0.00001  
    val tillväxtfaktorVikt  = 0.0002  
  
    def ät(mat: Double): Unit = {  
        if (ålder < slutaVäxaÅlder) minLängd += tillväxtfaktorLängd * mat  
        minVikt += tillväxtfaktorVikt * mat  
    }  
  
    def fyllÅr: Unit = ålder += 1  
  
    def tillstånd: String = s"Tillstånd: $minVikt kg, $minLängd cm, $ålder år"  
}
```

# Tillstånd kan förändras indirekt genom metodanrop

```
1 scala> val björn = new Människa(födelseVikt=3.5, födelseLängd=52.1)
2 björn: Människa = Människa3e52
3
4 scala> björn.tillstånd
5 res0: String = Tillstånd: 3.5 kg, 52.1 cm, 0 år
6
7 scala> for (i <- 1 to 42) björn.fyllÅr
8
9 scala> björn.tillstånd
10 res2: String = Tillstånd: 3.5 kg, 52.1 cm, 42 år
11
12 scala> björn.ät(mat=5000)
13
14 scala> björn.tillstånd
15 res3: String = Tillstånd: 4.5 kg, 52.1 cm, 42 år
```

# Metoden `isInstanceOf` och rot-typen `Any`

```
1 scala> class X(val i: Int)
2
3 scala> val a = new X(42)
4 a: X = X@117b2cc6
5
6 scala> a.isInstanceOf[X]
7 res0: Boolean = true
8
9 scala> val b = new X(42)
10 b: X = X@61ab6521
11
12 scala> b.isInstanceOf[X]
13 res1: Boolean = true
14
15 scala> a == b
16 res2: Boolean = false
17
18 scala> a.i == b.i
19 res3: Boolean = true
```

- Ett objekt skapat med **new** X är en instans av **typen** X.
- Detta kan testas med metoden `isInstanceOf[X]: Boolean`

# Metoden `isInstanceOf` och rot-typen `Any`

```
1 scala> class X(val i: Int)
2
3 scala> val a = new X(42)
4 a: X = X@117b2cc6
5
6 scala> a.isInstanceOf[X]
7 res0: Boolean = true
8
9 scala> val b = new X(42)
10 b: X = X@61ab6521
11
12 scala> b.isInstanceOf[X]
13 res1: Boolean = true
14
15 scala> a == b
16 res2: Boolean = false
17
18 scala> a.i == b.i
19 res3: Boolean = true
```

- Ett objekt skapat med **new** X är en instans av **typen** X.
- Detta kan testas med metoden `isInstanceOf[X]: Boolean`
- Typen **Any** är supertyp till **alla** typer och kallas för **rot-typ** i Scalas typhierarki.

```
1 scala> a.isInstanceOf[Any]
2 res4: Boolean = true
3
4 scala> b.isInstanceOf[Any]
5 res5: Boolean = true
6
7 scala> 42.isInstanceOf[Any]
8 res6: Boolean = true
```

(se quickref sid 4, mer om detta i w07)

# Överskugga toString

Alla objekt får automatiskt en metod `toString` som ger en sträng med objektets unika identifierare, här `Gurka@3830f1c0`:

```
1 scala> class Gurka(val vikt: Int)
2
3 scala> val g = new Gurka(42)
4 g: Gurka = Gurka@3830f1c0
5
6 scala> g.toString
7 res0: String = Gurka@3830f1c0
```

Man kan **överskugga** den automatiska `toString` med en **egen implementation**. Observera nyckelordet **override**.

```
1 scala> class Tomat(val vikt: Int){override def toString = s"Tomat($vikt g)}
2
3 scala> val t = new Tomat(142)
4 t: Tomat = Tomat(142 g)
5
6 scala> t.toString
7 res1: String = Tomat(142 g)
```

# Objektfabrik i kompanjonsobjekt

- Om det finns ett objekt i samma kodfil med samma namn som klassen blir det objektet ett s.k. **kompanjonsobjekt** (eng. *companion object*).
- Ett kompanjonsobjekt får **accessa privata medel** i den klass till vilken objektet är kompanjon.
- Kompanjonsobjekt är en bra plats för s.k. **fabriksmetoder** som skapar instanser. Då slipper vi skriva **new**.

```
1  scala> :paste    // måste skrivas tillsammans annars ingen kompanjon
2
3  class Broccoli(var vikt: Int)
4
5  object Broccoli {
6      def apply(vikt: Int): Broccoli = new Broccoli(vikt)
7  }
8
9  scala> val b = Broccoli(420)
10 b: Broccoli = Broccolila6f2363
```

# Kompanjonsobjekt kan accessa privata medlemmar

```
class Gurka(startVikt: Double) {
  private var vikt = startVikt
  def åt(tugga: Int): Unit = if (vikt > tugga) vikt -= tugga else vikt = 0
  override def toString = s"Gurka($vikt)"
}

object Gurka {
  private var totalVikt = 0.0
  def apply(): Gurka = {
    val g = new Gurka(math.random * 0.42 + 0.1)
    totalVikt += g.vikt // hade blivit kompileringsfel om ej vore kompanjon
    g
  }
  def rapport: String = s"Du har skapat ${totalVikt.toInt} kg gurka."
}
```

```
1 scala> val gs = Vector.fill(1000)(Gurka())
2 gs: scala.collection.immutable.Vector[Gurka] =
3   Vector(Gurka(0.49018400799506734), Gurka(0.2462822679714138), Gurka(0.173913
4
5 scala> println(Gurka.rapport)
6 Du har skapat 305 kg gurka.
```

# Förändringsbara och oföränderliga objekt

Ett **oföränderligt objekt** där nya instanser skapas i stället för tillståndsändring "på plats".

```
class Point(val x: Int, val y: Int) {  
  def moved(dx: Int, dy: Int): Point = new Point(x + dx, y + dy)  
  
  override def toString: String = s"Point($x, $y)"  
}
```

Ett **förändringsbart** objekt där **tillståndet uppdateras**.

```
class MutablePoint(private var x: Int, private var y: Int) {  
  def move(dx: Int, dy: Int): Unit = {x += dx; y += dy} // Mutation!!!  
  
  override def toString: String = s"MutablePoint($x, $y)"  
}
```



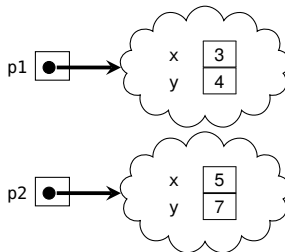
# Oföränderliga objekt

```
1 scala> var p1 = new Point(3, 4)
2 p1: Point = Point(3, 4)
3
4 scala> val p2 = p1.moved(2, 3)
5 p2: Point = Point(5, 7)
6
7 scala> println(p1)
8 Point(3, 4)
9
10 scala> p1 = new Point(0, 0)
11 p1: Point = Point(0, 0)
```

# Oföränderliga objekt

```
1 scala> var p1 = new Point(3, 4)
2 p1: Point = Point(3, 4)
3
4 scala> val p2 = p1.moved(2, 3)
5 p2: Point = Point(5, 7)
6
7 scala> println(p1)
8 Point(3, 4)
9
10 scala> p1 = new Point(0, 0)
11 p1: Point = Point(0, 0)
```

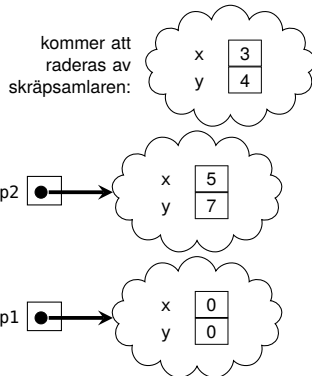
Minnessituationen efter rad 7:



# Oföränderliga objekt

Minnessituationen efter rad 10:

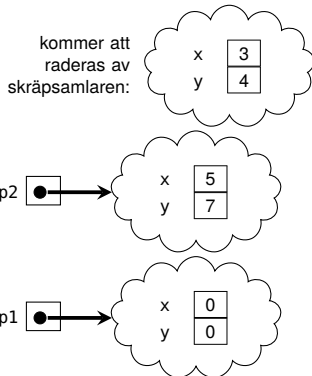
```
1 scala> var p1 = new Point(3, 4)
2 p1: Point = Point(3, 4)
3
4 scala> val p2 = p1.moved(2, 3)
5 p2: Point = Point(5, 7)
6
7 scala> println(p1)
8 Point(3, 4)
9
10 scala> p1 = new Point(0, 0)
11 p1: Point = Point(0, 0)
```



# Oföränderliga objekt

Minnessituationen efter rad 10:

```
1 scala> var p1 = new Point(3, 4)
2 p1: Point = Point(3, 4)
3
4 scala> val p2 = p1.moved(2, 3)
5 p2: Point = Point(5, 7)
6
7 scala> println(p1)
8 Point(3, 4)
9
10 scala> p1 = new Point(0, 0)
11 p1: Point = Point(0, 0)
```

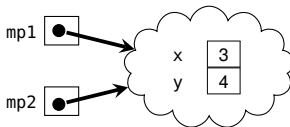


Vi kan **lugnt dela referenser** till vårt oföränderliga objekt eftersom det **aldrig** kommer att ändras.

# Förändringsbara objekt

```
1 scala> val mp1 = new MutablePoint(3, 4)
2 mp1: MutablePoint = MutablePoint(3, 4)
3
4 scala> val mp2 = mp1
5 mp2: MutablePoint = MutablePoint(3, 4)
6
7 scala> mp1.move(2,3)
8
9 scala> println(mp2)
10 MutablePoint(5, 7)
```

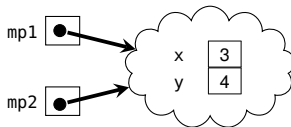
Minnessituationen efter rad 4:



# Förändringsbara objekt

```
1 scala> val mp1 = new MutablePoint(3, 4)
2 mp1: MutablePoint = MutablePoint(3, 4)
3
4 scala> val mp2 = mp1
5 mp2: MutablePoint = MutablePoint(3, 4)
6
7 scala> mp1.move(2,3)
8
9 scala> println(mp2)
10 MutablePoint(5, 7)
```

Minnessituationen efter rad 4:

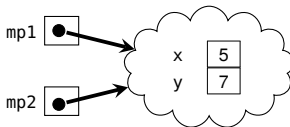


**Varning!** Vem som helst som har tillgång till en referens till ditt förändringsbara objekt kan **manipulera** det, vilket ibland ger överaskande och **problematiska** konsekvenser!

# Förändringsbara objekt

```
1 scala> val mp1 = new MutablePoint(3, 4)
2 mp1: MutablePoint = MutablePoint(3, 4)
3
4 scala> val mp2 = mp1
5 mp2: MutablePoint = MutablePoint(3, 4)
6
7 scala> mp1.move(2,3)
8
9 scala> println(mp2)
10 MutablePoint(5, 7)
```

Minnessituationen efter **rad 7**:



**Varning!** Vem som helst som har tillgång till en referens till ditt förändringsbara objekt kan **manipulera** det, vilket ibland ger överaskande och **problematiska** konsekvenser!

# Case-klasser



# Vad är en case-klass?

- En **case**-klass är ett smidigt sätt att skapa **oföränderliga objekt**.
- Kompilatorn ger dig **en massa "godis"** på köpet (ca 50-100 rader kod), inkl.:
  - klassparametrar blir automatiskt **val**-attribut, alltså **publika** och **oföränderliga**,
  - en automatisk **toString** som visar klassparametrarnas värde,
  - ett automatiskt **kompanjonsobjekt** med **fabriksmetod** så du slipper skriva **new**,
  - automatiska metoden **copy** för att skapa kopior med andra attributvärden, m.m...  
(Mer om detta i w06 & w11, men är du nyfiken kolla på uppgift 2d) på sid 261.)

# Vad är en case-klass?

- En **case**-klass är ett smidigt sätt att skapa **oföränderliga objekt**.
- Kompilatorn ger dig **en massa "godis"** på köpet (ca 50-100 rader kod), inkl.:
  - klassparametrar blir automatiskt **val**-attribut, alltså **publika** och **oföränderliga**,
  - en automatisk **toString** som visar klassparametrarnas värde,
  - ett automatiskt **kompanjonsobjekt** med **fabriksmetod** så du slipper skriva **new**,
  - automatiska metoden **copy** för att skapa kopior med andra attributvärden, m.m...  
(Mer om detta i w06 & w11, men är du nyfiken kolla på uppgift 2d) på sid 261.)
- Det **enda** du behöver göra är att lägga till nyckelordet **case** före **class**...

```
scala> case class Point(x: Int, y: Int)
```

```
scala> val p = Point(3, 5)  
p: Point = Point(3,5)
```

```
scala> p. // tryck TAB och se lite av allt case-klass-godis  
scala> Point. // tryck TAB och se ännu mer godis
```

```
scala> val p2 = p.copy(y=30)  
p2: Point = Point(3,30)
```

# Exempel på case-klasser

```
case class Person(namn: String, ålder: Int) {  
  def fyllerJämt: Boolean = ålder % 10 == 0  
  def hyllning = if (fyllerJämt) "Extra grattis!" else "Vi gratulerar!"  
  def ärLikaGammalSom(annan: Person) = ålder == annan.ålder  
}  
  
case class Point(x: Int = 0, y: Int = 0) {  
  def distanceTo(other: Point) = math.hypot(x - other.x, y - other.y)  
  def dx(d: Int): Point = copy(x + d, y)  
  def dy(d: Int): Point = copy(y=y + d) //namngivet arg. och defaultarg.  
}  
object Point {  
  def origin = new Point()  
}
```

```
1 scala> Point().dx(10).dy(10).dx(32)  
2 res0: Point = Point(42,10)  
3  
4 scala> Point(3,4) distanceTo Point.origin  
5 res1: Double = 5.0
```

# Synlighet av klassparametrar i klasser & case-klasser

**private[this]** är **ännu** mer privat än **private**

```
class Hemlis(private val hemlis: Int) {  
  def ärSammaSom(annan: Hemlis) = hemlis == annan.hemlis // Funkar!  
}  
  
class Hemligare(private[this] val hemlis: Int) {  
  def ärSammaSom(annan: Hemligare) = hemlis == annan.hemlis //KOMPILERINGSFEL  
}
```

Vad händer om man inte skriver något? Olika för klass och case-klass:

```
class Hemligare(hemlis: Int) { // motsvarar private[this] val  
  def ärSammaSom(annan: Hemligare) = hemlis == annan.hemlis //KOMPILERINGSFEL  
}  
  
case class InteHemlig(seMenInteRöra: Int) { // blir automatiskt val  
  def ärSammaSom(annan: InteHemlig): Boolean =  
    seMenInteRöra == annan.seMenInteRöra  
}
```

# Samlingar

# Vad är en samling?

En **samling** (eng. *collection*) är en datastruktur som kan innehålla många element av **samma typ**.

# Vad är en samling?

En **samling** (eng. *collection*) är en datastruktur som kan innehålla många element av **samma typ**.

*Exempel:*

Heltalsvektor:      **val** xs = Vector(-1, 2, 3, 42, 0)

# Vad är en samling?

En **samling** (eng. *collection*) är en datastruktur som kan innehålla många element av **samma typ**.

*Exempel:*

Heltalsvektor: **val** xs = Vector(-1, 2, 3, 42, 0)

Samlingar implementeras med hjälp av klasser.

I standardbiblioteken `scala.collection` och `java.util` finns **många färdiga samlingar**, så man behöver sällan implementera egna.



# Vad är en samling?

En **samling** (eng. *collection*) är en datastruktur som kan innehålla många element av **samma typ**.

*Exempel:*

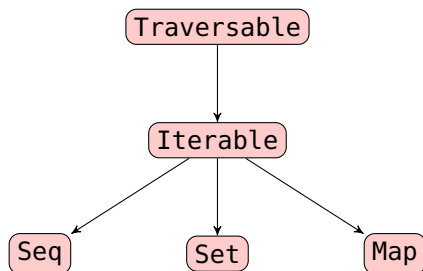
Heltalsvektor: **val** xs = Vector(-1, 2, 3, 42, 0)

Samlingar implementeras med hjälp av klasser.

I standardbiblioteken `scala.collection` och `java.util` finns **många färdiga samlingar**, så man behöver sällan implementera egna.

Om man behöver en egen, speciell datastruktur är det oftast bäst att skapa en klass som *innehåller* en *färdig* samling och utgå från färdiga metoder.

# Hierarki av samlingstyper i `scala.collection`



Traversable har metoder som är implementerade med hjälp av:

```
def foreach[U](f: Elem => U): Unit
```

Iterable har metoder som är implementerade med hjälp av:

```
def iterator: Iterator[A]
```

Seq: ordnade i sekvens

Set: unika element

Map: par av (nyckel, värde)

Samlingen **Vector** är en Seq som är en Iterable som är en Traversable.

# Använda iterator

Med en iterator kan man **iterera** med **while** över alla element, men endast **en gång**; sedan är iteratorn "förbrukad". (Men man kan be om en ny.)

```
1 scala> val xs = Vector(1,2,3,4)
2 xs: scala.collection.immutable.Vector[Int] = Vector(1, 2, 3, 4)
3
4 scala> val it = xs.iterator
5 it: scala.collection.immutable.VectorIterator[Int] = non-empty iterator
6
7 scala> while (it.hasNext) print(it.next)
8 1234
9
10 scala> it.hasNext
11 res1: Boolean = false
12
13 scala> it.next
14 java.util.NoSuchElementException: reached iterator end
15   at scala.collection.immutable.VectorIterator.next(Vector.scala:674)
```

**Normalt** behöver man **inte** använda iterator: det finns oftast färdiga metoder som gör det man vill, till exempel foreach, map, sum, min etc.

# Några användbara metoder på samlingar

## Traversable

<code>xs.size</code>	antal elementet
<code>xs.head</code>	första elementet
<code>xs.last</code>	sista elementet
<code>xs.take(n)</code>	ny samling med de första n elementet
<code>xs.drop(n)</code>	ny samling utan de första n elementet
<code>xs.foreach(f)</code>	gör f på alla element, returtyp Unit
<code>xs.map(f)</code>	gör f på alla element, ger <b>ny</b> samling

## Iterable

<code>xs.zip(ys)</code>	ny samling med par (x, y); "zippa ihop" xs och ys
<code>xs.zipWithIndex</code>	ny samling med par (x, index för x)
<code>xs.sliding(n)</code>	ny samling av samlingar genom glidande "fönster"

## Seq

<code>xs.length</code>	samma som <code>xs.size</code>
<code>xs :+ x</code>	ny samling med x sist efter xs
<code>x ++ xs</code>	ny samling med x före xs

# Några användbara metoder på samlingar

## Traversable

<code>xs.size</code>	antal elementet
<code>xs.head</code>	första elementet
<code>xs.last</code>	sista elementet
<code>xs.take(n)</code>	ny samling med de första n elementet
<code>xs.drop(n)</code>	ny samling utan de första n elementet
<code>xs.foreach(f)</code>	gör f på alla element, returtyp Unit
<code>xs.map(f)</code>	gör f på alla element, ger <b>ny</b> samling

## Iterable

<code>xs.zip(ys)</code>	ny samling med par (x, y); "zippa ihop" xs och ys
<code>xs.zipWithIndex</code>	ny samling med par (x, index för x)
<code>xs.sliding(n)</code>	ny samling av samlingar genom glidande "fönster"

## Seq

<code>xs.length</code>	samma som <code>xs.size</code>
<code>xs :+ x</code>	ny samling med x sist efter xs
<code>x ++ xs</code>	ny samling med x före xs

**Minnesregel** för `++` och `:+`    **Colon on the collection side**

# Några användbara metoder på samlingar

## Traversable

<code>xs.size</code>	antal elementet
<code>xs.head</code>	första elementet
<code>xs.last</code>	sista elementet
<code>xs.take(n)</code>	ny samling med de första n elementet
<code>xs.drop(n)</code>	ny samling utan de första n elementet
<code>xs.foreach(f)</code>	gör f på alla element, returtyp Unit
<code>xs.map(f)</code>	gör f på alla element, ger <b>ny</b> samling

## Iterable

<code>xs.zip(ys)</code>	ny samling med par (x, y); "zippa ihop" xs och ys
<code>xs.zipWithIndex</code>	ny samling med par (x, index för x)
<code>xs.sliding(n)</code>	ny samling av samlingar genom glidande "fönster"

## Seq

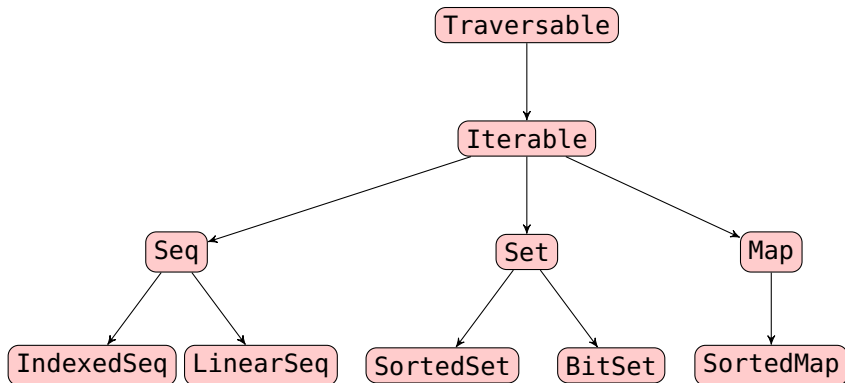
<code>xs.length</code>	samma som <code>xs.size</code>
<code>xs :+ x</code>	ny samling med x sist efter xs
<code>x ++ xs</code>	ny samling med x före xs

**Minnesregel** för ++ och :+    **Colon on the collection side**

Prova fler samlingsmetoder ur snabbreferensen: <http://cs.lth.se/quickref>

# Mer specifik samlingstyper i `scala.collection`

Det finns **mer specifika subtyper** av Seq, Set och Map:



**Vector** är en **IndexedSeq** medan **List** är en **LinearSeq**.

# Några oföränderliga och förändringsbara sekvenssamlingar

`scala.collection.immutable.Seq.`

`IndexedSeq.`

**Vector**  
**Range**

`LinearSeq.`

**List**  
**Queue**

`scala.collection.mutable.Seq.`

`IndexedSeq.`

**ArrayBuffer**  
**StringBuilder**

`LinearSeq.`

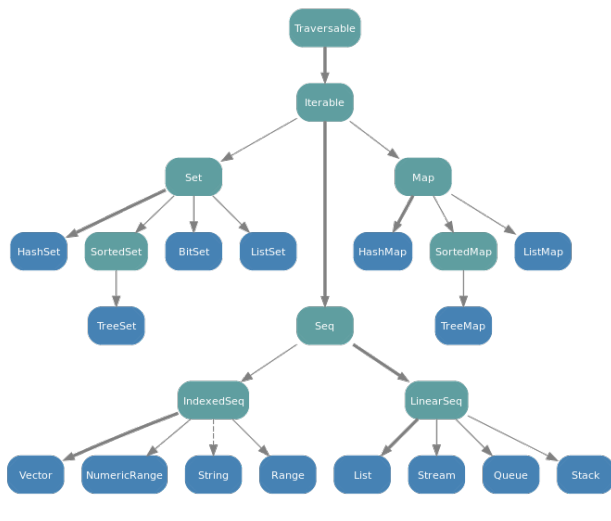
**ListBuffer**  
**Queue**

Studera samlingars egenskaper här:

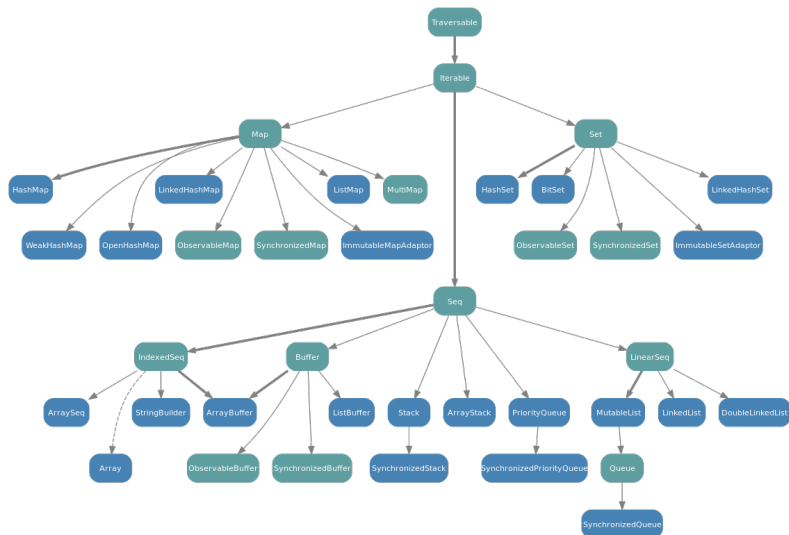
[docs.scala-lang.org/overviews/collections/overview](https://docs.scala-lang.org/overviews/collections/overview)



# scala.collection.immutable



# scala.collection.mutable



# Vector eller List?

[stackoverflow.com/questions/6928327/when-should-i-choose-vector-in-scala](https://stackoverflow.com/questions/6928327/when-should-i-choose-vector-in-scala)

- 1 If we only need to transform sequences by operations like map, filter, fold etc: basically it does not matter, we should program our algorithm generically and might even benefit from accepting parallel sequences. For sequential operations List is probably a bit faster. But you should benchmark it if you have to optimize.
- 2 If we need a lot of random access and different updates, so we should use vector, list will be prohibitively slow.
- 3 If we operate on lists in a classical functional way, building them by prepending and iterating by recursive decomposition: use list, vector will be slower by a factor 10-100 or more.
- 4 If we have an performance critical algorithm that is basically imperative and does a lot of random access on a list, something like in place quick-sort: use an imperative data structure, e.g. ArrayBuffer, locally and copy your data from and to it.

[stackoverflow.com/questions/20612729/how-does-scalas-vector-work](https://stackoverflow.com/questions/20612729/how-does-scalas-vector-work)

Mer om tids- och minneskomplexitet i fördjupningskursen och senare kurser.

# Speciella metoder på förändringsbara samlingar

# Mängd: snabb innehållstest, garanterat dubblettfri

# Den fantastiska nyckel-värde-tabellen Map

# Integrerad utvecklingsmiljö (IDE)

# Välja IDE



# Denna veckas övning: data

- Kunna skapa och använda tupler, som variabelvärden, parametrar och returvärden.
- Förstå skillnaden mellan ett objekt och en klass och kunna förklara betydelsen av begreppet instans.
- Kunna skapa och använda attribut som medlemmar i objekt och klasser och som som klassparametrar.
- Beskriva innebörden av och syftet med att ett attribut är privat.
- Kunna byta ut implementationen av metoden `toString`.
- Kunna skapa och använda en objektfabrik med metoden `apply`.
- Kunna skapa och använda en enkel case-klass.
- Kunna använda operatortnotation och förklara relationen till punktnotation.
- Förstå konsekvensen av uppdatering av föränderlig data i samband med multipla referenser.
- Känna till och kunna använda några grundläggande metoder på samlingar.
- Känna till den principiella skillnaden mellan `List` och `Vector`.
- Kunna skapa och använda en oföränderlig mängd med klassen `Set`.
- Förstå skillnaden mellan en mängd och en sekvens.
- Kunna skapa och använda en nyckel-värde-tabell, `Map`.
- Förstå likheter och skillnader mellan en `Map` och en `Vektor`.

# Denna veckas laboration: pirates

- Kunna använda en integrerad utvecklingsmiljö (IDE).
- Kunna använda färdiga funktioner för att läsa till, och skriva från, textfil.
- Kunna använda enkla case-klasser.
- Kunna skapa och använda enkla klasser med föränderlig data.
- Kunna använda samlingstyperna `Vector` och `Map`.
- Kunna skapa en ny samling från en befintlig samling.
- Förstå skillnaden mellan kompileringsfel och exekveringsfel.
- Kunna felsöka i små program med hjälp av utskrifter.
- Kunna felsöka i små program med hjälp av en debugger i en IDE.