Funkcionális programozás a gyakorlatban

Iván Szabolcs

2023 tavasz

Funkcionális programozás

- a C/C++, Java alapvetően imperatív nyelvek
  - változók, mutable data, mutable state, side-effects
  - for, while ciklusok, bejáró változók
  - utasítások
- a (pure) funkcionális nyelvekben
  - „változók” helyett „értékek”
  - „matematikai” függvények: az output csak az inputtól függ
  - nincs mellékhatás, globális state
  - fókusz: „mit”, nem „hogyan”
  - nyelvi szinten támogatott a függvény típus mint paraméter és mint visszatérési érték is
  - (általában) rövidebb kód
  - könnyebb reasoning a kódról, jobban automatizálható
  - könnyebb tesztelés
  - könnyebb párhuzamosítás (big data, blockchain)
  - kifejezések

a gyakorlatban

- A kurzuson – az általános paradigma megismerése mellett – a Scala nyelvet fogjuk használni
- JVM-en fut
- 100% Java-kompatibilis bytecodeot fordít
- hozzáférés a teljes Java osztálykönyvtárhoz
- rapid fejlesztés
- néha azért lesz más FP nyelv (pl Haskell) megfelelő is előadáson

típusok

egy funkcionális programozási nyelvben jellemzően vannak

- built-in alaptípusok
- ha σ és τ típusok, akkor $\sigma \rightarrow \tau$ egy függvénytípus
- σ típust vár, τ típust ad vissza

Scalában például van

- Int, Double, Float, Char, String típus
- input stringnek a hosszát visszaadó függvény típusa pl String $\Rightarrow$ Int típusú
- „hozzáadok egyet" függvény Int $\Rightarrow$ Int típusú
- „hozzáadok ötöt" függvény Int $\Rightarrow$ Int típusú
- „bejön az input n szám, visszaadom az „adj hozzá n-t" függvényt"

$$\text{Int} \Rightarrow (\text{Int} \Rightarrow \text{Int})$$

értékek

egyelőre Scalában mindent egy `object Main extends App{ ... }` belsejébe fogunk kódolni

erre még visszatérünk később

```
object Main extends App {
  val welcome: String = "Hello World" //value definition with a type
  println(welcome)                    //prints Hello World
}
```

- val – érték deklaráció
- print, println – konzolra kiírás
- „Hello World" – kifejezés

```
object Main extends App {
  val welcome = "Hello " + "World" //value with inferred type
  println(welcome)                 //prints Hello World
}
```

függvények

```
object Main extends App {
  def adjadOssze(x: Int, y: Int): Int = //függvény deklaráció
  {
    x + y //NINCS return
  }
  println(adjadOssze(3, 4)) // prints 7
}
```

- substitution model
- kifejezéssé átírás

$$\text{def } f(x_1 : T_1, \ldots, x_n : T_n) = M$$

$$f(a_1, .., a_n) \ \triangleright \ M[x_1/a_1, \ldots, x_n/a_n]$$

$$\text{adjadOssze}(3, 4) \ \triangleright \ 3 + 4 \ \triangleright \ 7$$

feltételek

```
def parosE(n: Int) = {                    //String is inferred
  if (n % 2 == 0) "Páros" else "Páratlan" //az Int nem Boolean
}
println( parosE(2) ) //prints Páros
```

- $\text{if}(b : \text{Boolean})E_1 \text{ else } E_2$

$$\text{if}(\text{true})E_1 \text{ else } E_2 \triangleright E_1$$

$$\text{if}(\text{false})E_1 \text{ else } E_2 \triangleright E_2$$

$$\frac{B \triangleright B'}{\text{if}(B)E_1 \text{ else } E_2 \ \triangleright \ \text{if}(B')E_1 \text{ else } E_2}$$

$\text{parosE}(2)$

$\text{if}(2\%2 == 0)$ "Páros" else "Páratlan"

$\text{if}(0 == 0)$ "Páros" else "Páratlan"

$\text{if}(\text{true})$ "Páros" else "Páratlan"

"Páros"

call by value vagy call by name

- CBV: előbb kiszámoljuk a kifejezés értékét, utána adjuk oda a függvénynek
- CBN: magát a kifejezést adjuk át, a függvény minden pontján kiértékeljük, ahol szükséges

Scala default: CBV. Hogy CBN legyen: $x : T$ helyett $x :\Rightarrow T$ a fejlécbe

```
def thisOrThat(test: Boolean, a: => String, b: => String) =
  if (test) a else b

def loop(): String = loop() // végtelen ciklus, ha kiértékeljük
                             // fordul () nélkül is
val result = thisOrThat(true, "First", loop())
println(result) //prints First
```

call by value vagy call by name

Ökölszabályok

- ha biztos szükség lesz a paraméter értékére: call by value
- ha legfeljebb egyszer és lehet, hogy nem is: call by name

rekurzió

A funkcionális paradigmára a rekurzió jellemző az iteráció helyett

```
def fib(n: Int): Int = //rekurzív függvénynek KELL type
  if (n < 2) n else fib(n - 1) + fib(n - 2)

println(fib(6)) // prints 8
```

$$
\begin{aligned}
\mathrm{fib}(3) &\triangleright \mathrm{if}(3 < 2)\ 3\ \mathrm{else}\ \mathrm{fib}(3-1) + \mathrm{fib}(3-2) \\
&\triangleright \mathrm{if}(\mathrm{false})\ 3\ \mathrm{else}\ \mathrm{fib}(3-1) + \mathrm{fib}(3-2) \\
&\triangleright \mathrm{fib}(3-1) + \mathrm{fib}(3-2) \\
&\triangleright \mathrm{fib}(2) + \mathrm{fib}(3-2) \\
&\triangleright \left(\mathrm{if}(2 < 2)\ 2\ \mathrm{else}\ \mathrm{fib}(2-1) + \mathrm{fib}(2-2)\right) + \mathrm{fib}(3-2) \\
&\triangleright \left(\mathrm{if}(false)\ 2\ \mathrm{else}\ \mathrm{fib}(2-1) + \mathrm{fib}(2-2)\right) + \mathrm{fib}(3-2) \\
&\triangleright \left(\mathrm{fib}(2-1) + \mathrm{fib}(2-2)\right) + \mathrm{fib}(3-2)
\end{aligned}
$$

telik a verem!

tail rekurzió

Ha a függvénytörzsben rekurzív hívás csak „végeredmény” pozícióban szerepel, akkor tail rekurzív az implementáció

hogy ez miért jó?

```
def sum(n: Int): Int =
  if (n <= 0) 0 else n + sum(n - 1)

println(sum(20000)) //StackOverflowError
```

```
sum(5) ▹ if( 5 <= 0 ) 0 else 5 + sum( 5-1 )
       ▹ if( false ) 0 else 5 + sum( 5-1 )
       ▹ 5 + sum( 5-1 )
       ▹ 5 + sum( 4 )
       ▹ 5 + { if( 4 <= 0 ) 0 else 4 + sum( 4-1 ) }
       ▹ 5 + { if( false ) 0 else 4 + sum( 4-1 ) }
       ▹ 5 + { 4 + sum( 4-1 ) } //note: a kapcsos marad, amíg nincs kiértékelve a belső
       ▹ 5 + { 4 + sum( 3 ) }
       ▹ 5 + { 4 + { if( 3 <= 0 ) 0 else 3 + sum( 3-1 ) } }
       ▹ 5 + { 4 + { if( false ) 0 else 3 + sum( 3-1 ) } }
       ▹ 5 + { 4 + { 3 + sum( 3-1 ) } } //látványosan ,,hízik'' a kifejezés
       ▹ 5 + { 4 + { 3 + sum( 2 ) } }
       ▹ 5 + { 4 + { 3 + { if( 2 <= 0 ) 0 else 2 + sum( 2-1 ) } } }
       ▹ 5 + { 4 + { 3 + { if( false ) 0 else 2 + sum( 2-1 ) } } }
       ▹ 5 + { 4 + { 3 + { 2 + sum( 2-1 ) } } }
       ▹ 5 + { 4 + { 3 + { 2 + sum( 1 ) } } }
       ▹ 5 + { 4 + { 3 + { 2 + { if( 1 <= 0 ) 0 else 1 + sum( 1-1 ) } } } }
       ▹ 5 + { 4 + { 3 + { 2 + { if( false ) 0 else 1 + sum( 1-1 ) } } } }
       ▹ 5 + { 4 + { 3 + { 2 + { 1 + sum( 1-1 ) } } } }
       ▹ 5 + { 4 + { 3 + { 2 + { 1 + sum( 0 ) } } } }
       ▹ 5 + { 4 + { 3 + { 2 + { 1 + { if( 0 <= 0 ) 0 else 1 + sum( 0-1 ) } } } } }
       ▹ 5 + { 4 + { 3 + { 2 + { 1 + { if( true ) 0 else 1 + sum( 0-1 ) } } } } }
       ▹ 5 + { 4 + { 3 + { 2 + { 1 + { 0 } } } } } // finally, térhetünk vissza
       ▹ 5 + { 4 + { 3 + { 2 + { 1 + 0 } } } }
       ▹ 5 + { 4 + { 3 + { 2 + { 1 } } } }
       ▹ 5 + { 4 + { 3 + { 2 + 1 } } }
       ▹ 5 + { 4 + { 3 + { 3 } } }
       ▹ 5 + { 4 + { 3 + 3 } }
       ▹ 5 + { 4 + { 6 } }
       ▹ 5 + { 4 + 6 }
       ▹ 5 + { 10 }
       ▹ 5 + 10
       ▹ 15
```

Tail rekurzívan

```
// tailSum(n, acc) visszaadja 1+2+...+n + acc -ot
def tailSum(n: Int, acc: Int): Int =
  if (n <= 0) acc else tailSum(n - 1, acc + n)

def tailSum(n: Int): Int = tailSum(n, 0)

println(tailSum(20000)) //prints 200010000
```

```
tailSum(5) ▹ tailSum(5,0)
  ▹ if( 5<=0 ) 0 else tailSum(5-1, 0+5)
  ▹ if( false ) 0 else tailSum(5-1, 0+5)
  ▹ tailSum(5-1, 0+5)
  ▹ tailSum(4, 0+5)    //első argumentum kiértékelve
  ▹ tailSum(4, 5)      //második argumentum kiértékelve
  ▹ if( 4<=0 ) 5 else tailSum(4-1, 5+4)
  ▹ if( false ) 5 else tailSum(4-1, 5+4)
  ▹ tailSum(4-1, 5+4)
  ▹ tailSum(3, 5+4)
  ▹ tailSum(3, 9)
  ▹ if( 3<=0 ) 9 else tailSum(3-1, 9+3)
  ▹ if( false ) 9 else tailSum(3-1, 9+3)
  ▹ tailSum(3-1, 9+3)
  ▹ tailSum(2, 12)
  ▹ if( 2<=0 ) 12 else tailSum(2-1, 12+2)
  ▹ if( false ) 12 else tailSum(2-1, 12+2)
  ▹ tailSum(2-1, 12+2)
  ▹ tailSum(1, 12+2)
  ▹ tailSum(1, 14)
  ▹ if( 1<=0 ) 14 else tailSum(1-1, 14+1)
  ▹ if( false ) 14 else tailSum(1-1, 14+1)
  ▹ tailSum(1-1, 14+1)
  ▹ tailSum(0, 14+1)
  ▹ tailSum(0, 15)
  ▹ if( 0<=0 ) 15 else tailSum(0-1, 15+0)
  ▹ if( true ) 15 else tailSum(0-1, 15+0)
  ▹ 15
```

nesting

```
def tailSum(n: Int): Int = {
  @tailrec //annotáció, compile error lesz, ha mégse tail rekurzív
  def tailSum(n: Int, acc: Int): Int =
    if (n <= 0) acc else tailSum(n - 1, acc + n)
  tailSum(n, 0)
}
println(tailSum(20000)) //still prints 200010000
```

- scopeon belül deklarált függvények, értékek,... nem láthatóak kintről
- ha több kifejezés van egymás után egy {}-blokkban, akkor az egész blokk értéke az utolsó kifejezés értéke lesz

tuple

```
typedef struct IntPair = {
  int x;
  int y;
} IntPair;

IntPair pair;
pair.x = 3;
pair.y = 4;
```

```
val (x,y) = (3,4)
val (x1: Int, y1: Int) = (5,6)
val (x2,y2): (Int, Int) = (7,8)
val pair = (9,10)
val pair2: (Int, Int) = (11,12)
println(pair._1) // prints 9
val (anInt, aString) = (42, "Sanyi")
```

match: switch on steroids
(not even in its final form)

```
// fizzBuzz
val pair = (n % 7, n % 10)
pair match {
  case (0, 7) => "FizzBuzz"
  case (_, 7) => "Fizz"
  case (0, _) => "Buzz"
  case _ => n.toString
}
```

iteratív ciklusból tail rekurzió

```
int x = 3;
int y = 1;
for (int i = 0; i < n; i++) {
  x = x + y;
  y = y * 2 + 1;
}
...
```

```
@tailrec
def forLoop(x: Int, y: Int, i: Int): (Int, Int) =
  if (i >= n) (x, y)
  else forLoop(x + y, y * 2 + 1, i + 1)
val (x, y) = forLoop(3, 1, 0) // ebbe a tuple-ba kerül a "végérték"
```

countPrimes

```
int countPrimes(int a, int b) {
  int count = 0;
  for (int i = a; i <= b; i++ ) {
    if (isPrime(i)) count++;
  }
  return count;
}
```

```
def countPrimes(a: Int, b: Int): Int = {
  def loop(i: Int, count: Int): Int =
    if (i > b) count
    else if (isPrime(i)) loop(i + 1, count + 1)
    else loop(i + 1, count)
  loop(a, 0)
}
```

scalában szerencsére általában nem kell ilyet kézzel csinálni, hanem. . .

range, for comprehension, foreach

…van Range típus!

```
val szamok = 1 to 10 // 1,2,...,10
val kevesebbSzamok = 1 until 10 // 1,2,...,9
val lepeskozzel = 1 to 10 by 4 //1, 5, 9

// "for comprehension": for side effects only
for (i <- lepeskozzel) {
  println(i) //prints 1, 5 and then 9
} // Unit, always yields ()
// this also works ofc
for (i <- 1 to 10 by 4) {
     println(i)
}
// compiles into
(1 to 10 by 4).foreach(println)
```

és foreach:

- kap egy `Int => Any` függvényt,
- azt meghívja a Range összes elemén,
- az eredményeket eldobja
- végeredmény Unit lesz
- a for comprehension egy kényelmes syntax foreachelni

Vector[T]

- feldolgozandó értékek jöhetnek egy „tömbben” is
- T helyére mehet a konkrét típus, hogy milyen elemek vannak benne
- immutable!

```
val data: Vector[Int] = Vector[Int](1,4,2,8,5,7)
// ha a generic típust ki tudja következtetni, nem kell odaírjuk
val data = Vector[Int](1,4,2,8,5,7)
val data = Vector(1,4,2,8,5,7)

// ezen is van foreach
def printVector(input: Vector[Int]) =
  for (i <- input) println(i)
printVector(data) //prints 1, 4, 2, 8, 5, 7
// megy így is
data.foreach(println)
```

mit tud még pl a Vector[T]

```
val data = Vector(1,4,2,8,5,7)
data :+ 9 // Vector(1,4,2,8,5,7,9), új
8 +: data // Vector(8,1,4,2,8,5,7), új (syntaxról később)
data ++ Vector(3,5) // Vector(1,4,2,8,5,7,3,5)
data(3) // 8
data.contains(8) // true
data.size // 6
data.sum // 27 -- nem minden típusra fordul, Intre igen
data.max // 8 -- same, note: üres Vector()-ra error!
data.min // 1 -- same
data.drop(2) // Vector(2,8,5,7), dropja az első két elemet
data.take(2) // Vector(1,4), első két elem egy vektorban
data.dropRight(2) // Vector(1,4,2,8)
data.takeRight(2) // Vector(5,7)
```

filter

- predikátum: Boolean kimenetű függvény

```
val data = Vector(1,4,2,8,5,7)
def isEven(i: Int) = i % 2 == 0 // ez Boolean

data.filter(isEven) //Vector(4,2,8)
data.filterNot(isEven) //Vector(1,5,7)
data.partition(isEven) //(Vector(4,2,8), Vector(1,5,7)) tuple

data.filter(i => i % 2 == 0) //megy így is
data.filter( _ % 2 == 0) //megy így is, ha egyszer használjuk

// írjuk ki a data-beli páros számokat
data.filter(isEven).foreach(println) //ok
for(i <- data) if(isEven(i)) println(i) //same, don't
for(i <- data.filter(isEven)) println(i) //same, don't
for(i <- data if isEven(i)) println(i) //same, ok
//last two compile to the first
```

van más is, ami predikátumot kaphat

```
val data = Vector(1,4,2,8,5,7)
def isEven(i: Int) = i % 2 == 0

data.count(isEven) //3, ennyire igaz
data.exists(isEven) //true, van amire igaz
data.forall(isEven) //false, nem mindre igaz

data.filter(isEven).size //don’t
data.filter(isEven).size > 0 //don’t
data.filterNot(isEven).isEmpty //don’t
```

map

- map: ha `Vector[T]`, akkor kap egy `f:T => U` függvényt
- `U` lehet bármilyen típus
- kiértékeli `f`-et az összes elemére a vektornak — eddig mint a foreach
- az eredményeket egy `Vector[U]`-ban visszaadja

```
val data = Vector(1,4,2,8,5,7)
def dup(n: Int) = n * 2

data.map(dup) //Vector(2,8,4,16,10,14)
data.map( i => i * 2 ) //same with lambda
data.map( _ * 2 ) //same

data.map( i => i.toString ) //Vector("1","4","2","8","5","7")

val mappedData = for (i <- data) yield i * 2 //Vector(2,8,4,16,10,14)
// compiles to map
```

van más is, ami függvényt kap

```
val tuples = Vector((1,4), (2,8), (5,7)) // Vector[(Int,Int)]

tuples.max // (5,7) -- tuplét lexikografikusan rendez by def, ha tud
tuples.maxBy( tuple => tuple._2) // (2,8)
tuples.maxBy( _._2 ) // same

tuples.sorted // Vector((1,4),(2,8),(5,7)), új ez is
tuples.sortBy( _._2 ) // Vector((1,4),(5,7),(2,8))
```

zip

- összerak egy Vector[T]-t és egy Vector[U]-t egy Vector[(T,U)]-ba
- amelyik hosszabb, levágja a végét

```
val data = Vector(1,4,2,8,5,7)
val names = Vector("a","b","c")

names.zip(data) //Vector(("a",1),("b",4),("c",2))

data.zip(0 until data.size)
//Vector((1,0),(4,1),(2,2),(8,3),(5,4),(7,5)) - indexek!

data
  .zip(0 until data.size) //indexekkel
  .maxBy( _._1 ) //adat szerinti max: (8,3)
  ._2 // 3 -- a maximális elem indexe

data.zip(data.drop(1))
// Vector((1,4), (4,2), (2,8), (8,5), (5,7))
```

Set

van még más konténer is, pl.

- List – van headje ami egy elem és tailje ami a maradék lista
- Set – halmaz, egyenlő elemből csak egy van benne
- String – karakterkonténerként is működik

```
val aSet = Set(10,20,30,10,20,30,40,50)
println(aSet)
// HashSet(10, 20, 50, 40, 30)
```

toSet, toList, toVector

```
"sanyiferitibi" count (_ == 'i') // 4
"sanyiferitibi" toSet
//  HashSet(e, s, n, y, t, f, a, i, b, r)
aSet.toVector
// Vector(10, 20, 50, 40, 30)
```

„struct" – „szorzat típus"

```
case class Vektor2D(x: Int, y: Int) //és kész is, két adattag
// immutable
def add(v1: Vektor2D, v2: Vektor2D) = //Vektor2D is inferred
  Vektor2D(v1.x + v2.x, v1.y + v2.y) // van "getter"

val u = Vektor2D(0, 1) // nincs new
val v = u.copy(x = 3) // Vektor2D(3, 1)
val v2 = u.copy(y = 3, x = 1) // Vektor2D(1, 3)
val w = add(u, v)

println(w) // prints Vektor2D(3, 2)
val z = Vektor2D(3, 2)
println(w == u) //prints true !
```

- egyenlőség „okos", érték szerint hasonlít össze
- println is pretty-printed

match

Javában egy ősosztályt kapó függvényben ha alosztályra akarunk szűkíteni, az kb. így megy:

```
class Pet // a Pet.java -ban
class Dog extends Pet ... void woof() // a Dog.java -ban
class Cat extends Pet ... void meow() // a Cat.java -ban

void saySomething(Pet pet) { // valami .java-ban
  if (pet instanceof Cat) {
    Cat cat = (Cat)pet;
    cat.meow();
  } else if (pet instanceof Dog) {
    Dog dog = (Dog)pet;
    dog.woof();
  } else {
    System.out.println("Unknown pet type: " + pet);
  }
}
```

trait („unió”, „összeg típus”) + match

Scalában ugyanez:

```
// mehet minden egy fileba
trait Pet // kb "interface"
case class Dog() extends Pet { // a Dog az egy Pet, nincs adattagja
  def woof() = println("Woof")
}
case class Cat() extends Pet { // a Cat az egy Pet
  def meow() = println("Meow")
}

def saySomething(pet: Pet) = pet match {
  case cat: Cat => cat.meow()
  case dog: Dog => dog.woof()
  case _ => println(s"Unknown pet type $pet")
}

saySomething(Cat()) // prints "Meow"
```

match

```
case class Cat(name: String, age: Int)

def morcostKeresem(cat: Cat) = cat match {
  case Cat("Morcos", x) => println(s"Megvan Morcos! $x éves")
  case _ => println("Ez nem Morcos")
}

morcostKeresem(Cat("Morcos", 3)) //Megvan Morcos! 3 éves
morcostKeresem(Cat("Nyuszi", 4)) //Ez nem Morcos

def kismacska(cat: Cat) = cat match {
  case Cat(_, x) if x < 3 => true // if guard
  case _ => false
}
kismacska(Cat("Morcos", 3)) // false
kismacska(Cat("Pihe", 1)) // true
```

trait methods

```
trait Pet {
  def name: String // parameterless method:
  def speak: String // side-effect: (), no side-effect: no ()
  def legs: Int = 4 // if not implemented, this is the default
}
case class Cat(name: String, age: Int) extends Pet { // name itt, ok
   override def speak = "Meow"
}
case class Dog(name: String, age: Int) extends Pet {
   override val speak = "Woof" // lehet val is
}
// val-t deffel nem overrideolhatunk
case class Fish(name: String) extends Pet {
   override def speak = ""
   override def legs = 0
}
```

match, sealed trait

```
def doSomethingWithMyPet(pet: Pet) = pet match {
  case Cat("Morcos", _) => println("Megvan Morcos!")
  case Dog(_, x) if x < 4 => println("Találtam egy kiskutyát")
  case dog: Dog => println("Találtam egy kutyát")
  case Cat(_, 1) => println("Kismacska")
  case _ => println(s"Egy ${pet.name} névre hallgató jószág")
}
```

```
sealed trait Pet // csak ugyanebben a file-ban lehet extendelni
case class Cat() extends Pet // same file
case class Dog() extends Pet
case class Gothi() extends Pet
// így biztos, hogy senki nem származtat le belőle máshol még valamit
// és (ha nincs if guard) próbálja ellenőrizni a matchek teljességét
```

Vissza a kollekciókhoz

collect

```
val pets = Vector(Cat("Morcos",3), Dog("Kutya",10), Cat("Pihe",1))
pets.collect { case cat: Cat => cat.name }
// Vector("Morcos","Pihe")
pets :+ Dog("másik kutya", 20) collect { // jó ez pont nélkül is
  case cat: Cat => cat.name
  case Dog(_,x) if x > 15 => "Vén kutya"
}
// Vector("Morcos", "Pihe", "Vén kutya")
```

reduce, foldLeft

```
val numbers = Vector(1,4,2,8,5,7)

numbers.reduce( (x,y) => x * y ) // 2240

numbers.filter( _ > 10 ).reduce( _ * _ )
// throws UnsupportedOperationException, runtime

numbers.foldLeft(1)( _ * _ ) // 2240
numbers.filter( _ > 10 ).foldLeft(1)( _ * _ ) // 1
```

- reduce csak ugyanolyan típus lehet, mint a kollekció alap elemei
- foldLeft bármilyen típusba tud aggregálni

foldLeft

```
val pets = Vector(Cat("Morcos",3), Dog("kutya",10), Cat("Pihe",1))

pets.foldLeft(0)( (sum, pet) => sum + pet.age ) //14
pets.foldLeft("")( (names,pet) => names + pet.name ) // MorcoskutyaPihe
pets.foldRight("")( (pet,names) => names + pet.name ) //PihekutyaMorcos
```

case object – singleton

Paraméter nélküli case classt minek példányosítsunk sokszor? Úgyis minden példánya teljesen egyforma

```
sealed trait IntLista
case object UresLista extends IntLista
case class NemuresLista(head: Int, tail: IntLista) extends IntLista

def printLista(list: IntLista) : Unit = list match {
   case UresLista => ()
   case NemuresLista(x, UresLista) => print(x)
   case NemuresLista(x, t) => print(x); print(","); printLista(t)
}
```

def, val, lazy val

```
case class Vektor(x: Double, y: Double) {
   val length_val = Math.sqrt(this.x * this.x + this.y * this.y)
   def length_def = Math.sqrt(this.x * this.x + this.y * this.y)
   lazy val length_lazy_val = Math.sqrt(this.x * this.x + this.y * this.y)
}
val v = Vektor(1.0, 1.0)
println(v.length_def)    //prints 1.4142135623730951
println(v.length_val)    //prints 1.4142135623730951
println(v.length_lazy_val) //prints 1.4142135623730951
```

- val: konstruáláskor megkapja az értéket, eltárolja egy mezőben
- def: minden egyes híváskor újra kiszámítja az értéket és visszaadja, nincs mező
- lazy val: az első híváskor számolja ki az értéket, ekkor eltárolja egy mezőben és a következőnél ezt adja vissza

def, val, lazy val

```
case class Vektor(x: Double, y: Double) {
  val length_val = { println("val"); Math.sqrt(this.x * this.x + this.y * this.y) }
  def length_def = { println("def"); Math.sqrt(this.x * this.x + this.y * this.y) }
  lazy val length_lazy_val = {
    println("lazy val"); Math.sqrt(this.x * this.x + this.y * this.y)
  }
}
val v = Vektor(1.0, 1.0) //val
println("def: " + v.length_def) //def
println("val: " + v.length_val)
println("lazy val: " + v.length_lazy_val) //lazy val
println("def: " + v.length_def) //def
println("val: " + v.length_val)
println("lazy val: " + v.length_lazy_val)
println("def: " + v.length_def) //def
println("val: " + v.length_val)
println("lazy val: " + v.length_lazy_val)
```

- Ha nincs mellékhatás, akkor () nélkül deklaráljuk, bármi is
- Ha később mégis átírjuk egyikről a másikra, a hívó kódban nem kell változzon semmi

def with args

```
case class Vektor(x: Double, y: Double) {
   def plus(that: Vektor) = Vektor(this.x + that.x, this.y + that.y)
}
val v1 = Vektor(1.0, 2.0)
val v2 = Vektor(3.0, 4.0)
println( v1.plus( v2 ) ) //instead of addOssze(v1, v2). prints Vektor(4.0,6.0)
println( v1 plus( v2 ) ) //ha kimarad a pont, azt SOKSZOR oda tudja helyezni a fordító
println( v1 plus v2 )  //na ez már kinéz valahogy. Ha unáris => SOKSZOR nem kell zárójel
```

```
case class Vektor(x: Double, y: Double) {
   def +(that: Vektor) = Vektor(this.x + that.x, this.y + that.y)
}
val v1 = Vektor(1.0, 2.0)
val v2 = Vektor(3.0, 4.0)
println( v1 + v2 )  //cool syntax
```

Actually, a $3 + 4$ is $3. + (4)$ az Int osztályban

Legnagyobb Mazsola

Java:

```
Mazsola legnagyobb( Mazsola[] mazsolak ) {
   Mazsola top = null;
   for( Mazsola current: mazsolak ) {
      if( current == null ) continue;
      if( top == null || top.meret < current.meret ) top = current;
   }
   return top;
}
```

null checkek mindenhol

Option

```
trait Option[A]
case object None[A] extends Option[A] // does not compile actually
case class Some[A](value: A) extends Option[A]
```

pattern matchelni is lehet épp rá

```
def legnagyobb(mazsolak: Vector[Option[Mazsola]]): Option[Mazsola] =
mazsolak
.fold[Option[Mazsola]](None)((acc, current) => (acc, current) match {
   case (None, _) => current
   case (_, None) => acc
   case (Some(top), Some(curr)) if (top.meret < curr.meret) => current
   case _ => acc
})
```

ez így tkp az imperatív kód átirata folddá, működik, de nem idiomatikus

Option

```scala
trait Option[A] {
   def foreach[B](f: A => B): Unit
   def filter(p: A => Boolean): Option[A]
   def map[B](f: A => B): Option[B]
}
case object None[A] extends Option[A] {
   override def foreach[B](f: A => B): Unit = ()
   override def map[B](f: A => B): Option[B] = None[B]
   override def filter(p: A => Boolean) = None
   // note: does not compile
}
case class Some[A](value: A) extends Option[A] {
   override def foreach[B](f: A => B): Unit = { f(value); () }
   override def map[B](f: A => B): Option[B] = Some(f(value))
   override def filter(p: A => Boolean) = if (p(value)) this else None
}
```

mint egy 0 vagy 1 elemű lista

Enumerátorokat egymásba ágyazhatsz

pontosvesszővel elválasztva

```
for (
i <- 1 to 10;
j <- 1 to i
) println(i + " * " + j + " = " + (i * j))
```

vagy kapcsosban, pontosvessző nélkül („this is the way”)

```
for {
   i <- 1 to 10
   j <- 1 to i
} println(s"$i * $j = ${i * j}") //string interpolátor
```

`(1 to 10) foreach { i => (1 to i) foreach { j => println(s"$i * $j = ${i * j}") } }`

map egymásba ágyazva nem type checks

ez fordul. de miért ez lesz?

```
val mapmap = for {
  i <- 1 to 10
  j <- 1 to i
} yield (i * j)
println(mapmap) //prints Vector(1,2,4,3,6,9,...)
```

hiszen ha mapokat rakunk egymásba, az eredmény egy List of List kéne legyen

```
val mapmap2 = (1 to 10).map(i => (1 to i).map(j => (i * j)))
println(mapmap2) //prints Vector(Vector(1), Vector(2, 4), Vector(3, 6, 9),...
```

well, mert ha több enumerátor van egy for-yield kifejezésben, akkor csak az utolsó lesz map, a többi…

flatMap

$\text{List}[T].\text{flatMap}[U](f : T \Rightarrow \text{List}[U]) : \text{List}[U]$

- végigmegy a listán (amiben T típusú elemek vannak), minden e elemére kiértékeli $f(e)$-t
- ami egy-egy lista lesz (U típusú elemeké)
- és ezeket az eredménylistákat egy listává fűzi össze (U típusú elemekévé)

```
val mapmap2 = (1 to 10).flatMap(i => (1 to i).map(j => (i * j)))
println(mapmap2) //prints Vector(1, 2, 4, 3, 6, 9,...
```

$\text{for } (x_1 \leftarrow c_1; x_2 \leftarrow c_2; \ldots; x_n \leftarrow c_n) \text{ yield } E(x_1, \ldots, x_n)$

nem más, mint

$c_1.\text{flatMap}(x_1 \;\Rightarrow\; c_2.\text{flatMap}(x_2 \;\Rightarrow \ldots \;\Rightarrow\; c_n.\text{map}(x_n \;\Rightarrow\; E(x_1, \ldots, x_n))))$

actual egymásba ágyazott „ciklusok”, minden kombinációra kiértékeli a kifejezést és az eredményeket egyetlen hosszú „lapos” listában adja vissza

Option

```
def legnagyobb(mazsolak: Vector[Option[Mazsola]]): Option[Mazsola] = {
  val actualMazsolak =
    for {
      mazsolaOption <- mazsolak
      mazsola <- mazsolaOption
    } yield mazsola
  if (actualMazsolak.isEmpty) None
  else Some(actualMazsolak.maxBy(_.meret))
}
```

for-yield comprehension különböző konténerek közt: a külsőé lesz a típus

Option

```
for {
   mazsolaOption <- mazsolak
   mazsola <- mazsolaOption
} yield mazsola
```

```
mazsolak
.flatMap( mazsolaOption => mazsolaOption.map( mazsola => mazsola ))
```

```
mazsolak
.flatMap( mazsolaOption => mazsolaOption )
```

```
mazsolak.flatten
```

Option

Ha egy Vector[Option[T]]-ből Vector[T]-t akarunk készíteni, arra a flatten metódus lesz való:

```
def legnagyobb(mazsolak: Vector[Option[Mazsola]]): Option[Mazsola] = {
  val actualMazsolak = mazsolak.flatten
  if (actualMazsolak.isEmpty) None
  else Some(actualMazsolak.maxBy(_.meret))
}
```

```
def legnagyobb(mazsolak: Vector[Option[Mazsola]]): Option[Mazsola] =
   mazsolak.flatten.maxByOption(_.meret)
   // this is the way
```

Az Optionnek van még toList, toVector metódusa is.

Map

- Map[Key,Value] generikus típus, kulcs-érték párokat tartalmaz („asszociatív tömb”)

```
val counts: Map[Char, Int] = Map( 'a' -> 7, 'b' -> 8, 'c' -> 10, 'a' -> 9 )
println( counts ) //prints Map(a -> 9, b -> 8, c -> 10)
```

```
counts.get('a') // Some(9)
counts.get('d') // None
counts('a') // 9
counts('d') // NoSuchElementException
counts.getOrElse('a', 0) // 9, same as get('a').getOrElse(0)
counts.getOrElse('d', 0) // 0
counts.contains('a') // true
counts.contains('d') // false
```

Map (van SortedMap is)

```
counts.updated('a', 10) // Map(a -> 10, b -> 8, c -> 10)
counts.removed('b') // Map(a -> 9, c -> 10)
Map('a' -> 3, 'b' -> 5) ++ Map('b' -> 6, 'c' -> 9)
// Map('a' -> 3, 'b' -> 6, 'c' -> 9)
```

```
filter(p: ((A, B)) => Boolean): Map[A, B]
map[A2, B2](f: ((A, B)) => (A2, B2)): Map[A2, B2]
foreach[C](f: ((A, B)) => C): Unit
flatMap[A2, B2](f: ((A,B)) => Map[A2, B2]): Map[A2, B2]
```

ezek úgy viselkednek, ahogy pl. egy Vector[(A,B)]-n elképzelné az ember

```
val counts = Map( 'b' -> 8, 'a' -> 7, 'c' -> 10 )
val hop = "abbabaca"
println( hop.map( counts ) ) //ArraySeq(7, 8, 8, 7, 8, 7, 10, 7)
```

Map, Set, Vector

```
val keys = Vector("egy", "kettő", "három")
val values = Vector("one", "two", "three")
val pairs = keys zip values
val theMap = pairs.toMap
// Map(egy -> one, kettő -> two, három -> three)
theMap.keySet
// Set(egy, kettő, három)

val map2 = theMap.withDefaultValue("sanyi")
map2("egy") // one
map2("kilenc") // sanyi

val map3 = theMap.withDefault(key => key + " " + key)
map3("egy") // one
map3("kilenc") // kilenc kilenc
```

Map, Set, Vector

Kicsit összetettebb, de hasznos függvények:

```
val data = Vector("egy", "kettő", "három", "négy", "öt", "hat")
data.groupBy(_.length)
// HashMap(5 -> Vector(kettő, három), 2 -> Vector(öt), 3 ->
    Vector(egy, hat), 4 -> Vector(négy))
```

```
data.groupMap(_.length)(_.reverse)
// Map(2 -> Vector(tö), 3 -> Vector(yge, tah), 4 -> Vector(ygén), 5 ->
    Vector(őttek, moráh))
```

groupBy(f) == groupMap(f)(identity)

```
data.groupMapReduce(_.length)(identity)((left, right) => left + "," +
    right)
// Map(2 -> öt, 3 -> egy,hat, 4 -> négy, 5 -> kettő,három)
```

curry

```
def f1(x: Int, y: Int): Int = x + y

def f2(x: Int): Int => Int = {
  y => x + y
}

def f3(x: Int)(y: Int): Int = x + y

f1(2,3) // 5
f1(2,4) // 6

val g = f2(2) // note: val!
g(3) // 5
g(4) // 6

val h = f3(2) _ // _ nélkül: hiányolja a függvényhívás argumentumát
h(3) // 5
h(4) // 6
```

partially applied functions, type inference, defaults

```
def multiply(x: Double)(y: Double): Double = x * y
def tax: Double => Double = multiply(1.27)
// or: def tax = multiply(1.27) _
tax(100.0) //127.0
```

```
def fold[B](init: B)(op: (B,A) => B): B
fold(0)( _ + _ ) // ok

def fold[B](init: B, op: (B,A) => B): B
fold(0, _ + _ ) // does not compile till Scala 3
fold[Int](0, _ + _ ) // ok
```

```
def f(x: Int, y: Int = 2 * x) = x + y // does not compile
def f(x: Int)(y: Int = 2 * x) = x + y // ok, call f(3)(4) or f(3)()
```

impliciteknél ide még visszatérünk később

multi varargs

```
def f(x: Int*) // kb x: Array[Int]
= x.sum

f(1,2,3,4) // 10
```

vararg csak az utolsó paraméter lehet

```
def f(x: Int*)(y: Int*) = x.sum * y.sum //mindkettő utolsó lol
f(1,2,3,4)(5,6,7) // 180
```

`apply` nevű metódus: az objektumot függvényként lehet hívni

```
case object mySizeOf {
   def apply(s: String) = s.length
}

println( mySizeOf("dinnye") ) //prints 6
```

Ha a Scala fordító lát egy `expression()` alakú kifejezést, ahol `expression` egy objektum, akkor abból egy `expression.apply()` hívás lesz. (Aminek deklarálunk `apply` metódust, minden olyan helyre odaadhatjuk, ami ilyen típusú függvényt vár.)

```
val strings = Vector("one", "two", "three")
strings.map(mySizeOf)
// Vector(3, 3, 5)
```

mert ez az `apply: String => Int`, ezért `Vector[Int]` lesz

Companion object

Lehet egyszerre valamit osztályként és objektumként is definiálni:

```
trait MyList
case object Ures extends MyList
case class Nemures(head: Int, tail: MyList) extends MyList

object MyList {
   def apply() = Ures
   def apply(a: Int) = Nemures(a, Ures)
   def apply(a: Int, b: Int): Nemures = Nemures(a, Nemures(b, Ures))
   def apply(a: Int, b: Int, c: Int): Nemures = Nemures(a, MyList(b,c))
   // vagy
   def apply(ints: Int*) = ints.foldLeft(Ures: MyList)( (acc,elem) => Nemures(elem,acc))
}

println(MyList(1,4,2,3)) // prints Nemures(1,Nemures(4,Nemures(2,Nemures(3,Ures))))
```

kb. a companion objectbe szervezhetjük ki, ami Javában static lenne

Egyelőre tehát azt tudjuk, hogy a generic $\text{List}[T]$ osztálynak van pl. egy konstruktora, ami T-ből készít $\text{List}[T]$-t.

private

```scala
case class Pont(private val x: Int, y: Int) {
   def +(that: Pont) = Pont(this.x + that.x, this.y + that.y)
   private def *(c: Int) = Pont(c * x, c * y)
   def double = *(2)
}

val p = Pont(1,0)
val q = Pont(0,1)
println(p.x) // nem fordul
println(p.y)
println(p + q)
println(p.double)
println(p * 2) // nem fordul
```

constructors

```
case class Pont(x: Int, y: Int)

case object Pont {
  def apply(x: Int, y: Int): Pont = // Pont(2,1) // infinite loop
    new Pont(x,y) // OK
  def apply(x: Int): Pont = Pont(x,x)
}

val p = Pont(1,0) //Pont(1,0)
val q = Pont(3) //Pont(3,3)
```

constructor param check + fix

```
case class Pont(x: Int, y: Int)

case object Pont {
  def apply(x: Int, y: Int): Pont =
    new Pont(0 max x ,0 max y)
    // ha nem tudjuk értelmesen fixelni a paramétert,
    // dobhatunk kivételt is, ha problémás
    // IllegalArgumentException pl jó erre
  def apply(x: Int): Pont = Pont(x,x)
}

val p = Pont(-1, 10) // Pont(0, 10)
val q = Pont(0, -1) // Pont(0, 0)
val r = Pont(3) // Pont(3, 3)
val t = Pont(-2) // Pont(0, 0)
val s = new Pont(-1, 10) // Pont(-1, 10)!
```

constructor param check + fix

```
case class Pont private (x: Int, y: Int)

case object Pont {
  def apply(x: Int, y: Int): Pont =
    new Pont(0 max x, 0 max y)
  def apply(x: Int): Pont = Pont(x,x)
}

val p = Pont(-1, 10) // Pont(0, 10)
val q = Pont(0, -1) // Pont(0, 0)
val r = Pont(3) // Pont(3, 3)
val t = Pont(-2) // Pont(0, 0)
val s = new Pont(-1, 10) // nem fordul, good
new Vector(1,2,3) // ez se fordul btw
```

generics

```
trait List[T]
case class Ures[T]() extends List[T] //case object nem lehet generic..ezt még megoldjuk
case class Nemures[T](head: T, tail: List[T]) extends List[T]

val list: List[Int] = Nemures(4,Nemures(2,Nemures(1,Ures()))) //! Int is inferred
val list = Nemures(4,Nemures(2,Nemures(1,Ures()))) //!! Int is inferred
```

```
trait List[T] {
   def map[U](f: T=>U): List[U] //this keeps getting better and better: generic method!!
   def filter(p: T=>Boolean): List[T]
}
case class Ures[T]() extends List[T] {
   override def map[U](f: T=>U) = Ures[U]()
   override def filter(p: T=>Boolean) = Ures[T]()
}
case class Nemures[T](head: T, tail: List[T]) extends List[T] {
   override def map[U](f: T=>U) = Nemures(f(head), tail map f)
   override def filter(p: T=>Boolean) =
   if(p(head)) Nemures(head, tail filter p) else tail filter p
}
val list = Nemures(4, Nemures(3, Nemures(1, Ures()))) //ok, List[Int]
println(list map { _.toBinaryString }) //ok, List[String], ("100","11","1")
println(list filter { _ > 3 }) // ok, List[Int], (4)
```

List[T]

A built-in List[T] osztály generic, van benne map, filter, reduce, foldLeft és foldRight is. Egyetlen case object van az üres listára, a Nil, ami minden típusú üres listának megfelel.

A Nemures osztály neve pedig :: mmint az a neve, hogy „::"

```
val list = 1 :: 4 :: 2 :: 8 :: 5 :: 7 :: Nil // ez kérem List[Int] lesz akkor
val list2 = "sanyi" :: Nil //ez meg List[String]
println( list map {_.toBinaryString } ) // List("1", "100", "10", "1000", "101", "111")
println( list filter { _ > 5 } ) // List(8,7)
println( list reduce { _+_ } ) // 27
println( list match {
   case Nil => "üres"
   case head :: tail => "nemüres"
})
```

Hogy miért is felelhet meg a `Nil` minden `T` típusra `List[T]`-nek, azt nemsokára meglátjuk

$\text{List}[T]$ konstruktor

Lehet listát így is példányosítani:

```
val list = List(1,4,2,8,5,7)
val list2 = 1 :: 4 :: 2 :: 8 :: 5 :: 7 :: Nil
println(list == list2) //prints true
```

Nézzük csak meg, hogy ezek miért is működnek

Syntax sugar: kettőspont balra ragasztja a metódus nevét

A kettőspontra végződő nevű metódusokat nem ponttal az objektum után, hanem simán az objektum elé írjuk...

tehát ez a két alak ugyanaz:

```
val list2 = 1 :: 4 :: 2 :: 8 :: 5 :: 7 :: Nil
val list3 = Nil.::(7).::(5).::(8).::(2).::(4).::(1)
```

well, ok.

és a „nemüres" osztályt is ugyanúgy ::-nek hívják, mint a metódust, ami egy ::-t ad vissza. ok

egyébként van a List osztálynak ::: nevű metódusa, ami listákat fűz össze, pl. $\mathrm{List}(1,4) ::: \mathrm{List}(2,8) = \mathrm{List}(1,4,2,8)$.

java arrays

```java
void setToZero(Object[] array, int i) {
   array[i] = new Integer(42);
}
String[] stringArray = new String[]{"egy", "kettő"};
setToZero(stringArray, 0);
// ???
```

guess:

- nem fordul le
- stringArray [Integer(42), "kettő"] lesz, String[]
- stringArray [Integer(42), "kettő"] lesz, Object[]
- valami más

java arrays

```
void setToZero(Object[] array, int i) {
   array[i] = new Integer(42);
}
String[] stringArray = new String[]{"egy", "kettő"};
setToZero(stringArray, 0);
// ???
```

guess:

- nem fordul le
- stringArray [Integer(42), "kettő"] lesz, String[]
- stringArray [Integer(42), "kettő"] lesz, Object[]
- valami más ArrayStoreException

Javában ha C alosztálya D-nek, akkor a C[] is alosztálya D[]-nek

`C <: D  =>  C[] <: D[]`

és ez gondokat tud okozni

scala arrays

```scala
trait A
case class B(value: Int) extends A

def f(array: Array[A]) = array foreach println

val arrayB = Array(B(3), B(5))
f(arrayB)
```

guess?

scala arrays

```
trait A
case class B(value: Int) extends A

def f(array: Array[A]) = array foreach println

val arrayB = Array(B(3), B(5))
f(arrayB)
```

guess?

nem fordul le

Scalában ha `C <: D`, attól még `Array[C]` NEM `Array[D]`

scala vectors

```
trait A
case class B(value: Int) extends A

def f(vector: Vector[A]) = vector foreach println

val vectorB = Vector(B(3), B(5))
f(vectorB)
```

guess?

scala vectors

```
trait A
case class B(value: Int) extends A

def f(vector: Vector[A]) = vector foreach println

val vectorB = Vector(B(3), B(5))
f(vectorB)
```

guess?

prints B(3), then B(5)

Scalában ha `C <: D`, akkor `Vector[C] <: Vector[D]`

variance

A `C[A]` generikus típus…

- kovariáns, ha `A <: B => C[A] <: C[B]`
- kontravariáns, ha `A <: B => C[B] <: C[A]`
- invariáns, ha `A <: B` nem ad hierarchiát `C[A]` és `C[B]` között

Pl. Scalában az Array invariáns, a Vector, List, Option kovariáns

variance

```
trait Opsn[A] {
   def map[B](f: A => B): Opsn[B]
}
case class Szam[A](value: A) extends Opsn[A] {
   override def map[B](f: A => B): Opsn[B] = Szam(f(value))
}
case class Nan[A]() extends Opsn[A] {
   override def map[B](f: A => B): Opsn[B] = Nan[B]()
}
trait Pet {
   def name: String
}
case class Cat(name: String) extends Pet
case class Dog(name: String) extends Pet

def f(petOption: Opsn[Pet]): Opsn[String] = petOption.map(_.name)
val catOption = Szam(Cat("Tom"))

f(Szam(Cat("Tom"))) // OK, Szam("Tom")
f(catOption) // nem fordul, az Opsn[Cat] az NEM Opsn[Pet]
```

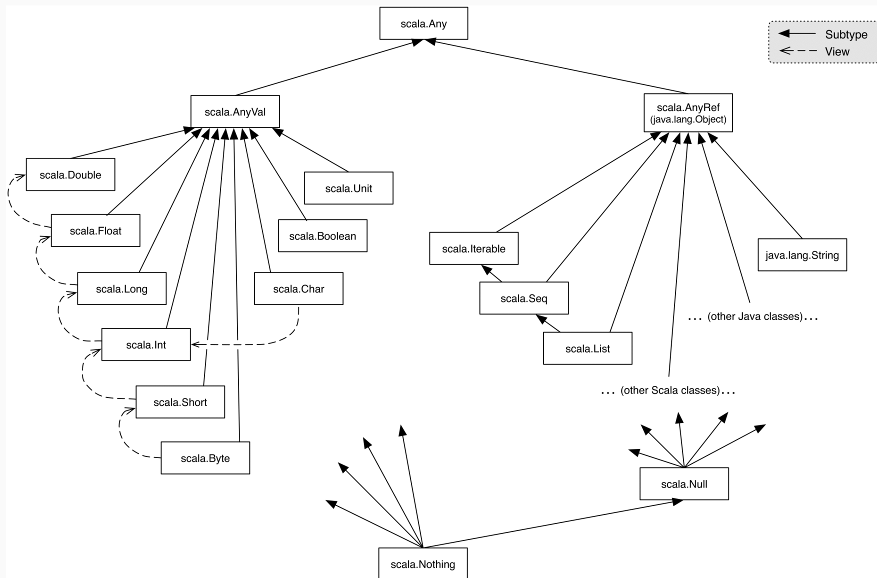
variance

```
trait Opsn[+A] { // +A: A-ban a típus legyen kovariáns
   def map[B](f: A => B): Opsn[B]
}
case class Szam[A](value: A) extends Opsn[A] {
   override def map[B](f: A => B): Opsn[B] = Szam(f(value))
}
case class Nan[A]() extends Opsn[A] {
   override def map[B](f: A => B): Opsn[B] = Nan[B]()
}
trait Pet {
   def name: String
}
case class Cat(name: String) extends Pet
case class Dog(name: String) extends Pet

def f(petOption: Opsn[Pet]): Opsn[String] = petOption.map(_.name)
val catOption = Szam(Cat("Tom"))

f(Szam(Cat("Tom"))) // OK, Szam("Tom")
f(catOption) // OK, Opsn[Cat] <: Opsn[Pet]
```

Scala types



(source: scala-lang.org)

variance

```
trait Opsn[+A] { // +A: A-ban a típus legyen kovariáns
   def map[B](f: A => B): Opsn[B]
}
case class Szam[A](value: A) extends Opsn[A] {
   override def map[B](f: A => B): Opsn[B] = Szam(f(value))
}
case object Nan extends Opsn[Nothing] {
   // object még mindig nem lehet generikus, de ez így nem is az
   override def map[B](f: Nothing => B): Opsn[B] = this
}
trait Pet {
   def name: String
}
case class Cat(name: String) extends Pet
case class Dog(name: String) extends Pet

def f(petOption: Opsn[Pet]): Opsn[String] = petOption.map(_.name)
val catOption = Szam(Cat("Tom"))

f(Szam(Cat("Tom"))) // OK, Szam("Tom")
f(catOption) // OK, Opsn[Cat] <: Opsn[Pet]
f(Nan) // OK, Nan extends Opsn[Nothing] <: Opsn[Pet], mert kovariáns
```

variance

nem mindent lehet egyből kovariánssá tenni

```
trait Opsn[+A] { // +A: A-ban a típus legyen kovariáns
  def map[B](f: A => B): Opsn[B]
  def getOrElse(default: => A): A
}
// nem fordul
```

„covariant type A occurs in contravariant position in type ⇒A of value default”

- Ha $A <: B$ és $C <: D$, akkor nem $A \Rightarrow C <: B \Rightarrow D$
- ahol egy $\text{Pet} \Rightarrow \text{Pet}$ függvényt várunk, oda
  - jó egy $\text{Pet} \Rightarrow \text{Cat}$ függvény
  - nem jó egy $\text{Pet} \Rightarrow \text{Any}$ függvény
  - nem jó egy $\text{Cat} \Rightarrow \text{Any}$ függvény
  - nem jó egy $\text{Cat} \Rightarrow \text{Pet}$ függvény
  - jó egy $\text{Any} \Rightarrow \text{Cat}$ függvény

A függvények a bejövő argumentumaikban kontravariánsak, a kimenőben kovariánsak:

ha $A <: B$ és $C <: D$, akkor $B \Rightarrow C \ <: \ A \Rightarrow D$

variance – type upper bounds

```
trait Opsn[+A] { // +A: A-ban a típus legyen kovariáns
   def map[B](f: A => B): Opsn[B]
   def getOrElse[B >: A](default: => B): B //upper bound
}

case class Szam[A](value: A) extends Opsn[A] {
   override def map[B](f: A => B): Opsn[B] = Szam(f(value))
   override def getOrElse[B >: A](default: => B) = value
}
case object Nan extends Opsn[Nothing] {
   override def map[B](f: Nothing => B): Opsn[B] = this
   override def getOrElse[B](default: => B) = default
}
trait Pet {
   def name: String
}
case class Cat(name: String) extends Pet
case class Dog(name: String) extends Pet

def f(petOption: Opsn[Pet]): Pet = petOption.getOrElse(Cat("Omega"))
val catOption = Szam(Cat("Tom"))

f(Szam(Cat("Tom"))) // OK,Cat(Tom)
f(catOption) // OK,Cat(Tom)
f(Nan) //OK, Cat(Omega)
```

variance

- a `map[B](f: A => B): Opsn[B]`-vel miért nem volt gond?
- mert az egy $(A \Rightarrow B) \Rightarrow B$ metódus
- A itt egy függvény bal oldalának a bal oldalán van, két kontravariáns lépés kovariáns lépés lesz
- Ha $A <: C$, akkor $C \Rightarrow B \;\; <: \;\; A \Rightarrow B$ és ezért $(A \Rightarrow B) \Rightarrow D \;\; <: \;\; (C \Rightarrow B) \Rightarrow D$
- szerencsére a fordító segít, ha kovariáns paramétert kontravariáns pozícióba (vagy fordítva) teszünk

java stream API

```
List<Ruha> ruhak = new ArrayList<>();
ruhak.add( new Ruha("piros", 40));
ruhak.add( new Ruha("zöld", 42));
ruhak.add( new Ruha("piros", 38));
ruhak.add( new Ruha("fehér", 42));
ruhak.add( new Ruha("fehér", 40));
ruhak.add( new Ruha("zöld", 42));

Stream<Ruha> ruhaStream = ruhak.stream();
ruhaStream.forEach(ruha -> System.out.println(ruha));
// vagy: ruhaStream.forEach(System.out::println);

ruhaStream
.filter( ruha -> ruha.getMeret() == 40)
.forEach( System.out::println );
// IllegalStateException: stream has already been operated upon or
    closed
```

java stream API

```
List<Ruha> ruhak = new ArrayList<>();
ruhak.add( new Ruha("piros", 40));
ruhak.add( new Ruha("zöld", 42));
ruhak.add( new Ruha("piros", 38));
ruhak.add( new Ruha("fehér", 42));
ruhak.add( new Ruha("fehér", 40));
ruhak.add( new Ruha("zöld", 42));

ruhak
.stream()
.filter( ruha -> ruha.getMeret() == 40)
.forEach( System.out::println);

//OK, piros/40 és fehér/40
```

java stream API

```
public static void szures(List<Ruha> ruhak, Predicate<Ruha> p) {
   ruhak
   .stream()
   .filter(p)
   .foreach(System.out::println);
}

szures(ruhak, ruha -> ruha.getMeret() == 40); //OK, ez egy Predicate
// Ruha => bool
Predicate<Ruha> negyvenes = ruha -> ruha.getMeret() == 40; //így is
szures(ruhak, negyvenes);
```

```
ruhak
.stream()
.filter(ruha -> ruha.getSzin().equals("piros"))
.forEach(ruha -> ruha.setMeret(ruha.getMeret() + 1));
// visszahat, nem klónoz
```

java stream API

```
ruhak
.stream()
.filter(negyvenes)
.map(Ruha::getSzin()) // Stream<String> lesz
.forEach(System.out::println)
```

```
Set<String> negyvenesRuhakSzinei =
ruhak
.stream() //Stream<Ruha>
.filter(negyvenes) //Stream<Ruha>
.map(Ruha::getSzin) //Stream<String>
.collect(Collectors.toSet()); //Set<String>
// van még pl. Collectors.toList() is
```

java stream API

```
// átlagosnál kisebb ruhákat tegyük egy listába
long darab = ruhak.stream().count();
int osszeg =
ruhak
.stream()
.map(Ruha::getMeret) // Stream<Integer>
.reduce(0,(x,y) -> x + y); //Integer, BinaryOperator<Integer>
double atlag = (double)osszeg / darab;
List<Ruha> kisruhak =
ruhak
.stream()
.filter(ruha -> ruha.getMeret() < atlag)
.collect(Collectors.toList());
```

```
.reduce(0, Integer::sum) //Integer, BinaryOperator<Integer>
.mapToInt(Ruha::getMeret).sum();
```

java stream API

```
// első piros ruhát írjuk ki
Optional<Ruha> elsoPiros = ruhak
.stream()
.filter(ruha -> ruha.getSzin().equals("piros"))
.findFirst();

if (elsoPiros.isEmpty()) {
   System.out.println("Nincs piros ruha!");
} else {
   System.out.println("Az első piros ruha: "+ elsoPiros.get());
}
```

```
String szoveg =
elsoPiros // Optional<Ruha>
.map(ruha -> "az első: " + ruha) // Optional<String>
.orElse("Nincs!") // String
```

java stream API

```
// ruha színeket Stringbe összefűzve
String szinek = ruhak
.stream()
.collect(
  () -> new StringBuilder(), //Supplier
  // vagy: StringBuilder::new,
  (acc, ruha) -> { acc.append(ruha.getSzin()); }, //BiConsumer
  (acc1, acc2) -> { }acc1.append(acc2); } //BiConsumer
)
.toString();
```

```
ruhak
.stream()
.map(Ruha::getSzin)
.collect(Collectors.joining(",","(",")")); //mkString("(",",",")")
```

java stream API

```
long szavazas(String[] szavazatok) {
   return Arrays.stream(szavazatok)
   .filter(word -> "lovasszekér".equals(word)).count();
   // vagy: "lovasszekér"::equals
}
int countCsakSzamok(String[] texts) {
   return Arrays.stream(texts)
   .filter( word -> word.matches("\\d*")).count();
}
Map<String, Integer> gyakorisagok(String szoveg) {
   Arrays.stream(szoveg.split(" "))
   .collect(
   HashMap::new,
   (acc, word) -> { acc.put(word, acc.geOrDefault(word, 0) + 1 );}
   (acc1, acc) -> {}
   );
}
```

java stream API

```
String leghosszabbMacskas(String[] szovegek) {
   return Arrays.stream(szovegek)
   .filter( word -> word.toLowerCase().contains("macska"))
   .max( //kb. a maxBy
   Comparator.comparingInt(String::length) //kb. az Ordering.by
   // vagy (left, right) -> left.length() - right.length()
   ).orElse(null);
}
```

SortedSet

```
import scala.collection.immutable.SortedSet //ez már nincs a Predefben

val theSet = SortedSet(1,4,2,8,5,7)

println( theSet )              //TreeSet(1, 2, 4, 5, 7, 8)
println( theSet filter { _ > 3 } ) //TreeSet(4, 5, 7, 8)
println( theSet map { _ * 2 } ) //TreeSet(2, 4, 8, 10, 14, 16)
println( theSet flatMap { x => Set(x, 2*x) } )
  // TreeSet(1, 2, 4, 5, 7, 8, 10, 14, 16)
```

```
val stringSet = SortedSet()(
  (x: String, y: String) => x.length - y.length
)
val plusSet = stringSet ++ Vector("dinnye", "alma", "körte")
println( plusSet ) //TreeSet(alma, körte, dinnye)
println( plusSet.contains("sanyi") ) //true!!!
```

Try[+T]

```
import scala.util.Try
val tryE: Try[A] = Try( e ) // típus persze inferelhető
```

e call by name paraméter

```
trait Try[+A]
case class Success[+A](value: A) extends Try[A]
case class Failure[+A](problem: Throwable) extends Try[A]

object Try {
  def apply[A](r: => A): Try[A] = {
    try Success(r) catch {
      case NonFatal(e) => Failure(e)
    }
  }
}
```

Try[+T]

amit pl. nem kap el egy Try doboz:

- VirtualMachineError
- ThreadDeath
- InterruptedException
- LinkageError

..ezeket vsz jó ötlet is nem elkapni

Try[+T]

```
.foreach[B]( f: A => B ): Unit
 // Success(value): f(value)
 // Failure(problem): ()

.map[B]( f: A => B ): Try[B]
 // Success(value): Try(f(value))
 // Failure(problem): Failure[B](problem)

.flatMap[B]( f: A => Try[B] ): Try[B]
 // Success(value): f(value)
 // Failure(problem): Failure[B](problem)

.isSuccess: Boolean
.isFailure: Boolean
.get: A //failure throws problem
```

Try[+T]

```
.filter(p: A => Boolean): Try[A]
 // Success(value): if p(value) this else
     Failure(NoSuchElementExcepction)
 // Failure(problem): this

.getOrElse(default: => A): A
 // Success(value): value
 // Failure(problem): default

.orElse(default: => Try[A]): Try[A]
 // Success(value): this
 // Failure(problem): try default catch { e: NonFatal => Failure(e) }

.toOption: Option[A]
 // Success(value): Some(value)
 // Failure(problem): None
```

Try[+T]

```
.recover[B >: A](pf: PartialFunction[Throwable, B]): Try[B]
// Success(value): this
// Failure(problem): ~ Try[B](pf(problem)) if defined or this if not

.recoverWith[B >: A](pf: PartialFunction[Throwable, Try[B]]): Try[B]
// Success(value): this
// Failure(problem) ~ pf(problem) or this

.transform[B](s: A => Try[B], f: Throwable => Try[B]): Try[B]
// Success(value): s(value) // this flatmap s
// Failure(problem): f(problem) // in a box

.fold[B](fa: Throwable => B, fb: A => B): B
// Success(value): fb(value), if ex -> fa(ex)
// Failure(problem): fa(problem)
```

Try[+T]

```
def toInt(s: String): Try[Int] = Try {
  Integer.parseInt(s.trim)
}
val a = toInt("1") // Success(1)
val b = toInt("nem") // Failure(NFE)

toInt(x) match {
  case Success(i) => println(i)
  case Failure(s) => println(s"failed: $s")
}

val y = for {
  a <- toInt(stringA)// use .getOrElse for defaulting if you wish so
  b <- toInt(stringB)
  c <- toInt(stringC)
} yield a + b + c
// y is either Success(a+b+c) or the first Failure
// do some code here, no need for "finally"s
```

Future[A]

```scala
import scala.concurrent.Future
import scala.concurrent.ExecutionContext.Implicits.global //laters

[..code..]
val futureResult = Future( [..time-consuming computation..] )
[..code..]
```

- ha a fő szál terminál, akkor a futó Future-ök is

Future[A]

```scala
import scala.concurrent.Future
import scala.concurrent.ExecutionContext.Implicits.global //laters
import scala.annotation.tailrec

@tailrec
def mainLoop(): Unit = { // I/O miatt illik a ()
  println("welcome text")
  val line = StdIn.readline() // String
  line.trim.toUpperCase match {
    case "Q" => println("bye"); ()
    case _ => Future(println(processInput(line))); mainLoop()
  }
}
```

így a UI nem fog blokkolni

Future[A]

ez is monád

```
.map[B](f: A = B): Future[B]
  // ha kész lesz, mappeli az eredményt, szintén Future-ben

.foreach[B](f: A => B): Unit
  // ha kész lesz, applyolja rajta f-et, szintén Futute-ben

.flatMap[B](f: A => Future[B]): Future[B]
  // ha kész lesz, flatmapi az eredményt (külső doboz nélkül)

for (result <- Future(processInput(line))) {
  println(result)
}
[..code..] // does not get blocked
```

par

```
val v: Vector[Int] = [...]
val parV = v.par // now it's parallel

for (value <- v) { // the usual stuff
  doStuffWith(value)
}
for (value <- parV) { // the (nondet) parallel stuff
  doStuffWith(value) // on desktop: use all them cores
}
```

működik: Range, Vector etc.

van alapból: ParHashMap, ParHashSet, ParIterable, ParMap, ParRange, ParSeq, ParSet, ParVector, ...

scala.collection.parallel.immutable._

par

```
.filter
.map
.foreach
.flatMap

.sum
.max
.min

.reduce[B >: A](op: (B,B) => B) //!!! associativity
```

view

```
val longVector = (1 to 10000000).toVector

val start = System.currentTimeMillis()

longVector
  .map( _ * 2 )
  .map( _ + 1 )
  .filter( _ % 3 == 0 )
  .map( _.toBinaryString )
  .find( _.startsWith("1010") )
  .foreach(println)

val end = System.currentTimeMillis()

println(s"Elapsed time: ${end - start} ms")

// 1292ms
```

view

```
val longVector = (1 to 10000000).toVector

val start = System.currentTimeMillis()

longVector
  .view
  .map( _ * 2 )
  .map( _ + 1 )
  .filter( _ % 3 == 0 )
  .map( _.toBinaryString )
  .find( _.startsWith("1010") )
  .foreach(println)

val end = System.currentTimeMillis()

println(s"Elapsed time: ${end - start} ms")

// 12ms
```

algebrai adattípus

```
trait List[+T]
case object Nil extends List[Nothing]
case class ::[T](head: T, tail: List[T]) extends List[T]

trait Tree
case class Leaf(data: Int) extends Tree
case class Inner(left: Tree, data: Int, right: Tree) extends Tree
```

egy trait „összeg típus”: az értéktartománya az őt extendelő osztályok értéktartományainak (diszjunkt) uniója

egy case class „szorzat típus”: az értéktartománya a mezői értéktartományainak direkt szorzata

algebrai adattípus

ha egy adattípus-rendszert fel lehet építeni az elemi adattípusokból (Int, Char, Double...) úgy, hogy minden résztvevő típus a többiek (és az elemiek) összege vagy szorzata (kombinálni ér), akkor ők algebrai adattípusok

```
Tree = Int + Tree x Int x Tree
List[T] = 1 + T x List[T] // 1: üres szorzat -- case object
Nothing = 0 // üres összeg -- nincs bele tartozó objektum
Option[T] = 1 + T     Try[T] = Throwable + T     Either[A,B] = A + B
```

```
Tree = Leaf + Inner
Leaf = Int
Inner = Tree x Int x Tree // szét tudjuk szedni

List[T] = 1 + ::[T]
::[T] = T x List[T] // mind vagy összeg, vagy szorzat
```

Scalában az összeg típust traittel, a szorzatot case classal oldhatjuk meg

algebrai adattípus

Egy algebrai adattípus

- ha szorzat, akkor a mezői immutable mezők legyenek
- az értéktartományába tartozó objektumok mindig reprezentálhatók faként
- ha egy függvénnyel feldolgozzuk, akkor
    - matchelünk rá
    - ha összeg típus, akkor az összeg minden típusára lesz egy case
    - ha szorzat, akkor az adattagjait bindeljük a pattern változóira
    - jellemzően rekurzívan
    - Scalában ha van értelme, tagfüggvényekkel is lehet

comparable / ordered

Ha az algebrai adattípusaink „külső" típusai amiket használunk, mind rendezhetőek, akkor az algebrai adattípusunk is az lehet:

```
trait Tree extends Ordered[Tree]

case class Leaf(data: Int) extends Tree {
   override def compare(right: Tree): Int = right match {
      case _: Inner => -1 // a levél mondjuk kisebb
      case Leaf(data) => this.data - that.data
   }
}
case class Inner(left: Tree, data: Int, right: Tree) extends Tree {
   override def compare(right: Tree): Int = right match {
      // másik case class: a case classok egy fix sorrendjében
      case _: Leaf => 1 // az Inner nagyobb, mint a Leaf
      case Inner(left2, data2, right2) =>
         // adattagonként lexikografikusan
         (left, data, right) compare (left2, data2, right2)
   }
```

LazyList

(régebben Stream)

```
LazyList[T] = 1 + T x (=> LazyList[T]) // kb.
```

így lehet a lista akár „végtelen” is

```
val fibs: LazyList[BigInt] =
  BigInt(0) #:: // ez a lazy list :: -je
  BigInt(1) #::
  fibs.zip(fibs.tail).map { n => n._1 + n._2 }
// note: List-tel ez NPE
println(fibs) // LazyList(<not computed>)
val fibStart = fibs take 5
println(fibStart) // LazyList(<not computed>)
for (i <- fibStart) println(i) // 0, 1, 1, 2, 3
println(fibStart) // LazyList(0, 1, 1, 2, 3)
println(fibs) // LazyList(0, 1, 1, 2, 3, <not computed>)
```

LazyList

```
val randoms: LazyList[Int] = Random.nextInt() #:: randoms
for (i <- randoms take 5) print(i + " ")
for (i <- randoms take 5) print(i + " ")
// -1261526257 -1261526257 -1261526257 -1261526257 -1261526257
// -1261526257 -1261526257 -1261526257 -1261526257 -1261526257
```

```
val ints = LazyList.from(0)
for (i <- ints take 5) println(i) // 0 1 2 3 4
val randoms = LazyList.continually(Random.nextInt()) // => T
for (i <- randoms take 3) println(i)
// -1113209229 874937190 -1865170664
for (i <- randoms take 3) println(i)
// -1113209229 874937190 -1865170664
```

LazyList

```
val rand = LazyList.continually(Random.nextInt())
val randWithI = rand zip LazyList.from(1)
val firstDiv10 = randWithI.find(pair => pair._1 % 10 == 0)
println(firstDiv10)
// Some((-963703010,28)) -> a 28. random szám volt az első 0-ra végződő

val randPlus = rand.map(i => i + 1)
for(i <- randPlus take 5) println(i) // kiír ötöt

val odds = rand.count(i % 2 == 1) // infinite loop
```

Monád

egy monád egy M generikus osztály,

- $M[T]$-nek van egy $\mathrm{unit}(value : T) : M[T]$ metódusa
(ez általában egy apply metódus egy companion objectben: pl. a List-nél a $\mathrm{List}(3)$ a 3 Int-ből készít egy egyelemű $\mathrm{List}[\mathrm{Int}]$et)
- és egy $\mathrm{flatMap}[U](f:\ T \Rightarrow M[U]) :\ M[U]$ metódusa
(a List $\mathrm{flatMap}$-je pont ilyen szignatúrájú: kap egy függvényt, ami minden elemhez rendel egy listát, és ezeknek a listáknak az összefűzöttjét adja vissza)
- amik teljesítik a monád axiómákat:

$$\mathrm{unit}(x).\mathrm{flatMap}(f) = f(x) \qquad \text{(balegység)}$$

$$m.\mathrm{flatMap}(\mathrm{unit}) = m \qquad \text{(jobbegység)}$$

$$m.\mathrm{flatMap}(f).\mathrm{flatMap}(g) = m.\mathrm{flatMap}(x \Rightarrow f(x).\mathrm{flatMap}(g)) \qquad \text{(asszociativitás)}$$

A List egy monád. Van még sok.