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
```

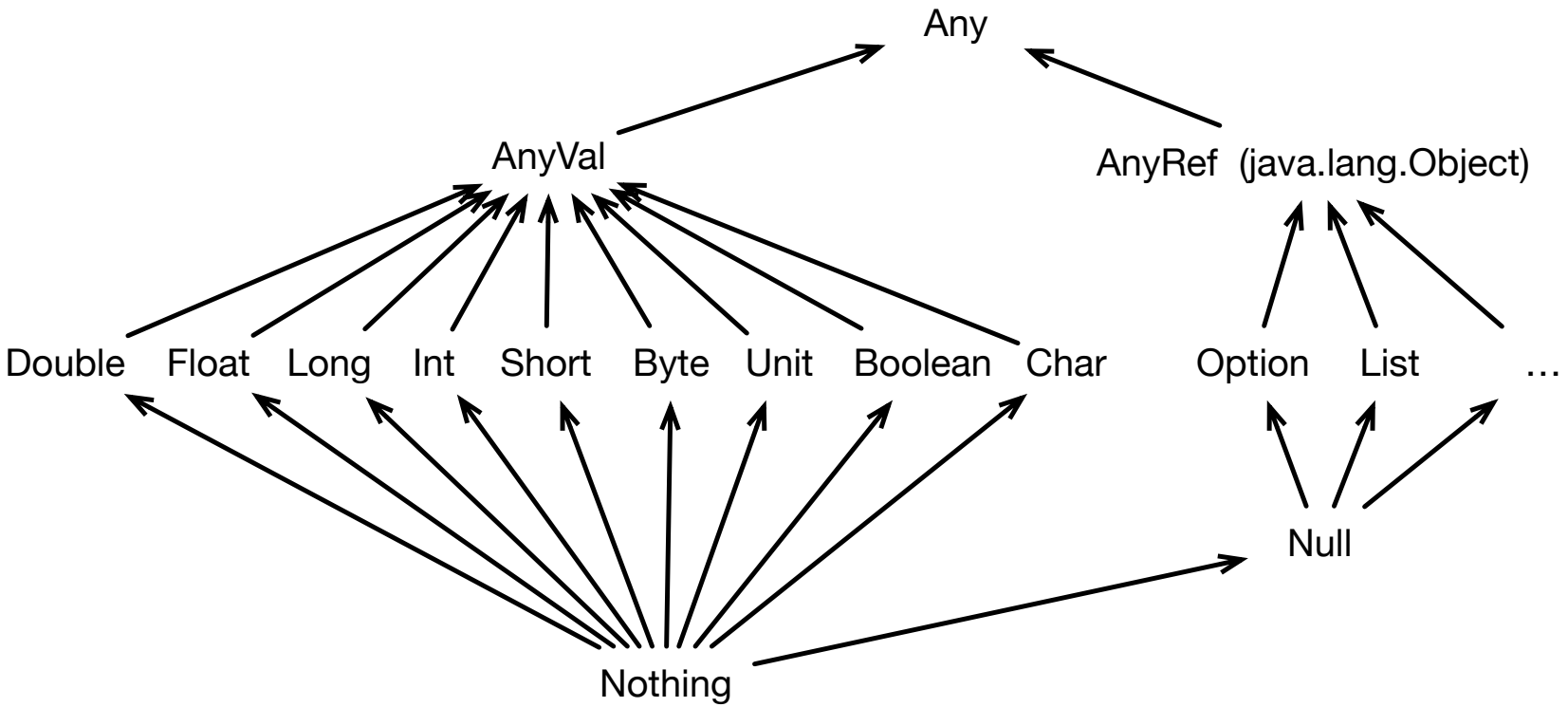
  - peavad leiduma `f :: α -> β` ja `g :: γ -> δ`
  - kuna rakendame `x` mõlemale, siis peab α `=` γ
  - kuna `(+) :: Num a =>a -> a -> a` siis β `=` δ ja `Num` α
  - s.t. `foo :: Num` β `=>` α `->` β

- (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

```scala
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.
- 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#
```

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.
  - `Priter[Koduloom]` saame kasutada kui nõutakse `Priter[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 { ... }
```

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

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`