- 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> res1.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: jäävad kaks 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`