Seis

- Baastest:
  - 34 sai läbi (kõik kes esitasid)
  - 5 pole teinud
- Kontrolltöö:
  - 9 sai 25..23 punkti
  - 7 sai 21..12 punkti
  - 11 sai 11..0 punkti
  - 12 ei teinud kt-d

Kava

1. Näide Scala programmist. (1. loeng)
2. Kiire ülevaade peamsitest omadustest.
3. Vaatame mõnda keele aspekti detailselt.

Scala

```
object MinuEsimeneScalaProgramm {
  def main(args: Array[String]): Unit = {
    println("Tere maailm!")
  }
}
```

Scalas ...

1. meetodid on seotud *objektidega*
2. peameetodi nimi on `main`
3. peameetodi ainsa argumendi tüübiks on `Array[String]`
4. peameetodi tagastustüübiks on `Unit`

Näide: *Evil Hangman*

```
def main(args: Array[String]): Unit = {
  // loeme sõnastiku
  val wordSet: Set[String] =
    io.Source.fromFile("s6nad.txt").getLines.toSet

  // äraarvatava sõna pikkus
  print("Mitu tähte: ")
  val wordLength: Int = StdIn.readInt()

  // jätame alles õige pikkusega sõnad
  val filteredWords = wordSet.filter(_.length == wordLength)

  if (filteredWords.nonEmpty)
    play(new GameState(filteredWords))
  else
    println("Kahjuks sellise pikkusega sõnu pole.")
}
```

Abifunktsioonid

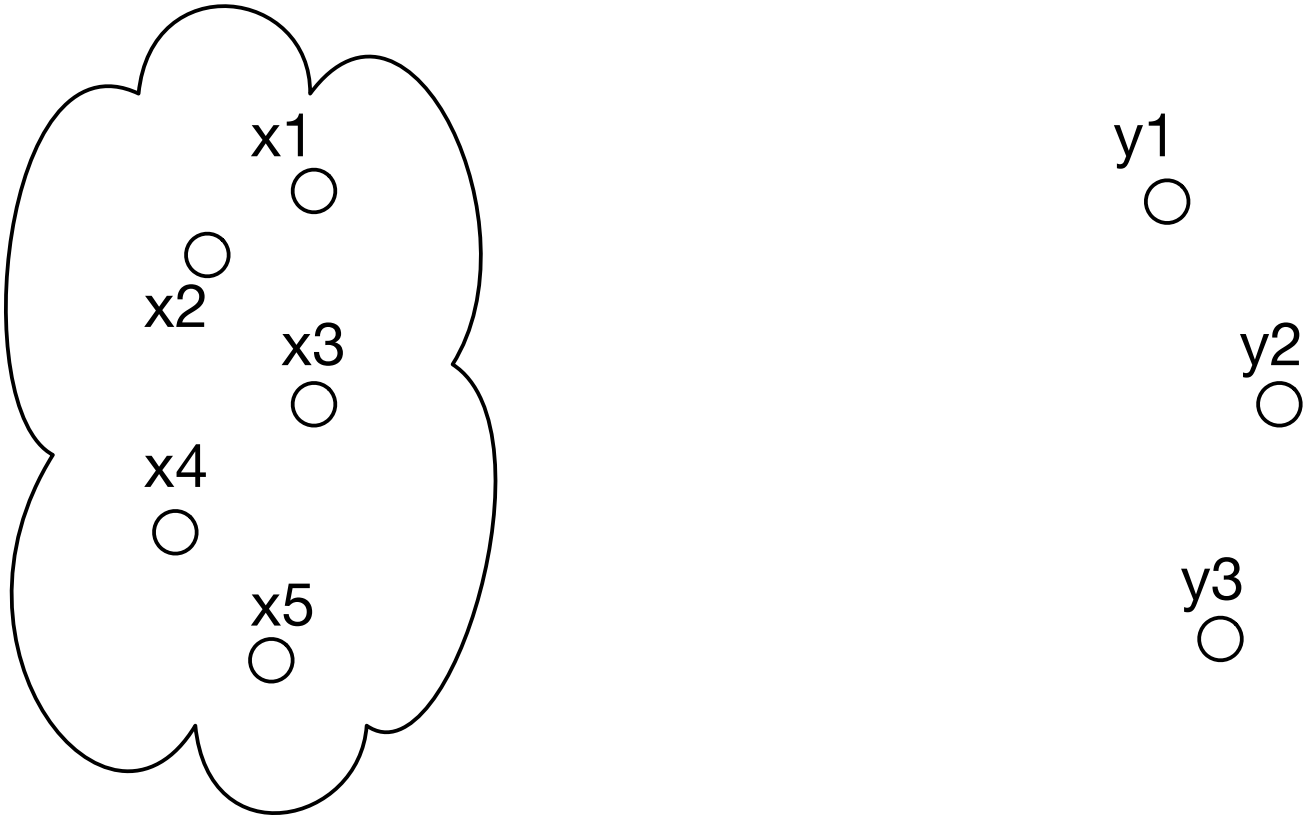
```
def readChar: Char = {
    print("Paku täht: ")
    try {
      StdIn.readChar()
    } catch {
      case _: Throwable => println("Viga!"); readChar
    }
  }

sealed trait Status
case object Correct extends Status
case object Wrong extends Status
```

Mängu loogika

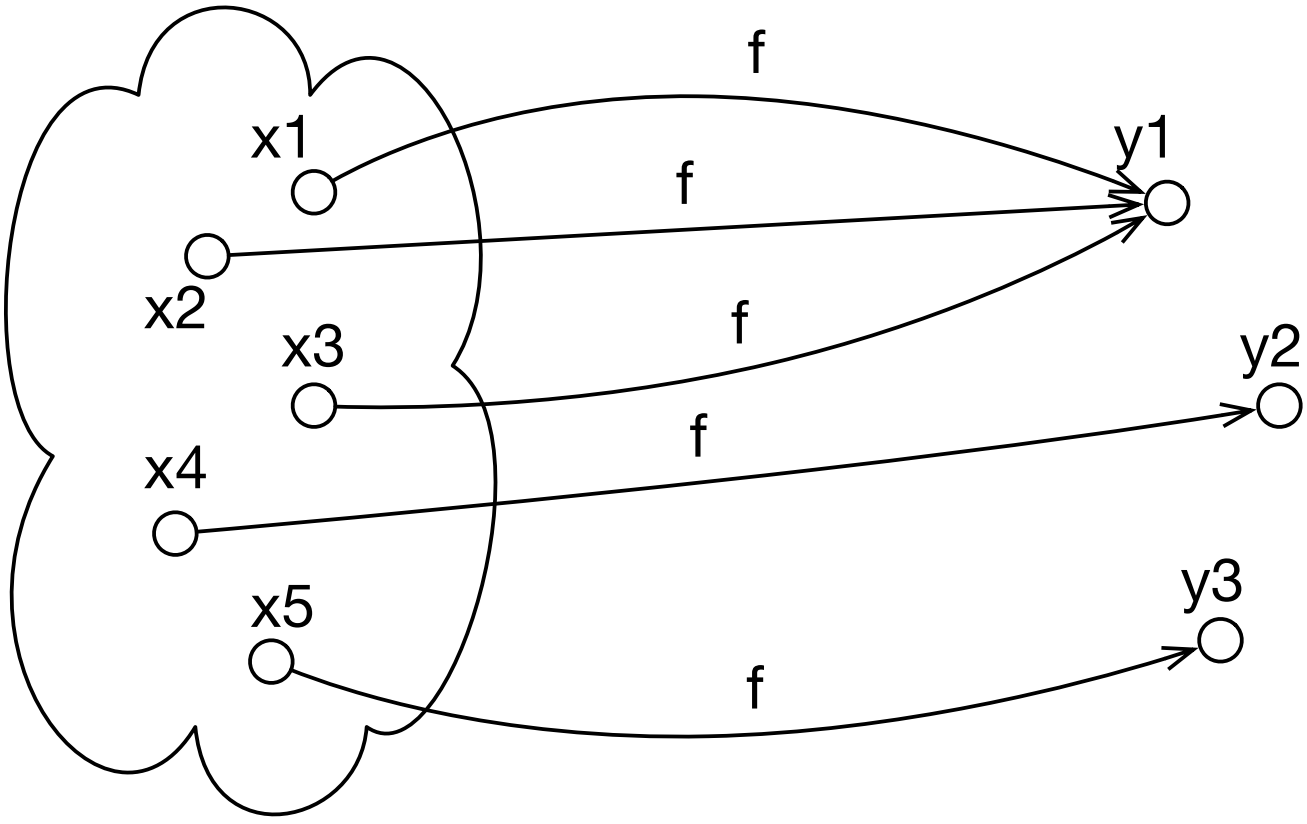
```
class GameState(
    var candidateWords: Set[String],
    var movesLeft: Int   = 21,
    var guessed: Set[Char] = SortedSet.empty)
{

  def blankOtherChars(word: String, charSet: Set[Char]): String =
    word.map(c => if (charSet(c.toLower)) c else '_')

  // Nüüd asendame selles sõnes kõik mitte-pakutud tähed alakriipsuga.
  def blankedWord: String =
    blankOtherChars(candidateWords.head, guessed)

  // Kui meie sõnes pole enam ühtegi alakriipsuga peidetud täht jäänud,
  // siis on sõne ära arvatud ja mäng on läbi.
  def allGuessed: Boolean = !blankedWord.contains('_')
```

```
// Mängu juhtimise meetod, mis teeb ühe käigu ära.
def move(guess: Char): Status = {

  // Nüüd siis peamine töö. Tahame pakkumise järgi liigitada sõnad.
  val groups: Map[String, Set[String]] =
    candidateWords.groupBy(blankOtherChars(_, Set(guess)))

  // Kui on liigitatud, siis tuleks lihtsalt valida need sõnad,
  // kus on kõige rohkem kandidaatsõnu.
  val (pattern, newWords): (String, Set[String]) =
    groups.maxBy{ case (pat, set) => set.size }

  candidateWords = newWords
  movesLeft     -= 1
  guessed       += guess

  if (pattern.contains(guess))
    Correct
  else
    Wrong
}
```

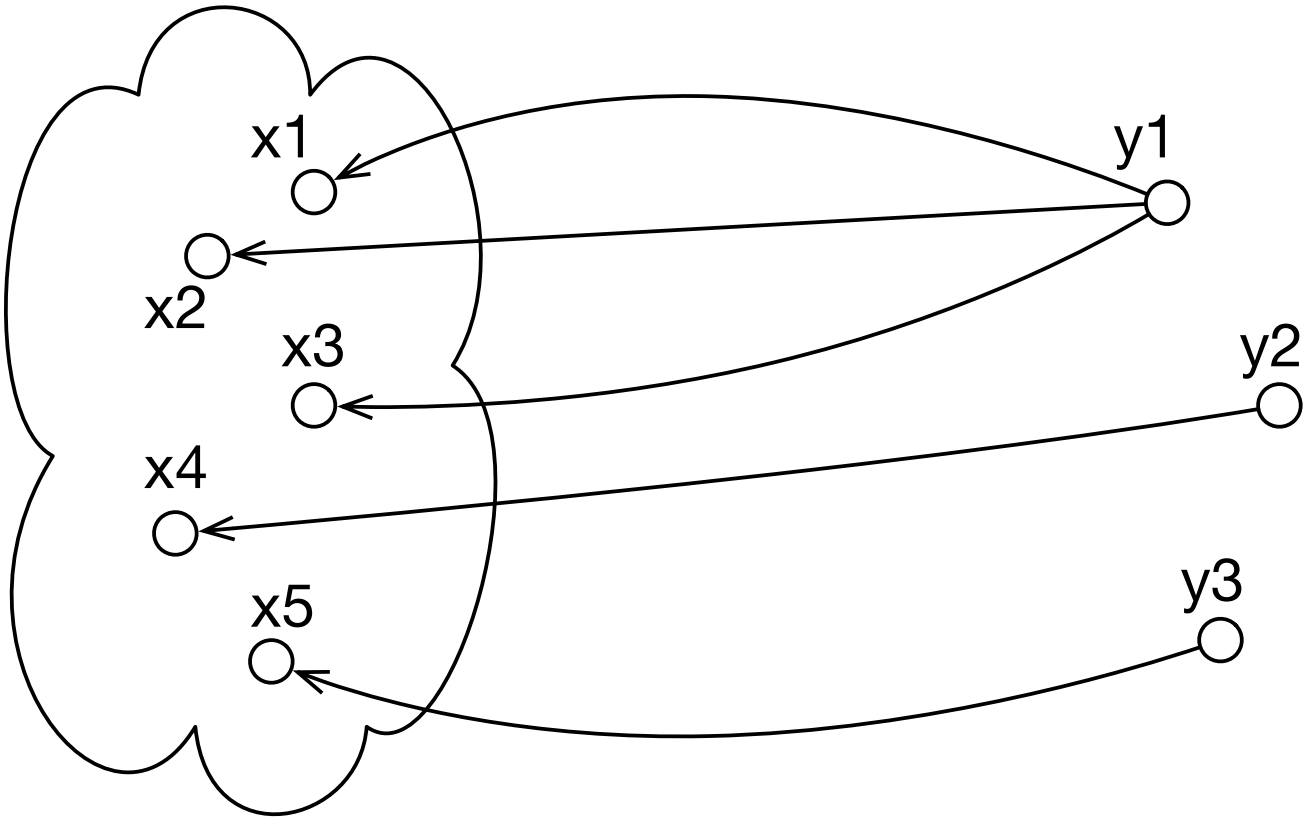
groupBy (1)



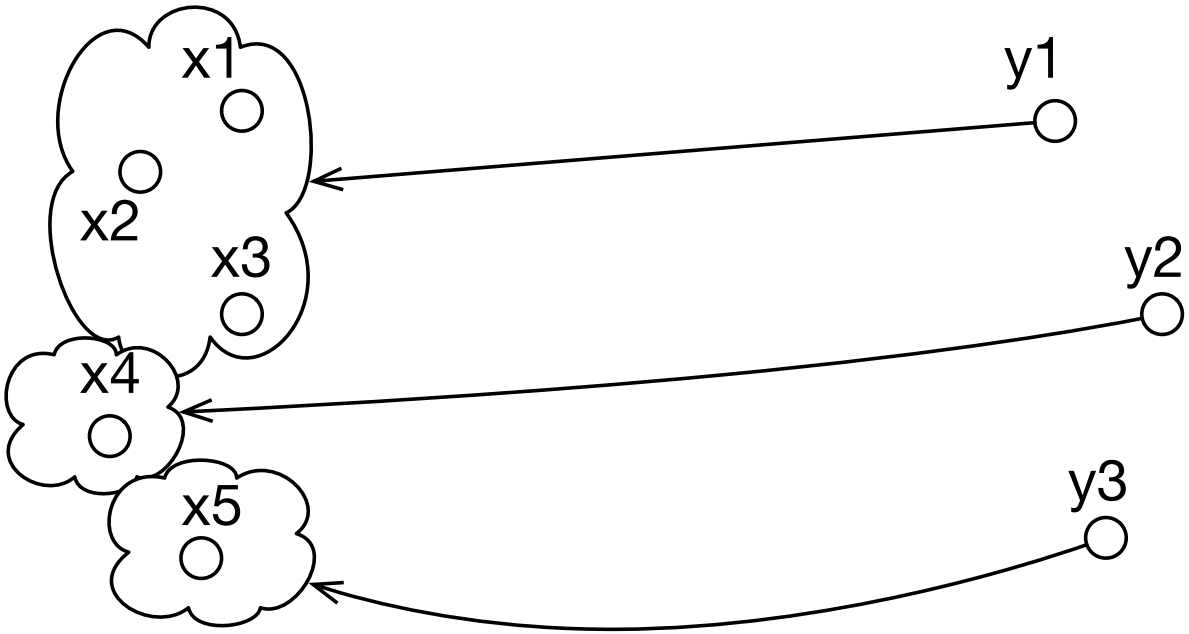
groupBy (2)



groupBy (3)



groupBy (4)



Suhtlus kasutajaga

```
def play(gs: GameState): Unit = {
  if (gs.movesLeft == 0) {
    println("Kaotasid! 6ige vastus oli: " + gs.candidateWords.head)
    return
  }

  printf("Pakutud tähed: %s\nPakkumisi jäänud: %d\n\n%s\n",
    gs.guessed.mkString(", "), gs.movesLeft, gs.blankedWord.mkString(" "))

  if (gs.allGuessed) {
    println("Palju õnne!")
    return
  }

  gs.move(readChar) match {
    case Correct => println("V2ga tubli!")
    case Wrong => println("Vale t2ht!")
  }

  play(gs)
}
```

Scala ülevaade

1. Meetodid, muutujad ja väärtused. Süntaks
2. Lihtsad tüübid ja väärtused.
3. OOP, `case`-klassid ja mustrisobitus.
4. Puhta Scala väärtustamine.
5. Keerulisemad tüübid.
6. Nähtavus, implitsiitsus.

Meetodid

deklaratsioon, näiteks: `def +(x:Int): Unit = { println(x); x+1 }`

meetodi kutse:

$\texttt{nimi}_{\texttt{objekt}}\texttt{.nimi}_{\texttt{meetod}}\texttt{(avaldis}_1\texttt{, ..., avaldis}_n\texttt{)}$

näiteks: `o.+(5)`

infixne kutse (kui on üks argument):

$\texttt{nimi}_{\texttt{objekt}}\ \texttt{nimi}_{\texttt{meetod}}\ \texttt{avaldis}$

näiteks: `o + 5`

Väärtused ja muutujad

Objektide (ja meetodide) sisse saab defineerida väärtuseid, muutujaid ja meetode.

```
object ScalaProgramm {
  var muutuja: Int = 10
  val v22rtus: Int = 100
  def main(args: Array[String]): Unit = {
    muutuja = 20
    val summa = muutuja + v22rtus
    println(summa) // prindib 120
  }
}
```

Sulud ja meetodite kutsed

- Kui meetodil puuduvad argumendid, jätame enamasti sulud ära.
- Sulge on soovitatav kasutada, kui meetod muudab objekti.

```
object Sulud {
  def x(): Int = 5
  def y  : Int = 5
  val z  : Int = 5

  def main(args: Array[String]): Unit = {
    val summa = x + x() + y + z
//    summa += y() + z()  <- ei tööta
  }
}
```

Tüübituletus

- Scala:

```
def foo(x) = x.f + x.g
```

  - `x`-l peavad olema meetodid (või väljad) `f` ja `g`
  - klasse, mis defineerivad `f` ja `g` võib olla palju
  - ka viise, kuidas `f` ja `g` defineerida on palju
  - meetodi `f` tagastusväärtus peab omama meetodit `+`
  - klasse, mis defineerivad `+` on palju
  - ???

- Haskell:

```
foo x = f x + g x
```

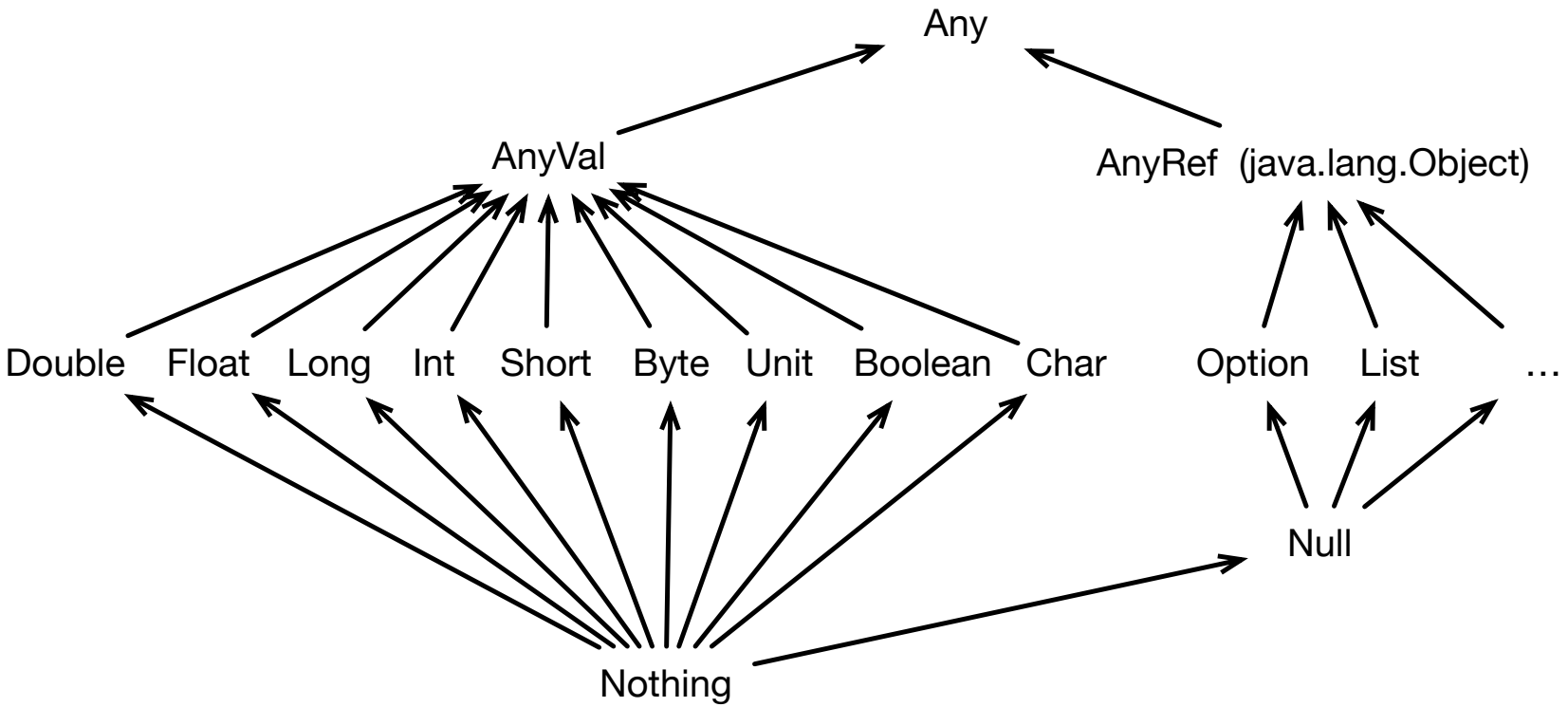
  - peavad leiduma `f :: α -> β` ja `g :: γ -> δ`
  - kuna rakendame `x` mõlemale, siis peab α `=` γ
  - kuna `(+) :: Num a =>a -> a -> a` siis β `=` δ ja `Num` α
  - s.t. `foo :: Num` β `=>` α `->` β

- (need olid lihtsustatud näited)

Süntaks

- Tingimusavaldised:
  ```
  if(e) s1 else s2
  ```
- tsükklid (lihtsustatult):
  ```
  for (muutuja <- algus to l6pp) { ... } // kaasa-arvatud
  for (muutuja <- algus until l6pp) { ... } // välja-arvatud
  for (muutuja <- kollektsioon) { ... }
  ```
- funktsiooni tüüp:
  ```
  tyyp =>tyyp2
  ```
- lambdad:
  ```
  (muutuja: tyyp, muutja2: tyyp2) => keha
  ```
- blokid (eraldi ridadel võib semikoolonid ära jätta):
  $\{e_1; e_2; \ldots e_n;\}$

Alamtüüpimine



Avaldised

Väärtused (`AnyVal`):

- `() : Unit`
- `true : Boolean`, `false : Boolean`
- `'a' : Char`, `'b' : Char`, ...
- `1 : Byte`, `2 : Short`, `3 : Int`, `3 : Long`
- `1f : Float`, `2 : Double`

- Väärtustel saab samuti meetodeid välja kutsuda. N: `1.+(2)`
- Aritmeetikafunktsioonid. N: `math.max(math.Pi, x*4)`

Klassid ja Objektid

```
class MinuKlass {
  val viis: Int = 5
  def lisaviis(x: Int): Int = x+viis
}
```

- Klassid on objektide tüübid.

```
var o : MinuKlass = null
```

- Objekte saab luua võtmesõnaga **new**.

```
o = new MinuKlass
```

Konstruktorid

- Peamine konstruktor — argumendid klassi nime järel ja keha meetodite ja väljadega segamini.

```
class K(nimi: String, synniaasta: Int) {
  println("Loodi klass K("+ nimi +"," + synniaasta + ")")
  def umbkaudneVanus(): Int = LocalDate.now.getYear - synniaasta
}
```

- Abikonstruktor.

```
class K(nimi: String, synniaasta: Int) {
  def this() = this("nipitiri",LocalDate.now.getYear)
  println("Loodi klass K("+ nimi +"," + synniaasta + ")")
  def umbkaudneVanus(): Int = LocalDate.now.getYear - synniaasta
}
```

- Konstruktori väljakutse:

```
  val x = new K("Donald", 1934)
  val y = new K // = new K("nipitiri", 2018)
```

Case klassid ja objektid

```
trait List[+A] // Scala listide lihtsustus
case object Nil                          extends List[Nothing]
case class Cons[A](x:A, xs: List[A]) extends List[A]

val list = Cons(1, Cons(2, Cons(3, Nil)))
```

- Nagu algebralised andmestruktuurid Haskellis.
- Ei kasutata võtmesõna **new**.
- Defineerib võrduse, mis sõltub konstruktorite argumentidest:

```
Cons(3, Nil) == Cons(3, Nil)
Cons(3, Nil) != Nil
```

- Defineerib `toString` meetodi:

```
list.toString = "Cons(1,␣Cons(2,␣Cons(3,␣Nil)))"
```

Traitid ja pärimine

- Liideste asemel on Scalas **`trait`**-id:

```
trait T {
  var q: Char
  def f(x: Int): Double
  def g(x: Int): Double = f(10) / 2
}
```

- Abstraktsed klassid:

```
abstract class Q {
  def h(x: Int): Boolean
}
```

- Klass laiendab kuni ühte klassi ja suvaline arv *trait*-e.

```
class W extends Q with T {
  override var q: Char = 'a'
  override def f(x: Int): Double = x.toDouble % 10
  override def g(x: Int): Double = f(10) / 3
  override def h(x: Int): Boolean = f(x) > 100
}
```

Traitide lineariseerimine

- Traitid väldivad mitmese pärimise probleeme kasutades lineariseerimist.
- Objekti meetodi kutsel tehakse valik vastavalt eelistusjärjekorrale

Näide

```
class A
trait B extends A
trait C extends A
trait D extends C
class E extends A with B with D
```

Järjekord: `E -> D -> C -> B -> A -> AnyRef -> Any`

Apply meetod

Kirjutades	`o(e)`
mõistab Scala seda nii:	`o.apply(e)`

- Kasutatakse näiteks massiivide (`Array`) ja kujundite (`Map`) juures.
- Kasutatakse andmestruktuuride koostamisel.

```
val a = Array(11,22,33) // Array.apply(11,22,33)
val b = a(1)            // a.apply(1) == 22
```

Mustrisobitus

```
def length[A](xs: List[A]): Int =
  xs match {
    case Nil         => 0
    case Cons(_, xs) => 1 + length(xs)
  }
```

Omistamisega meetodid

Kirjutades `l += e` või `l.+=(e)`
mõistab Scala seda nii: `l = l + e`

- Töötab suvalise sümbolitest koosneva operaatoriga.
- Avaldis `l` väärtustakse üks kord.

```
class Q(val y:Int) {
  def +(x: Int): Q = new Q(x+y)
}

var x = new Q(10)
x += 10
println(x.y) // trükitakse 20
```

Puhta Scala Väärtustamine

- Aritmeetika väärtustatakse samamoodi nagu Haskellis.
- Blokid:
  - `{ e }` → `e`
  - `{ val x =v; es }` → `{ es`$[x \to v]$` }` (kui `v` on normaalkujul)
  - `{ val x =`e_1`; es }` → `{ val x =`e_2`; es }`
  - `{ def f ...; es }` → `{ es }` (kui `f` ei sisaldu `es`-s)
  - `{ def f ...;` es_1 `}` → `{ def f ...;` es_2 `}`
  - `{ class k ...; es }` → `{ es }` (kui `f` ei sisaldu `es`-s)
  - `{ class k ...;` es_1 `}` → `{ class k ...;` es_2 `}`
  - Sarnaselt klassidele käitume case-klasside ja objektide puhul.
- Argumendid väärtustatakse enne rakendust.
- Funktsioon **def** `fn(`x_1`,...,`x_n`) =` e:

$$\texttt{fn}(v_1, \ldots, v_n) \to e[x_1 \to v_1] \ldots [x_n \to v_n]$$

Näited

```
e = {
  def mul(x: Int, y: Int) = x * y
  def add(a: Int, b: Int) = a + b
  add(5, mul(3,8))
}
```

```
add(5, mul(3,8))
        ↓(x*y)[x->3][y->8]
   add(5, 3*8)
        ↓
    add(5, 24)
        ↓(a+b)[a->5][b->24]
      5+24
        ↓
       29
```

Puhta Scala Väärtustamine

- Meetodid `class K(p1,...,pm) { def fn(x1,...,xn) = e; ... }`:

$$\texttt{new K}(w_1,\ldots,w_m).\texttt{fn}(v_1,\ldots,v_n)$$
$$\downarrow$$
$$e[x_1 \to v_1]\ldots[x_n \to v_n][p_1 \to w_1]\ldots[p_m \to w_m][\texttt{this} \to \texttt{new K}(w_1,\ldots,w_m)]$$

- Väljad `class K(p1,...,pm) { val n = e; ... }`:

$$\texttt{new K}(w_1,\ldots,w_m).\texttt{n}$$
$$\downarrow$$
$$e[p_1 \to w_1]\ldots[p_m \to w_m][\texttt{this} \to \texttt{new K}(w_1,\ldots,w_m)]$$

- Sarnaselt käitume case-klasside ja objektide puhul.

Näited

```
e = {
  class Kala(nimi: String) {
    def tee(mida: String) =  nimi+"␣"+mida+"b."
  }
  val nemo = new Kala("Neemo")
  nemo.tee("uju")
}
```

```
      nemo.tee("uju")
            ↓
new Kala("Neemo").tee("uju")
            ↓ (nimi+"␣"+ mida+"b.")[nimi->"Neemo"][mida->"ujub"]
 ("Neemo"+"␣"+ "uju"+"b.")
            ↓
       "Neemo␣ujub."
```

Näited

```
e = {
  class Kala(nimi: String) {
    def tee(mida: String) = nimi+"␣"+mida+"b."
    def lenda = this.tee("lenda")
  }
  new Kala("Doris").lenda
}
```

```
    new Kala("Doris").lenda
                ↓ this.tee("lenda")[this->new Kala("Doris")]
new Kala("Doris").tee("lenda")
                ↓ (nimi+"␣"+mida+"b.")[mida->"lenda"][nimi->"Doris"]
  ("Doris"+"␣"+"lenda"+"b.")
                ↓
        "Doris␣lendab."
```

Tüübiparameetrid (lihtne vorm)

- Kandilistes sulgudes, komadega eraldatult.
- Kasutatakse sõnu: geneeriline ja polümorfism.
- Meetodi või klassi nime järel.

```
class A[T](val x:T) {
  def g[U](f: T => U): U = f(x)
}
val a = new A[Int](5)
println(a.g[String](x => x.toString+"#")) // trükib 5#
```

- Püütakse tuletada

```
val a = new A(5)
println(a.g(x => x.toString+"#")) // trükib 5#
```

Parameetrite kitsendamine

- ülevalt: `T <: U`

```
trait Koduloom
trait Kass extends Koduloom
trait Koer extends Koduloom
class LoomaWrapper[L <: Koduloom](p: L) {
  def loom: L = p
}
```

- alt: `T >: U`

```
trait Järjend[B] {
  def lisaAlgusesse[U >: B](e: U): Järjend[U]
}
val kassid : Järjend[Kass] = ???
val loomad : Järjend[Koduloom] = kassid.lisaAlgusesse[Koduloom](???)
```

- implitsiitselt teisendatav tüüp: `T <% U`
- `T : M` — leidub implitsiitne väärtus `M[T]`

Klasside variantsus

Oletame, et `U <: V` ehk `U` ülemklass on `V`.

	Klassi definitsioon	Tähendus
kovariantsus	`C[+T]`	`C[U] <: C[V]`
kontravariantsus	`C[-T]`	`C[U] >: C[V]`
invariantsus	`C[T]`	`C[U]` ja `C[V]` pole seotud

- Listid on kovariantsed (`List[+A]`).
  - `List[Koer]` saame kasutada kui nõutakse `List[Koduloom]`
- Printerid on kontravariantsed.
  - `Printer[Koduloom]` saame kasutada kui nõutakse `Printer[Kass]`

```
class Printer[-A] {
  def print(p: A): Unit
}
```

Tüübi alias ja abstraktsed andmetüübid

- Tüübile saab anda uue nime:

```
type uusNimi = vanaTyyp
```

- Traiti sees võimaldab luua abstraktseid andmetüüpe:

```
trait MyList[T] {
  type S
  def empty: S
  def insert(t: T, s: S): S
  def foldr[Q](t: Q, op: T => Q => Q, s:S): Q
}
```

Karrimine ja mitu argumendi komplekti

Vaatame foldLeft tüüpi:

```
def foldLeft[B](z: B)(op: (B, A) => B): B
```

- Kõigepealt võtab ühe argumendi z siis op.
- Paindlikum, kuna järgnev komplekt võib sõltuda eelmustest:

```
trait Q {
  type t
  var e : t
}

object Test {
  def f(q: Q)(v: q.t): q.t = {
    v
  }
}
```

Meetodi nähtavus

võtmesõna	**nähtavus**
`private[this]`	ainult sama objekt
`private`	sama klassi objektid
`protected`	lisaks alamklassidest
`private[paketiNimi]`	nimetaud paketist
	(vaikeväärtus) kõik

*Mixin*id — motivatsioon

```
trait Part {
  def prääks
  def lenda
  def joonista
}
class SinikaelPart extends Part { ... }
class TuliPart     extends Part { ... }
class KummiPart    extends Part { ... } // ei prääksu (vaid piiksub)
class PeibutusPart extends Part { ... } // ei tee häält ega lenda
...
```

- Koodi taaskasutus problemaatiline.
- Pole tüübi järgi eristust, kes lendab ja kes prääksub.
- Uue meetodi lisamine tüütu.

Java-lik lahendus — *strategy pattern*

```
class Part(p:PrääksuStrateegia, l:LennuStrateegia) {
  def prääks
  def lenda
  def joonista
}
class SinikaelPart extends Part(prääksuVõime     , lennuVõime)
class TuliPart     extends Part(prääksuVõime     , lennuVõime)
class KummiPart    extends Part(piiksuVõime      , lennuVõimePuudu)
class PeibutusPart extends Part(prääksuVõimePuudu, lennuVõimePuudu)
...
```

- Lahendab taaskasutuse probleemi.

Liittüübiga lahendus

```
trait Prääks      {  def prääks    }
trait PiiksPrääks {  def prääks    }
trait Lenda       {  def lenda     }
trait Part        {  def joonista }

class SinikaelPart extends Part with Prääks      with Lenda
class TuliPart     extends Part with Prääks      with Lenda
class KummiPart    extends Part with PiiksPrääks with Lenda
class PeibutusPart extends Part
```

- Koodi taaskasutus ok.
- Tüübi järgi eristust, kes lendab.
- Uue meetodi lisame vaid sinna kuhu vaja.
- Väga palju klasse.

Mixin lahendus

```
trait Prääks      { def prääks    }
trait PiiksPrääks { def prääks    }
trait Lenda       { def lenda     }
trait Part        { def joonista }

val skp = new Part with Prääks      with Lenda { ... }
val tp  = new Part with Prääks      with Lenda { ... }
val kp  = new Part with PiiksPrääks with Lenda { ... }
val pp  = new Part                             { ... }
```

- Nii vähe klasse kui ise soovime.
- Tüübid paindlikud.

```
def lennuta(p: Lenda) = ...
```

Klasside ja Traitide lineariseerimine

- Traitide sees võib olla implementatsioon.
- Alamtrait saab implementatsiooni muuta (**override**).
- Tekib küsimus: milline implementatsioon peale jääb.
- Vastus: Vastavalt lineariseerimie järjekorrale.

Näide:

```
trait A                          { ... }
trait B extends A                { ... }
trait C extends A with B         { ... }
trait D extends B                { ... }
class E extends A with D with C { ... }
```

Mis järjekorras meetode klassidest/traitidest otsitakse?

Lineariseerimise algoritm l

$$C_0 \texttt{ extends } C_1 \texttt{ with } C_2 \texttt{ with } \ldots \texttt{ with } C_n$$

- Kui $n = 0$, siis $l(C_0) := C_0 \to \texttt{AnyRef} \to \texttt{Any}$
- Kui $n > 0$:
  1. Võtame $k := \texttt{AnyRef} \to \texttt{Any}$
  2. Teeme tsükli $i \leftarrow \{1, 2 \ldots n\}$
     1. Leitakse $x := l(C_i)$
     2. Eemaldame x-st need, mis leiduvad k-s.
     3. Uuendame $k := x \to k$
  3. $l(C_0) = C_0 \to k$

Lineariseerimine

Tahame teada E lineariseerimist aga arvutame hoopis järjest (alates A-st).

```
trait A                              { ... }
trait B extends A                    { ... }
trait C extends A with B             { ... }
trait D extends B with C             { ... }
class E extends A with D with C { ... }
```

- $l(A) = \ldots$ $l(A) = A \to \texttt{AnyRef} \to \texttt{Any}$
- $l(B) = \ldots$ $l(B) = \ldots \to l(A)$ $l(B) = \ldots \to A \to \texttt{AnyRef} \to \texttt{Any}$
  $l(B) = B \to A \to \texttt{AnyRef} \to \texttt{Any}$
- $l(C) = \ldots$ $l(C) = \ldots \to l(A)$ $l(C) = \ldots \to A \to \texttt{AnyRef} \to \texttt{Any}$
  $l(C) = \ldots \to B \to A \to \texttt{AnyRef} \to \texttt{Any}$ $l(C) = C \to B \to A \to \texttt{AnyRef} \to \texttt{Any}$
- $l(D) = \ldots$ $l(D) = \ldots \to l(B)$ $l(D) = \ldots \to B \to A \to \texttt{AnyRef} \to \texttt{Any}$
  $l(D) = \ldots \to C \to B \to A \to \texttt{AnyRef} \to \texttt{Any}$
  $l(D) = D \to C \to B \to A \to \texttt{AnyRef} \to \texttt{Any}$
- $l(E) = \ldots$ $l(E) = \ldots \to l(A)$ $l(E) = \ldots \to A \to \texttt{AnyRef} \to \texttt{Any}$
  $l(E) = \ldots \to D \to C \to B \to A \to \texttt{AnyRef} \to \texttt{Any}$
  $l(E) = E \to D \to C \to B \to A \to \texttt{AnyRef} \to \texttt{Any}$

Objektide võrdsus

```
class Any {
  final def ==(that: Any): Boolean
  def equals(that: Any): Boolean
  ...
}
class AnyRef extends Any {
  final def eq(that: Any): Boolean
  def equals(that: Any): Boolean = this eq that
  ...
}
```

`AnyVal`ide puhul:

- Nii `==` kui `equals` võrdlevad sisuliselt.

`AnyRef`ide puhul:

- `eq` võrdleb viitasid
- `equals` tuleks üle laadida sisulise võrsusega, vaikimisi sama mis `eq`.
  - Ärge unustage ka `hashCode` üle defineerida!
- `x == y` $\Longleftrightarrow$ `if (x eq null) y eq null else x equals y`

Mustrisobitus

- Saab kasutada lausena:

```
var sign = ...
val c: Char = ...
c match {
  case '+' => sign = 1
  case '-' => sign = -1
  case _  => sign = 0
}
```

- Saab kasutada avaldisena:

```
val c: Char = ...
val sign =
  c match {
    case '+' => 1
    case '-' => -1
    case _  => 0
  }
```

- Nägime, et saab sobitada literaalide ning alakriipsuga.

Mustrisobitus

- Mustrid võivad sisaldada valvureid ja muutujaid:

```
var sign = 0
var s = ""
StdIn.readChar() match {
  case '+' => sign = 1
  case '-' => sign = -1
  case c if Character.isDigit(c) => s :+= c
  case _  => sign = 0
}
```

- Mustrid üle ennikute, listide ja `case`-klasside:

```
list match {
  case Nil => 0
  case x :: y :: xs => x + y
}
tuple match {
  case (x, 1) => x
  case (x, y) => x/y
}
```

Mustrisobitus

- Mustrisobitus üle tüüpide:

```
val a : Any = ???
a match {
  case x: Int    => x
  case x: String => x.length
  case _         => 1
}
```

- Kuna geneerikute (v.a. massiivid) tüübid kustutatakse, ei soovitata mustrisobitust üle nende teha.

```
Map(1 -> 'x', 2 -> 'c') match {
  case x: Map[Int,String] => println(1) // sobitub
  case _                  => println(2)
}
```

Mustrisobitus

- Mustrid kasutamine väljaspool **case**-avaldist:

```
val (x, y) = (x, y)
for ((k,v) <- paarid) ...
for ((k,v) <- paarid if v != 0) ...
```

- Saame ise mustreid luua:

```
object at {
  def unapply(arg: Email): Option[(String, String)] =
    Some(arg.n, arg.h)
}
class Email(val n: String, val h: String)
val mail = new Email("kalmera", "ut.ee")

mail match {
  case n at "ut.ee" => "kohalik"
  case _            => "mujalt"
}
```

Mustrisobitus

- Suvaline arv argumente mustris:

```
object set {
  def unapplySeq[T](arg: Set[T]): Option[Seq[T]] =
    Some(arg.toSeq)
}
val s = Set(1, 2, 3)

s match {
  case set(x, y, z) => "kolm"
  case _            => "????"
}
```

- for-"laused" implementeeritakse foreach-iga, s.t.

```
// for (x <- p) { ... } ==  p.foreach(x => ... )
trait MyIterable[+T]{
  def foreach[U](f: T=>U): Unit
}
```

- Näiteks arvuvahemikud:

```
case class MyRange(begin: Int, end: Int, step: Int = 1)
  extends MyIterable[Int]
{
  override def foreach[U](f: Int => U): Unit = {
    if (begin <= end) {
      f(begin)
      MyRange(begin+step, end, step).foreach(f)
    }
  }
}
```

Näiteks

```
for (x <- MyRange(1,2)) { println(x) }
```
↓ desugar `for`
```
MyRange(1,2).foreach(println(_))
```
↓ `1<=2`
```
println(1); MyRange(1+1,2).foreach(println(_))
```
↓ `1+1<=2`
```
println(1); println(2); MyRange(2+1,2).foreach(println(_))
```
↓ `2+1>2`
```
println(1); println(2); ()
```

- Süntaktilise suhkru eemaldamine toimub kogu programmis enne väärtustamise algust.

Implitsiitsed klassid

Tahame täiendada olemasolevat tüüpi mingi meetodiga. Näiteks:

```
5 korda println("Hei!")
```

Seda teeme järgneva klassi deklareerimisega:

```
implicit class IntLisaksKorda(x: Int) {
  def korda[A](f: => A): Unit = {
    (0 to x).foreach( _ => f)
  }
}
```

- Meetodi (näites `korda`) puudumisel pakendatakse väärtus (`Int`) väärtused implitsiitsesse klassi (`IntLisaksKorda`).
- Implitsiitsedel klassidel on kitsendused:
  - Ei või olla välimises skoobis.
  - Täpselt üks mitte-`implicit` konstruktori argument.
  - Ei tohi olla sama nimega meetodi, välja ega objekti.

Näiteks

```
5 korda println("Hei")
```

↓ desugar infix

```
5.korda(println("Hei"))
```

↓ 5-l pole meetodit `korda`

```
new IntLisaksKorda(5).korda(println("Hei"))
```

```
(0 to 5).foreach(_ =>f)
```

↓ desugar infix

```
(0.to(5)).foreach(_ =>f)
```

↓ 0-l pole meetodit `to`

```
(RichInt(0).to(n)).foreach(_ =>f)
```

Lihtsustatud näide: arvuvahemike süntaks

```
implicit class Kuniga(i:Int) {
  def kuni(j: Int): Range = Range(i, j)
}

implicit class Sammga(r:Range) {
  def sammuga(j: Int): Range = Range(r.start, r.end, j)
}

def main(args: Array[String]): Unit = {
  val v1 = 1 kuni 10
  val v2 = 1 kuni 10 sammuga 2
  for (x <- v2)
    printf("%d␣", x)
  println()
}
```

Näiteks

```
1 kuni 10 sammuga 2
```

↓ desugar infix

```
1.kuni(10).sammuga(2)
```

↓ 1-l pole meetodit `kuni`

```
new Kuniga(1).kuni(10).sammuga(2)
```

↓ `new Kuniga(1).kuni(10)` pole meetodit `sammuga`

```
new Sammuga(new Kuniga(1).kuni(10)).sammuga(2)
```

Lihtsustatud näide: Listid

- Eesmärk:

```
object TestMyList {
  def main(args: Array[String]): Unit = {
    val v = MyList(1, 2, 3, 4)
    for (x <- v)
      printf("%d␣", x)
    println()
  }
}
```

Lihtsustatud näide: Listid

- Andmestruktuuri definitsioon:

```
sealed abstract class MyList[+T] extends MyIterable[T]

case object MyNil extends MyList[Nothing] {
  override def foreach[U](f: Nothing => U): Unit = ()
}

case class MyCons[T](head:T, tail: MyList[T]) extends MyList[T]
{
  override def foreach[U](f: T => U): Unit = {
    f(head)
    tail.foreach(f)
  }
}
```

- Konstrueerimine kasutab apply meetodit objekti sees

```
object MyList {
  def apply[T](xs: T*): MyList[T] =
    xs.foldRight[MyList[T]](MyNil)(MyCons.apply)
}
```

Näiteks

```
for(x <-MyList(1,2)) { println(x) }
```

↓ desugar for

```
MyList(1,2).foreach(println(_))
```

↓ MyList.apply implementatsioon

```
Seq(1,2).foldRight(MyNil)(MyCons(_,_)).foreach(println(_))
```

↓ foldRight

...

↓

```
MyCons(1,MyCons(2,MyNil)).foreach(println(_))
```

↓ MyCons.foreach implementatsioon

```
println(1); MyCons(2,MyNil).foreach(println(_))
```

↓ MyCons.foreach implementatsioon

```
println(1); println(2); MyNil.foreach(println(_))
```

↓ MyNil.foreach implementatsioon

```
println(1); println(2); ()
```

- Eesmärk:

```
object TestMap {
  def main(args: Array[String]): Unit = {
    val m = MyMap(4 -> 'd', 5 -> 'e', 1 -> 'a', 2 -> 'b', 3 -> 'c')
    m = m.add(6 -> 'a')
    for((x,y) <- m)
      printf("%d -> %c\n", x, y)
    print(m(2))
  }
}
```

- Ehk:

```
object TestMap {
  def main(args: Array[String]): Unit = {
    val m = MyMap((4, 'd'), (5, 'e'), (1, 'a'), (2, 'b'), (3, 'c'))
    m = m.add((6, 'a'))
    m     .withFilter{ case (x,y) => true; case _ => false }
          .foreach { case (x, y) => printf("%d -> %c\n", x, y) }
    print(m(2))
  }
}
```

Lihtsustatud näide: kujutised

```
sealed abstract class MyMap[T, +U] extends MyIterable[(T,U)] {
  def apply(p: T): U

  // peavalu: miskipärast vaja for ((x,y) <- map) ... jaoks
  def withFilter(p: ((T,U)) => Boolean): MyMap[T, U]
}


case class MyEmptyMap[T]() extends MyMap[T, Nothing] {
  override def apply(p: T): Nothing = throw new NoSuchElementException
  override def foreach[U](f: ((T, Nothing)) => U): Unit = ()
  override def withFilter(p: ((T, Nothing)) => Boolean) = this
}
```

```
case class MyConsMap[T, +U](left: MyMap[T, U],key: T, v: U,
  right: MyMap[T, U]) extends MyMap[T, U] {

  override def foreach[V](f: ((T, U)) => V): Unit = {
    left.foreach(f)
    f(key,v)
    right.foreach(f)
  }

  override def apply(p: T): U = {
    val ph = p.hashCode()
    val kh = key.hashCode()
    if (ph == kh)       v
    else if (ph < kh)   left(p)
    else                right(p)
  }

  // pole korrektselt implementeeritud: vaja for
  // ((x,y) <- map) jaoks, et teha mustrisobitust
  override def withFilter(p: ((T, U)) => Boolean): MyMap[T, U] = this
}
```

Konstrueerimine & operatsioonide lisamine

- Konstrueerimine sama nagu listide puhul.
- Operatsioonid teise, näiteks kompanjonobjekti:

```
object MyMap {
  def apply[T, U](xs: (T,U)*): MyMap[T, U] = {
    xs.foldLeft[MyMap[T,U]](MyEmptyMap())(add)
  }

  def add[T, U](m: MyMap[T,U], x: (T, U)): MyMap[T, U] = {
    ...
  }
}
```

Operatsioonide lisamine

- Soov on operatsioonid panna andmestruktuuri objekti:

```
sealed abstract class MyMap[T, U] extends MyIterable[(T,U)] {
  ...
  def add(x: (T, U)): MyMap[T, U]
}
```

  - Probleem: `U` pole enam kovariantne

- Trikk:

```
sealed abstract class MyMap[T, +U] extends MyIterable[(T,U)] {
  ...
  def add[V >: U](x: (T, V)): MyMap[T, V]
}
```

Lihtsustatud näide: kujutised

```
case class MyConsMap[T, +U]
  ...

  override def add[V >: U](x: (T, V)): MyMap[T, V] = {
    val ph = x._1.hashCode()
    val kh = key.hashCode()
    if (ph == kh)
      MyConsMap(left, x._1, x._2, right)
    else if (ph < kh)
      MyConsMap(left.add(x), key, v, right)
    else
      MyConsMap(left, key, v, right.add(x))

  }
}
```

Scala 2.7 (ja varem)

- Soov on operatsioonid tõsta klassihierarhias kõrgemale:

```
trait scala.Iterable[A] {
  // üks abstraktne meetod
  def elements: Iterator[A]

  // palju konkreetseid meetode
  def isEmpty: Boolean = ...
  def map[B](f: A => B): Iterable[B] = ...
  def dropWhile(p: A => Boolean): Iterable[A] = ...
}
```

- Täpsed tüübid lähevad kaduma:
  - `List(...).map(...): Iterable`
  - `HashSet(...).dropWhile(...): Iterable`
  - `TreeSet(...).map(...): Iterable`

Scala 2.8 kuni 2.12

- Kasutame implitsiitseid argumente:

```
trait MyCanBuildFrom[-From, -Elem, +To] {
  def apply(): MyBuilder[Elem, To]
}
trait MyBuilder[-Elem, +To] {
  def +=(elem: Elem): MyBuilder.this.type
  def clear(): Unit
  def result(): To
}


trait MyIterable[+A, CC[_], +C] {
  def map[B, That](f: A => B)
          (implicit bf: MyCanBuildFrom[MyIterable[A], B, That]): That
    val b = bf()
    for (x <- this)
      b += f(x)
    b.result
  }
  ...
}
```

Implitsiitsed parameetrid

Kuidas saab kirjutada:

```
val x1 = List(1,2,3).max
val x2 = List("aabits", "zorro").max
val x3 = List(false, true).max
```

Kui listidel (`List[+A]`) on meetod:

```
def max[B >: A](implicit cmp: Ordering[B]): A = ...
```

Järjestus on enam-vähem selline:

```
trait Ordering[A] { def compare(x: T, y: T): Int }
```

Kuskil on defineeritud:

```
implicit object A extends Ordering[Int] { ... }
implicit object B extends Ordering[Boolean] { ... }
implicit object C extends Ordering[String] { ... } //leks. järjestus
```

Scala 2.8 kuni 2.12

- Kasutame implitsiitseid argumente:

```
// Kuskil defineeritud:
implicit def cbfm[C,A,B]: MyCanBuildFrom[C, (A,B), MyMap[A,B]] = ...
implicit def cbfl[U]: MyCanBuildFrom[MyList[_], U, MyList[U]] = ...
implicit def cbfs: MyCanBuildFrom[MyList[_], Char, String] = ...

// Meie koodis:
val x : MyList[Int] = MyList(1,2,3,4)
val z : MyList[Char] = x.map(_.toChar)
val z : String = x.map(x => (x + 'a'.toInt).toChar)
val q : MyMap[Int, Char] = x.map(x => x -> (x+'a'.toInt).toChar)
```

- Ülipaindlik
- Veateated kohutavad, `map` tüüp kohutav
- Väga keeruline ja suur süsteem!

Kollektsioonid Scalas oli suur süsteem!

```
trait GenTraversableOnce[+A] extends Any

trait TraversableOnce[+A] extends Any with GenTraversableOnce[A]

trait GenIterable[+A] extends GenIterableLike[A, GenIterable[A]]
  with GenTraversable[A] with GenericTraversableTemplate[A, GenIterable]

trait GenericTraversableTemplate[+A, +CC[X] <: GenTraversable[X]]
  extends HasNewBuilder[A, CC[A] @uncheckedVariance]

trait GenIterableLike[+A, +Repr] extends Any
  with GenTraversableLike[A, Repr]

trait GenTraversableLike[+A, +Repr] extends Any
  with GenTraversableOnce[A] with Parallelizable[A, ParIterable[A]]

trait TraversableOnce[+A] extends Any with GenTraversableOnce[A]
```

Väga keeruline ja suur süsteem!

```
trait TraversableLike[+A, +Repr] extends Any with HasNewBuilder[A, Repr]
  with FilterMonadic[A, Repr] with TraversableOnce[A]
  with GenTraversableLike[A, Repr] with Parallelizable[A, ParIterable[A]]

trait Traversable[+A] extends TraversableLike[A, Traversable[A]]
  with GenTraversable[A] with TraversableOnce[A]
  with GenericTraversableTemplate[A, Traversable]

trait IterableLike[+A, +Repr] extends Any with Equals
  with TraversableLike[A, Repr] with GenIterableLike[A, Repr]

trait Iterable[+A] extends Traversable[A] with GenIterable[A]
  with GenericTraversableTemplate[A, Iterable]
  with IterableLike[A, Iterable[A]]
```

Uued Scala kollektsioonid (2.13)

- Aluseks võetud iteraatorid:

```
trait IterableOnce[+A] extends Any {
  def iterator(): Iterator[A]
}

trait Iterator[+A] extends IterableOnce[A] {
  // abstraktsed meetodid
  def hasNext: Boolean
  def next(): A

  def iterator() = this

  // konkreetsed meetodid
  def dropWhile(p: A => Boolean): Iterator[A] = ...
  ...
}
```

Uued Scala kollektsioonid (2.13)

- Kasutab kõrgemat järku tüübimuutujaid:

```
trait IterableOps[+A, +CC[_], +C] extends Any {
  protected[this] def coll: Iterable[A]

  def iterableFactory: IterableFactory[CC]
  protected[this] def fromSpecificIterable(coll: Iterable[A]): C
  protected[this] def newSpecificBuilder(): Builder[A, C]

  // konkreetsed meetodid
  def size: Int =
    if (knownSize >= 0) knownSize else coll.iterator().length

  def dropWhile(p: A => Boolean): C =
    fromSpecificIterable(View.DropWhile(coll, p))

  def map[B](f: A => B): CC[B] =
    iterableFactory.fromIterable(View.Map(coll, f))
}
```

Uued Scala kollektsioonid (2.13)

- Lihtsustatud näide:

```
trait MyIterableOps[+A, +CC[_], +C] extends MyIterable[A] {
  def ++[B >: A](suffix: MyIterable[B]): CC[B]
  def map[B](f: A => B): CC[B]
  def filter(p: A => Boolean): C
}
sealed abstract class MyList[+T]
  extends MyIterableOps[T, MyList, MyList[T]]
{
  // def ++[B >: T](suffix: MyIterable[B]): MyList[B]
  // def map[B](f: T => B): MyList[B]
  // def filter(p: T => Boolean): MyList[T]
  ...
}
```

Uued Scala kollektsioonid (2.13)

- Kasutab ära ülelaadimist

```
trait SortedSet[A] extends Set[A]
  with SortedSetOps[A, SortedSet, SortedSet[A]]

trait SortedSetOps[A, +CC[X] <: SortedSet[X], +C <: SortedSet[A]]
  extends SetOps[A, Set, C] with SortedOps[A, C]
{
  ...
}
```

- `SortedSet[A]`-l on kaks `map`-i

```
def map[B           ](f: A => B): Set[B]
def map[B : Ordering](f: A => B): SortedSet[B]
```

Uued Scala kollektsioonid (2.13)

- `BitSet`-id:

```
trait BitSet extends SortedSet[Int] with BitSetOps[BitSet]

trait BitSetOps[+C <: BitSet with BitSetOps[C]]
  extends SortedSetOps[Int, SortedSet, C]
{
  def map(f: Int => Int): C = ...
  ...
}
```

- `BitSet`-il on need `map`-id

```
  def map[B](f: Int => B): Set[B]
  def map[B : Ordering](f: Int => B): SortedSet[B]
  def map(f: Int => Int): BitSet
```

Eelnev definitsioon pole Scala 2.11-s mugavalt kasutatav:

```
scala> BitSet(1,3,5).map(_ + 1)
<console>:13: error: missing parameter type for
      expanded function ((x) => x.plus(1))
```

Tüübituletus versioonis 2.11:

1. Püüame leida meetodi tüübi kuju järgi: jääb mitu alternatiivi
2. Üritame leida argumendi tüüpi:
   - Lambda _ + 1 on ilma oodatava tüübita.
   - Ei oska tüüpi leida, kuna liitmine võib olla defineeritud mitmel tüübil.

Keeleuuendused >=2.12

1. Püüame leida meetodi tüübi kuju järgi: jäävad kaks alternatiivi

```
(Function1[Int, B]): SortedSet[B]
(Function1[Int, Int]): BitSet
```

2. Unifitseerime alternatiivide tüübid:

```
(Function1[Int, ?]): ?
```

3. Leiame argumendi tüübi, kasutades meetodite unifitseeritud tüüpi:
   - `_ + 1` tüübitakse oodatava tüübiga `Function1[Int, ?]`
   - See õnnestub tüübiga `Function1[Int, Int]`

4. Tüübitakse alternatiivid, kasutades argumendi tüüpi. Sobib `(Function1[Int, Int]): BitSet`

Mitme operatsiooni tegemine

- Tehes

```
List(1,2,3,4).filter(f).map(g)
```

tekitatakse iga operatsiooni järel uus list.

- Vahetulemust ei genereeri:

```
for (x <- List(1,2,3,4) if f(x)) yield g(x)
```

ehk

```
List(1,2,3,4).withFilter(f).map(g)
```

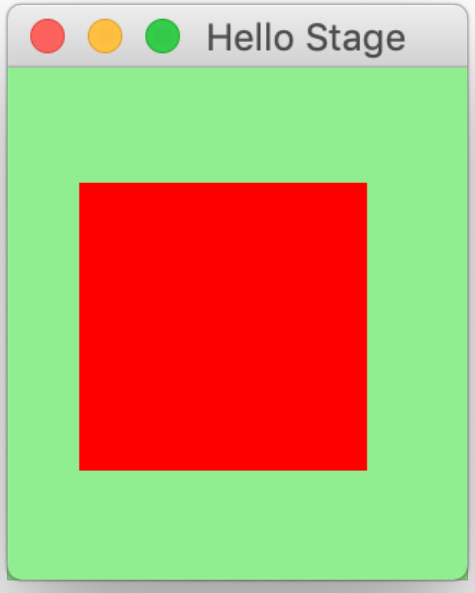
WithFilter

```
abstract class WithFilter[+A, +CC[_]] {
  def map[B](f: A => B): CC[B]
  def flatMap[B](f: A => MyIterable[B]): CC[B]
  def foreach[U](f: A => U): Unit
  def withFilter(q: A => Boolean): WithFilter[A, CC]
}

trait IterableOps[+A, +CC[_], +C] extends Any {
  ...
  def withFilter(q: A => Boolean): WithFilter[A, CC]
}
```

- Sama eesmärgiga on ka vaated (`View`).
  - parem implementatsioon 2.13-s

ScalaFX

```
object HelloStageDemo extends JFXApp {
  stage = new PrimaryStage {
    title = "Hello Stage"
    width = 160
    height = 200
    scene = new Scene {
      fill = LightGreen
      content = new Rectangle {
        x = 25
        y = 40
        width = 100
        height = 100
        fill = Red
      }
    }
  }
}
```



- `build.sbt`-sse lisada:

```
libraryDependencies += "org.scalafx" %% "scalafx" % "12.0.2-R18"

// Determine OS version of JavaFX binaries
lazy val osName = System.getProperty("os.name") match {
  case n if n.startsWith("Linux") => "linux"
  case n if n.startsWith("Mac")  => "mac"
  case n if n.startsWith("Windows") => "win"
  case _ => throw new Exception("Unknown platform!")
}

// Add dependency on JavaFX libraries, OS dependent
lazy val javaFXModules = Seq("base", "controls", "fxml",
                "graphics", "media", "swing", "web")
libraryDependencies ++= javaFXModules.map( m =>
  "org.openjfx" % s"javafx-$m" % "12.0.2" classifier osName
)
```

- Programm laiendab `JFXApp` trait-i.
- Kood kirjutada konstruktorisse, mitte `main`-i v.m.s.
- Kasutab `x_=(...)` meetodeid, näiteks

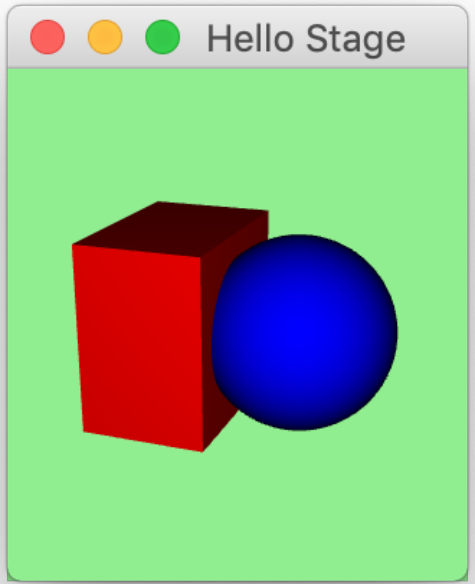
```
title = "Hello Stage"   ==>   title_=("Hello Stage")
```

Arhitektuur

- Stage – Aken ise
- Scene – Akna sisu
- Node – Akna element
  - Text
  - Shape (Line, Rectangle, . . .)
  - Chart (Bar, Pie, Line, Area, Bubble, Scatter)
  - Pane (VBox, HBox, StackPane, TilePane,GridPane . . .)
  - Control (Button, Lable, TreeView, TextArea . . .)
  - Canvas
  - ImageView
  - WebView

Shape3d

```
content = Seq(
  new Box() {
    transforms = Seq(new Translate(-1, 0, 0))
    material = new PhongMaterial(Color.Red)
    height = 3; width = 2; depth = 3
  },
  new Sphere() {
    radius = 1.5
    transforms =
      Seq(new Translate(1, 0, 0))
    material =
      new PhongMaterial(Color.Blue)
  }
)

camera = new PerspectiveCamera(true) {
  transforms =
    Seq(new Rotate(-20, Rotate.YAxis), new Rotate(-20, Rotate.XAxis),
      new Translate(0, 0, -15))
}
```



Kõrvalepõige: suhtlus kasutajaga

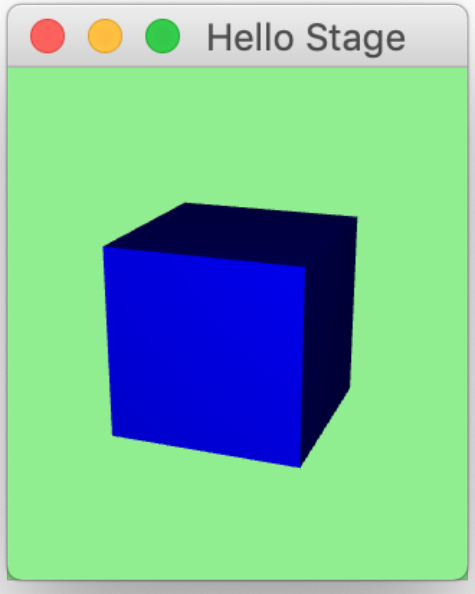
- Küsitlus
  - `if (tingimus) then tee_midagi()`
  - \+ sobiv, kui küsitleda harva
  - \- ei sobi, kui vaja kiiresti reageerid

- Sündmus
  - `override def onEvent(...) { ... }`
  - `handlers +={ ... }`
  - *Observer pattern*: registreerin vaatleja muutuva väärtuse A juurde, mis muudab väärtust B.
  - \+ sobiv, kui vaja kiiresti reageerida
  - \- tihti unustatakse *handler*ite eemaldamine

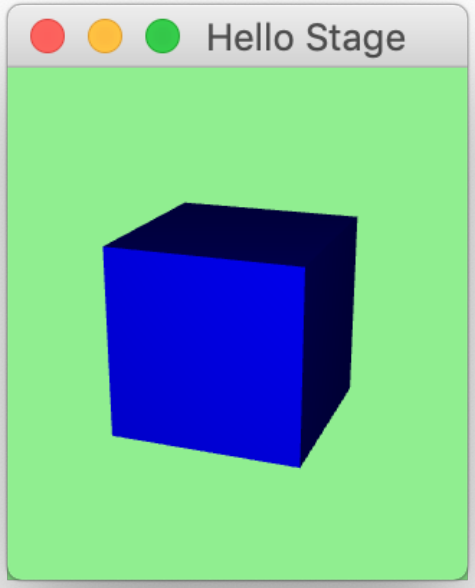
- Functional Reactive Programming (FRP)
  - `area <== base * height / 2`
  - `prop <== when(cond) choose(value1) otherwise(value2)`
  - FRP: registreerin B juurde seose, et ta võtaks väärtuse A-st.
  - \+ sobiv, kui vaja kiiresti reageerida
  - \+ sobiv lihtsate tingimuste jaoks
  - \+ pole vaja *handler*eid eemaldada
  - \- ei sobi tsükliliste sõltuvuste korral

Events

```
content =
  new Box() {
    material =
      new PhongMaterial(Color.Red)
    onMouseClicked = { _ =>
      material =
        new PhongMaterial(Color.Blue)
    }
    height = 3
    width = 3
    depth = 3
  }
```

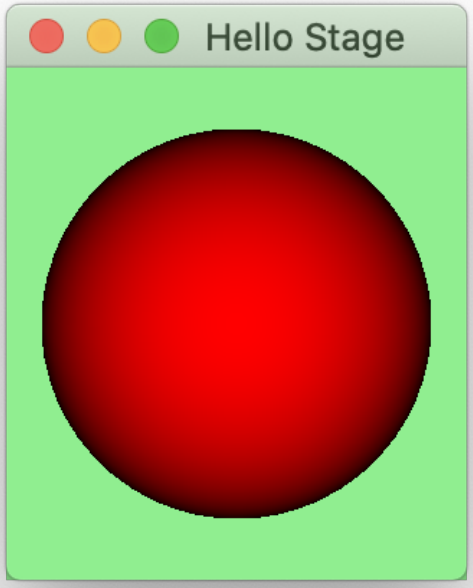


Property



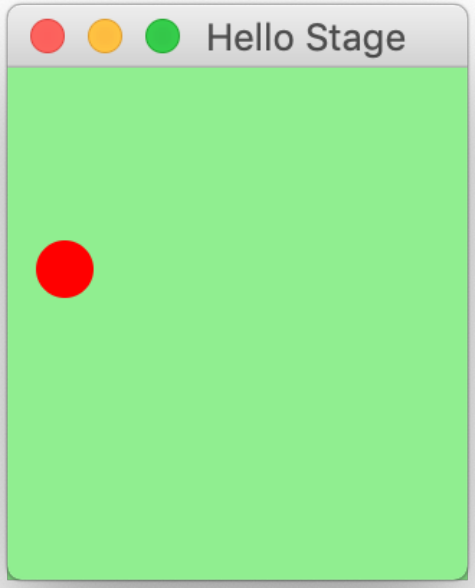

```
content =
  new Box() {
    material <==
      (when(hover)
        choose new PhongMaterial(Color.Red)
        otherwise new PhongMaterial(Color.Blue))

    height = 3; width = 3; depth = 3
  }
```

Animatsioonid




```
content =
  new Sphere() {
    material = new PhongMaterial(Color.Red)
    radius = 3
    onMouseClicked = { _ =>
      Timeline(at(3 s){radius -> 1}).play()
    }
  }
```

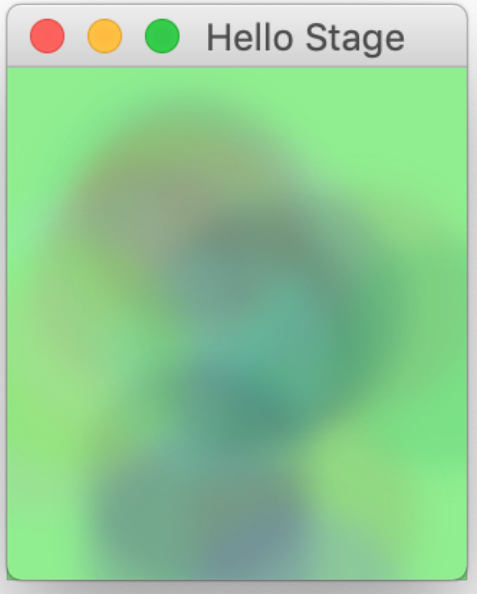
Animatsioonid 2




```
scene = new Scene {
  fill = LightGreen

  private val c = new Circle {
      centerX = 20
      centerY = 70
      radius = 10
      fill = Red
    }

  content = c

  onMouseClicked = { _ =>
    Timeline(at(0.5 s){c.centerY -> (height.value - 5)}).play()
  }
}
```

Animatsioonid 4

```
val cs = for (i <- 0 to 20) yield new Circle {
  centerX = random * 160
  centerY = random * 200
  radius = 40
  fill = color(random, random, random, 0.2)
  effect = new BoxBlur(10,10,3)
  onMouseClicked = handle {
    Timeline(at(3 s) {radius -> 0}).play()
  }
}
new Timeline{
  cycleCount = Timeline.Indefinite
  autoReverse = true
  keyFrames = for (c <- cs) yield at(20 s) {
    Set[KeyValue[_, _ <: Object]](
      c.centerX -> random * 160,
      c.centerY -> random * 200)}
}.play()
```



DelayedInit

Klassid ja objektid, mis pärivad `DelayedInit`, muudetakse nii: `code` $\Longrightarrow$ `delayedInit(code)`. S.t

```
trait C1 extends DelayedInit {
  println("C1 initsialiseerimine")
  def delayedInit(body: => Unit): Unit = {
    println("enne C2 initsialiseerimist")
    body       // C2 initsialiseerimine
    println("peale C2 initsialiseerimist")
  }
}

class C2 extends C1 {
  println("C2 initsialiseerimine")
}

object Test {
  def main(args: Array[String]): Unit = {
    val c = new C2
  }
}
```

App

`DelayedInit` kasutatakse ka trait-i `App` poolt.

```
trait App extends DelayedInit { // mõned detailid eemaldatud
  private val initCode = new ListBuffer[() => Unit]

  override def delayedInit(body: => Unit) {
    initCode += (() => body)
  }

  def main(args: Array[String]) = {
    for (proc <- initCode) proc()
  }
}

object Test extends App {
  val c = new C
}
```

Property

- Erinevad tüüpi omadused: `BooleanProperty`, `DoubleProperty`, `FloatProperty`, `IntegerProperty`, `LongProperty`, `StringProperty` ja `ObjectProperty`.
- Saab ka ise teha:

```
val speed1 = DoubleProperty(55)
val speed2 = new DoubleProperty(this, "speed2", 55)
```

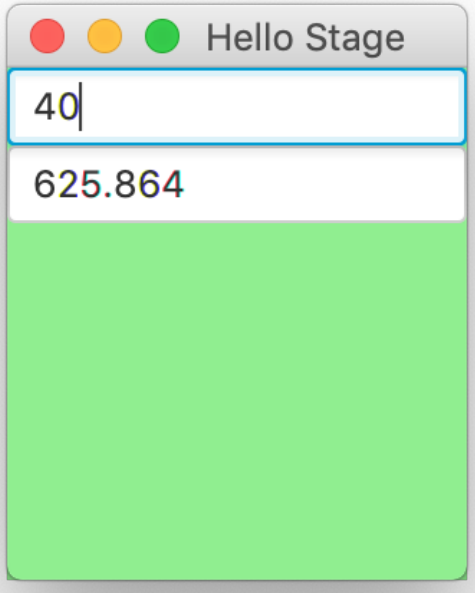
- Seotud sündmuste käsitlemisega:

```
prop.onChange { (source, oldValue, newValue) => doSomething() }
```

- Saab omavahel ühendada: `a <== b`, `a <==> b`

Omadused 2

```
scene = new Scene {
  val iF = new TextField(){
    maxWidth = 160
    text = "1"
  }

  val input1 = createIntegerBinding(
    () => Try(iF.text.value.toInt).getOrElse(0),
    iF.text)

  val outputText = new TextField(){
    maxWidth = 160
    text <== (input1 * 15.6466).asString()
  }

  content = new VBox(iF, outputText)
}
```



ScalaFX - Ensemble

Mitmelõimelisus

- Igal klassil (monitoril) on järgnevad meetodid:

```
def synchronized[A] (e: => A): A
def wait()
def wait(msec: Long)
def notify()
def notifyAll()
```

- Synchronized argument täidetakse teisi lõimi välistavalt: kaks lõime ei saa samaaegselt täita sama monitori *synchronized* koodi.

- Enamasti oodatakse mingi tingimuse C täitumist:

```
while (!C) wait()
```

Näide: BoundedBuffer

```
class BoundedBuffer[A](N: Int) {
  var in = 0, out = 0, n = 0
  val elems = new Array[A](N)

  def put(x: A) = synchronized {
    while (n >= N) wait()
    elems(in) = x ; in = (in + 1) % N ; n = n + 1
    if (n == 1) notifyAll()
  }

  def get: A = synchronized {
    while (n == 0) wait()
    val x = elems(out) ; out = (out + 1) % N ; n = n - 1
    if (n == N - 1) notifyAll()
    x
  }
}
```

Näide: BoundedBuffer

- BoundedBuffer kasutamine

```
import scala.concurrent.ops._
...
val buf = new BoundedBuffer[String](10)
spawn { while (true) { val s = produceString ; buf.put(s) } }
spawn { while (true) { val s = buf.get ; consumeString(s) } } }
```

- Spawn definitsioon

```
def spawn(p: => Unit) {
  val t = new Thread() { override def run() = p } t.start()
}
```

Sünkroniseeritud muutujad: SyncVar

```
class SyncVar[A] {
  var isDefined: Boolean = false
  var value: A = _

  def get = synchronized {
    while (!isDefined) wait()
    value
  }

  def set(x: A) = synchronized {
    value = x; isDefined = true; notifyAll()
  }

  def isSet: Boolean = synchronized {
    isDefined
  }
  def unset = synchronized {
    isDefined = false
  }
}
```

Tulevikuväärtused: `future`

- Väärtus, mida arvutatakse teises lõimes.

```
import scala.concurrent.ops._
...
val x = future(pikk_arvutus)
teine_pikk_arvutus
val y = f(x()) + g(x())
```

- `future` on defineeritud nii:

```
def future[A](p: => A): Unit => A = {
  val result = new SyncVar[A]
  fork { result.set(p) }
  (() => result.get)
}
```

Mitmelõimelisus

- Paketis `scala.concurrent` veel võimalusi: `Promise, Channel, Lock` jne.
- Keerukust aitab vähendada sobivama arvutusmudeli valimine: `Actor`id.
  - Sõnumite edastamisel põhinev mudel.
  - Aktorid reageerivad sissetulevale sõnumile, seejärel võivad teha kohalikke arvutusi ja omakorda saata sõnumeid.
  - Hea implementatsioon: Akka
  - Aktoreid saab viia ka teise arvutisse.

Akka

Aktoritel põhinev arvutusmudel:

- Programm on hulk aktoreid.
- Aktorile saab saata sõnumeid.
- Aktor töötleb sõnumeid järjest:
  - teeb kohalikke arvutusi ja
  - saadab sõnumeid teistele aktoritele.

Eelised teiste mudelite ees (reklaam):

- Kergekaaluline mudel.
- Koodi struktureerimisel üks kindel viis.
- Aktorid saab paigutada hajusalt.
- Aktorid saavad töötada paralleelselt.

Tegelikult:

- Sobib, kui vaja kiiresti reageerida reaalajas tekkivatele signaalidele.

Kasutusjuhud veebilehelt

- Transaction processing (Online Gaming, Finance/Banking, Trading, Statistics, Betting, Social Media, Telecom)
  - Scale up, scale out, fault-tolerance / HA
- Service backend (any industry, any app)
  - Service REST, SOAP, Cometd, WebSockets etc Act as message hub / integration layer Scale up, scale out, fault-tolerance / HA
- Concurrency/parallelism (any app)
  - Correct Simple to work with and understand Just add the jars to your existing JVM project (use Scala, Java, Groovy or JRuby)
- Simulation
  - Master/Worker, Compute Grid, MapReduce etc.

Kasutusjuhud veebilehelt (cont.)

- Batch processing (any industry)
  - Camel integration to hook up with batch data sources Actors divide and conquer the batch workloads
- Communications Hub (Telecom, Web media, Mobile media)
  - Scale up, scale out, fault-tolerance / HA
- Gaming and Betting (MOM, online gaming, betting)
  - Scale up, scale out, fault-tolerance / HA
- Business Intelligence/Data Mining/general purpose crunching
  - Scale up, scale out, fault-tolerance / HA
- Complex Event Stream Processing
  - Scale up, scale out, fault-tolerance / HA

Mis see praktiliselt tähendab?

```
class A {
  def m(...) {
    ???
  }
}
val a = new A
a.m(argument)
```

⇒

```
case class M(...)
class A extends Actor {
  def receive = {
    case M(...) => ???
  }
}
val a = ActorSystem("s").actorOf(Props[A], "a")
a ! M(argument)
```

või hajusa paigutuse puhul

```
case class M(...)
val a = ActorSystem("s")
        .actorSelection("akka://s@example.com:5678/user/A")
a ! M(argument)
```

- Sõnumi saatmine meetodiga '!'

```
def !(message: Any)(implicit sender: ActorRef =Actor.noSender): Unit
```

Väikseim töötav näide

```
import akka.actor.{Actor, ActorSystem, Props}

case class M(s: String)

class A extends Actor {
  def receive: PartialFunction[Any, Unit] = {
    case M(s) => println("Hello!")
  }
}

object Test extends App {
  val a = ActorSystem("s").actorOf(Props[A], "a")
  a ! M("Hi!")
}
```

Omavaheline suhtlus

```
case object Pall

class M(s6num: String) extends Actor {
  var vastane: ActorRef = _
  def receive: PartialFunction[Any, Unit] = {
    case a:ActorRef => vastane = a
    case Pall => println(s6num); Thread.sleep(1000); vastane ! Pall
  }
}

object Test extends App {
  val a = ActorSystem("s").actorOf(Props(classOf[M],"Ping"))
  val b = ActorSystem("s").actorOf(Props(classOf[M],"Pong"))
  a ! b
  b ! a
  a ! Pall
}
```

- Peale loomist suhtlus ainult sõnumitega!

Vastamine ja sünkroonne suhtlus

```
import akka.pattern._
import scala.concurrent.ExecutionContext.Implicits.global

class S extends Actor {
  var d: Int = 0
  def receive: PartialFunction[Any, Unit] = {
    case a: Int =>     d += a
    case _: Unit =>    sender ! d
  }
}

object Test extends App {
  implicit val t: Timeout = Timeout(10,TimeUnit.SECONDS)
  val s = ActorSystem("s").actorOf(Props(classOf[S]))
  s ! 10
  s ! 5
  val q : Future[Int] = s ? ()
  q.map(println)
}
```

Kõrvalepõige: Future ja Promise

- Kuidas teha samaaegset arvutust?
- Kuidas teha asünkroonseid meetodikutseid?
- Future!

Tulevikuväärtused

```
object Main {
  def pikk_arvutus1() = 1+1
  def pikk_arvutus2() = 2+2
  def main(args: Array[String]): Unit = {
    val a = pikk_arvutus1()
    val b = pikk_arvutus2()
    println(a+b)
  }
}
```

- Tahame a ja b arvutada samaaegselt.

Tulevikuväärtused

```
import scala.concurrent.duration.Duration
import scala.concurrent.{Await, ExecutionContext, Future}

object Main {
  def pikk_arvutus1() = 1+1
  def pikk_arvutus2() = 2+2
  def main(args: Array[String]): Unit = {
    val a = Future{pikk_arvutus1()}(ExecutionContext.global)
    val b = pikk_arvutus2()
    println(Await.result(a,Duration.Inf)+b)
  }
}
```

- `Future[Int]`: lubadus tagastada `Int` tüüpi väärtus.

Tulevikuväärtused elegantsemalt

```
object Main {
  implicit val ec: ExecutionContext = ExecutionContext.global
  def pikk_arvutus1() = Future{1+1}
  def pikk_arvutus2() = Future{2+2}
  def main(args: Array[String]): Unit = {
    val a = pikk_arvutus1()
    val b = pikk_arvutus2()
    val c = for (x <- a; y <- b) yield
      println(x+y)
    Await.ready(c,Duration.Inf)
  }
}
```

Future praktiliselt

- `Future[Int]`-l on meetodid
  - `def isCompleted: Boolean`,
  - `def value: Option[Try[Int]]` ja
  - `def onComplete[U](f: Try[Int] =>U)`
    `(implicit executor: ExecutionContext): Unit`
- `Try[A]` on `A` tüüpi väärtus või erind.
- Aga kuidas ise sellist mugavalt implementeerida?

Promise

- `Promise` – ühekordselt kirjutatav muutuja
  - `def tryComplete(result: Try[T]): Boolean`
  - `def isCompleted: Boolean`
  - `def future: Future[T]`

```
object Main {
  implicit val ec: ExecutionContext = ExecutionContext.parasitic
  def main(args: Array[String]): Unit = {
    val parv = Promise[Int]
    val farv = parv.future
    for (x <- farv)
      println(x)
    parv.tryComplete(Try{1})
  }
}
```

Promise ja Future erinevus

- `Future`-d on rangelt kapseldatud
  - Ei pea muretsema ootamise järjekorra pärast.
- `Promise`-d on kapseldamata
  - Programmeerija peab ise hoolitsema, et ta ei hakkaks ootama iseenda järele.

Vastamine ja sünkroonne suhtlus

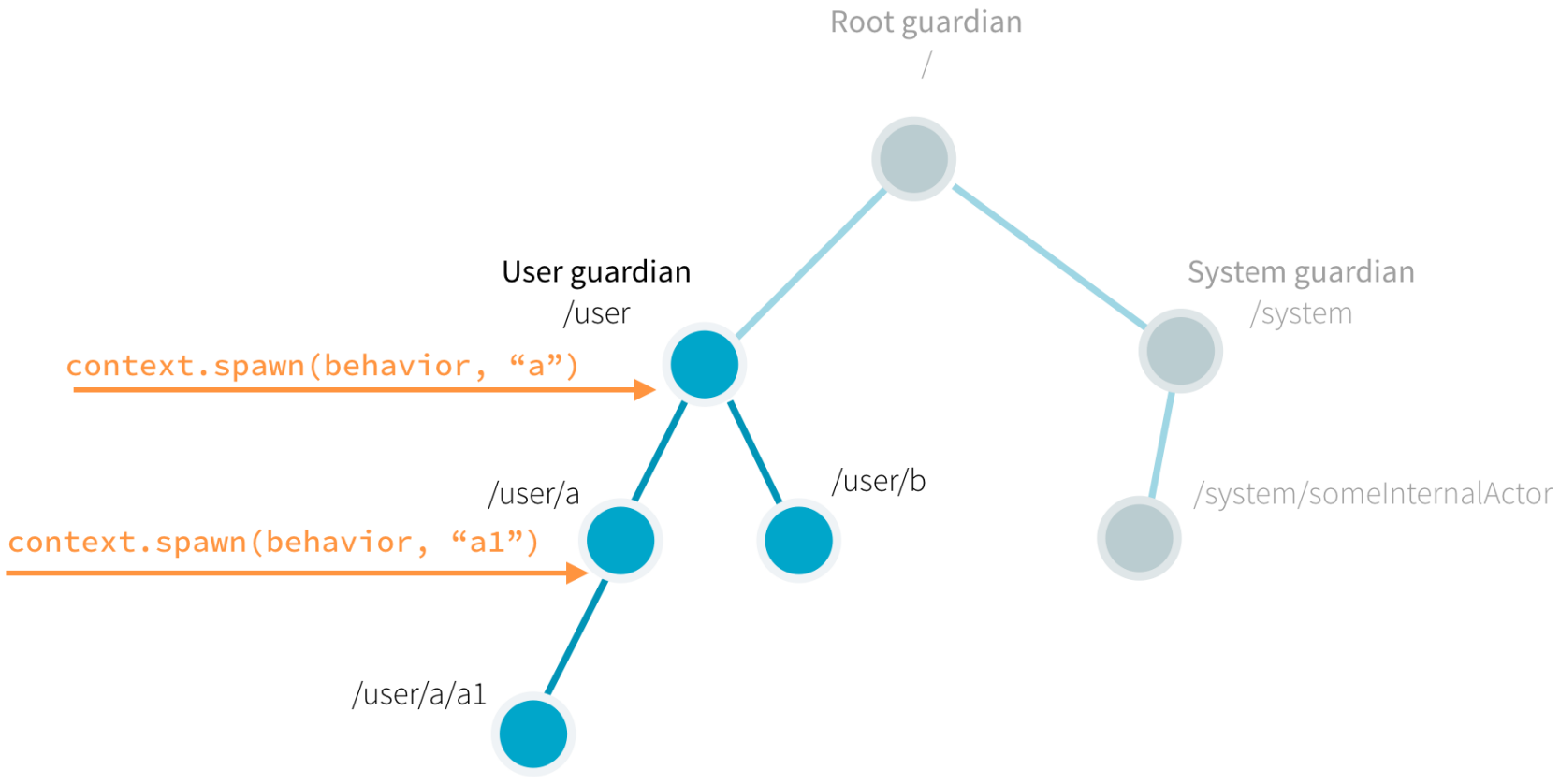
```
class S extends Actor {
  var d: Int = 0
  def receive: PartialFunction[Any, Unit] = {
    case a: Int =>     d += a
    case _: Unit =>    sender ! d
  }
}

object Test extends App {
  implicit val t: Timeout = Timeout(10,TimeUnit.SECONDS)
  val s = ActorSystem("s").actorOf(Props(classOf[S]))
  s ! 10
  s ! 5
  val q : Future[Int] = s ? ()
  q.map(println)
}
```

Edastamata sõnumid

- Sõnumid, mida ei saa edastada saadetakse aktorile `deadLetters`
- Sõnumid saab kätte importides `akka.actor.DeadLatter`.
- Võrguühenduse probleemide korral võivad sõnumid ka kaduma minna.

```
val as = ActorSystem("s")
as.eventStream.subscribe(minuDeadLetterAktor, classOf[DeadLetter])
```

Aktorite hierahia



- Valvurid (i.k. *Guardian*) vastutavad aktorite korrapärase sulgemise eest.

Perioodilised ja ajastatud sõnumid

```
import scala.concurrent.duration._
import system.dispatcher
...

// ajastatud sõnum
system.scheduler.scheduleOnce(50 milliseconds, testActor, "foo")

// korduv sõnum
val c = system.scheduler.scheduleOnce(50 milliseconds) {
  testActor ! System.currentTimeMillis
}

// tühistamine
c.cancle
```

Aktori peatamine

- stop

```
val actorSystem = ActorSystem("s")
val a = actorSystem.actorOf(Props[A]);
...
actorSystem.stop(a);
```

- PoisonPill

```
a ! PoisonPill
```

- gracefulStop

```
import akka.pattern.gracefulStop

val stopped: Future[Boolean] =
    gracefulStop(a, 2 seconds)(actorSystem)
```

Aktorite mudel

+ Mitmelõimelisusest tulenevad probleemid lahendatud.
+ Süsteemi hajusus kergelt muudetav.
+ Selge viis teatud ülesannete lahendamisel.

- Pole tüübitud
  - Akka Typed!

Akka Typed

```
object MinuAktor {
  def apply(): Behavior[String] =
    Behaviors.setup(context => new MinuAktor(context))
}

class MinuAktor(context: ActorContext[String])
  extends AbstractBehavior[String](context)
{
  override def onMessage(msg: String): Behavior[String] =
    msg match {
      case "start" =>
        println("bla")
        this
    }
}

object Main extends App {
  val testSystem = ActorSystem(MinuAktor(), "s")
  testSystem ! "start"
}
```

Akka Signaalid:

- ChildFailed,
  - erindi tõttu lapsaktoris
- PostStop, PreRestart, PreRestart, Terminated,
- DeleteEventsCompleted, DeleteEventsFailed,
- DeleteSnapshotsCompleted, DeleteSnapshotsFailed,
- RecoveryCompleted, RecoveryFailed,
- SnapshotCompleted, SnapshotFailed.

Aktor mis ei võta sõnumeid vastu

```
object Supervisor {
  def apply(): Behavior[Nothing] =
    Behaviors.setup[Nothing](context => new Supervisor(context))
}
class Supervisor(context: ActorContext[Nothing]) extends
   AbstractBehavior[Nothing](context)
{
  context.log.info("Application␣started")

  override def onMessage(msg: Nothing): Behavior[Nothing] = {
    // Pole vaja sõnumeid töödelda
    Behaviors.unhandled
  }

  override def onSignal: PartialFunction[Signal, Behavior[Nothing]] = {
    case PostStop =>
      context.log.info("Application␣stopped")
      this
  }
}
```