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] = ...
  ...
}
```