# Vecka 9. Mängder och tabeller

## Programmering, grundkurs (pgk)

Björn Regnell

Datavetenskap, LTH, Lunds universitet
https://cs.lth.se/pgk

EDAA45, Lp1-2, HT2024

Kompilerad den 14 maj 2025

9 Mängder och tabeller
- Veckans uppgifter
- Kodgranskning
- `scala.collection`
- Repetition: sekvens
- Mängd
- Nyckel-värde-tabell
- Tips inför veckans uppgifter
- Serialisering och deserialisering

# Omkontroll

- De som ej deltog i kontrollskrivningen ska delat på **omkontrollskrivning** **Torsd. 14/11 kl 13:00-18:00, E:1147**
- Detta obligatoriska moment krävs för godkänt på kursen.
- Du som inte deltog vid ordinarie skrivning ska ha fått mejl från `bjorn.regnell@cs.lth.se` med instruktioner. Om du ej fått mejl men ska delta: mejla Björn snarast!

# Veckans uppgifter

# Övning: `lookup`

- Uppgifterna innehåller delar som är **nödvändiga** för laborationen.
- På övningen tränar du på **mängder** och **nyckel-värde-tabeller**.
- Du ska skapa en klass `FreqMapBuilder` som bygger upp en tabell med ordfrekvenser, som behövs på labben.

# Laboration: `words`

- Denna uppgift handlar om analys av naurligt språk (eng. *Natural Language Processing, NLP*).
- Svara på frågorna:
  - Hur vanligt är ett visst ord i en given text?
  - Vilket är det vanligaste ordet som följer efter ett visst ord?
  - Hur kan man generera ordsekvenser som liknar ordföljden i en given text?
- Använda mängd för unika ord.
- Använda nyckel-värde-tabell för att för varje ord i en lång text räkna antalet förekomster av detta ord.

# Laboration: `words`

Lärandemål:

- Kunna skapa och använda nyckel-värde-tabeller med samlingstypen `Map`.
- Kunna skapa och använda mängder med samlingstypen `Set`.
- Förstå skillnader och likheter mellan en sekvens och en mängd.
- Förstå likheter och skillnader mellan en sekvens av par och en nyckel-värde-tabell.
- Kunna implementera algoritmer som använder nästlade strukturer.

Uppgifter:

- Dela upp en sträng i ord.
- Skapa ordfrekvenstabeller för böcker som ditt program laddar ner från nätet via projektet Gutenberg med fria böcker.
- Skapa frekvenstabeller för ordföljder, s.k. *n-gram*.
- Skriv ut intressant statistik om ordvalen i olika böcker, t.ex. ur könsrollsperspektiv.
- Valfri uppgift: Gör en bot som genererar slumpvisa, artificiella meningar som liknar mänskligt språk.

# Kodgranskning

# Att läsa kod

En förutsättning för att kunna *skriva* bra kod är att kunna *läsa* kod aktivt. Du behöver kontinuerlig träning i att **läsa** kod! (inte bara skriva)

**Hur läsa kod?**

# Att läsa kod

En förutsättning för att kunna *skriva* bra kod är att kunna *läsa* kod aktivt. Du behöver kontinuerlig träning i att **läsa** kod! (inte bara skriva)

**Hur läsa kod?**

- Skapa överblick av vilka kodfiler som finns och vad som finns i vilken fil.
- Studera dokumentation om vad som är **syftet** med olika abstraktioner.
- Studera klassparametrar och metodhuvuden. **Typerna** är dina **tankeverktyg**: ”Vad kommer in?” och ”Vad kommer ut?”
- Ställ dig under läsningen frågorna:
  ”Vad finns?” och ”Vad fattas?” för att du ska kunna lösa din uppgift.
- Läs iterativt. Vänta med implementationsdetaljer. Hoppa mellan deklaration och användning.
- Studera **beroenden**: `Matrix -> Life -> LifeWindow -> Main`
- Två strategier i din stegvisa läsning som kan mixas:
  Bottom-Up: Börja med delar med **minst** beroenden till andra delar.
  Top-Down: Börja med de delar som används av **huvudprogrammet**.
- Var aktiv när du läser! Anteckna; skriv ner frågor; experimentera i REPL.

# Kodgranskning i industriell systemutveckling

- Ett effektivt sätt att upptäcka fel är att människor **noga läser igenom** sin egen och andras kod, och försöker hitta relevanta **problem och förbättringsmöjligheter**.
- Man blir ofta ”hemmablind” när det gäller ens egen kod. Därför kan någon annans, oberoende granskning med ”nya, friska” ögon vara mycket fruktbar.
- I samband med kodgranskning kan man med fördel försöka bedöma huruvida koden är:
  - lätt att läsa,
  - lätt att ändra i,
  - annat som är viktigt för den framtida utvecklingen.
- Ofta hittar man vid granskning även enkla programmeringsmisstag, så som felaktiga villkor och loop-räknare som inte räknas upp på rätt sätt etc.

# Nyttan med kodgranskningar

Väl genomförda kodgranskningar är effektiva och nyttiga:

- Bra på att upptäcka problem, även sådana som varken kompilator eller testning hittar.
- Sprider kunskap inom en arbetsgrupp, speciellt mellan erfarna och juniora utvecklare.
- En god kodgranskningsprocess främjar en kultur av gemensamt ägarskap och respektfull samverkan.

# Kodgranskning under grupplaborationen snake

Grupplaborationen snake går över två läsveckor (w10–w11).
Du ska:

- delta i framtagande av en gemensam checklista för kodgranskning,
- granska minst en annan gruppmedlems kod,
- bjuda in minst en annan gruppmedlem att granska din kod,
- ge konstruktiv feedback,
- ta emot konstruktiv feedback och försöka förbättra din kod,
- på redovisning redogöra för nyttan och utmaningarna med kodgranskningar utifrån dina egna erfarenheter.

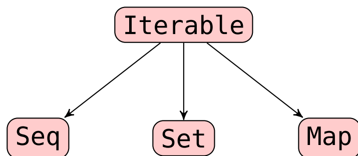
Mer om kodgranskning i efterföljande kurser, t.ex. ”Programvaruutveckling i grupp” och ”Programvaruutveckling för stora system”

# Gästföreläsning om kodgranskningar i praktiken

- **Sprid info till alla**: På andra föreläsningen i nästa vecka kommer **gäst från industrin**: den erfarne utvecklaren **Gustaf Lundh** från Axis och gästföreläser om kodgranskningar i praktiken.
- Gästföreläsningen ger dig en flygande start inför de kodgranskningar du ska genomföra i grupplabben `snake`.
- **Missa inte det!**

# scala.collection

# Hierarki av samlingstyper i `scala.collection` v2.13




`Iterable` har metoder som är implementerade med hjälp av:

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

De konkreta samlingarna är uppdelade i dessa paket:

`scala.collection.immutable` där flera är **automatiskt** importerade

`scala.collection.mutable` som **måste importeras** explicit

(undantag: primitiva förändringsbara `scala.Array` är automatiskt synlig)

# Metoden `iterator` ger en ”engångs-iterator”

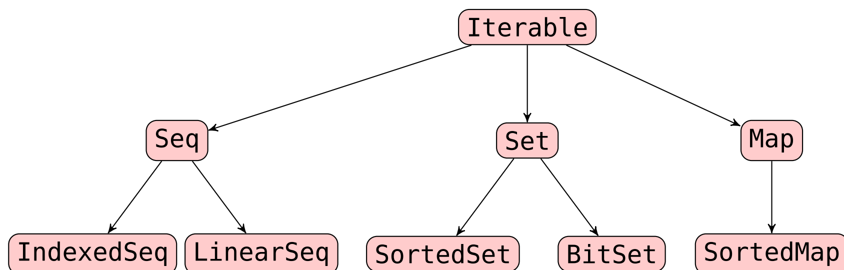
Med `iterator` kan man iterera med **`while`**, men endast **en gång**; sedan är iteratorn ”förbrukad”. (Men man kan be om en ny.) Används ”under huven” i samlingsbiblioteket för att implementera andra metoder.

```
scala> val xs = Vector(1,2,3,4)
val xs: Vector[Int] = Vector(1, 2, 3, 4)

scala> val it = xs.iterator
val it: Iterator[Int] = <iterator>

scala> while it.hasNext do print(it.next)
1234

scala> it.hasNext
val res0: Boolean = false

scala> it.next
java.util.NoSuchElementException: next on empty iterator
```

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

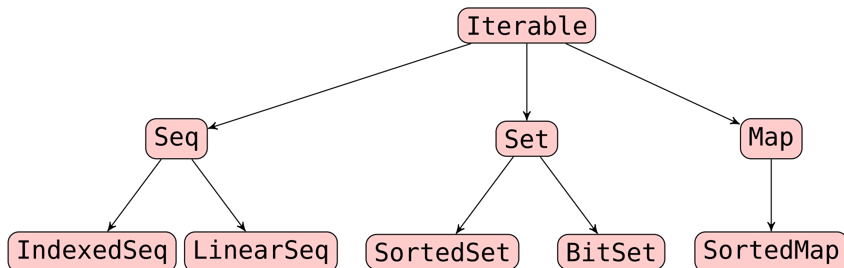
# Mer specifika samlingstyper i `scala.collection`

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



# Mer specifika samlingstyper i `scala.collection`

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




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

docs.scala-lang.org/overviews/collections-2.13/overview.html

# 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
```

Fördjupning: Studera samlingars prestanda-egenskaper här:

docs.scala-lang.org/overviews/collections-2.13/performance-characteristics.html

# Några användbara metoder på samlingar

| | | |
|---|---|---|
| **Iterable** | `xs.size` | antal elementet |
| | `xs.head` | första elementet |
| | `xs.last` | sista elementet |
| | `xs.take(n)` | ny samling med de första n elementet |
| | `xs.drop(n)` | ny samling utan de första n elementet |
| | `xs.foreach(f)` | gör f på alla element, returtyp `Unit` |
| | `xs.map(f)` | gör f på alla element, ger ny samling |
| | `xs.filter(p)` | ny samling med bara de element där p är sant |
| | `xs.groupBy(f)` | ger en `Map` som grupperar värdena enligt f |
| | `xs.mkString(",")` | en kommaseparerad sträng med alla element |
| | `xs.zip(ys)` | ny samling med par (x, y); ”zippa ihop” xs och ys |
| | `xs.zipWithIndex` | ger en `Map` med par (x, index för x) |
| | `xs.sliding(n)` | ny samling av samlingar genom glidande ”fönster” |
| **Seq** | `xs.length` | samma som `xs.size` |
| | `xs :+ x` | ny samling med x sist efter xs |
| | `x +: xs` | ny samling med x före xs |

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

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

Digga denna: `https://youtu.be/Lm9JWlEMHjo?si=sNdn_ZDaORlGr3lt`

# Repetition: sekvens

# Repetition: Vad är en sekvens?

- En sekvens är en **följd av element** som
  - är **numrerade** (t.ex. från noll), och
  - är av en viss **typ** (t.ex. heltal).

# Repetition: Vad är en sekvens?

- En sekvens är en **följd av element** som
  - är **numrerade** (t.ex. från noll), och
  - är av en viss **typ** (t.ex. heltal).
- En sekvens kan innehålla **dubbletter**.
- En sekvens kan vara **tom** och ha längden noll.
- Exempel på en icke-tom sekvens med dubbletter:

```
scala> val xs = Vector(42, 0, 42, -9, 0, 13, 7)
val xs: Vector[Int] = Vector(42, 0, 42, -9, 0, 13, 7)
```

# Repetition: Vad är en sekvens?

- En sekvens är en **följd av element** som
  - är **numrerade** (t.ex. från noll), och
  - är av en viss **typ** (t.ex. heltal).
- En sekvens kan innehålla **dubbletter**.
- En sekvens kan vara **tom** och ha längden noll.
- Exempel på en icke-tom sekvens med dubbletter:

```
scala> val xs = Vector(42, 0, 42, -9, 0, 13, 7)
val xs: Vector[Int] = Vector(42, 0, 42, -9, 0, 13, 7)
```

- **Indexering** ger ett element via dess ordningsnummer:

```
1  scala> xs(2)
2  val res0: Int = 42
3
4  scala> xs.apply(2)
5  val res1: Int = 42
```

# En sträng är också en `IndexedSeq[Char]`

Det sker vid behov **implicit konvertering** från `String` till `IndexedSeq[Char]`.

```
scala> val x: IndexedSeq[Char] = "hej"
val x: IndexedSeq[Char] = hej
```

Detta gör att **alla samlingsmetoder på Seq även funkar på strängar** och även flera andra smidiga strängmetoder erbjuds **utöver** de som finns i `java.lang.String` genom klassen `StringOps`.

```
scala> "hej".  //tryck på TAB och se alla strängmetoder
JLine: do you wish to see all 248 possibilities (42 lines)?
```

Detta är en stor fördel med Scala jämfört med många andra språk, som har strängar som inte kan allt som andra sekvenssamlingar kan.

# Konvertera mellan olika samlingstyper

- För vanligt förekommande konverteringar finns metoderna `toVector`, `toList`, `toArray`, `toBuffer`, `toMap`, `toSeq`, `toIndexedSeq`, `toSet`, `toString`
- Metoden `to` (ny från Scala 2.13) tar ett **kompanjonsobjekt** ur samlingsbiblioteket som argument och kan användas för konvertering till godtycklig samlingstyp.
- Detta kräver kopiering om underliggande representation är olika och samlingen är förändringsbar.
- Kan användas för att t.ex. konvertera mellan oföränderlig och förändringsbar samling:

```
scala> val ms = Set(1,2,3).to(collection.mutable.Set)
val ms: scala.collection.mutable.Set[Int] = HashSet(1, 2, 3)
```

# Mängd

# Vad är en mängd?

- En **mängd** är en samling **unika** element av en viss **typ**.
- En mängd kan alltså inte innehålla dubbletter:

```
scala> Set(1,1,2,2,3,3,4,4,5,5)
val res0: Set[Int] = HashSet(5, 1, 2, 3, 4)
```

# Vad är en mängd?

- En **mängd** är en samling **unika** element av en viss **typ**.
- En mängd kan alltså inte innehålla dubbletter:

```
scala> Set(1,1,2,2,3,3,4,4,5,5)
val res0: Set[Int] = HashSet(5, 1, 2, 3, 4)
```

- En mängd är **inte** en sekvens: du kan inte utgå från att elementen ligger i någon viss ordning, t.ex. den ordning som de ges vid konstruktion; en mängd har ej längd, men en **storlek**; metoden `size` ger antalet element men metoden `length` saknas.
- En mängd kan vara **tom** och har då storleken `0`.

# Vad är en mängd?

- En **mängd** är en samling **unika** element av en viss **typ**.
- En mängd kan alltså inte innehålla dubbletter:

```
scala> Set(1,1,2,2,3,3,4,4,5,5)
val res0: Set[Int] = HashSet(5, 1, 2, 3, 4)
```

- En mängd är **inte** en sekvens: du kan inte utgå från att elementen ligger i någon viss ordning, t.ex. den ordning som de ges vid konstruktion; en mängd har ej längd, men en **storlek**; metoden `size` ger antalet element men metoden `length` saknas.
- En mängd kan vara **tom** och har då storleken `0`.
- Man kan gå igenom element i **någon** ordning (exakt vilken är ej def.), med till exempel `xs.map(f)` eller `for (x <- xs) yield f(x)`

# Vad är en mängd?

- En **mängd** är en samling **unika** element av en viss **typ**.
- En mängd kan alltså inte innehålla dubbletter:

```
scala> Set(1,1,2,2,3,3,4,4,5,5)
val res0: Set[Int] = HashSet(5, 1, 2, 3, 4)
```

- En mängd är **inte** en sekvens: du kan inte utgå från att elementen ligger i någon viss ordning, t.ex. den ordning som de ges vid konstruktion; en mängd har ej längd, men en **storlek**; metoden `size` ger antalet element men metoden `length` saknas.
- En mängd kan vara **tom** och har då storleken `0`.
- Man kan gå igenom element i **någon** ordning (exakt vilken är ej def.), med till exempel `xs.map(f)` eller `for (x <- xs) yield f(x)`
- Det går **inte** att indexera i en mängd med `apply`, som i stället ger **innehållstest**: `Set(1,2,3).apply(3) == true`
- En mängd `Set[T]` med element av typen `T` kan således ses som ett **predikat för innehållstest**: alltså en funktion `T => Boolean` som är `true` om elementet finns annars `false`

# Oföränderlig mängd

- **Skapa**:

```
scala> var xs = Set("gurka", "tomat", "banan", "pingvin")
```

- **Läsa**: avgöra medlemskap

```
scala> xs("gurka")
val res1: Boolean = true
```

- **Uppdatera**: lägg till element (händer inget om redan finns)

```
scala> xs = xs + "jordekorre" // en ny, delvis förändrad
```

- **Ta bort**: (om finns, annars händer inget)

```
scala> xs = xs - "gurka" // en ny, delvis förändrad
```

SLUT = Skapa, Läsa, Uppdatera, Ta bort    CRUD = Create, Read, Update, Delete

# Mysteriet med de försvunna elementen

Vad händer här?

```
scala> val xs1 = Vector(1,2,3,4,5,6)
scala> xs1.map(_ % 2).count(_ == 0)
val res0: Int = 3                          // antalet jämna t
scala> val xs2 = Set(1,2,3,4,5,6)
scala> xs2.map(_ % 2).count(_ == 0)
val res1: Int = 1                          // varför?
```

# Mysteriet med de försvunna elementen

Vad händer här?

```
scala> val xs1 = Vector(1,2,3,4,5,6)
scala> xs1.map(_ % 2).count(_ == 0)
val res0: Int = 3                                      // antalet jämna t
scala> val xs2 = Set(1,2,3,4,5,6)
scala> xs2.map(_ % 2).count(_ == 0)
val res1: Int = 1                                      // varför?
```

Mängdegenskaper ger att `xs2.map(_ % 2) == Set(0, 1)`
Fundera alltid noga på om du **riskerar att förlora duplikat** som du egentligen hade velat behålla!

# Mysteriet med de försvunna elementen

Vad händer här?

```
scala> val xs1 = Vector(1,2,3,4,5,6)
scala> xs1.map(_ % 2).count(_ == 0)
val res0: Int = 3                           // antalet jämna t
scala> val xs2 = Set(1,2,3,4,5,6)
scala> xs2.map(_ % 2).count(_ == 0)
val res1: Int = 1                           // varför?
```

Mängdegenskaper ger att `xs2.map(_ % 2) == Set(0, 1)`
Fundera alltid noga på om du **riskerar att förlora duplikat** som du egentligen hade velat behålla!
Använd `toSeq` på mängd om du behöver sekvensegenskaper:

```
scala> xs2.toSeq.map(_ % 2).count(_ == 0)
val res1: Int = 3         // med toSeq blir det som vi ville
```

# Förändringsbar mängd

Med en **förändringsbar** mängd kan man stegvis utöka på plats.

```
1  scala> val mängd = scala.collection.mutable.Set.empty[Int]
2
3  scala> for i <- 1 to 1_000_000 do mängd.addOne(i)
4
5  scala> mängd.contains(-1)   // samma som mängd(-1) eller mängd.apply(-1)
```

En **mängd** är **snabb** på att avgöra om ett element **finns eller inte** i mängden. Ingen linjärsökning krävs eftersom den smarta implementationen av datastrukturen medger snabb uppslagning (eng. *lookup*) av ett element.

# Förändringsbar mängd

Med en **förändringsbar** mängd kan man stegvis utöka på plats.

```
1  scala> val mängd = scala.collection.mutable.Set.empty[Int]
2
3  scala> for i <- 1 to 1_000_000 do mängd.addOne(i)
4
5  scala> mängd.contains(-1)   // samma som mängd(-1) eller mängd.apply(-1)
```

En **mängd** är **snabb** på att avgöra om ett element **finns eller inte** i mängden. Ingen linjärsökning krävs eftersom den smarta implementationen av datastrukturen medger snabb uppslagning (eng. *lookup*) av ett element.

Men i en sekvens krävs linjärsökning vid innehållstest:

```
1  scala> val sekvens = (1 to 1_000_000).toVector
2
3  scala> sekvens.contains(-1)   // kräver linjärsökning ända till slutet
```

# Förändringsbar mängd

Med en **förändringsbar** mängd kan man stegvis utöka på plats.

```
1  scala> val mängd = scala.collection.mutable.Set.empty[Int]
2
3  scala> for i <- 1 to 1_000_000 do mängd.addOne(i)
4
5  scala> mängd.contains(-1)   // samma som mängd(-1) eller mängd.apply(-1)
```

En **mängd** är **snabb** på att avgöra om ett element **finns eller inte** i mängden. Ingen linjärsökning krävs eftersom den smarta implementationen av datastrukturen medger snabb uppslagning (eng. *lookup*) av ett element.

Men i en sekvens krävs linjärsökning vid innehållstest:

```
1  scala> val sekvens = (1 to 1_000_000).toVector
2
3  scala> sekvens.contains(-1)   // kräver linjärsökning ända till slutet
```

Övning: Testa själv att mäta tidsskillnaden med hjälp av:

```
def nanos(b: => Unit) = { val t0 = System.nanoTime; b; System.nanoTime - t0 }
```

# Speciella metoder på förändringsbar mängd

Förändringsbara mängder har metoder som ändrar på plats:

```
scala> val s = scala.collection.mutable.Set.empty[Int]

scala> s.addOne(1)     // finns även under namnet += om du gillar operator-notion
val res0: scala.collection.mutable.Set[Int] = HashSet(1)

scala> s.addOne(2).addOne(3).addOne(3).addOne(42) // addOne returnerar this
val res1: scala.collection.mutable.Set[Int] = HashSet(1, 2, 3, 42)

scala> res0.eq(res1)  // samma instans av mutable.Set (ingen ny har skapats)
val res2: Boolean = true

scala> s.addAll(Vector(3, 4, 5))  // finns även ++= om du gillar operator-notion
val res3: scala.collection.mutable.Set[Int] = HashSet(1, 2, 3, 4, 5, 42)

scala> s.subtractOne(1).subtractAll(List(1,2,3))  // finns även -= och --=
val res4: scala.collection.mutable.Set[Int] = HashSet(4, 5, 42)

scala> s.filterInPlace(_ > 4)
val res5: scala.collection.mutable.Set[Int] = HashSet(5, 42)
```

`addOne`, `addAll`, `subtractOne`, `subtractAll`, `filterInPlace` returnerar `this` så du kan ändra på plats med kedjade anrop med punktnotation.

# Nyckel-värde-tabell

# Vad är en nyckel-värde-tabell?

- En **nyckel-värde-tabell** är en samling element som är **par** med: en **nyckel** av någon typ K och ett **värde** av någon typ V.
- En sådan tabell kan skapas ur en sekvens av par `(k, v)` där k är en nyckel och v är ett värde:

```
1 scala> val ålder = Map("Björn" -> 42, "Sandra" -> 35, "Kim" -> 19)
2 val ålder: Map[String, Int] = Map(Björn -> 42, Sandra -> 35, Kim -> 19)
```

- Tabellens nycklar utgör en mängd som ges av metoden `keySet`; nycklarna är **unika**.
- Elementen utgör **inte en sekvens** och har ingen speciell ordning; en nyckel-värde-tabell har ej längd, men en **storlek**; metoden `size` ger antalet element.

# Vad är en nyckel-värde-tabell?

- En **nyckel-värde-tabell** är en samling element som är **par** med: en **nyckel** av någon typ K och ett **värde** av någon typ V.
- En sådan tabell kan skapas ur en sekvens av par `(k, v)` där k är en nyckel och v är ett värde:

```
scala> val ålder = Map("Björn" -> 42, "Sandra" -> 35, "Kim" -> 19)
val ålder: Map[String, Int] = Map(Björn -> 42, Sandra -> 35, Kim -> 19)
```

- Tabellens nycklar utgör en mängd som ges av metoden `keySet`; nycklarna är **unika**.
- Elementen utgör **inte en sekvens** och har ingen speciell ordning; en nyckel-värde-tabell har ej längd, men en **storlek**; metoden `size` ger antalet element.
- En tabell kan ses som en uppslagsfunktion (eng. *dictionary*): alltså en funktion K `=>` V som ger ett värde givet en nyckel.

# Den fantastiska nyckel-värde-tabellen `Map`

- En **nyckel-värde-tabell** (eng. *key-value table*) är en slags generaliserad vektor där man kan ”indexera” med godtycklig typ.
- Kallas även **hashtabell** (eng. *hash table*), **lexikon** (eng. *Dictionary*) eller **mapp** (eng. *Map*) (det blir lätt sammanblandning med metoden `map`).
- Om man vet nyckeln kan man slå upp värdet **snabbt**, på liknande sätt som indexering sker snabbt i en vektor givet heltalsindex.
- Denna datastruktur är **mycket användbar** och fungerar som en slags databas i kombination med filtrering, registrering, etc.

# Oföränderlig nyckel-värde-tabell

- **Skapa**: ge par till metoden `apply`

```
scala> var födelse = Map("C" -> 1972,  "C++" -> 1983, "C#" -> 2000,
  "Scala" -> 2004, "Java" -> 1995, "Javascript" -> 1995, "Python" -> 1991)
```

- **Läsa**: slå upp ett värde med hjälp av en nyckel

```
scala> val year = födelse.apply("Scala")
val year: Int = 2004
```

- **Uppdatera**: lägga till ett par, ersätta ett par

```
scala> födelse = födelse + ("Kotlin" -> 2011)
födelse: Map[String, Int] = HashMap(Scala -> 2004, C# -> 2000, Python -> 1991,
Javascript -> 1995, C -> 1972, C++ -> 1983, Kotlin -> 2011, Java -> 1995)
```

- **Ta bort** ett par via nyckeln (om finns, annars händer inget)

```
scala> födelse = födelse - "Python"
födelse: Map[String, Int] = HashMap(Scala -> 2004, C# -> 2000,
Javascript -> 1995, C -> 1972, C++ -> 1983, Kotlin -> 2011, Java -> 1995)
```

# Fler exempel nyckel-värde-tabell

Några ofta förekommande metoder på tabeller:

- `xs.keySet` ger en mängd av alla nycklar
- `xs.map(f)` kör funktionen f på alla par (key, value) i **någon** ordning
- `xs.map((k, v) => k -> f(v))` kör funktionen f på alla **värden**

```
scala> val färg = Map("gurka" -> "grön", "tomat"->"röd", "aubergine"->"lila")
val färg: Map[String, String] =
  Map(gurka -> grön, tomat -> röd, aubergine -> lila)

scala> färg("gurka")
val res0: String = grön

scala> färg.keySet
val res1: Set[String] = Set(gurka, tomat, aubergine)

scala> val ärGrönSak = färg.map((k,v) => (k, v == "grön"))
val ärGrönSak: Map[String,Boolean] =
  Map(gurka -> true, tomat -> false, aubergine -> false)

scala> val baklängesFärg = färg.map((k, v) => k -> v.reverse)
val baklängesFärg: Map[String,String] =
  Map(gurka -> nörg, tomat -> dör, aubergine -> alil)
```

# Från sekvens av par till tabell

```
scala> val xs = Vector(("Kim",42), ("Pam", 42), ("Kim", 50), ("Pam", 50))
val xs: Vector[(String, Int)] =
  Vector((Kim,42), (Pam,42), (Kim,50), (Pam,50))

scala> xs.toMap
val res0: Map[String, Int] =
  Map(Kim -> 50, Pam -> 50) // inga dublettnycklar

scala> val grupperaEfterNamn = xs.groupBy(_._1)
grupperaEfterNamn: Map[String,Vector[(String, Int)]] =
  Map(Kim -> Vector((Kim,42), (Kim,50)), Pam -> Vector((Pam,42), (Pam,50)))

scala> val grupperaEfterÅlder = xs.groupBy(_._2)
grupperaEfterÅlder: Map[Int,Vector[(String, Int)]] =
  Map(50 -> Vector((Kim,50), (Pam,50)), 42 -> Vector((Kim,42), (Pam,42)))
```

# Övning: Implementera en `Multimap`

- Om du lägger till ett värde i en *vanlig* `Map` så ersätts värdet:

```
scala> val m = Map(1 -> 2, 1 -> 3, 2 -> 1, 2 -> 2)
val m: Map[Int, Int] = Map(1 -> 3, 2 -> 2)   //senaste värdet gäller
```

  ...men ibland vill vi i stället lagra alla tillagda värden.
- En **multimap** är en speciell nyckel-värde-tabell där värdena utgör en samling (ofta en mängd).
- En multimap samlar alla värden som har samma nyckel.

```
1  scala> val mm = Multimap(1 -> 2, 1 -> 3, 2 -> 1, 2 -> 2)
2  val mm: Multimap[Int, Int] = Multimap(1 -> Set(2, 3), 2 -> Set(1, 2))
```

Övning: Implementera en multimap som fungerar som ovan, med hjälp av en case-klass med attributet `toMap` som är en oföränderlig nyckel-värde-tabell där värdena är en mängd.
Tips: Använd `groupBy`

# Lösning: Multimap

```scala
case class Multimap[K, V] private (toMap: Map[K,Set[V]]):
  def apply(k: K): Set[V] = toMap(k)

  def +(kv: (K, V)): Multimap[K, V] = kv match
    case (k, v) if toMap.isDefinedAt(k) => Multimap(toMap.updated(k, toMap(k) + v))
    case (k, v) => Multimap(toMap + (k -> Set(v)))

  override def toString = toMap.mkString("Multimap(",", ",")")

object Multimap:
  def apply[K, V](kvs: (K,V)*): Multimap[K, V] =
    new Multimap(kvs.groupBy(_._1).map((k,xs) => k -> xs.map(_._2).toSet))
```

# Speciella metoder på förändringsbar tabell

Både `Set` och `Map` finns i **förändringsbara** varianter med extra metoder för uppdatering av innehållet ”på plats” utan att nya samlingar skapas.

```
scala> import scala.collection.mutable

scala> val mm = mutable.Map.empty[String, String]
val mm: scala.collection.mutable.Map[String, String] = HashMap()

scala> mm.addOne("hej" -> "svejs")
val res0: scala.collection.mutable.Map[String, String] =
  HashMap(hej -> svejs)

scala> mm.addAll(Seq("abra" -> "kadabra", "ada" -> "lovelace"))
val res1: scala.collection.mutable.Map[String, String] =
  HashMap(hej -> svejs, abra -> kadabra, ada -> lovelace)

scala> mm("abra")
val res2: String = kadabra
```

Metoden ++= samma som addAll; används gärna med operator-notation:

```
mm ++= Seq("hej" -> "san", "abra" -> "kada", "bra" -> "scala")
```

# Tips inför veckans uppgifter

# Övning: Förändringsbar lokalt, returnera oföränderlig

Om du vill implementera en imperativ algoritm med en föränderlig samling: Gör gärna detta **lokalt** i en **förändringsbar** samling och returnera sedan en **oföränderlig** samling, genom att köra t.ex. `toSet` på en mängd, eller `toMap` på en hashtabell, eller `toVector` på en `ArrayBuffer` eller `Array`.

Exempel där lösningen har nytta av lokal förändring på plats:

```
def kastaTärningTillsAllaUtfallUtomEtt(sidor: Int = 6): (Int, Set[Int]) = ???
```

```
1  scala> kastaTärningTillsAllaUtfallUtomEtt()
2  val res0: (Int, Set[Int]) = (13,HashSet(5, 1, 6, 2, 3))
```

# Övning: Förändringsbar lokalt, returnera oföränderlig

Om du vill implementera en imperativ algoritm med en föränderlig samling: Gör gärna detta **lokalt** i en **förändringsbar** samling och returnera sedan en **oföränderlig** samling, genom att köra t.ex. `toSet` på en mängd, eller `toMap` på en hashtabell, eller `toVector` på en `ArrayBuffer` eller `Array`.

```
def kastaTärningTillsAllaUtfallUtomEtt(sidor: Int = 6): (Int, Set[Int]) =
  /*
    låt s vara en tom förändringsbar heltalsmängd
    låt n vara noll
    så länge mängden s är mindre än sidor - 1 gör:
      lägg till ett nytt tärningskast i s
      uppdatera n så att vi räknar hur många slumptal som dragits
  */
  (n, s.toSet)   // notera toSet som ger oföränderlig mängd
```

```
1 scala> kastaTärningTillsAllaUtfallUtomEtt()
2 val res0: (Int, Set[Int]) = (13,HashSet(5, 1, 6, 2, 3))
```

# Lösning: Förändringsbar lokalt, returnera oföränderlig

Om du vill implementera en imperativ algoritm med en föränderlig samling: Gör gärna detta **lokalt** i en **förändringsbar** samling och returnera sedan en **oföränderlig** samling, genom att köra t.ex. `toSet` på en mängd, eller `toMap` på en hashtabell, eller `toVector` på en `ArrayBuffer` eller `Array`.

```
def kastaTärningTillsAllaUtfallUtomEtt(sidor: Int = 6): (Int, Set[Int]) =
  val s = scala.collection.mutable.Set.empty[Int] //förändringsbar lokalt
  var n = 0
  while s.size < sidor - 1 do
    s.addOne(util.Random.nextInt(sidor) + 1)
    n += 1
  (n, s.toSet)   // notera toSet som ger oföränderlig mängd
```

```
1  scala> kastaTärningTillsAllaUtfallUtomEtt()
2  val res0: (Int, Set[Int]) = (13,HashSet(5, 1, 6, 2, 3))
```

I veckans uppgifter används detta i en s.k. **builder**: Först bygga upp en förändringsbar struktur i `FreqMapBuilder` steg för steg, och sedan, då alla tillägg är gjorda, övergå till oföränderlig struktur `Map[String, Int]`.

# Metoden `sliding`

Metoden `sliding(n)` skapar med ett "glidande fönster" en sekvens av delsekvenser av längd n genom att "svepa fönstret" från början till slut:

```
scala> val xs = "fem myror är fler än fyra elefanter".split(' ').toVector
val xs: Vector[String] = Vector(fem, myror, är, fler, än, fyra, elefanter)

scala> xs.sliding(2).toVector
val res0: Vector[Vector[String]] =
  Vector(Vector(fem, myror), Vector(myror, är), Vector(är, fler),
      Vector(fler, än), Vector(än, fyra), Vector(fyra, elefanter))

scala> xs.sliding(3).toVector
val res1: Vector[Vector[String]] =
  Vector(Vector(fem, myror, är), Vector(myror, är, fler),
    Vector(är, fler, än), Vector(fler, än, fyra),
      Vector(än, fyra, elefanter))
```

Denna metod har du nytta av på veckans laboration!

(se fler exempel på övning)

# Metoderna zipWithIndex, groupBy

```
1  scala> val kort = Vector("Knekt", "Dam", "Kung", "Äss")
2
3  scala> val kortIndex = kort.zipWithIndex.toMap
4  kortIndex: Map[String,Int] = Map(Knekt -> 0, Dam -> 1, Kung -> 2, Äss -> 3)
5
6  scala> kortIndex("Kung") > kortIndex("Knekt")
7  res0: Boolean = true
8
9  scala> kortIndex.map(p => p._1 -> (p._2 + 11))
10
11 scala> val tärningskast = Vector(1,2,3,4,5,6,2,4,6)
12
13 scala> val grupperaStörreÄnFyra = tärningskast.groupBy(_ > 4)
14 grupperaStörreÄnFyra: Map[Boolean,Vector[Int]] =
15   Map(false -> Vector(1, 2, 3, 4, 2, 4), true -> Vector(5, 6, 6))
16
17 scala> val grupperaLika = tärningskast.groupBy(x => x)
18 grupperaLika: Map[Int,Vector[Int]] = Map(5 -> Vector(5), 1 -> Vector(1),
19   6 -> Vector(6, 6), 2 -> Vector(2, 2), 3 -> Vector(3), 4 -> Vector(4, 4))
20
21 scala> val frekvens = tärningskast.groupBy(x => x).map((k,v) => k -> v.size)
22 frekvens: Map[Int,Int] = Map(5 -> 1, 1 -> 1, 6 -> 2, 2 -> 2, 3 -> 1, 4 -> 2)
```

# Fler användbara samlingsmetoder

Exempel att öva på: räkna bokstäver i ord.
Undersök vad som händer i REPL:

```
val ord = "sex laxar i en laxask sju sjösjuka sjömän"
val uppdelad = ord.split(' ').toVector
val ordlängd = uppdelad.map(_.length)
val ordlängdMap = uppdelad.map(s => (s, s.size)).toMap
val grupperaEfterFörstaBokstav = uppdelad.groupBy(s => s(0))
val bokstäver = ord.toVector.filter(_ != ' ')
val antalX = bokstäver.count(_ == 'x')
val grupperade = bokstäver.groupBy(ch => ch)
val antal = grupperade.map(p => p._1 -> p._2.size)
//samma som ovan men utnyttjar "parameter untupling":
val antal2 = grupperade.map((k,v) => k -> v.size)
val sorterat = antal.toVector.sortBy(_._2)
val vanligast = antal.maxBy(_._2)
```

# Serialisering och deserialisering

# Serialisering och deserialisering

- Att **serialisera** innebär att **koda objekt** i minnet till en avkodningsbar **sekvens av symboler**, som kan lagras t.ex. i en fil på din hårddisk.
- Att **de-serialisera** innebär att **avkoda en sekvens av symboler**, t.ex. från en fil, och **återskapa objekt** i minnet.

# Läsa text från fil och URL

I paketet `scala.io` finns singelobjektet `Source` med metoderna `fromFile` och `fromUrl` för läsning från fil resp. från URL, alltså Universal Resource Locator, som börjar t.ex. med `http://`

```
def läsFrånFil(filnamn: String): String =
  val s = scala.io.Source.fromFile(filnamn)
  try s.mkString finally s.close // säkerställ stängning även vid krasch

def läsRaderFrånFil(filnamn: String): Vector[String] =
  val s = scala.io.Source.fromFile(filnamn)
  try s.getLines.toVector finally s.close

def läsFrånWebbsida(url: String): String =
  val s = scala.io.Source.fromURL(url)
  try s.mkString finally s.close

def läsRaderWebbsida(url: String, kodning: String = "UTF-8"): Vector[String] =
  val s = scala.io.Source.fromURL(url, kodning) // läs med given teckenkodning
  try s.getLines.toVector finally s.close
```

Se vidare veckans övning. Exempel på annan teckenkodning: `"ISO-8859-1"`

# Serialisering i modulen `introprog.IO`

- I kursens kodbibliotek `introprog` finns ett singelobjekt `IO` som samlar smidiga funktioner för serialisering och de-serialisering.
- Se api-dokumentation här:
  `http://cs.lth.se/pgk/api/`
  Sök på IO och klicka på singelobjektet.
- Se koden här:
  `https://github.com/lunduniversity/introprog-scalalib/blob/master/src/main/scala/introprog/IO.scala`
- Om du vill får du gärna använda `introprog.IO` istället för `scala.io.Source` på labben.