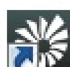
Bevezetés

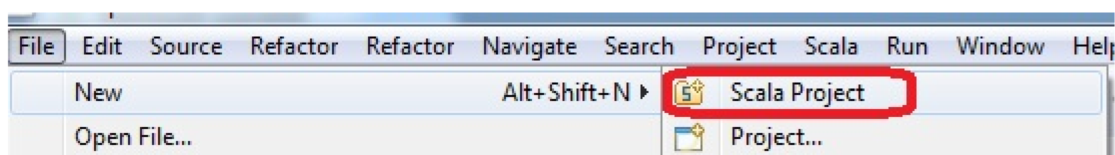
hogyne

Programozási környezet

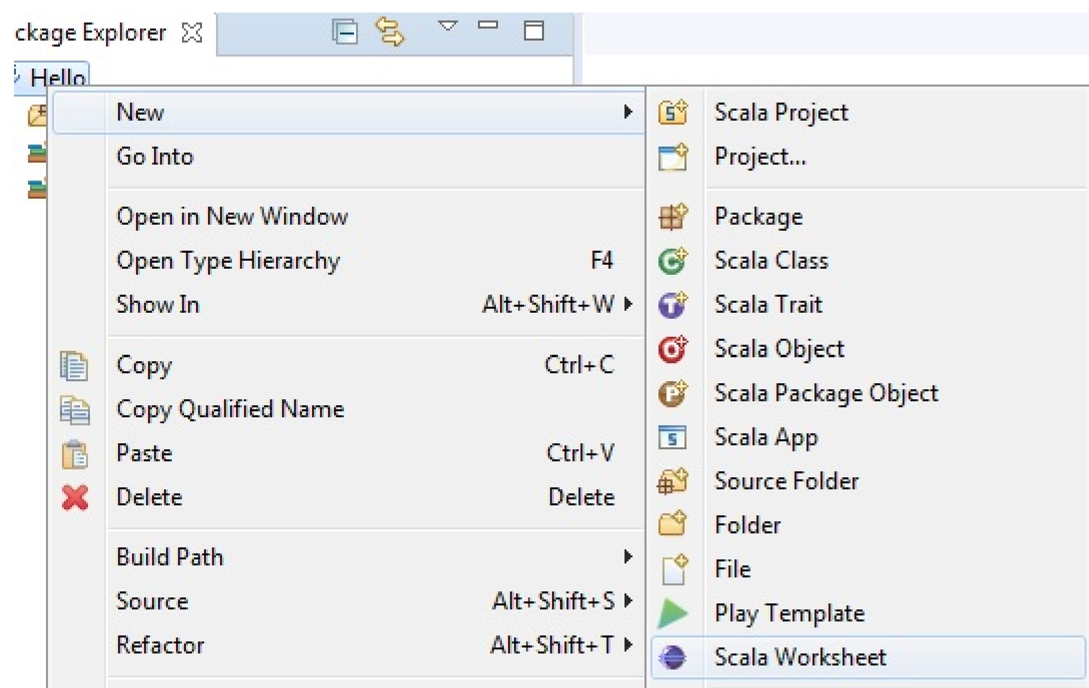
Töltsük le (mondjuk) a http://www.scala-ide.orgról az oprendszerünknek megfelelő Scala IDE csomagot. Unzip egy szimpatikus helyre. JDK 8 is legyen legalább a gépen. Mivel ez egy módosított Eclipse, az `eclipse.exe`vel indul, a parancsikonja ilyesmi:



Aki már látott Eclipset, ismerős lesz a használati mód (van persze IntelliJ Idea plugin is, a kabinetben Scala IDE lesz): `File/New/Scala Project`.



Én mondjuk a `Hello` nevet adtam a projektnek. A Java-ban szokásos módon hozhatnánk létre máris a packageket, osztályokat stb, de talán érdemes most előbb egy Worksheet-tel kezdeni, ennek is a HelloWorksheet nevet adom:



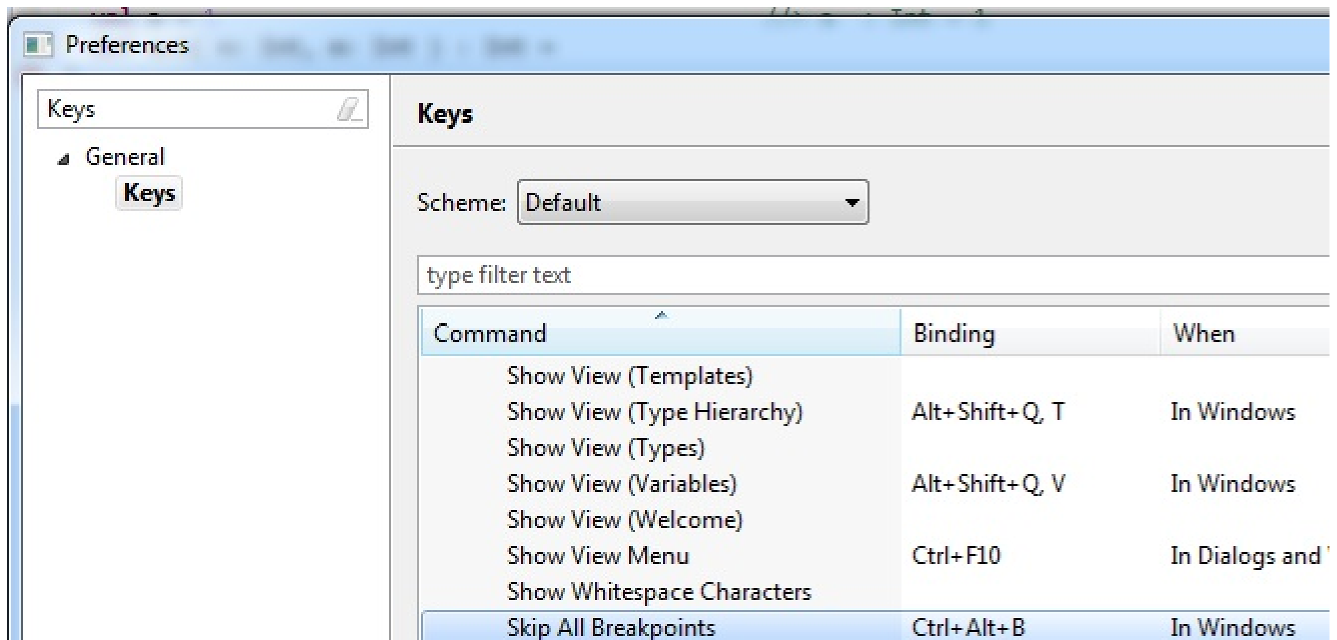
Egy Worksheet szemre és viselkedésre kb. olyan, mint egy interpreter. Valójában az történik, hogy a Scala a Worksheet-be gépelt kódot lefordítja, futtatja a lefordított kódot és a kimenetet visszaírja a Worksheet-re, ezért nem interpreternek, hanem REPL-nek (Read-Eval-Print Loop) szokták nevezni[1]. Ha a worksheetünk defaultból odakerülő `println` sora után beírjuk pl. azt, hogy `val a = 1` (ami létrehoz egy `a` nevű, `1` értékű hát... értéket, majd elmentjük, ezt kapjuk:

[1] Valójában a REPL konzolos. A Worksheet az inkább REPL on steroids `:D`

```
HelloWorksheet.sc
object HelloWorksheet {
  println("Welcome to the Scala worksheet")       //> Welcome to the Scala worksheet
  val a = 1                                        //> a  : Int = 1

}
```

Tehát, a rendszer kommentbe teszi az outputot a sorok után: kiírja a printelt szöveget és jelzi, hogy az `a` értéket `Int` típussal hozta létre, a konkrét értéke pedig `1` lett.

Mielőtt bármit is csinálnánk, Window/Preferencesben General/Keys, és unbindelni a Skip All Breakpoints Ctrl-Alt+B-jét, ha magyar billentyűzetkiosztással akarsz csinálni valamit, mert az bizony a cirmos nyitójel.



Első lépésként megírjuk az Ackermann-függvényt. Ugye ez az

$$\mathrm{ack}(n,m) = \begin{cases} m+1 & \text{ha } n = 0 \\ \mathrm{ack}(n-1,1) & \text{ha } n > 0 \text{ és } m = 0 \\ \mathrm{ack}(n-1,\mathrm{ack}(n,m-1)) & \text{ha } n,m > 0 \end{cases}$$

függvény. Azért szoktuk tanítani, mert nagyon gyorsan növekszik, meg nem primitív rekurzív, ilyesmi, most csak egy egyszerű rekurziós példaként van itt, amin majd a curryt is lehet demózni.

A worksheetben eddig is szereplő print meg értékadás után ennek egy Scala implementációja:

```
HelloWorksheet.sc
object HelloWorksheet {
  println("Welcome to the Scala worksheet")       //> Welcome to the Scala worksheet
  val a = 1                                        //> a  : Int = 1
  def ack( n: Int, m: Int ) : Int =
    if( n == 0 ) m+1
    else if( m == 0 ) ack( n-1 , 1 )
    else ack( n-1, ack( n, m-1 ) )               //> ack: (n: Int, m: Int)Int

  ack(1,1)                                         //> res0: Int = 3
  ack(2,1)                                         //> res1: Int = 5
  ack(2,2)                                         //> res2: Int = 7
  ack(3,3)                                         //> res3: Int = 61
}
```

Mit látunk a képen. Magának a függvénynek a definícióját a pirossal bekeretezett rész adja ofc, ezt vezeti be a `def` kulcsszó. A fejlécben azt látjuk, hogy az éppen definiált `ack` függvény két bejövő formális paramétert kér, az `n` és `m` `Int`eket. Javában ez `int ack(int n, int m)` vagy `Integer ack(Integer n, Integer m)` lenne, Scalában előbb adjuk meg a változó nevét, és utána kettősponttal a típusát. Ugyanez vonatkozik a függvény eredményének a típusára is, amit szintén kettősponttal adtunk meg a paraméterlista után, ez is egy `Int`.

A Scala egy típusos nyelv, minden bevezetett változónak, értéknek, függvénynek van egy típusa. Ennek némiképp ellentmondani látszik az előző sorban a `val a = 1`, ahol is egy szót nem szóltunk az `a` érték típusáról; de a REPL outputjából látszik, hogy a fordító `Int` típusúként hozza létre. A Scala fordítója egész okosan tud típusokat kikövetkeztetni: általában nem kell megadjuk egy függvény, vagy egy érték típusát, ha a fordító ki tudja következtetni, akkor nem reklamál, hanem odaírja. Jelen esetben mivel az 1 az egy egész, a fordító `Int` típusúra értékeli ki az `=` jobb oldalán álló részt (az 1-et), tehát a bal oldal típusát `Int`re guesseli. Általában elég jól működik a típus-kikövetkeztető (type inferencing) motorja, de amit pl. nem tud kezelni, az a rekurzív függvények esete: ha egy függvény rekurzív, akkor explicit ki kell írjuk a visszatérési értéket. Ha nem tesszük (mondjuk kitörlöm a `: Int` részt a fejlécben, akkor erről beszédesen tájékoztat is:

```
def ack( n: Int, m: Int ) =
  if( n == 0 ) m+1
recursive method ack needs result type , 1 )
  else ack( n-1, ack( n, m-1 ) )
```

Az `Int`ről még annyit érdemes tudni, hogy ez is egy osztály, ebben a nyelvben – szemben a Javával – egyáltalán nincsenek elemi típusok, nincs pl. `int`, `boolean`, `double`, `char`. Van viszont `scala.Int`, `scala.Boolean`, `scala.Double` és `scala.Char`, meg persze a teljesség igénye nélkül van ugyanitt `Float`, `Byte`, `Short` és `Long` is. A névterek kb. ugyanúgy működnek, mint ahogy Javában megszoktuk: ezek az osztályok pl. a `scala` csomagban vannak deklarálva, és explicit `import` nélkül is használhatjuk őket (mint Javában a `java.lang`ot is). Azon nem kell aggódni, hogy az elemi típusok osztályba wrappelésével veszítünk a hatékonyságból, mert a Scala fordító ezeket az osztályokat ahol csak lehet, elemi típusba fogja forgatni, tehát hatékonyságban itt pl. ugyanott leszünk, mintha Javában `int`tel implementáltuk volna.

Azt is láthatjuk, hogy a metódus típusa (ha kell) után egy `=` jelet kell kitettünk, majd ezek után jön a metódus törzse. Ami ebben a törzsben feltűnhet, hogy Javában nem pont így csinálnánk:

- Nincs `{`kapcsosba`}` rakva a törzs. Scalában ha egyetlen kifejezés szerepel akár több sorban is, akkor nem kell kitegyük a kapcsost. Most pl. az `if()..else if()..else..` az egyetlen kifejezés. Ha valaki szeretné, persze kiteheti, kapcsossal szervezhetünk *blokkokba* kifejezés-csoportokat.

- Nincs `return` (három helyen lehetne[2]). Opcionális, ha valakinek úgy olvashatóbb, kirakhatja a returnt is. Emögött az van, hogy Scalában minden *kifejezés* és minden rendelkezik valamilyen típussal. Konkrétan, az `if(B) E else F` konstrukció akkor valid, ha `B` típusa `Boolean`, ilyenkor az `E` és `F` kifejezéseknek is van valamilyen típusa, az egész `if-else` struktúra típusa pedig az `E` és `F` típusának *legspecifikusabb közös őse* lesz.

Tehát amit a Scala lát a metódus törzsében: az `m+1` rész típusa `Int` (mert `m` és `1` is `Int`, a `+` operátor pedig ezekből `Int`et készít), a két `ack(...)` részkifejezés típusa szintén `Int`, ezeknek pedig a közös őse megint csak az `Int` lesz, ami megfelel a metódus fejlécében szereplő típusnak;

[2] Erre még később visszatérünk

továbbá, az `n==0` és `m==0` tesztek pedig `Boolean` típust adnak vissza, tehát a kód szintaktikusan helyes.

Tesztelésképpen négy függvényérték-kiszámítást is kértem tőle, a worksheet jobb oldali komment oszlopában látszik mentés után, hogy mire értékelődik ki a függvény, jónak látszik.

Ezen a kurzuson az idő legnagyobb részében *pure* funkcionális programozni fogunk, ez (egyebek mellett) azt fogja jelenteni, hogy minden immutable lesz, a „változók” egyszer kapnak értéket, nincs utána újabb assignment. Az ilyen típusú változókat (attól, hogy nem változnak, én még változónak fogom hívni őket) vezeti be a `val` kulcsszó.

Ebben (és abban, hogy minden Scala program egyetlen kifejezés) az a jó, hogy magának a programnak a szemantikája definiálható a *helyettesítési modellben* (substitution model), ami lényegében a kifejezésünk lépésenkénti átírása néhány alapszabály mentén. Ha a kifejezésünk már nem írható át tovább, azt mondjuk rá, hogy *érték*. Maguk az átírási szabályok eléggé egyértelműek, egyelőre lássuk az Ackermann függvényhez kellő részt. Jelölje $M \triangleright N$ azt, hogy az M kifejezést átírjuk az N kifejezésre.

Egy `if(B) E else F` kifejezést a következőképp írunk át:

- Ha `B` még nem egy `Boolean` konstans, azon végzünk átírási lépést
- Ha `B==true`, akkor a kifejezést átírjuk `E`-re
- Ha `B==false`, akkor pedig `F`-re

Egy *függvényhívást* pedig a következőképp végzünk: ha van egy `def f(x1 : T1,...,xn : Tn) = F` függvénydefiníciónk valahol, és az aktuális kifejezésünk az `f(v1,...,vn)`, akor ezt a kifejezést `F[x1/v1,...,xn/vn]`-re írjuk át. Ez az utóbbi jelölés annyit tesz, hogy az `F`-ben az `xi`-k helyére a `vi`-ket helyettesítjük.

Nézzük ezt egy példán, mondjuk az `ack(2,0)` hívásnál mi történik:

$$\begin{aligned}&\texttt{ack}(2,0)\\ &\triangleright\,\texttt{if}(2 == 0)2+1\ \texttt{else if}\ (0 == 0)\texttt{ack}(2-1,1)\texttt{else ack}(2-1,\texttt{ack}(2,0-1))\end{aligned}$$

(ez volt a függvénytörzs-behelyettesítés, az ún. *β-redukció*) Most a $2 == 0$ részt értékeljük tovább[3], ami `false`:

$$\triangleright\texttt{if}(\texttt{false})2+1\ \texttt{else if}\ (0 == 0)\texttt{ack}(2-1,1)\texttt{else ack}(2-1,\texttt{ack}(2,0-1))$$

az `if false` miatt az első `else` utáni részre írunk át:

$$\texttt{if}\ (0 == 0)\texttt{ack}(2-1,1)\texttt{else ack}(2-1,\texttt{ack}(2,0-1))$$

a $0 == 0$ az épp igaz lesz:

$$\texttt{if}\ (\texttt{true})\texttt{ack}(2-1,1)\texttt{else ack}(2-1,\texttt{ack}(2,0-1))$$

az `if true` átírja az első ágra:

$$\texttt{ack}(2-1,1)$$

Ezen a ponton két lehetőségünk van.

- Előbb kiértékeljük a paramétereket, majd ezek után helyettesítünk be. Ez a konvenció a *Call-By-Value*, CBV.

[3] ezt a részt nem fejtjük ki, technikailag az `Int` osztály `==` metódusát β-redukálnánk most

- Behelyettesítünk így. Ez a konvenció a *Call-By-Name*, CBN.

A CBV konvenció szerint tehát előbb az `ack`(1,1) kifejezést kapjuk, majd az

```
if(1==0) 1+1 else if(1==0) ack(1-1,1) else ack(1-1,ack(1,1-1))
```

kifejezést. A CBN konvenció szerint pedig a

```
if(2-1==0) 1+1 else if(1==0) ack(2-1-1,1) else ack(2-1-1,ack(2-1,1-1))
```

kifejezést.

A Scala alapértelmezése a *Call-By-Value*, tehát előbb kiértékeli a paramétereket, aztán helyettesít be. A CBV-vel megkapjuk az első kifejezést, és ezzel folytatuk az átírást a korábbiak szerint...

```
ack( 1 , 1 )                                  //> res0: Int = 3
```

három lesz a vége.

Az egyik ereje a pure funkcionális paradigmának, hogy

teljesen mindegy, milyen sorrendben és hol végezzük az átírásokat,

a végeredmény akkor, *ha értéket kapunk*, mindig ugyanaz lesz.

Ez az ún. Church-Rosser tulajdonság, avagy „konfluencia”. Amiért pedig például örülünk neki: a kifejezésünk átírható résztermjeit elkezdhetjük párhuzamosítva átírni, ahogy akarjuk, garantáltan nem lesz belőle baj. Erről később többet fogunk látni.

Első megérzésből talán arra gondol az ember, hogy a CBV mindig „jobb” a CBN-nél, hiszen ha egy paramétert többször használunk, akkor a CBV csak egyszer kell átírja, a CBN meg annyiszor, ahányszor előfordul. Ez egyébként az esetek jó részében igaz is. Ahol esetleg különbség lehet: *ha a paramétert egy szálon egyáltalán nem használjuk*, akkor a CBN ki se értékeli egyáltalán. Ha minden szálon minden paraméter legalább egyszer szerepel (mint ahogy egyébként az `ack` függvényünkben is), akkor a CBV tényleg jobb lehet. De nézzük meg pl. a következő függvényt:

```
def f( n: Int, m : Int ) = if( n == 0 ) 0 else m//> f: (n: Int, m: Int)Int
```

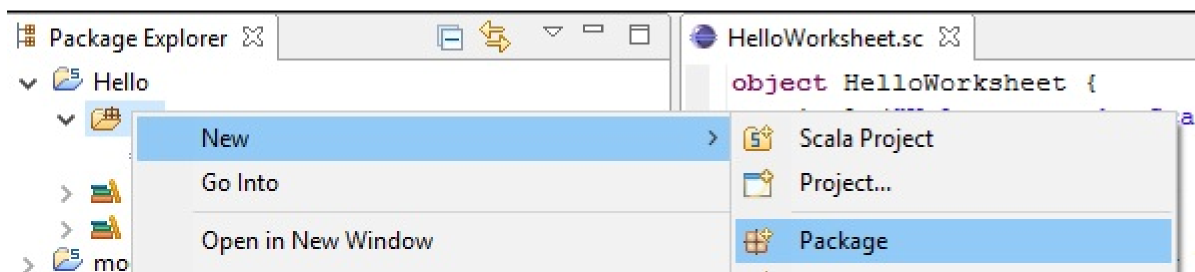
Eddig semmi veszélyes. Mivel a függvény nem rekurzív, a type checker ki is tudta következtetni, hogy `Int` a kimenet. Azt láthatjuk, hogy az `n` argumentumot mindig *pontosan egyszer* értékeljük ki, annak tehát mindegy, hogy CBV vagy CBN, de az `m`-et pedig *legfeljebb egyszer*: ha `n==0`, akkor meg se nézzük. Ilyenkor jobban hangzik a CBN. Főleg, ha még a következőt is hozzátesszük:

```
def loop : Int = loop                         //> loop: => Int
```

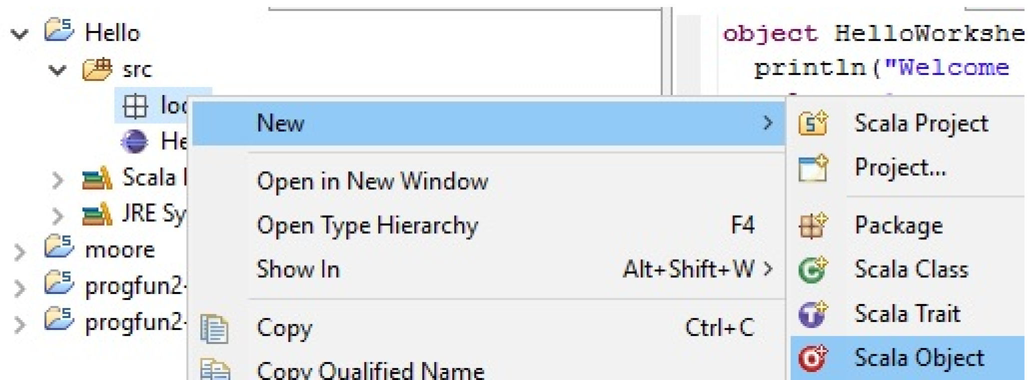
A kódról még azt láthatjuk, hogy ha egy függvény nem vár bejövő paramétereket, akkor nem kell kitegyük a zárójeleket. Lehetett volna `def loop() : Int = loop()` is a deklaráció. Szemantikailag ekvivalens a kettő, a *konvenció* (ami sokat dob a kód olvashatóságán) az, hogy *ha a függvénynek van mellékhatása*, akkor tegyük ki a zárójeleket, ha nincs, akkor ne tegyük ki. Egyelőre a pure funkcionális programozásban, amit nagyon próbálunk követni, kb az egyetlen, ami mellékhatást tud termelni, az a `println()` függvény lesz.

Mindenesetre a `loop` nem más, mint egy végtelen ciklus: folyamatosan önmagára írjuk át.

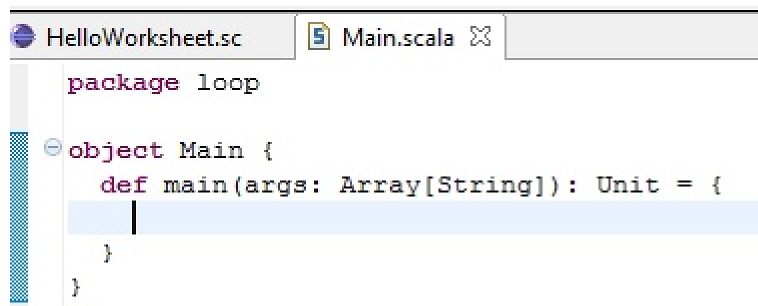
Hogy teszteljük, mi történik, ezt már ne a Worksheetünkben tegyük, mert a Worksheet nagyon rosszul tűri azt a végtelen ciklust, ami nem tölti a stacket és ez pont olyan. Tehát most már nem a REPLben homokozunk, hanem létrehozunk egy csomagot:



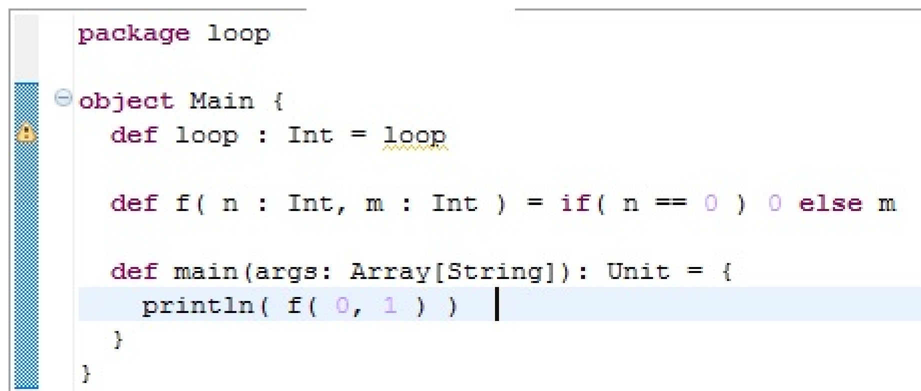
Én épp `loop`nak neveztem el. Hozzunk létre benne egy új `objekt`umot:



Elneveztem `Main`-nek, belenavigáltam, beírtam, hogy `main` és nyomtam ctrl-space auto kiegészítést:



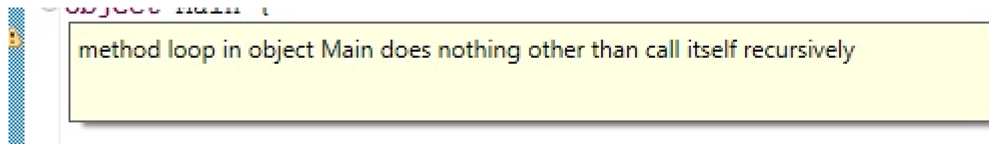
Beleírtam a Worksheet most érdekes részét:



Mármost ha van egy Scala `object`ünk, benne egy `def main(args: Array[String]): Unit` metódus, akkor ez egy valid belépési pont lesz (hasonlóan a Java `public void static main( String[] args )`ához). A még nem ismert részeket nemsokára megismerjük, egyelőre csak futtassuk:

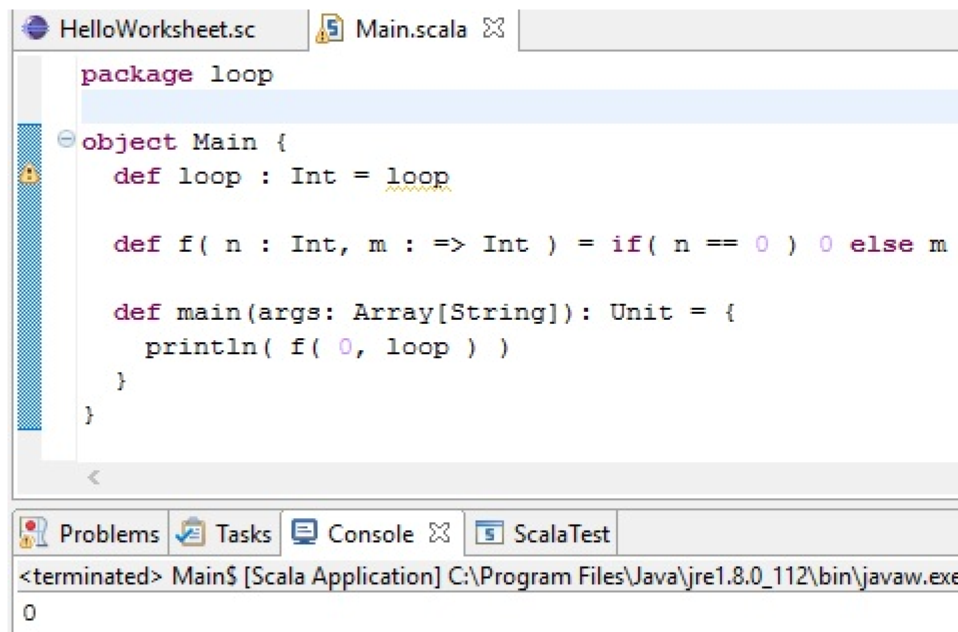
Lenn megkapjuk konzolban, hogy `0`, remek. Hiszen hát az `f(0,1)` az tényleg `0` kell legyen az `f` kódja szerint. De mi lesz `f(0,loop)`?

Amúgy „a fordító nem hülye” rovatban kiemelném ezt a warningot:



Ha átírjuk `f(0,loop)`-ra a hívást a `main` belsejében, és úgy futtatunk (most már elég a zöld play gombra kattogni hozzá, hogy van run konfigunk), az végtelen ciklus lesz. Persze, hiszen először az `f` argumentumait értékelné ki a VM, a nullával nincs is gond, de a `loop` értékelésekor elszáll végtelen ciklussal. Ez nem történne meg, ha call-by-name hívnánk: ekkor az átírás hagyná a `loop`ot ahogy van, az `f(0,loop)` átíródna `if(0==0) 0 else loop`-ra, majd a feltétel `true`, és kapnánk egy `0`-t. Ebben az esetben jobb lenne a call-by-name, mint a call-by-value.

Scalában az argumentum típusa elé `=>` jelet írva érjük el, hogy az adott argumentum call-by-name legyen:



És az outputból látjuk is, hogy ezúttal sikerült kiértékelni `0`-ra azt a hívást.

Most térjünk vissza erre a mondatra: „a Worksheet nagyon rosszul tűri azt a végtelen ciklust, ami nem tölti a stacket” és beszéljünk kicsit a *tail recursion*ről.

A fenti `loop` defnél világos, hogy mi történik:

$$\texttt{loop} \rhd \texttt{loop} \rhd \texttt{loop} \rhd \ldots,$$

legalábis elméleti szinten, abban a *substitution model*ben, mely szerint a kifejezések kiértékelése függvények esetén így történik, kicserélve a nevét a törzsére és helyettesítve a formális paramétereket

értékekkel (amik most nincsenek). Ha ebben a modellben gondolkodunk, akkor ez így valóban nem tölt semmilyen stacket, konstans memóriát használ.

Javában viszont ha megírjuk ezt a kódot:

```
public class Main {
  public static int loop() { return loop(); }
  public static void main(String[] args) {
    loop();
  }
}
```

akkor ha kipróbáljuk, kapunk is egy `StackOverflowErrort.` Mindez azért történik, mert a Scala alapból végez *tail recursion optimization*t, a Java pedig nem. Ez az optimalizálás arról szól, hogy ha egy `f` rekurzív függvény „csak” annyiban rekurzív, hogy a törzsében minden rekurzív hívás `return f(...)` alakú (imperatív környezetben), azaz a rekurzív hívás az utolsó, amit elvégez, majd ennek az eredményét változatlan formában visszaadja (az ilyen függvényeket hívják tail recursive függvénynek), akkor a függvényhívás *helyett* egyszerűen aktualizálja a formális paraméterek értékét és `goto`zik a függvénytörzs elejére. Például legegyszerűbb példánkban, a `loop`ban ahelyett, hogy folyamatosan elmentenénk a stacktrace-be, hogy hol járunk, majd beugrunk újra a függvénybe, ahelyett aktualizálhatnánk a paraméterlistát (ami nincs), majd gotozhatnánk a loop elejére és ennyi. Nem telik a stack (a függvény meg persze nem terminál).

Nézzünk egy eggyel összetettebb példát. Vegyük a faktoriálist kiszámító következő kódot:

```
def fact( n : Int ) : Int = if ( n <= 1 ) 1 else n*fact(n-1)
```

Ez *nem* tail recursive, a `fact` hívása nem az utolsó lépés. Ha pl. hívjuk a `fact(4)`-et, látjuk is, hogy mi történik:

$$\begin{aligned}\texttt{fact(4)} &\rhd \texttt{if (4 <= 1) 1 else 4*fact(4-1)}\\ &\rhd \texttt{4*fact(3)}\\ &\rhd \texttt{4*3*fact(2)}\\ &\rhd \texttt{4*3*2*fact(1)}\\ &\rhd \texttt{4*3*2*1} \rhd \texttt{24},\end{aligned}$$

tehát látszik, hogy kiértékelés közben nő a „stack”, $O(n)$ memóriát használunk el a `fact`(n) kiszámításakor, mert a visszatérési értékkel még végzünk valamit a későbbiekben.

Mivel a funkcionális paradigmában a rekurzió egy fő komponens (szemben az iterációval), így ezt hatékonyan kell csinálnunk. Sokszor ahhoz, hogy egy rekurzív függvényt tail rekurzívra írjunk át, be kell vezetnünk egy (vagy több) új paramétert, melyet „akkumulátorként” használva gyűjtünk le részeredményeket. Vegyük a faktoriálisnak a következő implementációját:

```
def fact2( n : Int, acc : Int ) : Int = if( n<= 1 ) acc else fact2( n-1, n*acc )
```

Nézzük csak, mi történik, ha kiértékeljük pl. `fact2(4,1)`-et:

```
fact2(4,1) ▷ if( 4<=1 ) 1 else fact2( 4-1, 4*1 )
           ▷ fact2( 3, 4 )
           ▷ if( 3<=1 ) 4 else fact2( 3-1, 3*4 )
           ▷ fact2( 2, 12 )
           ▷ fact2( 1, 24 )
           ▷ if( 1 <= 1 ) 24 else fact2( 1-1, 1*24)
           ▷ 24.
```

Mindenesetre ez a függvénydefiníció így tail rekurzív (ebben a paradigmában nem a `return` mondja meg, hogy mi az utolsóként kiértékelődő része egy kifejezésnek, de világos), és ennek megfelelően ez a függvény már konstans stack méretben kiértékelődik. A második argumentum akkumulátorként viselkedik: indul 1-ről, majd rászorzunk 4-gyel, utána még 3-mal, majd megint 2-vel, és ezt adjuk vissza; általában is a `fact2`(n, a) hívásának eredménye $n! * a$ lesz, ha $n \geq 0$.

Ezt az állítást most be is bizonyítjuk. Ezzel azt szeretném szemléltetni, hogy a funkcionális programok, kiváltképp a rekurzívak, melyekben nincs „változó” változó (mutable state) helyessége sokszor formálisan is bizonyítható, jellemzően felépítés szerinti indukcióval, ebben az esetben pl. teljes indukcióval.

Tehát, ha $n = 0$, akkor `fact2`(n,a) értéke

```
fact2(0,a) ▷ if ( 0 <= 1 ) a else fact2( 0-1, 0*a )
           ▷ a,
```

ami épp $0! * a = 1 * a = a$, ez az eset kész. Ugyanez történik $n = 1$-re is, megint a-t kapunk, ami $1! * a = 1 * a = a$. Most ha feltesszük, hogy igaz n-re, $n \geq 1$, nézzük meg $n + 1$-re is, ekkor:

```
fact2(n+1,a) ▷ if(n+1 <= 1 ) a else fact2( n+1-1, (n+1)*a )
             ▷ fact2( n, (n+1)*a ),
```

és erre az utóbbira, mivel az első argumentuma n, az akkumulátora pedig $(n + 1) * a$, alkalmazhatjuk az indukciós feltevést, és kapjuk, hogy értéke $n! * (n + 1) * a = (n + 1)! * a$ lesz, és pont ezt akartuk bizonyítani.

Ez tehát azt is jelenti, hogy az $n!$ értékét a `fact2(n,1)` hívással kapjuk meg. Persze jobb, ha a user elől elrejtjük a belső implementációs részleteket, ezt megtehetjük Scalaban:

```
def fact( n : Int ) = {
  def fact2( n : Int, acc : Int ) : Int = if( n<= 1 ) acc else fact2( n-1, n*acc )
  fact2( n, 1 )
}
```

Mit látunk itt pontosan? Definiálunk egy `fact` függvényt, ami egy `Int` paramétert vár. Ezen belül két kifejezést látunk, az első egy függvénydeklaráció: Scalaban lehet bárhol függvényt deklarálni, nem csak top-level blokkban, hanem akár más függvényeken belül is. Tehát itt van deklarálva a kétváltozós függvényünk, majd a következő kifejezés ezt hívja meg `fact2(n,1)`-képpen, amiről tudjuk, hogy ennek az értéke $n!$ lesz. A két (vagy több) egymás után írt kifejezésből álló blokk értéke mindig az utolsó kifejezés értéke lesz, tehát így az egész `fact` függvény értéke egy adott n inputra tényleg $n!$ lesz.

Még egy hasznos feature: az volt a fejünkben, hogy ez a `fact2` függvény tail rekurzív, erre szerettük volna megírni, hogy hatékonyan gazdálkodjon a memóriával futás közben a programunk. Ha tényleg az, akkor a Scala fordító arra is fordítja, de persze ha nem az, akkor nem

fog hibát dobni, miért is tenné. Van viszont egy annotáció, a `@tailrec`, ami pont erre való: ha eléírjuk egy függvénynek, ami *nem* tail rekurzív, akkor a fordító sipákolni fog, hogy nem tail rekurzív a függvény, amit mi (az annotáció szerint) annak hiszünk:

```
def fact( n : Int ) = {
  @tailrec
  def fact2( n : Int, acc : Int ) : Int = if( n<= 1 ) acc else fact2( n-1, n*acc )
  fact2( n, 1 )
}
```

Mivel a fordító nem kiabál velünk, az annotált `fact2` függvény tényleg tail rekurzív, tehát ki lesz optimalizálva. Remek.

Nem minden esetben könnyű persze tail rekurzív alakúra írni egy függvényt, és nem is minden függvényt lehet. A következő példában a Fibonacci függvényt, `Fib(0)=Fib(1)=1` és nagyobb n-re `Fib(n)=Fib(n-1)+Fib(n-2)`, fogjuk megírni tail rekurzív alakban és közben megismerkedünk a Scala néhány nyelvi elemével is.

Már felfigyelhettünk rá az IDEben, hogy legalább háromféle dolgot lehet létrehozni: `class`t, `object`et és `trait`et. Eddig `object`eket hoztunk létre jobbára. Ugyanolyan névvel létrehozhatunk egy `class`t és egy `object`et is egyszerre, ekkor kötelesek vagyunk mindezt egy forrásfile-ban tenni. Scala-ban nincs `static` kulcsszó, a legegyszerűbb úgy elképzelnünk az `object`et, hogy amit Javában `static` módosítóval látnánk el, az megy az `object`be, amit meg nem, hanem amiből az osztály minden példányának van sajátja, az pedig a `class`ba. Erről kicsit később pontosítunk, egyelőre jó lesz ennyit gondolnunk róla. A `trait` pedig majdnem olyan, mint egy `interface` azzal a különbséggel, hogy lehetnek benne implementált metódusok és megadott mezők is. . . tehát majdnem olyan, mint egy `abstract class` azzal a különbséggel, hogy egy osztály több `trait`et is tud `extend`elni. Egyelőre hagyjuk a `trait`et, maradjunk a `class`nál és az `object`nél.

Javában valami hasonlót implementálnánk Fibonaccira:

```
class Fib {
  public static int compute( int n ) {
     if ( n<=1 ) return 1;
     return compute( n-1 ) + compute( n-2 );
  }
}
//majd valahol lejjebb a main-ben
  Fib.compute( 5 ) //returns 8
```

Mivel ez egy számolós osztály, semmi belső állapottal, ezért `static`nak van értelme deklarálni ezt a metódusát, különben még egy példányt is létre kéne hoznunk belőle és `new Fib().compute( 5 )`-ként hívni. Mivel pedig `static`, így a `Fib`-nek az `object`jébe kerül. Sőt, hát mivel nemstatic tag vagy metódus nincs is, a `class`t létre se hozzuk, hanem csak objektumunk lesz (egyébként eddig is ezt csináltuk a Worksheettel, majd később a Main objektumunkkal is, most már világos, hogy miért: mert a `main` metódus is „static”, tehát `object`be tesszük a kódját).

Hasonlóan a `C++`hoz, a Scala is támogatja az *operator overloading*ot. Konkrétan az `operator()` overloadingjára itt az `apply` metódust kell definiálnunk: így pl. az `def apply( n: Int, m: Int ): Int` metódus definiálása egy objektumban vagy osztályban azt eredményezi, hogy az objektumon vagy az osztály példányát (legyen ez mondjuk `C`) mint „függvényt” is hívhatjuk így: `C(1,4)` és az eredmény egy `Int` lesz.

Tehát az első implementációnk egy példa hívással:

```
object Fib{
  def apply( n : Int ) : Int = if( n <= 1 ) 1 else Fib( n-1 )+Fib( n-2 )
}
object Main{
  def main(args: Array[String]): Unit = {
    println( Fib( 5 ) ) //prints 8
  }
}
```

Tehát még egyszer: azért írhatjuk a `main`be a `Fib(5)`-öt, mert a `Fib` egy `object` (ezért nem kell `new Fib`), és mert az `apply( n: Int )` metódusát definiáltuk (ezért valid a függvény-like szintaxis, az `Int` argumentumra).

Viszont ez az implementáció így nem is tail rekurzív. Ha elgondolkodunk rajta, hogy hogy tehetnénk azzá, rájöhetünk, hogy pl. *két* akkumulátorral igen: az egyik akkumulátor tárolná az aktuális, a második akkumulátor pedig az előző Fibonacci értéket. Hogy ez honnan jött, hát pl. ha egy imperatív megvalósítást veszünk:

```
if( n < 2 ) return 1; // alapesetek
int prev = 1;
int current = 1;
int i = 1; // most az ennyiedik Fibonacci a current erteke
while( i < n ){
  int tmp = prev;
  prev = current;
  current = current + tmp; // az eredeti current prevbe, az ujba az osszeg megy
  i = i + 1;
}
return current;
```

Itt is két változót használunk. Pont ezek lesznek a két akkumulátor; ha még faragunk egy kicsit a kódon, akkor az `i` változótól is meg tudunk szabadulni: ahelyett, hogy `i`-t visszünk 1-től n-ig, inkább n-t csökkentsük 1-re:

```
if( n < 2 ) return 1; // alapesetek
int prev = 1;
int current = 1;
while( n > 1 ){
  int tmp = prev;
  prev = current;
  current = current + tmp; // az eredeti current prevbe, az ujba az osszeg megy
  n = n - 1;
}
return current;
```

Ezt pedig már könnyen meg tudjuk írni az eddigiek alapján tail rekurzív formában is, hiszen van egy n változónk, azon megy a rekurzió leszállási feltétele, és van pár másik változónk, azok válnak akkumulátorrá:

```
object Fib{
  def apply( n : Int ) = {
```

```
    @tailrec
    def fib2( n : Int, prev : Int, current : Int ) : Int =
      if( n == 1 ) current else fib2( n-1, current, prev+current )
    if( n <= 1 ) 1 else fib2( n, 1, 1 )
  }
}
```

Így ez a tail rekurzív kód megint pontosan ugyanolyan hatékonyan számítja ki a Fibonaccit, mintha iteratívan írtuk volna meg egy imperatív nyelven.

Egyébként még egy nyelvi feature: természetesen mivel az `apply` metódusnak az a neve, hogy `apply`, így `Fib(5)` helyett írhatjuk azt is, hogy `Fib.apply(5)`. Sőt, az objektum és a tagfüggvénye közti pontot sem kötelező kitenni (soha), írhatjuk azt is, hogy `Fib apply(5)`. Továbbmegyek: ha egy függvénynek egy paramétere van, mint pl. most is, akkor ha nincs előtte kirakva a pont, az argumentum körüli zárójeleket is elhagyhatjuk és írhatjuk azt is, hogy `Fib apply 5`. Mind ugyanazt jelenti.

Fentebb láttuk, hogy egy tail rekurzív függvényt a fordító igazából átír úgy, hogy a rekurzív hívás helyén értékadások lesznek, aztán egy feltétel nélküli `goto` a törzs elejére, ami meg tulajdonképpen egy `while(true)` ciklus, tehát egy tail rekurzív kódot mindig át lehet írni imperatív `while` kódra. Ez visszafele is igaz: most írni fogunk Scalaban egy olyan `While( B ) P` konstrukciót, ami azt hajtja végre, amit elvárunk: kiértékeli a `B` feltételt, ha az igaz, akkor végrehajtja a `P` „utasítást”, majd kezdi elölről, ha pedig `B` hamis, leáll.

Az „utasítás”ról annyit, hogy Scalaban a legtöbb minden kifejezés; a „visszatérési érték nélküli” kifejezéshez legközelebb Scalában a `Unit` osztály van, egyetlen példánya a `()` (ha az egész osztálynak ez az egy példánya van, akkor minden `Unit` típusú kifejezés erre fog kiértékelődni, tehát nincs „informatív” visszatérési értéke: emiatt persze az `Unit` típusú kifejezések rendszerint valami mellékhatással rendelkeznek, hiszen különben minek értékelnénk ki őket). Egyébként a `Unit`ból a JVM számára leginkább `void` lesz.

Tehát, a fenti képletben `P` típusa `Unit`? Majdnem: `=>Unit`. Ugyanis ha call-by-value kapnánk meg, akkor előbb kiértékelnénk, akár kell, akár nem, pedig csak akkor szeretnénk, ha a feltétel igaz.

Most nézzük a `B` feltételt. Ezt újra és újra ki akarjuk értékelni minden „iterációban” – tehát ezt se adhatjuk át mezei `Boolean`ként, érték szerint, mert akkor már nem változhat az értéke. Ezt is `call-by-name` adjuk át, tehát típusa `=>Boolean` lesz.

A kódunk tehát:

```
object While {
  @tailrec
  def apply( b : =>Boolean )( p : =>Unit ) : Unit =
    if( b ) { p; While( b )( p ) } else ()
}
```

(Valójában itt az `else` ág nem kell, hiánya éppen `Unit`ra defaultol, ami most pont jó nekünk.)

Világos, mi történik: kiértékeljük a feltételt, ha hamis, akkor visszaadjuk az egyetlen lehetséges `Unit` értéket, ha meg igaz, akkor először „végrehajtjuk” `p`-t, majd ezek után rekurzívan újra hívjuk ugyanezzel a paraméterrel a „ciklusunkat”.

Ezt pl. egy `main`ből így is tudjuk hívni, hála a Scala zárójelezési engedékenységének:

```
var n = 6;
```

```
While( n > 0 ) { println( n ); n = n - 1; }
```

A fentiből a `var`t, mutable változót amennyire csak lehet, kerüljük, de hogy ugyanaz a feltétel másra értékelődhessen ki, most szükségünk van rá. Tehát deklarálunk egy mutable `var`t, 6ra inicializálva, és hívjuk a `While`...objektum...`apply` metódusát...az első argumentumba az `n>0`, `Boolean`-ra kiértékelődő feltételt tesszük, ez OK, a másodikba pedig hát egy kifejezésblokk van kapcsosban, ez lesz az `Unit`ra kiértékelődő call-by-name argumentum. És teljesen úgy néz ki, mint egy átlagos imperatív programozási nyelvben egy `while` ciklus, úgy is működik, tehát meg lehet építeni tetszőleges `while` ciklust tail rekurzív módon.

Tehát a tail rekurzió és a while ciklus kifejezőereje megegyezik.

Apropó, térjünk vissza egy kicsit erre a részre: „itt most az `else` nem kell" és nézzük meg az első implementációját az Ackermann-függvénynek még egyszer:

```
def ack( n : Int, m : Int ) : Int = {
  if( n == 0 ) m + 1
  else if ( m == 0 ) ack( n - 1 , 1)
  else ack( n - 1, ack( n , m - 1 ))
}
```

Páran biztos eljátszottak a gondolattal (én igen), hogy „á, itt az `else` fölösleges, szedjük ki" és kapták ezt a kódot:

```
def ack( n : Int, m : Int ) : Int = {
  if( n == 0 ) m + 1
  if ( m == 0 ) ack( n - 1 , 1)
  ack( n - 1, ack( n , m - 1 ))
}
```

Majd meglepve tapasztaltuk egy `ack(1,1)` hívásnál, hogy nem kapjuk vissza, hogy ez is három, hanem köszön egy StackOverflowError. Ez azért van, mert amit itt a második implementációban látunk, az egy függvényblokkban *három kifejezés*. Itt nem az történik, hogy ha az első `if` feltétele igaz, akkor visszaadjuk az `m+1` értéket! Hanem csak annyi, hogy van az első kifejezés, az `if(n==0) m+1`, ez ha `n==0`, akkor kiértékelődik `m+1`-re, ha pedig nem nulla, akkor (`else` ág híján) a `Unit` típus egyetlen objektumára, `()`-ra. (Egyébként ha ezt odaadjuk egy `val`nak, akkor a Scala típuskövetkeztetője `Any`re sorolja be, ami tényleg az `Int` és a `Unit` osztályok legspecifikusabb közös őse). Aztán, ennek a kifejezésnek az értékét mivel nem adjuk oda senkinek, eldobjuk és kiértékeljük a következőt *mindenképp*, akár igaz volt a feltétel az előbb, akár nem. Ha `m==0`, akkor még egy rekurzív hívásba is bemegyünk ekkor. De ha ezt a második `if`et végül sikerül kiértékelni, akkor ennek is eldobjuk az eredményét, és elkezdjük kiértékelni a harmadik kifejezést, az `ack(..)`-ot. Ha sikerülne kiértékelni, akkor az egész kifejezés-sorozatnak ez lenne az értéke: egy (kapcsosba zárt) kifejezés-sorozat értéke mindig a legutolsó kifejezésének az értéke (de ettől a többit is kiszámoljuk. Persze ha nincs mellékhatásuk, akkor amúgy teljesen feleslegesen tesszük ezt).

Tehát ha elhagyjuk az `else`t, akkor egy egész más szemantikájú programot kapunk. Az `else` az kell, period[4].

Most nézzük meg még egyszer a `While` objektumunk kódját. Szintaktikailag furcsán hathat az

[4] A teljesség kedvéért: igen, Scalaban is létezik a `return` kulcsszó, amivel itt is visszaadhatnánk az értékeket menet közben és nem kiszámolva a hátsó két kifejezés értékét. Viszont a Scala expertek egybehangzó véleménye az, hogy senki soha ne használjon `return`t :D

a két zárójel egymás mellett. Hogy megértsük, ez mit is jelent, először is írjuk meg a `While` objektumot *két* argumentumúra:

```
object While {
  @tailrec
  def apply( condition : => Boolean, statement : => Unit ) : Unit = {
    if( condition ){ statement ; While( condition, statement ) }
  }
}
```

Ez a kód ugyan tail rekurzív és egy `While` implementáció, de kissé kényelmetlen használni, mert pl. az előző kódot (amíg `n` pozitív, írd ki és csökkentsd) így kéne megírni, ha ezt használjuk:

```
var n = 6; // csak hogy forduljon
While( n > 0 , { println(n); n = n - 1 })
```

Tehát a `While` két argumentuma közé vesszővel, a hátsó után pedig még egy csukójel.

Ez elég kényelmetlennek hangzik, és itt jön be a képbe, hogy a Scala egy *funkcionális* nyelv, most abban az értelemben, hogy nyelvi szinten támogatja a *függvény típusok*at, amivel párban jár a *currying* módszere is. Nézzük sorjában.

Általában ha van egy kétváltozós f függvényem, ami kap egy A és egy B típusú inputot és készít egy C típusú outputot, azt $f : A \times B \to C$-cel szoktuk írni. A fenti `While` esetében $A =$ `Boolean` és $B = C =$ `Unit` (most az egyszerűség kedvéért tekintsünk el a call-by-name vs. call-by-value különbségtől). Egy ilyen kétváltozós függvényből készíthetek egy g *egyváltozós* függvényt, ami csak az első argumentumot kapja és visszatérési értéke egy *függvény* lesz, ami ha megkapja az f *második* argumentumát, akkor visszaadja az f értékét ezen a két argumentumon. Tehát g egy $A \to (B \to C)$ függvény. Scalában szintaktikailag például így adhatjuk ezt meg:

```
def g( a : A ) : B=>C = {
  b => f(a,b)
}
```

Általában ha `A` és `B` Scala típusok, akkor `A=>B` az összes olyan függvény típusa, aminek `A` az input és `B` az output típusa. Például ha a fenti f az összeadás (mondjuk `Intek` közt), akkor g egy `Int=>(Int=>Int)` függvény, amire pl. $g(1)$ az „adj hozzá 1-et” `Int=>Int` függvény, $g(6)$ meg az „adj hozzá 6-ot” függvény.

Ezt a folyamatot, amikor egy többváltozós függvényből több egyváltozósat készítünk (egy n-változós esetében $A_1 \times \ldots \times A_n \to B$ függvényből lesz egy $A_1 \to (A_2 \to \ldots \to (A_n \to B)\ldots)$ függvény, tehát szépen balról jobbra, egyesével adjuk oda az input argumentumokat mindig a soron következő függvénynek), nevezik *currying*nek, a fenti g pedig g *curried* változata.

Azt is láthatjuk a fenti snippetből, hogy egy függvényt hogy adhatunk meg: az input és az output közé tett `=>` jelekkel. A Scala fordító ennél egy még eggyel kényelmesebb és *majdnem* teljesen ekvivalens szintaxist is lehetővé tesz ugyanerre:

```
def g( a : A)( b : B ) : C = f(a,b)
```

és persze mivel a `C` típust ki tudja következtetni, ha `f` típusa (ami $A \times B \to C$ kell legyen) ismert, ezért a `: C` kimeneti típus el is hagyható. Hát ezt a szintaxist használtuk az első `While` implementációnkban.

Amiért azt mondom, hogy *majdnem* ekvivalens: a Scala számára a hosszabb változat `A=>(B=>C)` típusú, a rövidebb pedig `(A)(B)=>C` típusú. Nézzük meg a különbséget:

```
def add( x : Int, y : Int ) : Int = x + y
// def add( x : Int )( y : Int ) : Int = x + y // NEM FORDUL, ha add( x:Int, y:Int)
   is van!
def add( x : Int ) : Int=>Int = { //fordul
y => x+y
}
val add3 = add(3) //add3 : Int=>Int
def addd( x : Int )( y : Int ) = x + y
// val add4 = addd(4) // NEM FORDUL
val add4 = addd(4)(_) // OK
```

Tehát egyrészt nem lehet egyszerre `f(Int,Int)` és `f(Int)(Int)` is jelen, mert a fordító ugyanazt a (JVM) kódot generálja mindkettőnek, másrészt az első argumentumot ha rögzíteni akarjuk (mint pl. az „adj hozzá hármat” függvényt ahogy készítjük az összeadásból), akkor emiatt kicsit más szintaxist kell használnunk. A `_` jel itt egy *placeholder*: függvénydefiníciókban azt jelenti, hogy ide kell behelyettesíteni az inputot, tehát pl. `addd(4)(_)` az `x => addd(4)(x)` függvényre egy shorthand, `_+1` az `x => x+1` függvényre, `_>0` az `x => x>0` (mondjuk `Int=>Boolean` típusú) függvényre stb. Ha a kifejezésben a `_` *többször* szerepel, az azt jelenti, hogy a függvényem *annyi változós, ahányszor a `_` szerepel benne*, és a `_` első előfordulása helyére az első, második helyére a második,... argumentum helyettesítődik be. Tehát pl. `_+_` az `(x,y) => x+y` függvényre egy shorthand.

Nézzük még egy kicsit a `While` curried kódját, amit most már nagyon értünk:

```
object While {
  @tailrec
  def apply( condition : => Boolean ) : (=>Unit) => Unit = {
    statement => if( condition ){ statement; While(condition)(statement) }
  }
}
```

Ez így hívó oldalról már szép: a `While` egy objektum, ha hívjuk az `apply` metódusát mondjuk call-by-name egy `n>0` feltétellel (ami egy `Boolean` típusú kifejezés, tehát rendben van), akkor a kapott `While(n>0)` egy `(=>Unit)=>Unit` típusú függvény lesz, aminek tehát adhatunk call-by-name egy `Unit`típusú bármit (praktice: utasítást), és ezt fogja újra és újra kiértékelni, amíg az `n>0` feltétel igaz marad, aztán leáll. Tehát pl. elfogadja a `{ println(n); n=n-1 }` kifejezés-blokkot, ami `Unit` típusú azért, mert az értékadás típusa mindig `Unit` (!) és a két kifejezésből álló blokk utolsó kifejezése ez, tehát az egész típusa `Unit`. Tehát szintaktikailag van értelme annak, hogy `While( n>0 )({println(n);n=n-1})`, és az unáris függvényhívásra vonatkozó konvenció szerint a hátsó zárójelpárost akár el is hagyhatjuk, így lesz egy épp úgy kinéző kódunk, mint amit imperatív nyelvben csinálnánk, és a szemantikája is ugyanaz, csak ez tail rekurzív, nem iteratív.

(Egyébként mivel az `apply` most szintén unáris, így persze azt is írhatjuk, hogy `While {n>0} {println(n);n=n-1}`, ha akarjuk, az is ezt jelenti.)

Ami viszont magában a `While` implementációban csúnya: a `condition`t folyton csak dobáljuk tovább a következő és a következő hívásra is, holott nyilván ez ugyanaz a kifejezés lesz végig. Egy kis refactorral ezt is megúszhatjuk. Általában azokat a függvényeket, amik a törzsükben hivatkoznak egy rajtuk kívüli scope-ban deklarált értékre (így tehát „emlékeznek a környezetre,

amiben létrehozták őket”), úgy hívják, hogy *closure*. Tehát, ha egy olyan closure-t készítünk, ami emlékszik a `condition`-ra, akkor nem kell dobálja magának rekurzívan[5]:

```
object While {
  def apply( condition : => Boolean ) : (=>Unit) => Unit = {
    @tailrec def myClosure( statement : =>Unit ) : Unit =
      if( condition ){ statement ; myClosure( statement ) }
    myClosure
  }
}
```

Tehát: az `apply` metódusban *két* kifejezés szerepel. Az első egy `def`, ebben elkészítjük a `myClosure` nevű closure-unkat, ami látja a kintről jövő `condition` változót és azt is használja a törzsében. A második kifejezés `myClosure`, ez egyszerűen annyit mond, hogy az `apply` metódusunk ezt adja vissza: az előbb létrehozott `myClosure` függvényt. (Ami stimmel, mert ez egy megfelelő típusú függvény lesz.)

Sőt, ennél még eggyel tovább is mehetünk, hiszen láthatjuk, hogy az előbb létrehozott closure-ünk most még a `statement` argumentumot is csak dobálja saját magának változatás nélkül, sokkal szebb lenne, ha ez egy olyan változóban lenne, ami a függvénytörzs scopeon kívül van. Ezt pedig akár így is megtehetjük:

```
object While {
  def apply( condition : => Boolean ) : (=>Unit) => Unit =
    statement => {
      @tailrec def rec : Unit = if(condition){ statement; rec }
      rec
    }
}
```

Világos: az `apply` kap egy feltételt call-by-name. Mit ad vissza? Egy függvényt, ami kap egy utasítást call-by-name, és visszatérési értéke `Unit` (nemsokára lesz szó erről a típusról kicsit részletesebben, egyelőre gondolhatjuk, hogy ez így `void`). Konkrétan a függvény ha a `statement` utasítást kapja, akkor amit visszaad: először is létrehozunk egy `rec` nevű closure-t, ami ha a feltétel (kettővel kintebb van, látjuk erről a pontról) igaz, akkor kiértékeli a `statement`et (eggyel kintebb van, látjuk erről a pontról) majd rekurzívan hívja önmagát. Majd, miután létrehoztuk ezt a `rec` nevű closuret, vissza is adjuk mint eredményt.

És „kintről” persze ugyanúgy `While(n>0){println(n);n=n-1}` tudjuk hívni, mint eddig :)

És hogy mivel több egy closure, mint egy nested function? Mert ez is működik:

```
var n = 6
val whilePos = While( n > 0 )  // Oo.. okay
n = 10
whilePos { println(n); n = n-1 } // prints 10 9 8 .. 1
```

Ami tehát történik: a `While.apply` megkapja a feltételt, visszatér egy függvénnyel (ami egy utasítást ha kap szintén call-by-name, akkor azon végrehajtja mint ciklusmagon, amíg a feltétel igaz). Mivel a Scala funkcionális nyelv, ez pl. azt jelenti, hogy ezt a függvényt oda is adhatjuk mondjuk egy `whilePos` nevű változónak értékül. Megkaptuk tehát a metódus belsejében `def`elt függvényt értékként, és később (miután akár még az `n`-hez is hozzányúlunk, mindegy) ezt a

[5] persze mivel tail rekurzív, valójában most ebből nem lesz függvényhívás, de a paramétert akkor is updatelné

függvényt a példakód utolsó sorában meghívhatjuk az utasításunkra. Tehát a kapott closure „emlékszik” azokra a lokális változókra is, amiket látott, mikor létrehoztuk.

Mielőtt továbbmennénk és megismerkednénk a funkcionális paradigma egy (a rekurzió és hogy a függvények „first-class citizenek” melletti) újabb szokásos eszközével (a dekompozícióval), nézzünk előbb egy feladatot, mondjuk Javában. A task: legyen egy `Noti` (absztrakt osztály vagy interface, ízlés szerint), ennek két konkrét alosztálya mondjuk az `SMS` és az `Email`, előbbinek van egy `int` mezője, utóbbinak meg egy `String` mezője.

Írjunk olyan kódot, mondjuk egy `Handler` osztály `handle` metódusát, ami kap egy `Notit` és annak függvényében, hogy ez most mi, megjeleníti az adattagját (tehát pl. kiírja a számot vagy a stringet).

Két lehetséges megoldás is kínálkozik azok alapján, amit tanultunk Javából. Az egyik megoldás, hogy rádelegáljuk a konkrét osztályokra, hogy intézzék el ők a dolgot, ehhez az ősosztályban kell deklarálnunk ehhez egy függvényt:

```
interface Noti { void handle(); }
class SMS implements Noti {
  public final int number;
  public SMS( int number ){ this.number = number; }
  public void handle(){ System.out.println( number ); }
}
class Email implements Noti {
  public final String address;
  public Email( String address ){ this.address = address; }
  public void handle() { System.out.println( address ); }
}
class Handler {
  public void handle( Noti noti ) { noti.handle(); }
}
```

Ez azért működik, mert Javában minden nem `final` metódus virtuális, tehát majd futásidőben a paraméterként érkező `Notinak` a run-time típusa fogja eldönteni, hogy a `noti.handle();` az melyik alosztály konkrét metódusa legyen.

A másik megoldás, hogy nem az osztályokba tesszük a feldolgozó kódot, hanem a `Handler` függvény sorbapróbálja a szóba jövő osztályokat `instanceof` checkekkel, amelyik bejön, arra castolja az argumentumot és annak megfelelően kezeli le. Hogy eggyel szebb legyen, a kezelő függvényeket külön rakom:

```
interface Noti { }
class SMS implements Noti {
  public final int number;
  public SMS( int number ){ this.number = number; }
}
class Email implements Noti {
  public final String address;
  public Email( String address ){ this.address = address; }
}
class Handler {
  public void handle( SMS sms ){ System.out.println( sms.number ); }
  public void handle( Email email ){ System.out.println( email.address ); }
  public void handle( Noti noti ) {
    if( noti instanceof SMS ){ handle( (SMS)noti );}
```

```
    else if( noti instanceof Email ){ handle( (Email)noti );}
    else { /* ismeretlen Noti subclass handler code */ }
  }
}
```

Ez így nem csak azért szebb, mintha az `instanceof` checkbe raknánk castolás után egyetlen `handle(Noti)` metódussal az egész kezelőt, mert modulárisabb és könnyebb átlátni, hanem azért is, mert ha tudom, hogy egy SMS-t kapok, mint pl. egy `SMS sms = new SMS(7); new Handler().handle(sms);` kódban, akkor egyből az SMSt kapó handler függvény kapja meg a vezérlést, skippelve az `instanceof` checkeket. Ha a fenti `sms` változó típusa pedig `Noti` (de a kód többi része ugyanaz marad), akkor pedig a `handler(Noti)` kapja meg, az elvégzi a típusellenőrzést, és aztán dobja feljebb kezelésre.

Persze ha mondjuk az `SMS` osztálynak még további alosztályai is lesznek később, akkor a `handle(SMS)` függvényen belülre is kell tegyünk egy-egy `instanceof` checket az összes abból leszármazott osztályra...

A meglévő architektúrával kétféle változás történhet: vagy az osztályhierarchia változik (bejön pl. egy `IM` új noti alosztály), vagy a funkcionalitások köre bővül (bejön pl. egy másik feladat, mondjuk nem hogy printeljünk ki az egyetlen mezőt, hanem hogy szúrjuk be a konkrét osztálytól függő adattáblába egy adatbázisban).

Az első megvalósítás akkor jobb, ha az osztályhierarchia változik és a funkcionalitások köre (legalábbis ezeké) viszonylag fix, és logikailag tényleg „az osztályhoz tartozik szervesen”, ilyenkor az új alosztályban implementálni kell a leszármazott `handle` függvényt is. Tipikusan ilyen ugye pl. az `Object.toString()` is, azzal, hogy ennek van egy default implementációja. Ekkor a `Handler` kódjához nem kell nyúljunk. Ha viszont az elvárt funkcionalitások köre bővül, akkor kell egy új handler osztály, ami ugyanúgy egysoros, mint ez a másik, de egy másik (mondjuk `handle2() :P`) függvényét hívja a `Noti` interface-nek, és ezt a függvényt ekkor az összes konkrét osztályban implementálnunk kell. Ilyenkor ez nagy költség.

Ha viszont a hierarchia többé-kevésbé fix, és a funkcionalitások köre nő gyakrabban, akkor a második megvalósítás talán a jobb: egy új funkcióhoz egy ugyanekkora `AnotherHandler` osztályt kell szerkesszünk egy hosszú `instanceof` checkkel és egy-egy kezelő metódussal minden konkrét alosztálynak. Az osztályokhoz nem kell nyúlnunk ekkor. Ez a struktúra pedig az osztályhierarchia bővítését „tűri rosszul”: ha plusz osztályt teszünk be, minden handlert át kell írnunk, hogy működjön arra is (amelyiket nem írjuk át, annak mondjuk lefut az „ismeretlen Noti subclass handler code”-ja, tehát legalább lefordul).

Ennek a második megközelítésnek egy `instanceof` check nélküli megvalósítását adja a `Visitor` tervezési minta. Ebben a (konkrét) `Handler` osztály(ok) mind implementálnak egy `Visitor` interfacet, a `Noti` osztályok pedig egy `Visitable` interfacet. A lényeg, hogy az objektumokat (az `SMS`, `Email` stb. osztályúakat) úgy tekintjük, mint akiket végig lehet látogatni, a rajtuk műveleteket végzőket (mint a `Handler`) pedig úgy, mint akik meglátogatják őket. Ennek megfelelően a `Visitable` objektumoknak lesz egy `accept(Visitor)` függvényük, a `Visitor` objektumoknak pedig egy `visit(Visitable)` függvényük... és utóbbiaknak a `visit` függvényük minden alosztályra külön le is lesz specifikálva. Kód:

```
interface Visitable { public void accept( Visitor v ); }
interface Noti extends Visitable {}
class SMS implements Noti{
  public final int number;
  public SMS( int number ){ this.number = number; }
  public void accept( Visitor v ){ v.visit( this ); }
```

```
}
class Email implements Noti{
  public final String address;
  public Email( String address ){ this.address = address; }
  public void accept( Visitor v ){ v.visit( this ); }
}
interface Visitor{
  public void visit( SMS sms );
  public void visit( Email email );
  public void visit( Noti noti );
  public void handle( Noti noti );
}
class Handler implements Visitor {
  public void visit( SMS sms ){ System.out.println( sms.number ); }
  public void visit( Email email ){ System.out.println( email.address ); }
  public void visit( Noti noti ){ /* unknown Noti handler code */}
  public void handle( Noti noti ){ noti.accept(this); }
}
```

Nézzük, mi is történik akkor, ha létrehozunk egy `Noti sms = new SMS(42);`t és odaadjuk egy Handlernek `new Handler().handle( sms );`?

Először is, az már fordítási időben eldől, hogy mivel az `sms` változó deklarált típusa `Noti`, ezért a `Handler.handle( Noti )` metódust hívja meg. Ha `new Handler().visit( sms );`-t hívtunk volna, az is ugyanígy a default `Noti`-t váró metódust hívta volna, de nem ezt akarjuk, hanem hogy fusson rá az `SMS`re specifikált overloadra. Ekkor a `noti.accept(this);` hívás viszont már a `noti` (azaz az `sms`) futásidejű típusa alapján dől el, hogy melyik acceptről beszélünk: tehát az `SMS.accept(Visitor v);` fog futni (fordítási időben az dől el, hogy egy `Noti.accept(Visitor v)` kerül majd hívásra, de hogy a `Noti` melyik konkrét osztályának az `accept`je, az csak futás közben). Az `SMS.accept` pedig visszahívja a `v.visit(this);`-t – viszont mivel ez az `SMS` osztályon belül van deklarálva, ekkor a fordító tudja, hogy a `this` az egy `SMS` példány lesz, tehát a `Visitor`nak a `visit(SMS)` metódusát fogja hívni. (Ezért kell explicit felsorolnunk a `Visitor` interface-ben az összes leszármazott osztályt; ha nem tennénk, csak a `visit(Noti)` lenne ott, akkor az alosztálynak is a `visit(Noti)`ját hívná meg, hiába lenne abban az alosztályban egy `visit(SMS)` overload is.) Tehát végül is ráfut a vezérlés a `Handler.visit(SMS)` metódusra és at lekezeli szépen, ami a feladat.

Ezt a megoldást *double dispatch*nak hívják, amikor ezzel az oda-vissza hívással végül is megtudjuk mindkét objektum futásidejű típusát.

A `Visitor` pattern is jól tűri, ha új funkció kerül be: csak kell írnunk egy új `Handler` osztályt, és implementálni a `Visitor` összes függvényét a megfelelő módon. Új osztály megjelenésével a hierarchiában viszont ugyanúgy gondok vannak: az új osztály be kell kerüljön a `Visitor` interface-be argumentumként és lehet újraírni az összes handler kódot, mert már nem implementálnak mindent. Erre (mármint hogy legalább forduljon) az egy megoldás lehet, ha a `Visitor` nem interface lesz, hanem egy osztály, ami defaultból minden alosztályra az ős `visit`jét hívja:

```
interface Visitable { public void accept( Visitor v ); }
interface Noti extends Visitable {}
class SMS implements Noti{
  public final int number;
  public SMS( int number ){ this.number = number; }
```

```
  public void accept( Visitor v ){ v.visit( this ); }
}
class Email implements Noti{
  public final String address;
  public Email( String address ){ this.address = address; }
  public void accept( Visitor v ){ v.visit( this ); }
}
class Visitor{
  public void visit( SMS sms ){ visit((Noti)sms); }
  public void visit( Email email ){ visit((Noti)email);}
  public void visit( Noti noti ){ /*default do nothing*/ }
  public void handle( Noti noti ){ noti.accept(this); }
}
class Handler implements Visitor {
  public void visit( SMS sms ){ System.out.println( sms.number ); }
  public void visit( Email email ){ System.out.println( email.address ); }
}
```

Annyi előnye ennek a módszernek mindenképp van, hogy a `Handler` osztályban csak azt kell implementáljuk, amivel foglalkozni akarunk, a többire (beleértve a `handle` metódust is) teljesen jó lesz az ősosztály implementációja is. Egyébként az ANTLR Visitor generátorai pont ilyet generálnak, azzal a különbséggel, hogy `visitSMS`, `visitEmail` stb. nevű metódusokat hoznak létre.

Egy további megoldás, ha reflectiont használva megtudjuk, hogy melyik az a legspecifikusabb ősosztály, akire van visit metódusunk. Ekkor a `Noti`, `SMS` és `Email` osztályokba nem kell semmi, a handler osztály pedig:

```
class BaseReflectiveVisitor {
  public void handle( Object object){
    Class theClass = object.getClass();
    while( true ){
      try {
        Method method = getClass().getMethod("visit", theClass );
        try {
          method.invoke(this, object );
          return;
        } catch (IllegalAccessException e) {
          e.printStackTrace();
        } catch (IllegalArgumentException e) {
          e.printStackTrace();
        } catch (InvocationTargetException e) {
          e.printStackTrace();
        }
      } catch (NoSuchMethodException e) {
        theClass = theClass.getSuperclass();
      } catch (SecurityException e) {
        //nem jo vilag van
        e.printStackTrace();
      }
    }
  }
}
```

```
class NotiVisitor extends BaseReflectiveVisitor {
  public void visit( SMS sms ) { System.out.println( sms.number ); }
  public void visit( Email email ){ System.out.println( email.address ); }
}
```

Ez is világos: a `handle` metódus végigkérdezi magát a `Visitor`t, hogy a kezelendő objektum melyik első őse van kezelve egyáltalán `visit` metódussal, az első ilyet meg meg is hívjuk[6].

Na nézzük, hogy lehet ezt implementálni Scalaban.

NEMSOKÁRA, DE ELŐBB SBT

Mivel gyakran használunk különböző libeket is (mint ahogy pl. használjuk a ScalaTestet és a Scalacticot a kódjainkhoz tesztesetek gyártására), kulturált megoldás használni egy build toolt (az IDEn kívül), ami jobban manageli pl. a használt libeket (sok más dolog mellett), mint mondjuk az Eclipse. Magát a kódot persze a kedvenc IDEnkben (Eclipse, IntelliJ vagy akár VIM `:D`) tudjuk szerkesztgetni mellette.

A Scalához a kézenfekvő build tool az sbt (http://www.scala-sbt.org/index.html), amit admin jogok híján a kabinetben telepíteni ugyan nem tudunk, de portable is működik. Hogy felélesszük, bekonfiguráljuk és az Eclipse-el is kompatibilissé tegyük, a következő lépéseket kell tennünk:

1. Töltsük le a ZIPet vagy a TGZt innen és csomagoljuk ki bárhova. A példában nekem ez a `C:/Users/szabivan/sbt` lesz. (Ezt csak egyszer kell megcsinálni.) A továbbiakban ezt a helyet `SBT_DIR`nek hívom majd.

2. Ha új projektet akarunk létrehozni, annak is hozzunk létre valahol egy könyvtárat, nálam ez a `C:/Users/szabivan/dev/workspace-sbt/PatternMatch` lesz. Ezt meg `PROJECT_DIR`nek fogom hívni.

3. Ha van jogosultságunk változtatni a `PATH`on (vagy a linux ekvivalensén), akkor érdemes az `SBT_DIR/bin` könyvtárat beletenni. Ha nincs, legegyszerűbb persze létrehozni egy `sbt.bat`ot a `PROJECT_DIR`ben annyitartalommal, hogy `SBT_DIR/bin/sbt.bat`.

4. Ha most egy command promptból a `PROJECT_DIR`ből indítjuk az `sbt.bat`ot, az le is tölt sok mindent (eltart egy darabig), ami ahhoz kell, hogy maga az sbt működjön, mindenből lehúzza a legfrissebbet:

[6] A teljesség kedvéért persze pl. ha nem találunk ilyet egyáltalán, akkor végigjárhatnánk az összes implementált interface-t is, hátha.

```
C:\Windows\System32\cmd.exe - sbt
Microsoft Windows [verziószám: 6.1.7601]
Copyright (c) 2009 Microsoft Corporation. Minden jog fenntartva.

c:\Users\szabivan\Documents\sbt>sbt

c:\Users\szabivan\Documents\sbt>bin/sbt.bat
Java HotSpot(TM) 64-Bit Server VM warning: ignoring option MaxPermSize=256m; sup
Getting org.fusesource.jansi jansi 1.11 ...
downloading https://repo1.maven.org/maven2/org/fusesource/jansi/jansi/1.11/jansi
        [SUCCESSFUL ] org.fusesource.jansi#jansi;1.11!jansi.jar (181ms)
:: retrieving :: org.scala-sbt#boot-jansi
        confs: [default]
        1 artifacts copied, 0 already retrieved (111kB/209ms)
Getting org.scala-sbt sbt 0.13.13 ...
downloading https://repo.typesafe.com/typesafe/ivy-releases/org.scala-sbt/sbt/0.
        [SUCCESSFUL ] org.scala-sbt#sbt;0.13.13!sbt.jar (2063ms)
downloading https://repo1.maven.org/maven2/org/scala-lang/scala-library/2.10.6/s
        [SUCCESSFUL ] org.scala-lang#scala-library;2.10.6!scala-library.jar (120
downloading https://repo.typesafe.com/typesafe/ivy-releases/org.scala-sbt/main/0
        [SUCCESSFUL ] org.scala-sbt#main;0.13.13!main.jar (4571ms)
downloading https://repo.typesafe.com/typesafe/ivy-releases/org.scala-sbt/compil
..
        [SUCCESSFUL ] org.scala-sbt#compiler-interface;0.13.13!compiler-interfac
downloading https://repo.typesafe.com/typesafe/ivy-releases/org.scala-sbt/action
        [SUCCESSFUL ] org.scala-sbt#actions;0.13.13!actions.jar (2278ms)
downloading https://repo.typesafe.com/typesafe/ivy-releases/org.scala-sbt/main-s
        [SUCCESSFUL ] org.scala-sbt#main-settings;0.13.13!main-settings.jar (263
downloading https://repo.typesafe.com/typesafe/ivy-releases/org.scala-sbt/interf
        [SUCCESSFUL ] org.scala-sbt#interface;0.13.13!interface.jar (2015ms)
downloading https://repo.typesafe.com/typesafe/ivy-releases/org.scala-sbt/io/0.1
        [SUCCESSFUL ] org.scala-sbt#io;0.13.13!io.jar (2257ms)
downloading https://repo.typesafe.com/typesafe/ivy-releases/org.scala-sbt/ivy/0.
        [SUCCESSFUL ] org.scala-sbt#ivy;0.13.13!ivy.jar (3158ms)
downloading https://repo.typesafe.com/typesafe/ivy-releases/org.scala-sbt/loggin
        [SUCCESSFUL ] org.scala-sbt#logging;0.13.13!logging.jar (2097ms)
downloading https://repo.typesafe.com/typesafe/ivy-releases/org.scala-sbt/logic/
        [SUCCESSFUL ] org.scala-sbt#logic;0.13.13!logic.jar (2020ms)
downloading https://repo.typesafe.com/typesafe/ivy-releases/org.scala-sbt/proces
        [SUCCESSFUL ] org.scala-sbt#process;0.13.13!process.jar (2090ms)
downloading https://repo.typesafe.com/typesafe/ivy-releases/org.scala-sbt/run/0.
        [SUCCESSFUL ] org.scala-sbt#run;0.13.13!run.jar (2041ms)
downloading https://repo.typesafe.com/typesafe/ivy-releases/org.scala-sbt/comman
        [SUCCESSFUL ] org.scala-sbt#command;0.13.13!command.jar (3003ms)
downloading https://repo1.maven.org/maven2/org/scala-sbt/launcher-interface/1.0.
        [SUCCESSFUL ] org.scala-sbt#launcher-interface;1.0.0-M1!launcher-interfa
downloading https://repo.typesafe.com/typesafe/ivy-releases/org.scala-sbt/classp
        [SUCCESSFUL ] org.scala-sbt#classpath;0.13.13!classpath.jar (2037ms)
downloading https://repo.typesafe.com/typesafe/ivy-releases/org.scala-sbt/comple
        [SUCCESSFUL ] org.scala-sbt#completion;0.13.13!completion.jar (2429ms)
```

Ez is csk első alkalommal fog persze eddig tartani. Lépjünk ki az sbt-ből `exit`tel, még be kell állítsunk pár dolgot, hogy mindenünk meglegyen.

5. Ha olyan projectet akarunk készíteni, amit aztán meg tudunk nyitni sbt-ben (az IntelliJ elvileg meg tudja nyitni simán az sbt projectet, csak az Eclipse-nek kell plugin), akkor az itt leírt módon tegyük ezt meg. Én ezt projekt szinten adom meg, tehát a `PROJECT_DIR/project/plugins.sbt` file-ba beírom az

```
addSbtPlugin("com.typesafe.sbteclipse" % "sbteclipse-plugin" % "5.1.0")
```

sort.

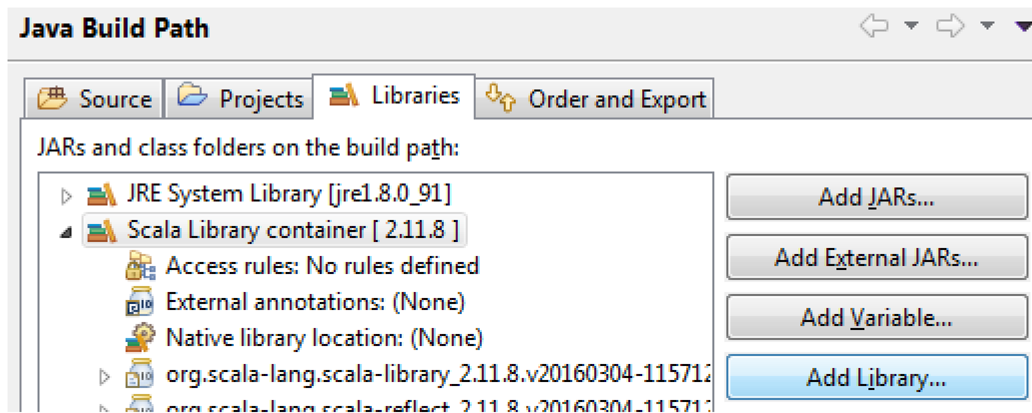
6. Ha ezt követően indítjuk a `PROJECT_DIR`ből az sbt-t, akkor ott az `eclipse` parancs beírásával generál nekünk egy Eclipse project filet is (amihez megint csak letölt pár dolgot előbb), kép:

```
C:\Windows\System32\cmd.exe - sbt

c:\Users\szabivan\dev\workspace-sbt\PatternMatch>sbt

c:\Users\szabivan\dev\workspace-sbt\PatternMatch>c:\Users\szabivan\sbt\bin\sbt.bat
Java HotSpot(TM) 64-Bit Server VM warning: ignoring option MaxPermSize=256m; support was removed
[info] Loading project definition from C:\Users\szabivan\dev\workspace-sbt\PatternMatch\project
[info] Updating {file:/C:/Users/szabivan/dev/workspace-sbt/PatternMatch/project/}patternmatch-bu
[info] Resolving org.fusesource.jansi#jansi;1.4 ...
[info] downloading https://repo.scala-sbt.org/scalasbt/sbt-plugin-releases/com.typesafe.sbteclip
t_0.13/5.1.0/jars/sbteclipse-plugin.jar ...
[info]  [SUCCESSFUL ] com.typesafe.sbteclipse#sbteclipse-plugin;5.1.0!sbteclipse-plugin.jar (222
[info] downloading https://repo1.maven.org/maven2/org/scalaz/scalaz-core_2.10/7.2.5/scalaz-core_
[info]  [SUCCESSFUL ] org.scalaz#scalaz-core_2.10;7.2.5!scalaz-core_2.10.jar(bundle) (1467ms)
[info] downloading https://repo1.maven.org/maven2/org/scalaz/scalaz-effect_2.10/7.2.5/scalaz-eff
[info]  [SUCCESSFUL ] org.scalaz#scalaz-effect_2.10;7.2.5!scalaz-effect_2.10.jar(bundle) (159ms)
[info] Done updating.
[info] Set current project to patternmatch (in build file:/C:/Users/szabivan/dev/workspace-sbt/P
> eclipse
[info] About to create Eclipse project files for your project(s).
[info] Updating {file:/C:/Users/szabivan/dev/workspace-sbt/PatternMatch/}patternmatch...
[info] Resolving org.fusesource.jansi#jansi;1.4 ...
[info] Done updating.
[info] Resolving org.scala-lang#scala-compiler;2.10.6 ...
[info] downloading https://repo1.maven.org/maven2/org/scala-lang/scala-library/2.10.6/scala-libr
[info]  [SUCCESSFUL ] org.scala-lang#scala-library;2.10.6!scala-library.jar(src) (416ms)
[info] downloading https://repo1.maven.org/maven2/org/scala-lang/scala-library/2.10.6/scala-libr
[info]  [SUCCESSFUL ] org.scala-lang#scala-library;2.10.6!scala-library.jar(doc) (3549ms)
[info] downloading https://repo1.maven.org/maven2/org/scala-lang/jline/2.10.6/jline-2.10.6-javad
[info]  [SUCCESSFUL ] org.scala-lang#jline;2.10.6!jline.jar(doc) (206ms)
[info] downloading https://repo1.maven.org/maven2/org/scala-lang/jline/2.10.6/jline-2.10.6-sourc
[info]  [SUCCESSFUL ] org.scala-lang#jline;2.10.6!jline.jar(src) (149ms)
[info] downloading https://repo1.maven.org/maven2/org/scala-lang/scala-reflect/2.10.6/scala-refl
[info]  [SUCCESSFUL ] org.scala-lang#scala-reflect;2.10.6!scala-reflect.jar(doc) (3565ms)
[info] downloading https://repo1.maven.org/maven2/org/scala-lang/scala-reflect/2.10.6/scala-refl
[info]  [SUCCESSFUL ] org.scala-lang#scala-reflect;2.10.6!scala-reflect.jar(src) (176ms)
[info] downloading https://repo1.maven.org/maven2/org/fusesource/jansi/jansi/1.4/jansi-1.4-javad
[info]  [SUCCESSFUL ] org.fusesource.jansi#jansi;1.4!jansi.jar(doc) (159ms)
[info] downloading https://repo1.maven.org/maven2/org/fusesource/jansi/jansi/1.4/jansi-1.4-sourc
[info]  [SUCCESSFUL ] org.fusesource.jansi#jansi;1.4!jansi.jar(src) (144ms)
[info] downloading https://repo1.maven.org/maven2/org/scala-lang/scala-compiler/2.10.6/scala-com
[info]  [SUCCESSFUL ] org.scala-lang#scala-compiler;2.10.6!scala-compiler.jar(src) (374ms)
[info] downloading https://repo1.maven.org/maven2/org/scala-lang/scala-compiler/2.10.6/scala-com
[info]  [SUCCESSFUL ] org.scala-lang#scala-compiler;2.10.6!scala-compiler.jar(doc) (1466ms)
[info] Successfully created Eclipse project files for project(s):
[info] patternmatch
>
```

7. Most már a (per pillanat tök üres) projektünket meg tudjuk nyitni Scala IDE-ben: a Scala IDE-ből az `Import existing project...`, majd kiválasztva a `PROJECT_DIR`t meg is nyitja. Jó eséllyel lesz sipákolás, hogy az a scala lib, amit a Scala IDE használ, nem ugyanaz, mint amit az sbt húzott le legfrissebbet. A képen pl. nekem a Scala IDE még 2.10-essel jött le, a legfrissebb meg a 2.11.8, amit az sbt töltött le. (A Scala IDE bundle nem mindig a legfrissebbel érkezik.)

   Errors (1 item)
   The version of scala library found in the build path (2.11.8) is incompatible with the one expected by scala IDE (2.10). Plea

   Használjuk inkább a 2.11-et, ha már egyszer lehúzta nekünk az sbt, szerkesszük át a Scala IDE-ben a build pathot, tegyük rá a 2.11.8-as (vagy amit az sbt épp lehúzott) Scala Library Containert:



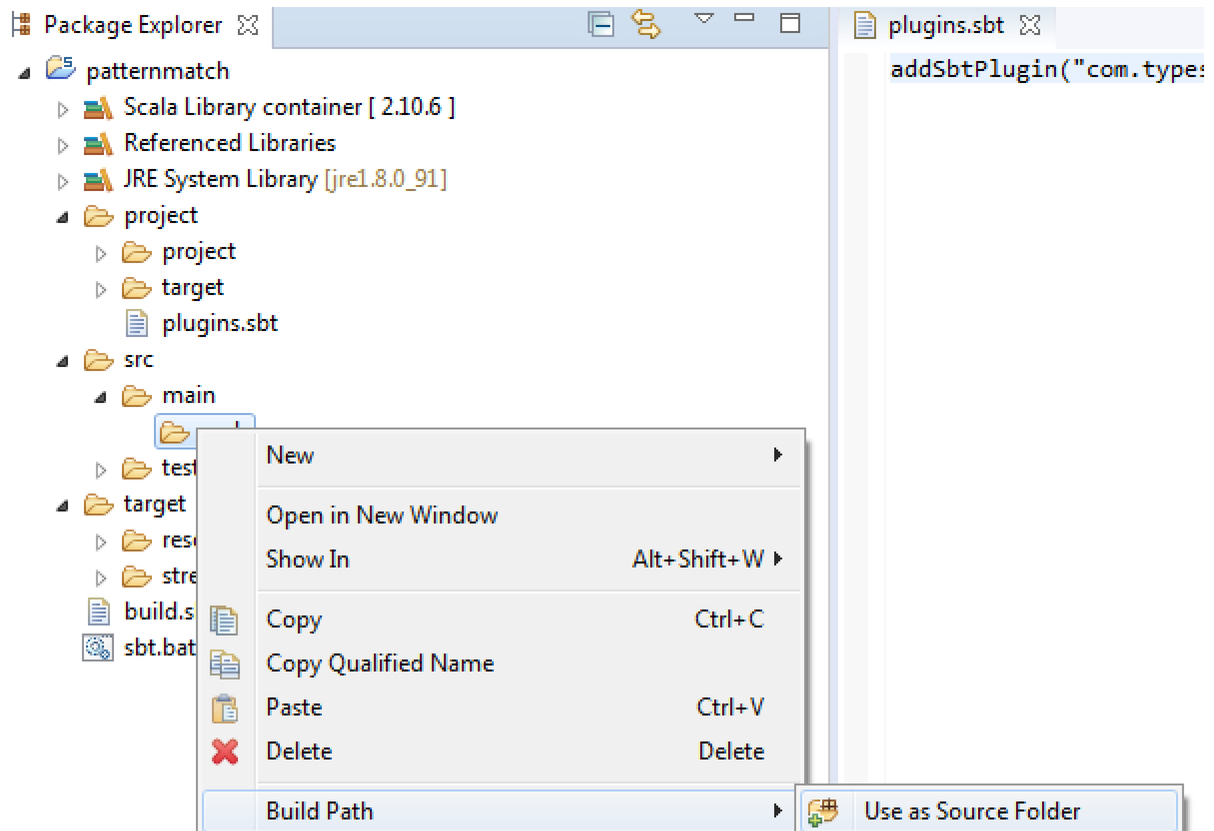
Elvileg most már fordulni fog a Scala IDE-ből is az sbt project.

8. A metódusaink és osztályaink tesztelésére a Scalatestet és a Scalacticot fogjuk használni. Hogy ez a két lib is rákerüljön a build pathra, az itt leírt módon hozzunk létre a `PROJECT_DIR/build.s` file-t ezzel a tartalommal:

```
scalaVersion := "2.11.1"
libraryDependencies += "org.scalactic" %% "scalactic" % "3.0.1"
libraryDependencies += "org.scalatest" %% "scalatest" % "3.0.1" % "test"
```

   Az első sor azért kell, hogy az sbt akkor is ragaszkodjon a frissebb scalához, ha az Eclipse netán mégiscsak felül akarná bírálni. Ha ezek után az sbt-vel újrageneráltatjuk az Eclipse projektet (az `eclipse` paranccsal megint, letölti ezeket is), akkor már a Scala IDEben (reload vagy refresh után) látjuk a két referenced libraryt, amit most adtunk hozzá.

9. Hogy az sbt-ből teszteljünk, a tesztеseteinket a `PROJECT_DIR/src/text/scala` könyvtárba hozzuk majd létre, magát a kódot pedig érdemes a `PROJECT_DIR/src/main/scala` könyvtárba pakolni, ezeket hozzuk is létre. Adjuk hozzá mint source foldert a build pathhoz.



Most már dolgozhatunk a Scala IDE-ben és könnyebb lesz majd tesztelni, ha közben fut az sbt, csak ki kell majd adjuk ott a `test` parancsot.

Tehát, a task az volt, hogy legyenek Notik, Emailek és SMSek, meg egy Handler, aki kezeli őket. Első közelítésben (mondjuk a `noti` package-en belül) hozzunk létre egy `Noti` nevű trait-et (egy trait kb. egy java interface, de lehetnek benne implementált metódusok is), és két leszármazott osztályt, megint csak úgy, hogy az `Email`nek legyen egy `address: String` immutable mezője, a `SMS`nek pedig egy `number: Int` mezője. Első változat, amin tudok mesélni a Scala osztályokról:

```
package noti

trait Noti

class Email( _address : String ) extends Noti {
  val address = _address
}

class SMS( _number : Int ) extends Noti {
  val number = _number
}
```

A szintaxis: ha valami üres törzsű (mint pl a `Noti` trait), ahhoz nem kell kapcsos, se a szokott módon pontosvessző. Másrészt nem kell minden egyes osztályt külön forrásfile-ba pakolni (de aki ezt szereti, tegye).

A `class` kulcsszóval deklaráltunk egy `Email` nevű osztályt. Az osztály neve utáni `( _address : String )` pedig a primary konstruktora. (!) Tehát ez Java-ban kb. így nézne ki:

```
class Email implements Noti {
  public final String address;
  Email( _address : String ) { this.address = _address; }
}
```

Még egyszer: a primary konstruktor argumentumlistája az osztálynév mellett közvetlenül. Az `extends` kulcsszót használjuk, akár egy traitből, akár egy másik classból „származunk le”, itt nincs külön `implements` és külön `extends` kulcsszó. A konstruktor kódja pedig *maga az osztály törzsének a tartalma*, tehát ha példányosítunk egy új `Email` objektumot, akkor létrehozunk neki egy `address` immutable mezőt, ami megkapja a konstruktorban érkező értéket.

Hogy lássuk, hogy írunk teszteseteket a kódjainkhoz, hozzunk létre mondjuk a `src/test/scala` forráskönyvtárban (adjuk hozzá ezt is a build pathban a source folderekhez) szintén a `noti` package-ben a `NotiTest` osztályt:

```
package noti

import org.scalatest._

class NotiTest extends FlatSpec {
  "An Email object" should "remember its constructor argument as a field" in {
    val address = "szabivan@inf.u-szeged.hu"
    val email = new Email( address )
    assert( email.address === address)
  }
```

```
}
```

Tehát első közelítésből egy teszteset file a `FlatSpec`ből származik és a konstruktorába (tehát az osztály törzsébe) írunk teszteseteket kb. ezzel a szintaxissal (részletesen később): a mostani egyetlen tesztesetünk azt nézi, hogy vajon sikerült-e a konstruktor argumentet lerakni a mezőbe.

És ha most sbt-ben beírjuk, hogy `test`, az sbt szépen lefordítja a teszteseteinket és hát mivel ügyesen lementettük a mezőt, az szép zöld lesz:

```
[info] Successfully created Eclipse project files for project(s):
[info] patternmatch
> test
[info] Compiling 1 Scala source to C:\Users\szabivan\dev\workspace-sbt\PatternM
[info] 'compiler-interface' not yet compiled for Scala 2.11.1. Compiling...
[info]   Compilation completed in 11.784 s
[info] Compiling 1 Scala source to C:\Users\szabivan\dev\workspace-sbt\PatternM
[info] NotiTest:
[info] An Email object
[info] - should remember its constructor argument as a field
[info] Run completed in 489 milliseconds.
[info] Total number of tests run: 1
[info] Suites: completed 1, aborted 0
[info] Tests: succeeded 1, failed 0, canceled 0, ignored 0, pending 0
[info] All tests passed.
[success] Total time: 20 s, completed 2017.02.28. 17:05:00
>
```

Persze elsőre ez is lassú, amikor először forgatunk tesztesetet. Ha eltoljuk (írjuk át, hogy mivel legyen egyenlő az érték), piros lesz:

```
[success] Total time: 1 s, completed 2017.02.28. 17:08:05
> test
[info] Compiling 1 Scala source to C:\Users\szabivan\dev\workspace-sb
[info] NotiTest:
[info] An Email object
[info] - should remember its constructor argument as a field *** FAIL
[info]   "...ivan@inf.u-szeged.hu[]" did not equal "...ivan@inf.u-sze
[info] Run completed in 360 milliseconds.
[info] Total number of tests run: 1
[info] Suites: completed 1, aborted 0
[info] Tests: succeeded 0, failed 1, canceled 0, ignored 0, pending 0
[info] *** 1 TEST FAILED ***
[error] Failed tests:
[error]         noti.NotiTest
[error] (test:test) sbt.TestsFailedException: Tests unsuccessful
[error] Total time: 1 s, completed 2017.02.28. 17:08:55
```

Egyelőre a tesztelésről ennyit. Térjünk vissza a `Noti` feladathoz.

Nagyon sok esetben ha az ember Javában fejleszt[7], akkor a konstruktorban adott paramétereket egyből egy-egy mezőbe rakja. Mivel ez ennyire gyakori minta, Scala-ban a konstruktor paraméterekből egyes esetekben automatikusan member fieldek generálódnak getterrel és esetleg setterrel:

Scala Cookbook 4.2

- A módosítószó nélküli konstruktor paraméterekhez nem generálódik se getter, se setter (ez volt az előbb az `SMS` és a `Email` esetében).
- Ha a paraméter `val` módosítót kap, akkor egy ugyanilyen nevű getter metódust készít hozzá.
- Ha a paraméter `var` módosítót kap, akkor kap getter és setter metódust; a setter metódus szintaktikailag úgy néz ki, mintha értéket adnánk ennek a mezőnek.
- Ha a paraméter ezen kívül még a `private` módosítót is megkapja (tehát `private var`

[7] a sok Java-hoz való méricskélés nem azért van, hogy ekézzem a Javát, hanem természetesen adódik a JVM miatt, ami a Scala osztályokat is futtatja és mert a Java és a Scala elég jól illeszthető emiatt is egymáshoz, erre még visszatérek

vagy `private val`), akkor a fieldet csak az osztályon belülről érhetjük el (`val` esetében csak getterrel, `var` esetében setterrel is).

Tehát a második változatunk a `Noti` esetére, ami az előzővel ekvivalens:

```
package noti
trait Noti
class Email( val address: String ) extends Noti
class SMS( val number : Int ) extends Noti
```

És ekkor pl. így tudjuk használni:

```
package noti

trait Noti
class Email( val address : String ) extends Noti
class SMS( val number : Int ) extends Noti

object Main extends App {
  val e = new Email( "sanyi@otthon.com" )
  println( e.address ) // OK, it works
  e.address = "ss"
      reassignment to val
}
```

Tehát a `val` mezőkből automatikusan generálódott egy field, amit olvasni tudunk, megváltoztatni nem. Ha `var` lenne, akkor fordulna az utolsó sor is.

A fenti példára: láthatjuk, hogy most nem egy `object`nek a `main` metódusát futtattam, hanem egy `App`-ot kiterjesztő objektumba írtam bele. Most már tudjuk, hogy az „objektum törzsébe beleírás” az a konstruáláskor lefutó kódot jelenti; tehát ha futtatható kódot szeretnék, akkor csak annyi dolgom van, hog az `App`ból származtassak egy objektumot, és ennek a konstruktorába írjam a main kódot, less boilerplate. (Erről egyelőre ennyit.)

Most, hogy az osztályaink megvannak, szeretnénk írni egy „ha a `Noti` amit kaptunk, az egy `Email`, akkor a címmel, ha meg `SMS`, akkor a számmal kezdjünk valamit” kódot. Mentalitásában leginkább az „instanceof-cast” módszerhez fog hasonlítani a megoldásunk, tehát a handler kód fogja megnézni, hogy ami bejött, az melyik leszármazott osztály, olyan osztályúnak fogja kezelni, és kinyeri a mezőket.

Hogy ezt meg tudjuk csinálni, megismerkedünk a rekurzió mellett a funkcionális toolbox egy másik alap elemével, a *mintaillesztéssel*. Kezdjük egy példán keresztül:

```
def doStuff( a : Any ) : Unit = a match {
  case 5 => println( "a == 5" )
  case true => println( "a is true" )
  case "5" => println( "a egy String, konkretan '5'")
  case s : String => println( "a egy String, de nem '5', hanem " + s )
  case _ => println("a fentiek kozul egyik sem, hanem " + a.toString() )
}
```

Tehát szintaktikailag egy mintaillesztés egy `match`-kifejezés: a match kulcsszó bal oldalán szerepel a kifejezés, amit illesztünk (most épp az `a`, ami `Any`, tehát bármilyen típusú lehet, nemsokára megnézzük a Scala típusokat is, egyelőre elég ennyit tudjunk róla), a `match` után pedig minták vannak felsorolva, `case minta => kifejezés` alakban. Egy ilyen kifejezés értékét úgy kapjuk, hogy a Scala megpróbálja illeszteni először az első mintát, hogy illeszkedik-e az inputra, ha

igen, akkor az első minta kifejezése lesz az érték, ha nem, akkor próbálja illeszteni a másodikat és így tovább. Ami fontos lehet:

- Nincs átmászkálás, mint a `switch/case` esetben Javában és Cben: a kifejezés értéke az a kifejezés lesz, ami a minta után, a következő mintáig tart.
- Ha egyetlen minta sem illeszkedik, akkor dob egy `MatchError`-t.

A teljesség igénye nélkül nézzük meg a fenti példákat, hogy mi is lehet minta:

- Literál (mint fentebb az `5`, `true`, `"5"`): ha egy konkrét objektumot írunk mintának, akkor a minta akkor illeszkedik, ha *egyenlő* az illesztett kifejezéssel (tehát lefut az `==` metódus, ami Scalában ugyanaz, mint az `equals`).
- Típusos változó (mint fentebb a `s : String`): ez minden `String` típusú kifejezésre illeszkedik, és illeszkedés esetén az `s` változót köti (*binds*): tehát `s` a case jobb oldalában már a „castolt” kifejezés értéke lesz. Ez így kb. ekvivalens az `if (a instanceof String ){ s = (String) a; ... }` kóddal.
- Típus nélküli változó (pl. `case s => ...`) mindenre illeszkedik, és köti az `s` változót. (Ennek összetett minták belsejében lesz értelme.) Azt érdemes tudni, hogy mintában a változónevek *kisbetűvel kell kezdődjenek*, és eltakarják a kintebbi blokkokban esetleg ugyanilyen néven deklarált változókat. (Tehát ha van egy `value` változóm még a match kifejezésen kívül deklarálva, akkor a `case value => ...` eset *nem* fog egyenlőséget tesztelni a kintebbi változóval, hanem illeszkedik és az új, belső `value` nevű változónak az illesztett kifejezés lesz az értéke a case jobb oldalán.)
- Változók helyett a `_` placeholdert is használhatjuk, ez mindenre illeszkedik és nem köt nevet, önmagában ez default ágként állhat (mint a fenti példában is), típusolni is lehet (`_ : String => println("ez egy String")`).
- *Stabil azonosító* is lehet, erre egy példa: `` case `value` => ... `` akkor illeszkedik, ha a matchen kívül deklarált `value` nevű `val` (!) változóéval egyenlő a kifejezés értéke. A nagybetűs változóneveket alapból stabil azonosítóként keresi a mintaillesztő, a kisbetűkeset backtickelni kell.

Az eddigiek alapján a `Noti` handlerjét így is megírhatjuk:

```
def checkNoti( n : Noti ) : Unit = n match {
  case e : Email => println( e.address )
  case s : SMS => println( s.number )
  case _ => println("Unknown Noti type!")
}

val v = new Email( "sanyi@otthon.com" )
val w = new SMS( 666 )

checkNoti( v ) // prints sanyi@otthon.com
checkNoti( w ) // prints 666
```

Ez így kb. egy az egyben megfelel a Javánál látott „instanceof/cast”nak: ha a noti egy Email, akkor tegyük le egy `e` változóba mint Emailt, annak már van `address` mezeje, írjuk ki, ha SMS, akkor kezeljük azt, ha egyik se, akkor tájékoztassunk valami useless szöveggel.

Ugyanakkor, ennél van egy még eggyel kulturáltabb és standardabb módszer, a `case class`ok használata. A case classról amit mindenképp érdemes tudni:

- Olyan, mint egy `class`, csak a deklarációjában `case class` vezeti be.
- Alapértelmezetten minden konstruktor argumentuma `val`, tehát getter generálódik hozzá.
- Lehet használni mintaillesztésben és a mintában kinyerni a mezőit[8].
- A fordító generál hozzá companion objectet, amiben implementál egy factoryt a companion object apply metódusán keresztül. Praktice ez azt jelenti, hogy létrehozásakor nem kell `new`ot használjunk.
- Az egyenlőséget (equalst) is kigenerálja hozzá a fordító, a megadott mezők mindegyikének egyenlőnek kell lenniük a két objektumban. A hashCode-ot is kigenerálja.
- Generál hozzá toString-et is, ami maga a konstruktorfejléc lesz.

Az IDEben is másmilyen fonttal van szedve a case class, hogy lássa az ember messziről.

Case classal és mintaillesztéssel a notis task:

```
package noti

trait Noti
case class Email( address : String ) extends Noti
case class SMS( number : Int ) extends Noti

object Main extends App {
  def checkNoti( n : Noti ) : Unit = n match {
    case Email( address ) => println( address )
    case SMS( num )      => println( num )
    case _               => println("Unknown Noti type!")
  }
  val v = Email( "sanyi@otthon.com" )
  val w = SMS( 666 )
  checkNoti( v ) // prints sanyi@otthon.com
  checkNoti( w ) // prints 666
}
```

Tehát pl. a `case SMS( num ) => println( num )` az annyit tesz, hogy ha az illesztett `n` kifejezés egy `SMS`, akkor annak az adattagjaira illeszti a belső mintákat. Most pl. az egyetlen `number` adattagra illeszti a `num` változót, ami mindenre illeszkedik, tehát az egész minta illeszkedni fog az `SMS(666)`-ra és a `num` változó felveszi a `666` értéket, amit a case jobb oldalán ki is írhatunk.

Case classok használatával már összetett mintákat is tudunk írni: pl.

- `Email(_)` illeszkedik minden `Email`re, nem bindeli az adattagot változóhoz.
- `SMS(666)` akkor illeszkedik, ha a bejövő `SMS` adattagja `666`.
- Ha van egy `case class Person( name : String, age : Int )`ünk, akkor pl. egy `Person("Sanyi",a)` minta akkor illeszkedik, ha amire illesztjük egy `Person`, aminek `name` mezője Sanyi, `age` mezője pedig illik az `a` változóra (az mindig teljesülni fog), ekkor bindeli az `a` változóba az `age` mező értékét.

[8] Ez a kigenerált `unapply` metódus miatt van, ami a mintaillesztés alatt hívódik btw.

Azt is érdemes tudnunk, hogy egy mintában egy változónevet csak egyszer használhatunk, tehát egyenlőség ellenőrzésére nem tudjuk így használni ezt a mechanizmust.

Amire gyakran használunk case classt: *algebrai adattípusok* definiálására. Kerülve a túlzott formalizmust, ha vannak alap típusaink (`Double`, `String`, `Noti`, minden ami már korábban definiálva volt), és „formális" típusneveink, `X1`,..., `Xn`, úgy, hogy minden `Xi` formálisan az `X`-ek és az alap típusok (akárhány tényezős) szorzatának összegeként vannak felírva, akkor az `Xi`-k algebrai adattípusok.

Ezt leginkább egy példán keresztül lehet megérteni: ha mondjuk egy `String` egy alaptípus, és a formális típusneveink a `Var`, `Form`, `True`, `False` `Or` és `Neg` úgy, hogy

- a `Var` a `String` (egytényezős) szorzata,
- a `Form` a `True`, `False`, `Var`, `Or` és `Neg` összege,
- a `True` és a `False` nullatényezős szorzatok,
- az `Or` a `Form` kéttényezős szorzata,
- a `Neg` pedig a `Form` egytényezős szorzata,

akkor ezt a következőképp értelmezhetjük: mind az öt most deklarált típus egy osztály; a *szorzat* azt jelenti, hogy a megadott típusú adattagokkal rendelkezik az adott osztály (pl. az `Or`-nak van két `Form` típusú adattagja, a `True`-nak nincs adattagja), az összeg pedig azt, hogy a tagok uniója az összegük (mondjuk az összeg tagjai leszármazott osztályai az összegnek, ha ez nem okoz körkörös függést). Tehát ha ezt a fenti adattípus-halmazt implementáljuk Scalában, akkor ezt kapjuk:

```
trait Form
case class Var( name: String ) extends Form
case class Neg( f : Form ) extends Form
case class Or( f : Form, g : Form ) extends Form
case object True extends Form
case object False extends Form
```

(note: az adattag nélküli osztály, ha állapota sincs, akkor egy singletonnal modellezhető, ezért `case object` a két konstans; ha nincs is adattag, akkor két `True` egymástól semmilyen módon nem lesz megkülönböztethető).

És máris van egy formula implementációnk, amikben változókat, negálást, vagyolást és a konstans igazat-hamist használhatjuk. Egy összetettebb formulát létrehozhatunk pl. így

```
val pandnot_qorfalse = And(Var("p"),Not(Or(Var("q"),False)))
```

A funkcionális toolbox leggyakoribb algebrai adattípusa a `List`: egy `List` a `Nil` és a `Cons` összege, a `Nil` az üres szorzat, a `Cons` meg egy alaptípus és a `List` szorzata. Pl. egy `Int`ekből álló lista:

```
trait IntList
case object Nil extends IntList
case class Cons( head: Int, tail: IntList)
```

és pl. az `(1,4,2)` listát létrehozni (most még kényelmetlenül) így lehet:

```
Cons( 1 , Cons( 4, Cons( 2, Nil ) ) )
```

Az üres lista a `Nil` objektum, a nemüres lista egy `Cons` példány, akinek az első eleme a `head` mezője, a többi része (tehát a második elemtől kezdődő rész-listája) a `tail` mezője.