CS-A1120 Ohjelmointi 2

Aalto-yliopisto
Perustieteiden
korkeakoulu

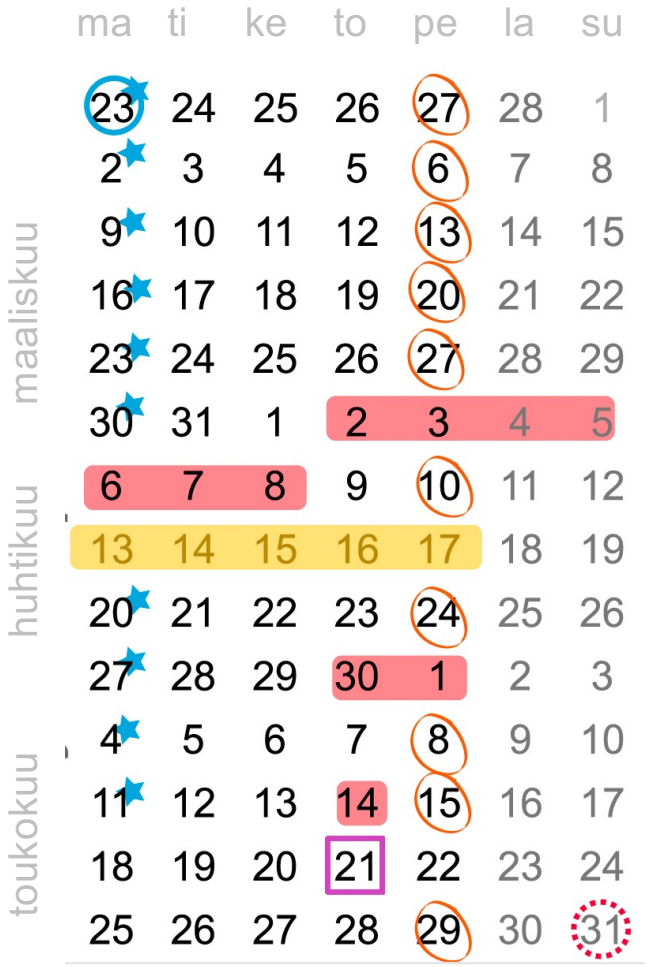
Luento 6
kevät 2026

Sanna Suoranta (suomeksi, AS2) ja
Lukas Ahrenberg (englanniksi, T1)

30.3.2026

https://presemo.aalto.fi/o2fi2026

Muistutus aikataulusta



Kierroksen 6 DL on 10.4 klo 18:00

Loma – ei opetusta

Tenttiviikko – ei opetusta

(70% DL)

Tentti – muista ilmoittautua! DL 14.5.

Seuraava luento vasta 20.4. ja se on T1:ssä

1. moduulin luentokyselytilastoja ☺

- **Kumpi on lemppari:**
  - Nolla 27% - yksi 63%
- **Lempiportti:**
  - AND 14%, OR 34%, NOT 10%,
  - XOR 17%, NAND 10%,
  - en tiedä 7%, ei mikään 7%
- **Onko kello:**
  - kiva 8%, ei kiva 31%, joskus kiva ja joskus ei 62%
- **Lemppari:**
  - 21% kirjaimet, 57%, kymmenjärjestelmän luvut 21% binääriluvut

Tänään O2:ssa

https://presemo.aalto.fi/o2fi2026

MODUULI 2

Kokoelmat ja funktiot

Aalto-yliopisto
Perustieteiden
korkeakoulu

MODUULI 2 – MOTIVAATIOKSI

- **Mitä kaipasit (ohjelmointimielessä) Assemblyä koodatessasi?**

https://presemo.aalto.fi/o2fi2026

Moduuli 2 – ohjelmoinnin abstraktiot ja analysointi

- **Olemme oppineet tietokoneen perusrakenteen, mutta**

- **Tehokas ohjelmointi vaatii abstraktioita (abstraction) algoritmeista ja datasta sekä implementoitavista ohjelmista malleja (model), jotka ovat**
  - Toistuvia (recurrent): käytännössä usein kohdattava tarve
  - Hyödyllisiä (useful) : auttaa hallitsemaan ja kapseloimaan (encapsulate) ongelman
  - Tehokkaita (efficient) tietokoneen suorittaa (ja ohjelmoijan työskentelyä ajatellen)

Korkean tason oppimistavoitteet 2. moduulille

- Kierros 6: kokoelmat ja funktiot
- Kierros 7: tehokkuus
- Kierros 8: rekursio

Tämän moduulin jälkeen

- Olet tutustunut Scala 3:n kokoelmien kehykseen (collection framework)
- Pystyt ohjelmoimaan funktionaalisella tyylillä
- Pystyt käyttämään rekursiivista ongelmanratkaisua ja –tietorakenteita
- Osaat analysoida perusfunktioiden tehokkuutta (efficiency) ja monimutkaisuutta (complexity)

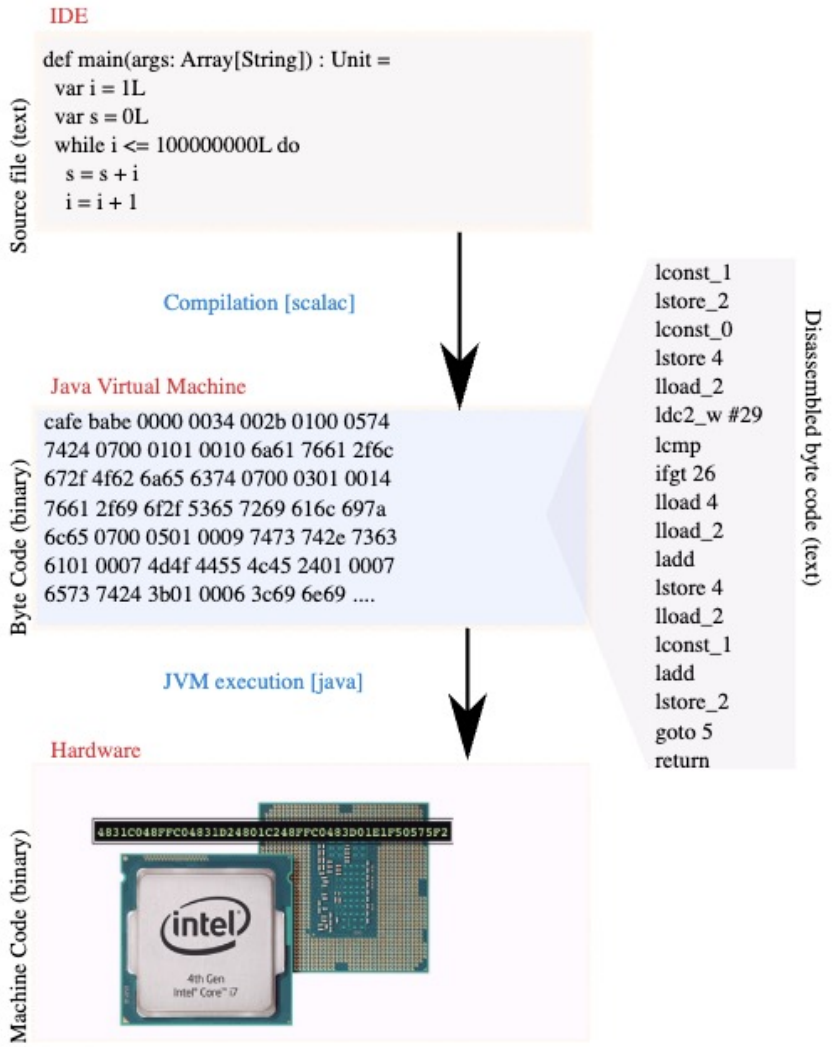
Kierroksen 6 oppimistavoitteet

Tämän kierroksen jälkeen

- **Osaat ohjelmoida funktionaalisella tyylillä**
- **Olet tutustunut nimettömien funktioiden käyttöön**
- **Osaat selittää korkeamman asteen funktioiden ominaisuudet**
- **Osaat määritellä funktion sivuvaikutuksia ja puhtaita funktioita**
- **Osaat toteuttaa Scalalla perustapauksen iteraattorista (iterator)**
- **Olet tietoinen Scalan kokoelmien ominaisuuksista**

Moduulista 1 moduulin 2: miten korkean tason kieli muutetaan konekoodiksi?

- **Prosessorille syötetään ohjelma konekoodi-binäärinä**
- **Viime kierroksella käytettiin assemblyä, ihmisluettavaa “konekoodia”**
  - Käännetään melkein 1:1:een käyttäen konekielen kääntäjää (assembler)
- **Korkean tason kielet käyttävät kääntäjiä (compilers) muuttamaan ohjelman**
  - tavukoodiksi ajettavaksi virtuaalikoneella
  - tai suoraan konekoodiksi tietylle arkkitehtuurille
  - (tai tulkkia, interpreter – melkein sama)



Abstraktio (yleistys, engl. abstraction)

- **Laitteisto toteuttaa koneiston laskentaa varten**
- **Yleismaailmallisuus (universality) ja ohjelmoitavuus (programmability) mahdollistavat ohjelmointirakenteet ja abstraktiot**
  - Meidän ei tarvitse ohjelmoida assembly-kielellä
  - Käytämme kääntäjiä (compiler) tai tulkkeja (interpreter) muuttamaan käskyt konekoodiksi
- **Tietorakenteiden abstraktio (data structure abstraction)**
  - Lisää korkean tason rakenteen muistissa oleville perustason sanoille
  - Tyypit, piirteet, rakenteet, luokat…
- **Kontrollivuon abstraktio (control flow abstraction)**
  - Mahdollistaa koodin uudelleen käytön (funktiot & aliohjelmat)
- **Ohjelmointikielet tarjoavat korkeamman tason ajattelun ja analyysin**

Tietorakenne-abstraktio

- **Erilaiset toteutukset ja tietorakenteet sopivat erilaiselle halutulle toiminnallisuudelle ja erilaisille datalle**
- **Tietorakenteet toimivat yleensä eri abstraktiotasoilla:**
  - Perustasolla, kaikki tulee tallennetuksi sekventiaaliseen muistiin sanoina (laitteistossa)
  - Loogisella/konseptuaalisella tasolla lisätään rakenne helpottaman datan kanssa työskentelyä
- **Esim. array (taulukko) voi olla aika ohut abstraktio (mutta se on silti abstraktio)**
  - Tarvitaan: alkuosoite, elementtien määrä, elementtien koko

Muuttuva((tila)inen) ja muuttumaton

- **Scala-kokoelmista löytyy molempia versioita**
  - Perusdatatyypeille vähän sama kuin `var` (muuttuja, engl. variable) ja `val` (arvo, engl. value)
- **Muuttuvaisen (mutable) kokoelman voi muuttaa**
- **Muuttumattomia (immutable) ei voi**
  - Muutosta varten luodaan uusi kokoelma
- **Scala käyttää oletuksena muuttumattomia kokoelmia (versiosta 2.13 alkaen)**
  - Eli saadaksesi muuttumattoman version, voit kirjoittaa suoraan `Seq(2,0,2,4)`
  - Poikkeus: `Array` on erityinen kokoelma ja se on aina muuttuva (vastaa Javan array:ta)

Muistutus: muuttujat ja parametrit sisältävät viittauksen objektiin, ei objektia itseään

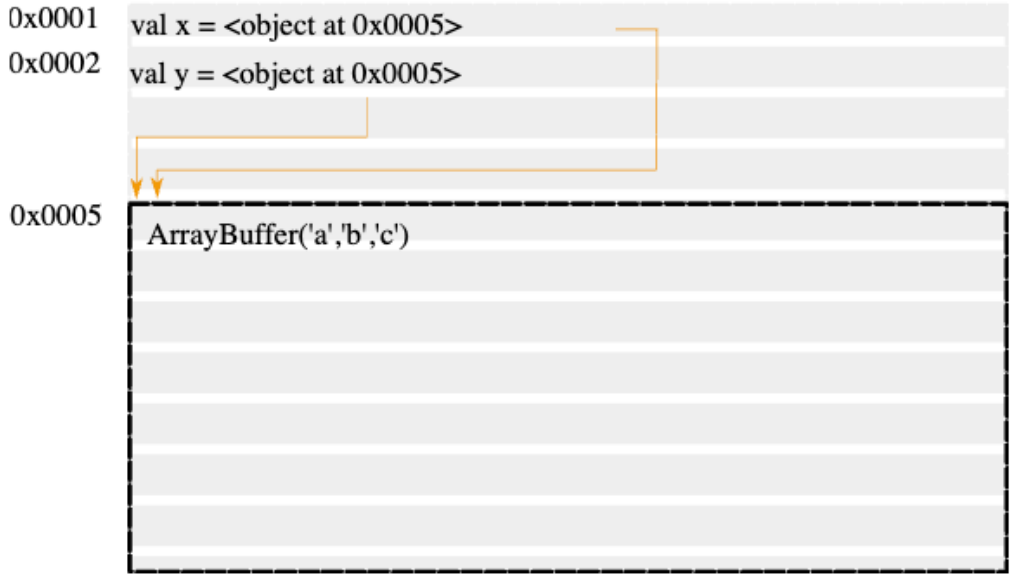
- **`val y = x` ei luo kopiota ArrayBuffer-objektista, se luo kopion ainoastaan viittauksesta (reference)**

```
import scala.collection.mutable.ArrayBuffer
val x = ArrayBuffer("a","b","c")
val y = x
println(s"x = $x")
println(s"y = $y")
x(0) = "CHANGED"
println(s"x = $x")
println(s"y = $y")
```

Tästä syystä muutokset näkyy:

```
x = ArrayBuffer(a, b, c)
y = ArrayBuffer(a, b, c)
x = ArrayBuffer(CHANGED, b, c)
y = ArrayBuffer(CHANGED, b, c)
```

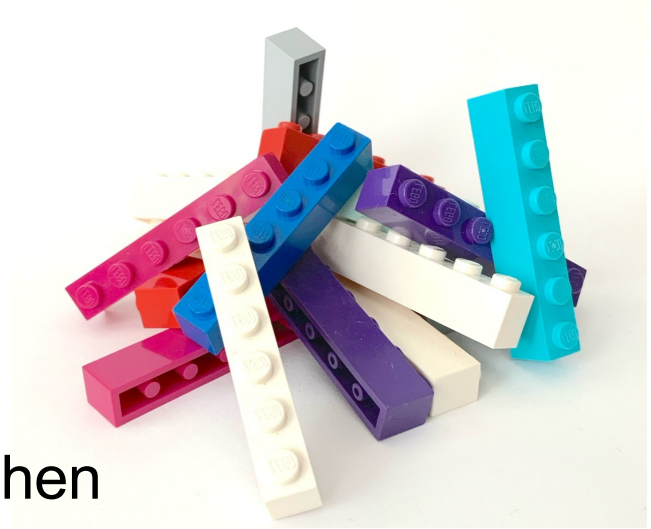
Yksinkertaistettu näkymä tietokoneen muistiin:



Keko- ja pinomuisti (heap and stack memory)

- **Käyttöjärjestelmät ja ohjelmointikielet tarjoavat abstraktioita kuinka ohjelmat käyttävät alla olevaa järjestelmämuistia**
- **Tuottaakseen esim. Dynaamisia rakenteita (dynamic structures) kuten puskureita tai alirutiineja**
- **Useimmat modernit järjestelmät jakavat muistin pinoksi (stack) ja keoksi (heap)**
  - (on myös tietorakenteita nimeltä pino ja keko, joihin palaamme myöhemmillä kursseilla)

Keko (heap)



- **Keko on (järjestyksetön) muistilohko**
- **Kun pyydämme kekomuistia esim. ArrayBufferin luomista varten Scalassa**
  - Järjestelmä varaa (allocate, reserve) osuuden keosta tätä varten ja antaa meille viittauksen siihen
  - Kun objektia ei enää tarvita, muisti voidaan vapauttaa muuhun käyttöön
  - Kekomuisti voi olla hankalaa hallita, ja lähestymistapoja on monia:
    - Scala,Java, Python,.. tekevät automaattista roskankeruuta (garbage collection)
    - Rust luottaa omistajuuteen välttääkseen roskankeruun kuormituksen
    - C-kielessä muisti pitää itse varata ja vapauttaa
- **Kekomuisti on kätevä dynaamisissa rakenteissa ja objekteissa, joita välitetään funktioiden välillä**

Pino (stack)

- **Pino on muistialue, jota hallitaan ‘vikana sisään, ekana ulos’ (last-in-first-out) –tavalla**
- **Kun uusi pinomuisti (pinokehys, stack frame) varataan**
  - Se laitetaan aina juuri edellisen palan jälkeen (päälle) pinoon
  - Mutta, kun kehyksiä voi vapauttaa vain päinvastaisesessa järjestyksessä (päällimmäinen ensin)
- **Pinomuisti on rajoittunut, mutta paljon yksinkertaisempi ja se voi olla nopeampi**
  - Ohjelmointikielet yleensä käyttää pinoallokaatiota automaattisesti esim. funktioita kutsuttaessa
  - Pinoa käytetään kun tallennetaan funktion tila ja paluuosoite

Funktionaalinen ohjelmointi

Aalto-yliopisto
Perustieteiden
korkeakoulu

Imperatiivinen ja funktionaalinen ohjelmointi

- **Scala tukee sekä imperatiivista että funktionaalista tyyliä**

Imperatiivinen ohjelmointi

- **Käyttää peräkkäisiä käskyjä**
  - Eka tämä, sitten tuo, sitten..
- **Muuttaa muuttujien arvoja (ohjelman tilaa)**
- **Käyttää muuttuvaisia tyyppejä**

```
val v = Vector(1.0, 2.5, 3.0)
var i = 0
var sumOfSquares = 0.0
while i < v.length do
  sumOfSquares += v(i)*v(i)
  i=i+1
```

Funktionaalinen ohjelmointi

- **Ilmaisee asiat funktioina**
  - Soveltaminen: $y = f(x)$
  - Kompositio: $h = g \circ f$
- **Muodostaa uusia arvoja vanhojen pohjalta**
- **Käyttää muuttumattomia tietotyyppejä**

```
val v = Vector(1.0, 2.5, 3.0)
val sumOfSquares = v.map(x=>x*x).sum
```

Funktionaalinen tyyli

- **Käytetään yleensä muuttumattomia tietotyyppejä ja `val`-muuttujia**
- **Silmukat toteutetaan rekursion avulla**
- **Luonteenomaista ovat**
  - Nimettömät funktiot
  - Korkeamman asteen funktiot
- **Funktionaalinen ohjelmointi ei vaikuta siihen, mitä voimme ohjelmoida, mutta vaikuttaa siihen, miten ohjelmoimme sen.**

```
val v = Vector(1.0, 2.5, 3.0)
// In the following,
// map is a higher-order function
// taking the anonymous function
// x=>x*x as a parameter
val squares = v.map(x=>x*x)
val sumOfSquares = squares.sum
```

Nimettömät funktiot (anonymous functions) (lambda-funktio, lambda, funktioliteraali)

- **Funktioita, joita emme vaivaudu nimeämään funktiota, ainoastaan määrittelemme sille rungon/toteutuksen (body)**
- **Miksi? Joskus funktiota tarvitaan vain kerran**

```
val v = Vector(3,4,5,2,8,7)
val evens = v.filter(x => x%2 == 0)
```

- **Rakenne: `(parametri) => (funktion runko)`**
  - Esim: `x => x%2 == 0`

Nimettömät funktiot (anonymous functions) (lambda-funktio, lambda, funktioliteraali)

- **Voi käyttää myös Scalan _-notaatiota**

```
val v = Vector(3,4,5,2,8,7)
val evens = v.filter(_%2 == 0)
```

- **Anonyymit funktiot ovat arvoja, ja funktion voi sijoittaa muuttujaan `val` ja `var` (eli 'nimetä'):**

```
val isEven : Int => Boolean = x=>x%2 == 0
val v = Vector(3,4,5,2,8,7)
val evens = v.filter(isEven)
```

Funktion tyyppi

- **Muoto A => B ilmaisee funktion tyypin, esim funktio**

```
scala> val isEven = (x: Int) => (x%2 == 0)
val isEven: Int => Boolean = Lambda$2155/0x00000008409b1040@418ee41b
```

  - on tyyppiä `Int => Boolean` eli funktio ottaa `Int`-tyyppisen arvon ja palauttaa `Boolean`-tyyppisen arvon
- **`Int => Boolean` tarkoittaa, että yllä `val`-muuttujaan on tallennettu viittaus funktioon, joka ottaa `Int`-luvun ja palauttaa `Boolean`-arvon**

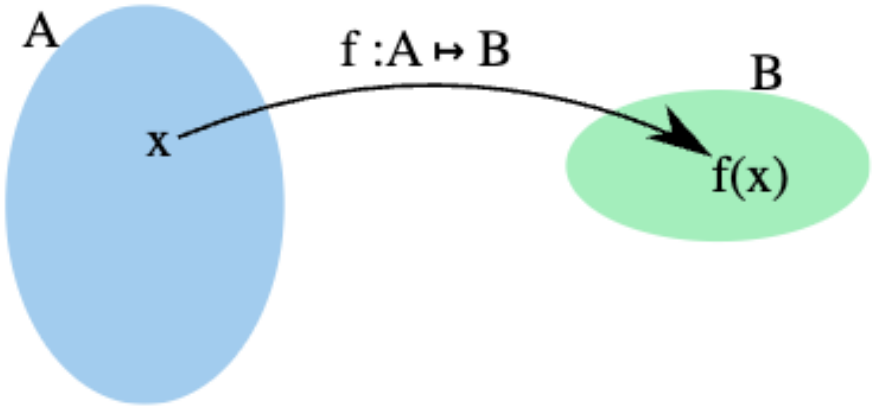
Funktion tyyppi

```
scala> val isEven = (x: Int) => (x%2 == 0)
val isEven: Int => Boolean = Lambda$2155/0x00000008409b1040@418ee41b
```

- **Vertaa kuinka matematiikassa kirjoitamme** $f:A \mapsto B$ f:A↦B **funktiosta, joka kuvaa arvoja alueesta A alueeseen B**

  Esim

  - $f: \mathbb{Z} \mapsto \{0,1\}$
  - isEven : Int ↦ Boolean



Korkeamman asteen funktiot (higher order functions)

- **Funktioita voi kohdella kuin arvoja (funktiot on ‘first class’)**
- **Korkeamman asteen funktiot**
  - ottavat muita funktioita parametrina tai
  - palauttavat uuden funktion
- **Esim. `filter`-funktiolla `Seq`:ssä**
  - on puumerkki (signature):
    ```
    def filter(pred: (A) => Boolean): Seq[A]
    ```
  - ottaa predikaattifunktion* (predicate function) `pred`, ja käyttää sitä määrittelemään mitä elementtejä filteröidään
  - Esim:
    ```
    scala> Seq(1,2,3).filter(x => x % 2 == 0)
    val res13: Seq[Int] = List(2)
    ```
  - Jossa nimetön funktio `x => x%2 == 0` on `pred`

*predikaattifunktio on funktio, joka palauttaa aina joko true tai false
(A) tarkoittaa, että A on mitä tahansa tyyppiä, sillä ei ole väliä

Funktio-objektit (function objects)

- **Mitä taikuutta tämä nyt on? Kuinka funktio (= ohjelmakoodia) voidaan antaa parametrina toiselle funktiolle?**
- **No, jos jollain parametrilla voi olla tyyppi `Int`, niin miksei voisi olla myös tyyppi `Int => Bool`**
- **Scalassa tämä toimii, koska funktiot ovat oikeasti objekteja, ja objektin voi lähettää parametrina**

```
object isEven extends Function1[Int,Boolean]:
  def apply(x: Int) : Boolean = (x % 2 == 0)
end isEven
val v = Vector(3,4,5,2,8,7)
val evens = v.filter(isEven)
```

Piirre `Function1` on yksiargumenttisille funktioille

- **Rakenne `val isEven : Int => Boolean = x=>x%2 == 0` on syntaktista sokeria (eli helpottaa koodin kirjoittamista)**

Kokoelmien käyttö funktionaalisessa tyylissä

- **Scala.collection tarjoaa monia hyödyllisiä metodeja:**
  - Forall, foreach, map, flatMap, groupBy, foldLeft, zip
  - https://a1120.cs.aalto.fi/2026/notes/round-functions--using.html
- **Harjoituksia varten voi olla hyödyllistä tutkia Scalan dokumentaatiota, esim.**
  - piirteet `Seq` ja `Map` (mitä metodeita näillä on)
  - metodit kuten `splitAs`, `keys`, `filter`, `toSet`, `toMap`, `toVector`, `indexWhere`, `take`, `min`, ja `max` (milloin ovat käytettävissä)
- **Jos kaipaat kertausta, CS-A1110 Ohjelmointi 1 kierros 6:**
  - https://plus.cs.aalto.fi/o1/2025/w06/ch03/

Sivuvaikutukset ja puhtaat funktiot

Aalto-yliopisto
Perustieteiden
korkeakoulu

Sivuvaikutukset (side effects)

- **Funktiolla on sivuvaikutuksia, jos se tekee jotain muuta kuin ottaa arvon ja palauttaa (jotenkin lasketun) tuloksen**
- **Sivuvaikutuksia ovat esim**
  - Funktio aiheuttaa jonkin havaittavan vuorovaikutuksen ympäristössään, esim. tulostaa jotain, kirjoittaa tiedostoon, odottaa käyttäjältä syötettä, tai aiheuttaa poikkeuksen
  - Se muuttaa jonkin ulkoisesti näkyvän objektin (tai globaalin muuttujan) tilaa
  - Se kutsuu jotain muuta metodia/funktiota, jolla on sivuvaikutuksia
- **Sitä, että suoritus vie aikaa ja käyttää muistia, ei lasketa sivuvaikutukseksi**

Sivuvaikutukset, esim

- **Näillä ei ole sivuvaikutuksia:**
- **A:**

```
def f(a: Int, b: Int): Int = a+b
```

- **B:**

```
var x = 0
x = if v > x then v else x
```

- **Miksi?**

- **Näillä on sivuvaikutuksia**
- **C:**

```
def g(a: Int, b: Int): Int =
  println(s"a = $a, b = $b")
  a+b
```

- **D:**

```
var x = 0
def h(v: Int): Int =
  if v > x then
    x = v
  x
```

Millä seuraavista on sivuvaikutuksia?

A)

```
def printDivisibleBy3(S : Seq[Int]) : Unit =
  S.filter(x=> (x % 3) == 0).foreach(x=>println(x))
```

B)

```
def cSum(A : Array[Int]) : Array[Int] =
  val B : Array[Int] = A
  var i = 1
  while i < A.length do
    B(i) = B(i-1) + A(i)
    i += 1
  B
```

C)

```
def zipWithReverse(S : Seq[Int]) : Seq[(Int,Int)] =
  S.zip(S.reverse)
```

D)

```
import scala.util.Random
def addNoise(S : Seq[Double]) : Seq[Double] =
  S.map(x => x + Random.nextGaussian())
```

Puhtaat funktiot (pure functions)

Funktio on puhdas (pure), jos

1. **Sillä ei ole sivuvaikutuksia ja**
2. **Kun se jokaisella kutsukerralla palauttaa saman arvon, kun parametrit ovat samat**

Mitkä näistä on puhtaita?

```
def f(a: Int, b: Int) =
   if a > b then a else b
```

```
println
```

```
Random.nextGaussian
```

```
java.time.LocalDateTime.now
```

Sulkeuma

Aalto-yliopisto
Perustieteiden
korkeakoulu

Sulkeuma (closure)

- **Ohessa (hidas) testi onko luku alkuluku:**

```
def isPrimeSlow(n: Int) : Boolean =
  require(n > 1)
  (2 until n).forall(divisor => n % divisor != 0)
```

- **Nimettömällä funktiolla `divisor => n % divisor != 0` ei ole parametria `n`**
- **`n` on vapaa muuttuja (free variable) tämän nimettömän function sulkeumassa**
  - Huom! Vapaat muuttujat on sidottu muuttujiin, ei muuttujien arvoihin, kun funktio luodaan
- **Funktion sulkeuma sisältää funktion itsensä ja sen viittausympäristön (referencing environment, scope)**
- **Funktio voidaan antaa argumenttina toiselle funktiolle, ja se voi silti vielä käyttää muuttujia ympäristöstä, jossa se luotiin**

Sulkeuma (closure)

- **Ohessa (hidas) testi onko luku alkuluku:**

```
def isPrimeSlow(n: Int) : Boolean =
  require(n > 1)
  (2 until n).forall(divisor => n % divisor != 0)
```

- **Nimettömällä funktiolla `divisor => n % divisor != 0` ei ole parametria `n`**
- **`n` on vapaa muuttuja (free variable) tämän nimettömän function sulkeumassa**
  - Huom! Vapaat muuttujat on sidottu muuttujiin, ei muuttujien arvoihin, kun funktio luodaan
- **Funktion sulkeuma sisältää funktion itsensä ja sen viittausympäristön (referencing environment, scope)**
- **Funktio voidaan antaa argumenttina toiselle funktiolle, ja se voi silti vielä käyttää muuttujia ympäristöstä, jossa se luotiin**

Esim. Viestin prefix (älä tee näin!)

- **Oletetaan, että haluamme tulostaa merkkijonon mutta lisätä rivinumeron ja prefiksin**
- **Esim. Prefix “A>” ja rivinumero 1, pitäisi funktion `prefixPrintln(“Hello”)` tuottaa `[1]A>Hello` ja uudestaan kutsuttuna `[2]A>Hello`**

```
// These var:s exist in the global scope
// [for the sake of demo, in general a bad idea]
var prefix = "A>"
var lines = 0
// println on the format [<lines>]<prefix><s>
def prefixPrintln(s: String) =
  lines +=1
  println(s"[$lines]$prefix$s")
```

```
scala> prefixPrintln("Hello")
[1]A>Hello

scala> prefixPrintln("World")
[2]A>World
```

Esim. Viestin prefix (älä tee näin!)

- **Globaali `var` on huono idea!**
- **`Var` globaalissa viittausympäristössä:**

```
scala> prefix = "B>"
prefix: String = B>

scala> lines = 4324324
lines: Int = 4324324

scala> prefixPrintln("Test")
[4324325]B>Test
```

- **Jos päivittää `prefix` tai `lines`-muuttujaa, muuttuu koko funktion käytös**

```
// These var:s exist in the global scope
// [for the sake of demo, in general a bad idea]
var prefix = "A>"
var lines = 0
// println on the format [<lines>]<prefix><s>
def prefixPrintln(s: String) =
  lines +=1
  println(s"[$lines]$prefix$s")
```

Esim. Objektin käyttö enkapsuloinnissa

- **Olio-ohjelmoinnin tapa ratkaista edellä ollut ongelma on luokka, jossa**
  - `prefix` annetaan parametrina ja
  - `lines` on enkapsuloitu
- **Toimii ihan hyvin, mutta sama voidaan saada aikaan käyttäen vain funktioita**

```
class PrefixPrinter(prefix: String):
  private var lines = 0L
  // apply method lets us call objects using ()
  def apply(s: String) =
    lines += 1
    println(s"[$lines]$prefix$s")
```

```
scala> val prefixPrintln =
PrefixPrinter("A>")
val prefixPrintln: PrefixPrinter =
  PrefixPrinter@7be7c052

scala> prefixPrintln("Hello")
[1]A>Hello

scala> prefixPrintln("World")
[2]A>World
```

Esim. Käyttäen sulkeumaa ja funktiota

- **Lines ja prefix voivat kuulua sellaisen funktion sulkeumaan, jonka palautamme tehdasfunktiosta (factory function)**

```
// Returns a println like function that prefixes everything
// with [n]`prefix`, where n is the line number,
// starting at 1
def createPrefixPrint(prefix: String): String => Unit =
  var lines = 0
  s =>                                  // s is the parameter
       lines+=1                         // first increase
       println(s"[$lines]$prefix$s") // then print

scala> val prefixPrintln = createPrefixPrint("A>")
val prefixPrintln: String => Unit = Lambda$1682/0x0000008408350400@25e796fe

scala> prefixPrintln("Hello")
[1]A>Hello

scala> prefixPrintln("World")
[2]A>World
```

Tämä palauttaa anonyymin funktion!

Esim. Käyttäen sulkeumia

- **Lines ja prefix voivat kuulua sellaisen funktion sulkeumaan, jonka palautamme tehdasfunktiosta (factory function)**

```
def createPrefixPrint(prefix: String): String => Unit =
  var lines = 0
  s =>                                  // s is the parameter
      lines+=1                          // first increase
      println(s"[$lines]$prefix$s") // then print
```

- **Tehtaan `createPrefixPrint` paluutyyppi on funktio `String => Unit`**
  - Eli palautettu arvo on funktio, joka ottaa parametriksi `String`-tyyppisen arvon ja palauttaa `Unit` (eli ei mitään)
  - Miksi `Unit`? No, se on `println`:n paluuarvon tyyppi
- **Palautettu arvo on nimetön funktio, joka alkaa `s =>`**

Useat parametri-luettelot ja osittainen soveltaminen

Aalto-yliopisto
Perustieteiden
korkeakoulu

Useat parametriluettelot (multiple parameter lists)

- **Yleensä, Scala-funktioille annetaan useampi parametri monikkona (tuple), esim:**

```
def sumTwo(a:Int, b:Int) : Int = a + b
```

- **Mutta, voimme antaa myös useita parametriluetteloita funktiolle, esim:**

```
def sumTwo(a:Int)(b:Int) : Int = a + b
```

- **Olet käyttänyt tätä jo, esim. foldLeft-metodi, joka löytyy Scalan iteroitavista kokoelmista.**
- **Matemaattisesti:**
  - `def sumTwo(a:Int, b:Int): Int` vastaa sumTwo:$Int \times Int \mapsto Int$
  - `def sumTwo(a:Int)(b:Int): Int` vastaa sumTwo:$Int \mapsto (Int \mapsto Int)$

Currying (hyvä tietää)

- **Vaihtamista useammasta parametrista sekvenssiin funktioita kutsutaan currying-nimellä (Haskell Curryn mukaan)**
  - Termi voi sekoittaa: useampi parametriluettelo toteutetaan toisin Scalassa
  - On olemassa metodi nimeltä curried, joka muuttaa tällaisen funktion sekvenssiksi lambda-funktioita
  - Tällaisen funktiosekvenssin kutsuminen näyttää samalta kuin useamman parametrilistan funktion kutsu

```
scala> def sumTwo(a:Int, b:Int): Int = a + b
def sumTwo(a: Int, b: Int): Int

scala> val sumTwoCurry = sumTwo.curried
val sumTwoCurry: Int => Int => Int =
  scala.Function2$$Lambda$1815/0x0000000840899840@225ac5e7

scala> sumTwoCurry(10)(11)
val res0: Int = 21
```

Funktion osittainen suorittaminen (partial application)

- **Useat parametriluettelot mahdollistaa suuremman joustavuuden, erityisesti funktionaalista tyyliä käytettäessä**
- **Voimme käyttää sitä osittaiseen suorittamiseen, jossa funktiota kutsutaan ensin osalla sen parametreistä, jolloin se palauttaa uuden funktion, sen sijaan että palauttaisi arvon**
- **Esim `def sumTwo` edellä, voimme luoda funktion sitomalla ensimmäisen argumentin johonkin arvoon (kuin b: Int => 5 + b):**

```
scala> val fivePlus = sumTwo(5)
val fivePlus: Int => Int = $Lambda$1348/0x00000008406e6840@5381cdb6
```

- **Saadaan nimetön funktio, jonka laitoimme muuttujaan**

```
scala> fivePlus(3)
val res: Int = 8
```

```
scala> fivePlus(7)
val res: Int = 12
```

Funktion osittainen soveltaminen (jatkuu)

- **Periaatteessa olisimme voineet tehdä funktion**

```
def fivePlus(b: Int): Int = 5 + b
```

  - Ihan sama olisi ollut (koska `def` on vähän eri kuin `val`)

```
val fivePlus: Int => Int = b => 5 + b
```

  - Mutta kun meillä oli jo funktio, jota voi käyttää myös tähän yksittäistapaukseen, niin meidän ei tarvitse

Osittainen suorittaminen -quiz

- **Oletetaan, että meillä on seuraava merkkijonojen yhdistämiseen**

```
def conc(x: String)(y: String): String = x + y
val partial = conc("A")
```

1. **Mikä on `partial`in tyyppi?**
2. **Mitä saadaan tulokseksi, kun ajetaan** `partial("B")`?
3. **Mitä saadaan tulokseksi, kun ajetaan** `partial("C") + partial("B")`?

https://presemo.aalto.fi/o2fi2026

Funktion kutsu ja argumentit

Aalto-yliopisto
Perustieteiden
korkeakoulu

Eri tapoja argumenttien evaluointiin

- **Monissa kielissä (ja Scalassa) on useampi tapa evaluoida funktiolle annetut argumentit:**
  - Call-by-value: parametrit evaluoidaan hanakasti eli heti (eager) ja väistämättä joka kutsukerralle eli tiukasti (strict) ennen kuin funktio suoritetaan (oletuksena Scalassa näin)
  - Call-by-name; parametrit evaluoidaan väljästi (non-strictly) eli ne evaluoidaan vasta, jos niitä tarvitaan (ei vielä laiskaa)
  - Call-by-need: parametrit evaluoidaan paitsi väljästi (edellinen tapaus) myös korkeintaan kerran eli laiskasti (lazy)
- **Call-by-name käytetään, kun parametrien laskenta on kallista (ja funktio käyttää sitä vaan joskus), tai**
- **Kun haluamme tehdä funktiolohkoja, jotka funktio suorittaa**

Call-by-name eli väljä evaluointi

- **Parametrin voi merkitä väljästi evaluoitavaksi (call-by-name) käyttämällä => ensimmäisenä tyypin määrittelyssä**
- **Esim. Toinen funktio joka tulostaa prefixin merkkijonoon käyttäen useampaa parametrilistaa, yhtä prefixille ja toista viestille. Tehdään vielä prefixistä call-by-name-tyyppinen:**

```
// prefix is call-by-name, and will be evaluated when needed
// (when it is printed)
// msg is call-by-value and stays the same
// as when when logMessage is called
def logMessage(prefix: => String)(msg: String): Unit =
  println(s"$prefix$msg")
```

  - Prefixillä on nyt tyyppi `=> String`

Call-by-name vs. Call-by-value

Call-by-name: `prefix: => String`

```scala
// Import now, which gives the current time
import java.time.LocalDateTime.now

// Log message function, note that prefix is call-by-name
def logMessage(prefix: => String)(msg: String): Unit =
  println(s"$prefix$msg")

// Create a log function that prefix everything
// with the current time
val logThis = logMessage(now().toString + ": ")

// Use it to log a few things
logThis("Captain's log")
logThis("Star-date...")
logThis("I can't remember!")
```

Output:

```
2025-03-29T16:49:40.486410873: Captain's log
2025-03-29T16:49:40.486757394: Star-date...
2025-03-29T16:49:40.486890486: I can't remember!
```

- **`prefix` evaluoidaan joka kerta kun `println` tarvitsee sitä**

Call-by-value: `prefix: String`

```scala
// Import now, which gives the current time
import java.time.LocalDateTime.now

// Log message function, note that prefix is call-by-value
def logMessage(prefix: String)(msg: String): Unit =
  println(s"$prefix$msg")

// Create a log function that prefix everything
// with the current time
val logThis = logMessage(now().toString + ": ")

// Use it to log a few things
logThis("Captain's log")
logThis("Star-date...")
logThis("I can't remember!")
```

Output:

```
2025-03-29T16:55:24.388120025: Captain's log
2025-03-29T16:55:24.388120025: Star-date...
2025-03-29T16:55:24.388120025: I can't remember!
```

- **`prefix` evaluoidaan kun `logThis` luodaan**

Call-by-name vs. Call-by-value

Call-by-name: `prefix: => String`

```scala
// Import now, which gives the current time
import java.time.LocalDateTime.now

// Log message function, note that prefix is call-by-name
def logMessage(prefix: => String)(msg: String): Unit =
  println(s"$prefix$msg")

// Create a log function that prefix everything
// with the current time
val logThis = logMessage(now().toString + ": ")

// Use it to log a few things
logThis("Captain's log")
logThis("Star-date...")
logThis("I can't remember!")
```

Output:

```
2025-03-29T16:49:40.486410873: Captain's log
2025-03-29T16:49:40.486757394: Star-date...
2025-03-29T16:49:40.486890486: I can't remember!
```

- **`prefix` evaluoidaan joka kerta kun `println` tarvitsee sitä**

Call-by-value: `prefix: String`

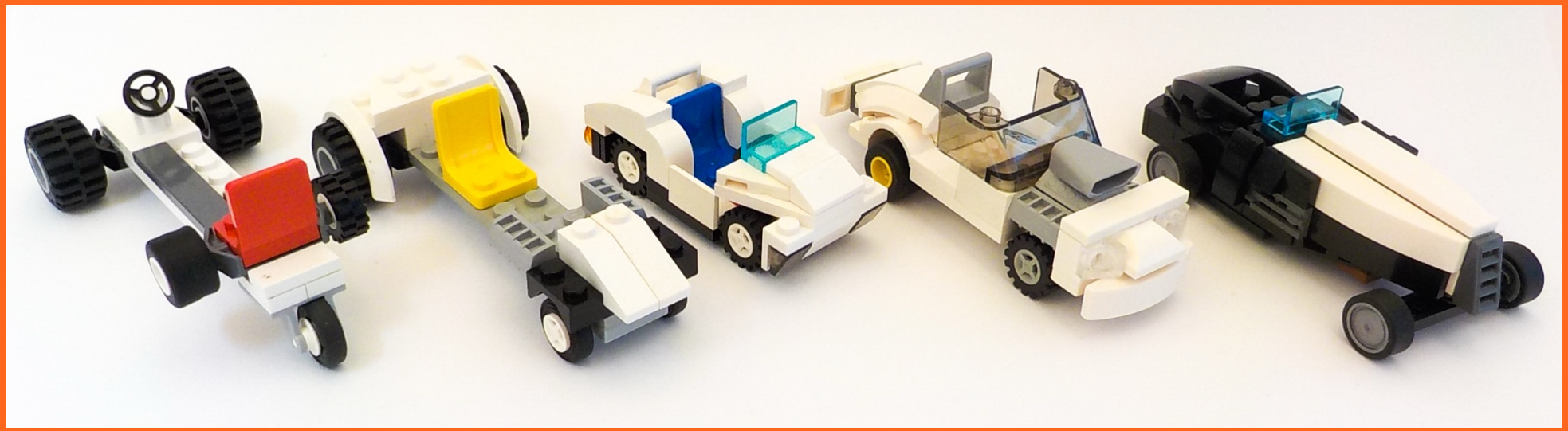
```scala
// Import now, which gives the current time
import java.time.LocalDateTime.now

// Log message function, note that prefix is call-by-value
def logMessage(prefix: String)(msg: String): Unit =
  println(s"$prefix$msg")

// Create a log function that prefix everything
// with the current time
val logThis = logMessage(now().toString + ": ")

// Use it to log a few things
logThis("Captain's log")
logThis("Star-date...")
logThis("I can't remember!")
```

Output:

```
2025-03-29T16:55:24.388120025: Captain's log
2025-03-29T16:55:24.388120025: Star-date...
2025-03-29T16:55:24.388120025: I can't remember!
```

- **`prefix` evaluoidaan kun `logThis` luodaan**

Iteraattorit



Aalto-yliopisto
Perustieteiden
korkeakoulu

Iteraattorit (iterators)

- **Iteraattorit ovat muuttuvia objekteja, jotka voi käydä läpi elementtejä yksi kerrallaan**
  - Niillä on sivuvaikutuksia
- **Ne voivat jäljitellä kokoelmien käytöstä ja niillä on samoja metodeita, mutta ne ei tallenna kaikkia elementtejä**
- **Iteraattorit ovat käteviä, kun haluamme**
  - käydä läpi kokoelman elementit
  - luoda sekvenssi tallentamatta kaikkia arvoja

```
scala> (1L to 100000000L).filter(_%2 ==0).sum
java.lang.OutOfMemoryError: Java heap space

scala> (1L to 100000000L).iterator.filter(_%2 ==0).sum
val res4: Long = 2500000050000000
```

`Iterator`

- **Iterator on muuttuva(tila)inen objekti**
- **Sillä on kaksi perusoperaatiota:**
  - `Next(): A` palauttaa iteraattorista seuraavan arvon
  - `hasNext: Boolean` kertoo, onko elementtejä vielä lisää
- **Voit käyttää `iterator`-metodia Scala-kokoelmasta saadaksesi iteraattorin**

Iteraattorit

- **Iteraattorin tekeminen ja seuraavan kysyminen eli next()**

```
val s = Seq("Hey","You") // Make sequence
val it = s.iterator // Get iterator
// Access one by one
println(it.next()) // "Hey"
println(it.next()) // "You"
println(it.next()) // This thrown an Exception (java.util.NoSuchElementException)
```

- **Käydään läpi niin kauan kuin `hasNext` on tosi**

```
val s = Seq("Hey","You") // Make sequence
val it = s.iterator // Get iterator
// Access one by one, as long as there are more
while it.hasNext do
  println(it.next())
```

- **Voi käyttää tavalliseen tapaan**

```
val s = Seq(1,2,3,4)
// Sum all even numbers
s.iterator.filter(_%2 == 0).sum
```

Iteraattorit

- **Miksi tästä pitäisi välittää?**

```
// Sum all even numbers in range
val r =(1L to 100000000L)
// The next will give java.lang.OutOfMemoryError
r.filter(_%2 ==0).sum
```

Note: java.lang.OutOfMemoryError: Java heap space

- **Sama iteraattorin avulla:**

```
// Sum all even numbers in range
val r =(1L to 100000000L)
r.iterator.filter(_%2 ==0).sum
// Will give 2500000050000000
```

Oman iteraattorin tekeminen

- **Vaikka yleensä käytämme olemassaolevien kokoelmien iteraattoreita, joskus haluamme tehdä oman**
  - Esim jos teemme uudenlaisen tietorakenteen, tai
  - Haluamme luoda (generate) mukautetun sarjan dataa tallentamatta sitä ensin
- **Scalassa iteraattori perii `Iterator`-piirteen ja sen pitää toteuttaa kaksi metodia:**
  - `hasNext: Boolean` kertoo, onko seuraava elementti tarjolla
  - `next(): A` palauttaa seuraavan vierailemattoman elementin, huomaathan eksplisiittisen (), jota käytetään Scalassa metodeille, jotka ei ota parametreja mutta joilla on sivuvaikutuksia

Iteraattori-esimerkki

- **Tehdään iteraattori, joka laskee sekvenssin ‘orbit’, jolla on seuraavat säännöt:**
- **Se ottaa positiivisen kokonaisluvun**
- **ja laskee, jos numero on:**
  - parillinen: jaa kahdella
  - pariton: kerro kolmella ja lisää yksi
- **Tämä on ‘Collatz conjencture’:**
  - Oletus: joskus päästään ykköseen

- **Formaalimmin: jollakin luvulla** $x_0 \in \mathbb{Z}^+$
- **Määritellään**

$$x_{n+1} = \begin{cases} x_n/2 & \text{jos } x_n \text{ on parillinen} \\ 3x_n + 1 & \text{jos } x_n \text{ on pariton} \end{cases}$$

- **Esim. X_0 = 5 antaa lukujonon 5,16,8,4,2**

```
scala> orbit(300).length
val res22: Int = 16

scala> orbit(299).length
val res23: Int = 117
```

https://en.wikipedia.org/wiki/Collatz_conjecture

Iteraattori-esimerkki

```
def orbitIterator(start: BigInt) : Iterator[BigInt] =
  require(start>0, "Must start from a positive value")
  // new is used to create a new object
  new Iterator[BigInt]:
    //`x` is the current value, initially `start`
    private var x = start
    // Our implementation of `hasNext`
    // Orbit ends if we reach 1
    def hasNext: Boolean = (x != 1)

    // Our impmenentation of `next()`
    // Calculated according to rule
    def next(): BigInt =
    // Temporarily hold the current value
    val result = x
    // Update current number
    x = if x%2 == 0 then x/2 else 3*x+1
    // Return previous value
    result
```

https://scastie.scala-lang.org/e8XvjgN6R2m5tlMZeM1UhQ

- **Iteraattori, joka laskee:**
  - parillinen: jaa kahdella
  - pariton: kerro kolmella ja lisää yksi

```
// Let's test:
val o5 = orbitIterator(5)
while o5.hasNext do
  println(o5.next())
// Or the compactly with for
for x <- orbitIterator(5) do
  println(x)

// do not print last 1
5
16
8
4
2
```

Iteraattori-quiz

- **Oletetaan, että määritellään funktio, joka luo iteraattorin bittien läpikäymiseen tavussa `Byte`:**

```
def bitsIt(w: Byte): Iterator[Boolean] =
  new Iterator[Boolean]:
    private var i: Int = 8
    def hasNext: Boolean = (i > 0)
    def next(): Boolean =
      i = i - 1
      (((w>>>i) & 0x1) == 0x1)
```

- **Jos käytämme `bitsIt` näin:**

```
val w: Byte = 7
var n = 0
for b <- bitsIt(w) do
  if b then n+=1
```

- **Mikä on arvo `n`, kun silmukka lopettaa?**

Uudelleen-käytettävät kokoelmat

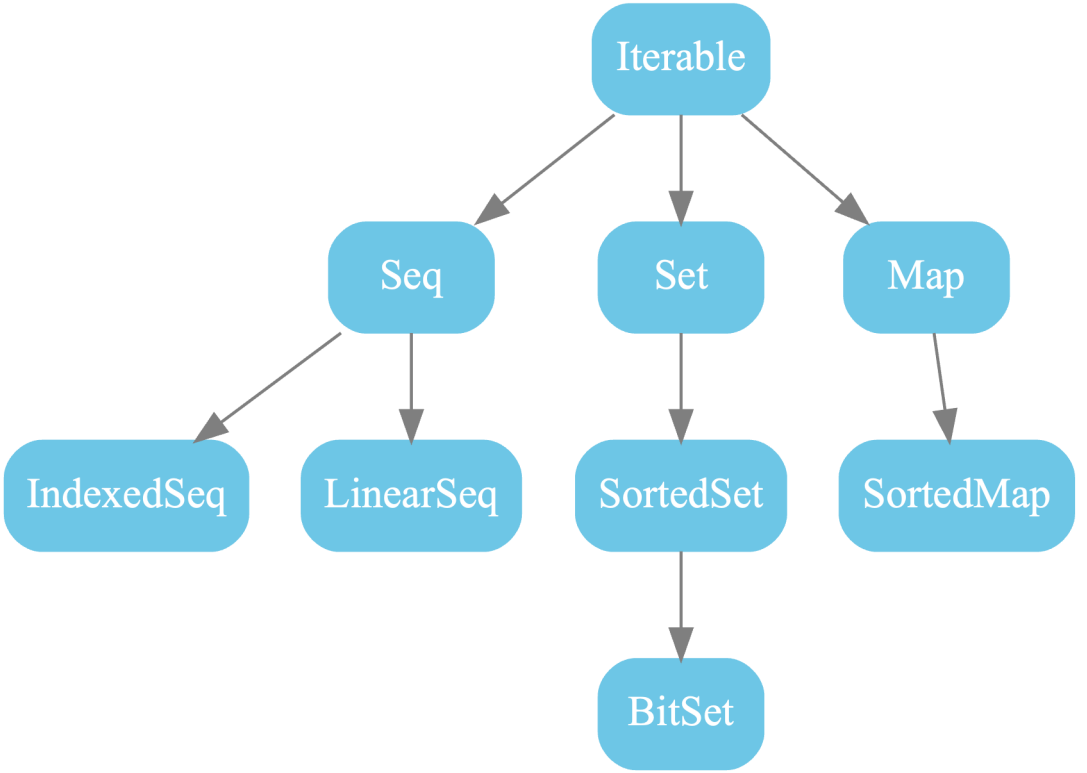
Aalto-yliopisto
Perustieteiden
korkeakoulu

Uudelleenkäytettävät kokoelmat

- **Yleisiä hahmoja ja abstraktioita**
  - Sarjat (sekvenssit, järjestykset, engl. sequences)
  - Joukot (engl. sets)
  - Hakurakenteet (engl. maps, associative arrays)
- **löytyvät monien modernien ohjelmointikielten peruskirjastosta (standard library)**
  - Hyödylliset abstraktiot ja menetelmät, sekä usein käytettävät funktiot

Scalan kokoelmat

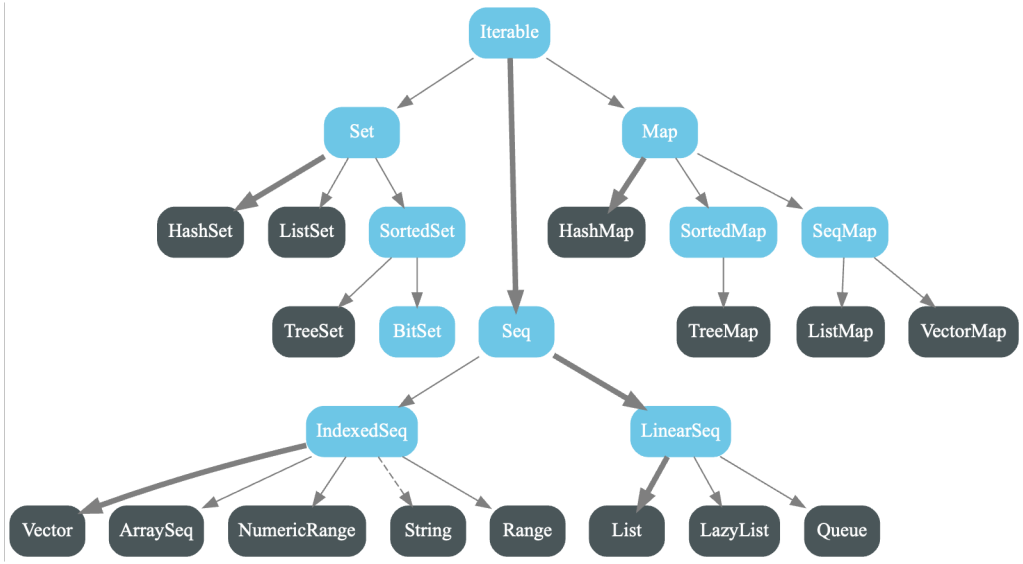
- **Yleinen ja yhteinen kehys datan kanssa työskentelyyn**



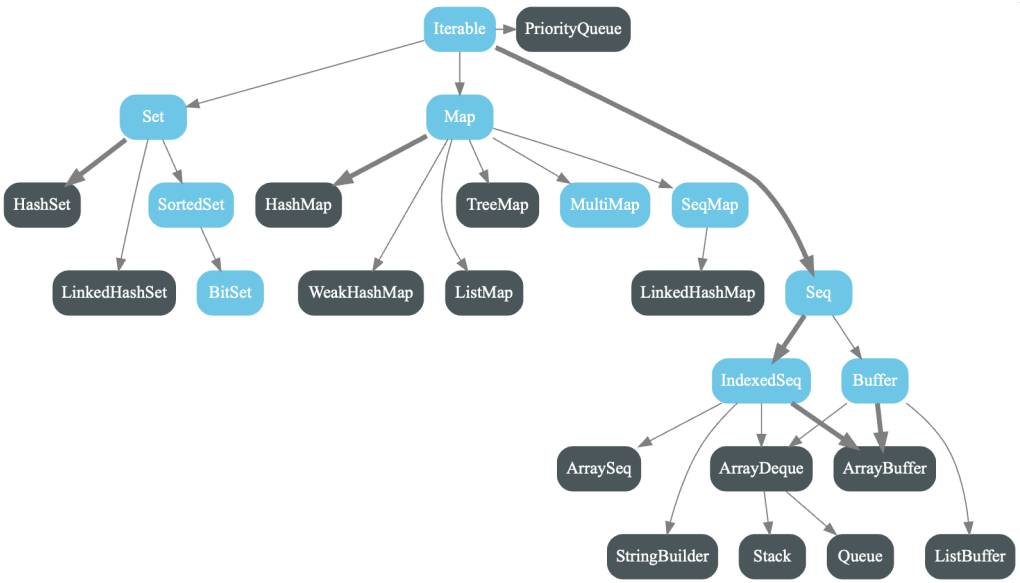
Kuva:
https://docs.scala-lang.org/scala3/book/collections-classes.html

Oikeastaan kaksi tyyppiä

Muuttumattomat (immutable)



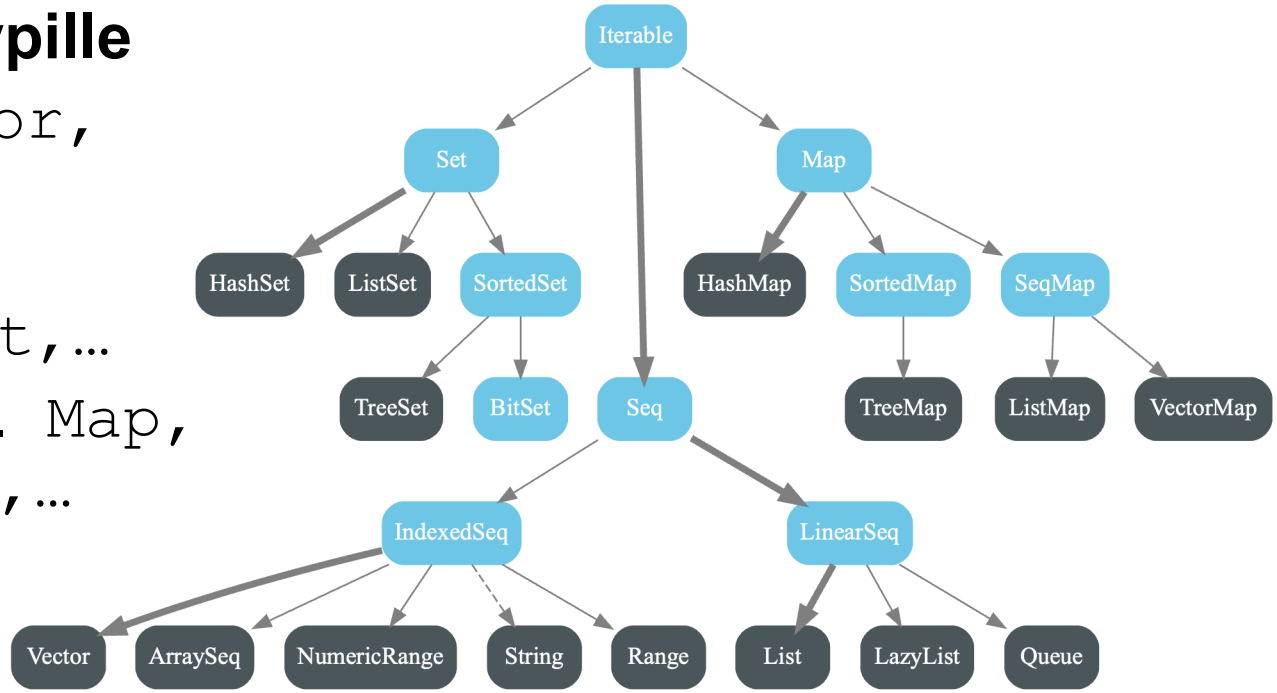
Muuttuvainen (mutable)



Kuva:
https://docs.scala-lang.org/scala3/book/collections-classes.html

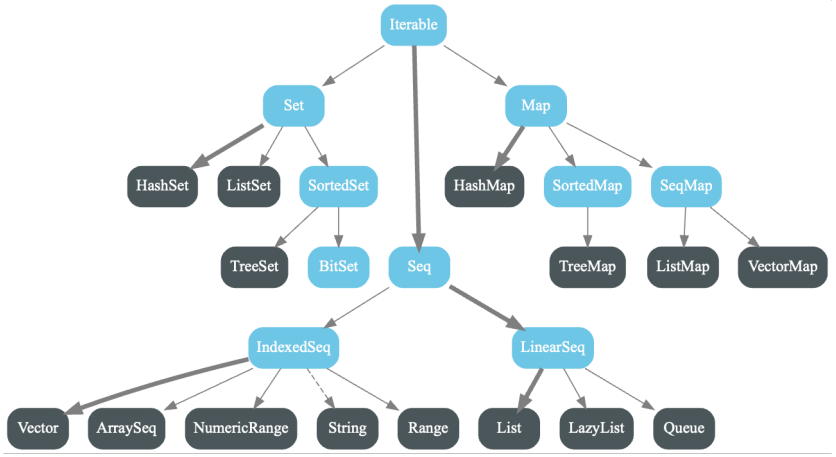
Niin monia vaihtoehtoja…

- **Sekä muuttuvaisia että muuttumattomia versioita**
- **Monia luokkia ja piirteitä jokaiselle kokoelmatyypille**
  - Sarjoille: esim. `Vector`, `List`, `ArraySeq`,…
  - Joukoille: esim. `Set`, `SortedSet`, `BitSet`,…
  - Hakurakenteille: esim. `Map`, `HashMap`, `ListMap`,…
- **Ihmetellään hetki tätä muuttumattomien haaraa:**



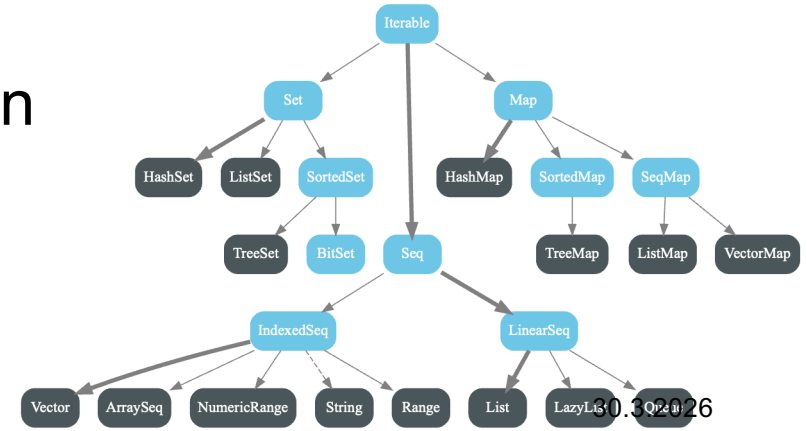
Kaikki tekee saman ainakin summan laskemisen tapauksessa, mutta mikä on ero?

```
scala> import scala.collection.immutable._
import scala.collection.immutable._

scala> Vector(1,2,3,4).sum
val res1: Int = 10

scala> List(1,2,3,4).sum
val res2: Int = 10

scala> ArraySeq(1,2,3,4).sum
val res4: Int = 10
```



Piirteet ja luokat

- **Korkean tason piirteet (trait) kuvaavat abstraktin rajapinnan (interface)**
- **Rajapinta on toteutettu konkreettisessa luokassa (concrete class)**
  - Esim. Map on abstrakti piirre, kun taas HashMap ja ListMap on kaksi eri konkreettista luokkaa sen toteuttamiseksi
  - Toteuttavat saman rajapinnan, mutta eroavat toisistaan taustalla olevalta tietorakenteiden toteutustavan osalta
- **Useimmilla piirteillä on kumppaniolio, joka mahdollistaa ilmentymän tekemisen**
  - Kumppaniolio valitsee konkreettisen olion



Piirteet ja luokat

- **Esimerkiksi**

```
scala> import scala.collection.immutable._
import scala.collection.immutable.*

scala> val m = Map("a"->1, "b"->2, "c"->8)
// Getting a Map, but which kind?
val m: scala.collection.immutable.Map[String,Int] = Map(a -> 1, b -> 2, c -> 8)
```

- **Mutta mikä konkreettilen luokka saadaan (jos ei erikseen tiettyä pyydetä)?**

```
scala> m.getClass
val res1: Class[? <: Map[String, Int]] = class scala.collection.immutable.Map$Map3
```

- **Näköjään Map3-luokka, saadaanko tämä aina?**

```
Map("a"-> 1, "b"-> 2, "c"-> 8, "f"-> 9, "k"-> -1).getClass
val res2: Class[? <: Map[String, Int]] = class scala.collection.immutable.HashMap
```

- **No ei… miksi?**

https://presemo.aalto.fi/o2fi2026

Sisäinen toteutus (internal implementation)

- **Jotkin luokat toteuttavat saman piirteen mutta käyttävät sisäisesti eri tietorakenteita tai algoritmeja**
  - Yleisille operaatioille on eri suorituskyky
- **Ei ole ratkaisua, joka sopii kaikkeen, kun dataa työstetään**
  - Sovelluksia ohjelmoidessa on tärkeää tietää tietorakenteen ominaisuudet
- **Esim. `ListBuffer` ja `ArrayBuffer` molemmat toteuttaa `Buffer`:n, mutta**
  - `ListBuffer` on nopeampi, kun halutaan lisätä elementtejä alkuun (tai loppuun), mutta hidas indeksoida (esim. etsi viides alkio)
  - `ArrayBuffer` on nopea indeksoida, mutta hidas, jos haluat lisätä elementin alkuun
- **Tehokkuusvertailu täällä:** https://docs.scala-lang.org/overviews/collections-2.13/performance-characteristics.html

Muuttuvatilaiset (mutable)

- **Scala.collection.mutablessa olevat luokat ovat muuttuvaisia**
  - Niitä voi muuttaa (päivittää) luomisen jälkeen

```
scala> val x = scala.collection.mutable.Seq(1,2,3,4)
val x: scala.collection.mutable.Seq[Int] = ArrayBuffer(1, 2, 3, 4)

scala> x(1) = 7 // We can change value at index 1

scala> x // Look, updated!
 val res1: scala.collection.mutable.Seq[Int] = ArrayBuffer(1, 7, 3, 4)
```

Muuttumattomat (immutable)

- **Scala.collection.inmutablessa olevat luokat ovat muuttumattomia**
  - Niitä EI voi muuttaa (päivittää) luomisen jälkeen

```
scala> val y = scala.collection.immutable.Seq(1,2,3,4)
val y: Seq[Int] = List(1, 2, 3, 4)

scala> y(1) = 7 // This won't work, look:
       ^ error: value update is not a member of Seq[Int]
       did you mean updated?

scala> val z = y.updated(1,7) // Can update index 1 to 7, but
val z: Seq[Int] = List(1, 7, 3, 4) // `updated` returns a new object

scala> y // Look, y is still the same
val res1: Seq[Int] = List(1, 2, 3, 4)
```

Miksi sekä muuttuvia että muuttumattomia?

- **On olemassa eri ohjelmointityylejä ja käyttötapauksia**
- **Muutuvuus mahdollistaa paikallisen päivityksen (esim. tilan säilyttämiseksi)**
  - Mutta vaatii ohjelmoijalta tietyn tason kurinalaoisuutta,
  - Esim. Herkkyyttä päivitystilauksille
- **Muuttumattomuus rajoittaa jonkin verran tapaa työskennellä kokoelmien kanssa**
  - Datan kuvaukset/muutokset (mapping/transform), pikemminkin kuin tilan
  - Voi olla vähemmän herkkä tietynlaisille virheille
  - Näppäriä funktionaalisessa ja rinnakkaisessa ohjelmoinnista

Ovatko muuttumattomat kokoelmat tehottomia?

- **Ei välttämättä (vaikka sellaiseksikin ne voi tehdä)**
- **Muista – saamme uuden kokoelman (tietorakenteen), emme välttämättä uutta joukkoa datasta!**
  - Koska data on muuttumatonta, osa datasta voidaan jakaa
- **Hyvin suunnitellut tietorakenteet voivat jakaa osarakenteita**
  - Muutokset ovat pienempiä
  - Esimerkkejä seuraa myöhemmillä kierroksilla

Harjoituksissa tällä viikolla, huomaa

- **Kurssimateriaalissa on esimerkkejä, joista on apua**
- **Tutustu Scalan dokumentaatioon, erityisesti `Seq`, `Set` ja `Map`**
- **Joissain harjoituksissa ei voi käyttää tiettyjä konstruktioita tai tietotyyppejä**
  - Esim `var` tai `Array`
  - Tarkistin katsoo, että näitä ei ole käytetty, ja antaa 0 pistettä, jos on
  - Yksikkötestit **eivät** tarkista näitä, ne vain tarkistavat oikeellisuutta, ei sitä, miten toteutus on tehty
- **Lue ohjeet ja annettu koodi huolella**

Ekstraa: hahmonsovitus (pattern matching)

- **Yleisesti kokoelmat sisältävät datan monikkoja (tuple)**
- **Silloin on näppärää käyttää hahmonsovituslausekkeita (pattern matching case expression)**
  - Ei tarvitse `_._1`-rakenteita ja vastaavia
- **Esim. Elementtien vaihtaminen päittäin tai summan laskeminen (“tavalliset” kaksiosaiset monikot):**

```
scala> val l = List(("apple",3.5), ("banana", 2.1), ("orange", 2.5))
val l: List[(String, Double)] = List((apple,3.5), (banana,2.1),
  (orange,2.5))
scala> l.map((x,y) => (y,x))  // Swap positions
val res0: List[(Double, String)] = List((3.5,apple), (2.1,banana),
  (2.5,orange))
scala> l.map((_,x) => x).sum // Sum of second elements
val res1: Double = 8.1
```

Hahmonsovitus (jatkuu)

- **Sisäkkäisille monikoille tarvitaan tapauslauseke (case expression)**
- **Esim. Indeksit arvojen perusteella (otetaan ne indeksit, joissa ”hinta on alle kolme”):**

```
// Zip previous list with its index
scala> val dataIndex = l.zipWithIndex
val dataIndex: List[((String, Double), Int)] = List(((apple,3.5),0),
    ((banana,2.1),1), ((orange,2.5),2))

// Pick out indices of values with p < 3; note the { case }
scala> dataIndex.filter{case ((n,p),i) => p < 3}.map((_,i)=>i)
val res3: List[Int] = List(1,2)
```

Hahmonsovitus (jatkuu) – `unapply`

- **Hahmonsovituksen tapauslausekkeet toimivat mille tahansa objektille `unapply`-metodilla**
- **Luodaan automaattisesti tapausluokille (case classes)**

```
scala> case class Person (fname:String, lname:String)

scala> val people = Seq(Person("Gwen", "Stacy"),
  Person("Miles", "Morales"), Person("Peter", "Parker"))

scala> people.map{case Person(_,last) => last}
val res11: Seq[String] = List(Stacy, Morales, Parker)
```

- **Eli mitäs tämä tekee:**

```
scala> people.filter{case Person(f,l) => f.head != l.head}
```