Funkcionalno programiranje

Crte (Traits)

Motivacija

- Višestruko nasle ivanje nije mogu e u jeziku Scala
  - izbegavanje nepogodne dijamantske strukture
- Rešenje u novijim OO-jezicima
  - interfejsi
  - apstraktan ugovor – potpuno razdvajanje od implementacije
- Interfejsi (u strogom smislu) imaju mane
  - nemaju implementaciju (novije verzije Jave dozvoljavaju)
  - nemaju polja
  - implementiraju e klase moraju da pruže implementaciju
  - ako klase nisu srodne, ponavljanje koda

Crte (traits)

- Na in proširivanja funkcionalnosti
  - umesto nasle ivanja/implementacije, **umetanje** (mix in)
  - proširivanje interfejsa (skupa javnih metoda) dodavanjem crta
  - primena u srodnim ili nesrodnim klasama
  - pretpostavka o mogu nostima konkretne klase
  - bolja reupotreba postoje eg koda
  - crte su tako e (apstraktni) tip podataka

```
trait Filozofska {
   def filozofiraj() {
      println("Zauzimam memoriju, dakle postojim!")
   }
}

class Osoba extends Filozofska
val o = new Osoba
val f : Filozofska = o
```

Osobine crta

- Podrazumevana nadklasa crte je `AnyRef`
- Nadklasa može da se eksplicitno navede
  - smisao: ograni iti skup klasa koje mogu da poseduju datu crtu
- Umetnute metode se mogu nadja ati u konkretnoj klasi
  - polimorfno, na isti na in kao i za obi ne klase (`override`)
- Umetanje pomo u
  - `extends`
  - `with`
- Mogu posedovati polja (stanja)

Osobine crta

- Umetanje nije isto što i (višestruko) nasle ivanje
  - crta "ne zna" kako   e biti umetnuta,
    ali ipak poseduje referencu ka `super` tipu
  - linearizacija – redosled umetanja je bitan

- Sintaksno, crte deluju kao klase, ali
  - Ne mogu imati parametre primarnog konstruktora
  - Dinami ko odre ivanje `super` objekta (kod klasa je stati ko)

- Nije mogu e
  - Umetnuti crtu u `final` klasu
  - Umetnuti crtu koja nadja ava `final` metodu

Osobine crta

- Mešanje sa `extends`

```
trait T
class A extends T
```

  - A implicitno nasle uje superklasu crte T
  - U ovom slu aju A nasle uje AnyRef

- Mešanje sa `with`

```
trait T
class X
class Y extends X with T
```

  - Y explicitno nasle uje X, ali ne i superklasu crte T

Primer

```
class Point(val x: Int, val y: Int)
class Rectangle(val topLeft: Point,
                val bottomRight: Point)
{
  def left = topLeft.x
  def right = bottomRight.x
  def width = right - left
  // itd...
}
abstract class Component
{
  def topLeft: Point
  def bottomRight: Point
  def left = topLeft.x
  def right = bottomRight.x
  def width = right - left
  // itd...
}
```

Primer

```
class Point(val x: Int, val y: Int)
class Rectangle(val topLeft: Point,
                val bottomRight: Point) {
   def left = topLeft.x
   def right = bottomRight.x
   def width = right - left
   // itd...
}
abstract class Component
{
   def topLeft: Point
   def bottomRight: Point
   def left = topLeft.x
   def right = bottomRight.x
   def width = right - left
   // itd...
}
```

```
trait Rectangular {
   def topLeft: Point
   def bottomRight: Point
   def left = topLeft.x
   def right = bottomRight.x
   def width = right - left
   // itd...
}

class Rectangle(val topLeft: Point,
                val bottomRight: Point)
      extends Rectangular   { //... }

abstract class Component
         extends Rectangular { //... }
```

Primer – crta Ordered

- esta potreba da se utvrdi redosled ure enih objekata
  - < , <=, >, >=, ...
- Crta Ordered
  - metoda `compare` je apstraktna (treba da vrati <0, 0 ili >0)
  - sve ostale metode (operatori) se definišu preko nje

- Pogodnost za programere
  - implementacija svih operatora je automatski dostupna
  - implementacija ne optere uje konkretnu klasu
- Oprez
  - `equals` se ne realizuje na osnovu `compare`
  - programer mora sam da implementira metodu `equals`
  - razlog: `equals` zahteva i proveru tipa, ne samo vrednosti a zbog brisanja tipa, `Ordered` to ne može da izvede

Primer – crta Ordered

- Bez crte Ordered

```
class Rational(n: Int, d: Int) {
   // ...
   def < (that: Rational) = this.numer * that.denom >
                            that.numer * this.denom
   def > (that: Rational) = that < this
   def <= (that: Rational) = (this < that) || (this == that)
   def >= (that: Rational) = (this > that) || (this == that)
}
```

- Optere uje se interfejs klase Rational
  - postaje tzv. "debeli" interfejs
  - kod je manje pregledan
  - zna ajna koli ina koda se ponavlja kod klasa koje imaju istu funkcionalnost

Primer – crta Ordered

- Uz primenu crte Ordered

```
class Rational(n: Int, d: Int) extends Ordered[Rational] {
   // ...
   def compare(that: Rational) = (this.numer * that.denom) –
                                 (that.numer * this.denom)
}

scala> val half = new Rational(1, 2)
half: Rational = 1/2
scala> val third = new Rational(1, 3)
third: Rational = 1/3

scala> half < third
res5: Boolean = false
scala> half > third
res6: Boolean = true
```

Kompozicija modifikacija klase

- Druga zna ajna osobina crta
  je mogu nost **kompozicije modifikacija** klasa
- Crte dozvoljavaju da se menjaju metode klasa
  - kompozicija: rezultat jedne izmene prenosi se na narednu
- Metode koje u estvuju u lancu kompozicije
  moraju biti ozna ene sa `abstract override`
  - crta mora biti umetnuta u klasu sa konkretnom definicijom metode

Kompozicija modifikacija klase

- Posmatra se red celih brojeva.
- Dodaju se modifikacije
  - dupliraj umetnutu vrednost
  - inkrementiraj umetnutu vrednost
  - filtriraj negativne umetnute vrednosti

```
abstract class IntQueue {
   def get(): Int
   def put(x: Int)
}

import scala.collection.mutable.ArrayBuffer
class BasicIntQueue extends IntQueue {
   private val buf = new ArrayBuffer[Int]
   def get() = buf.remove(0)
   def put(x: Int) { buf += x }
}
```

Kompozicija modifikacija klase

```
trait Doubling extends IntQueue {
   abstract override def put(x: Int) { super.put(2 * x) }
}
```

- `extends`: eksplicira kog tipa mora biti klasa u koju se ume e ova crta
- `super`: može, iako je metoda apstraktna
  - podrazumeva se poziv druge umetnute crte ili klase

```
trait Incrementing extends IntQueue {
   abstract override def put(x: Int) { super.put(x + 1) }
}

trait Filtering extends IntQueue {
   abstract override def put(x: Int) { if (x >= 0) super.put(x) }
}
```

Kompozicija modifikacija klase

```
val q = new BasicIntQueue with Doubling
                          with Incrementing
                          with Filtering

q.put(10); q.put(5); q.put(0); q.put(-1)
for(i <- 1 to 3) println( q.get() )

//> 22
//| 12
//| 2
```

- Redosled umetanja je bitan:
  "primena u obrnutom redosledu navo enja"
- `with Filtering with Doubling with Incrementing`
- `22, 12, 2, 0`

- Nije samo obrnut redosled u pitanju: vrši se linearizacija

Linearizacija

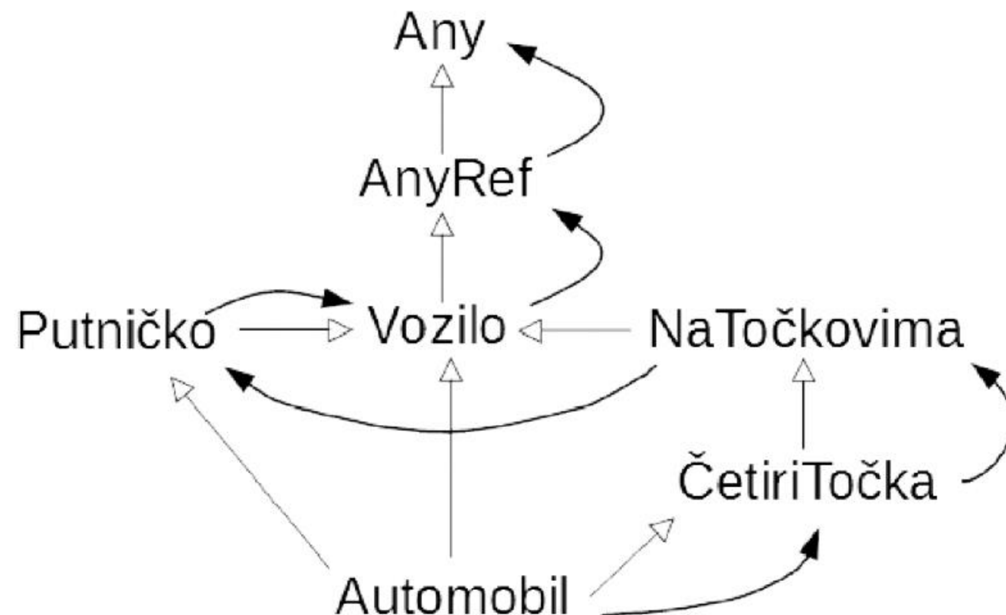
- Zašto višestruko izvo enje nije adekvatno?
  - sve roditeljske klase neke klase su "na istom nivou"
    - poziv `super.metoda` bi izvršio samo jednu (filtriranje, dupl. ili ink.)
  - `super` je unapred odre en
  - OO pristup sa izvo enjem ovde nije intuitivan

- Jezik Scala primenjuje linearizaciju kod umetanja
- Kada se instancira klasa, pravi se linearni poredak svih njenih nasle enih klasa ili crta
  - svaka pojava `super` referiše na prethodni tip iz tog poretka

Linearizacija

- Klasa X se u linearnom poretku nalazi pre njene nadklase ili umetnutih crta

```
class Vozilo
trait Putnicko extends Vozilo
trait NaTockovima extends Vozilo
trait CetiriTocka extends NaTockovima
class Automobil extends Vozilo with Putnicko with CetiriTocka
```



Linearizacija

```
class Vozilo { def m = { println("Vozilo"); }

trait Putnicko extends Vozilo {
  abstract override def m = { println("Putnicko"); super.m }
}

trait NaTockovima extends Vozilo {
  abstract override def m = { println("NaTockovima"); super.m }
}

...

val a = new Automobil
//> a  : w06.traits.Automobil =
w06.traits$$anonfun$main$1$Automobil$1@38082d64

a.m                                               //> Automobil
                                                  //| CetiriTocka
                                                  //| NaTockovima
                                                  //| Putnicko
                                                  //| Vozilo
```

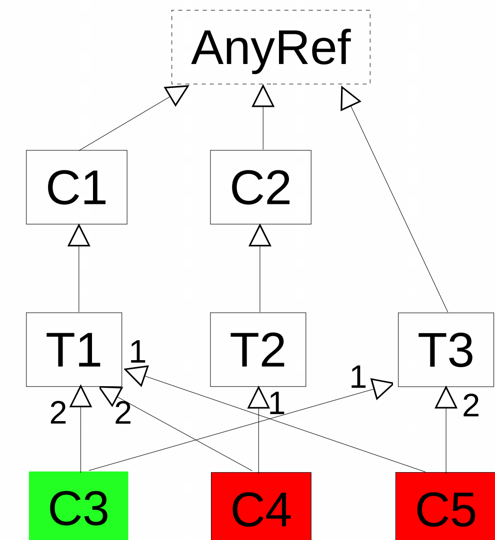
Formiranje hijerarhije

- Crta može da se izvede iz proizvoljnog broja drugih crta

```
trait A
trait B
trait C
...
trait D extends A with B with C ...
```

- Ali ako su crte izvedene iz klasa, klase moraju biti srodne

```
class C1
class C2
trait T1 extends C1
trait T2 extends C2
trait T3
class C3 extends T1 with T3
class C4 extends T1 with T2 // greska
class C5 extends T3 with T1 // greska
```



Formiranje hijerarhije

```
class C extends T1 with T2 with T3 ...
```

- Generalno pravilo: super-klasa od T1 mora biti podklasa super-klasa od T2, T3 ...

Formiranje hijerarhije

- Još jedan primer

```
class D1
class D2 extends D1
class D31 extends D2
class D32 extends D2
trait T31 extends D31
trait T32 extends D32
trait T1 extends D1
trait T2 extends D2
```

✓ `class D4 extends T31 with T2 with T1`

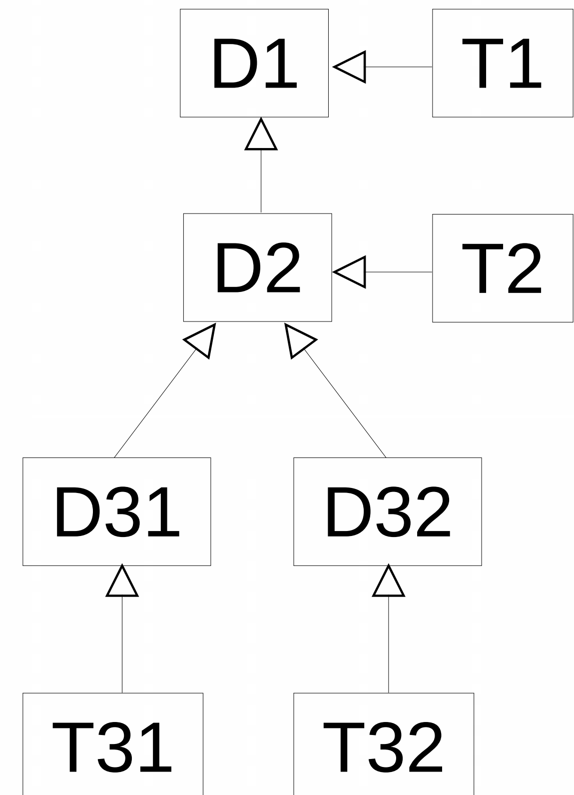
✓ `class D5 extends T31 with T1 with T2`

✗ `class D6 extends T2 with T31 with T1`

✗ `class D7 extends T2 with T1 with T31`

✗ `class D8 extends T1 with T2 with T31`

✗ `class D9 extends T1 with T31 with T2`



Konfliktne situacije

- Greška je ako klasa ume e crte koje deklarišu metodu istog imena i potpisa, a razli itog tipa

```
trait T1 { def m = 1 }
trait T2 { def m = 2.0 }
class C extends T1 with T2
```

- Greška je ako klasa ume e crte koje definišu metodu istog imena, potpisa i tipa

```
trait T1 { def m = 1.0 }
trait T2 { def m = 2.0 }
class C extends T1 with T2
```

- ... ali, tada je mogu e nadja ati

```
class C extends T1 with T2 { def m = 3.0 }
```

Kompozicija bez formiranja hijerarhije

- U konkretan objekat može da bude umetnuta crta
  - Ne mora cela klasa da poseduje datu crtu
- Umetanje se vrši prilikom stvaranja objekta
- Primer:

```
class X
trait Y
val y = new X with Y
y: X with Y = $anon$1@32a72c4

val x = new X
x: X = X@5934ca1e
val y = x with Y
<console>:1: error: ';' expected but 'with' found.
val y = x with Y
          ^
```

Pozivanje konkretne nasle ene metode

- Mogu e je birati koja nasle ena metoda se poziva.

```
object traittest {
  trait Covek { def pozdrav = "Zdravo"  }

  trait Majka extends Covek { override def pozdrav = "[toplo] Zdravo" }

  trait Otac extends Covek {
    override def pozdrav = "[autoritativno] Zdravo"
  }
  class Dete extends Covek with Majka with Otac {
    override def pozdrav = "[veselo] Zdravo"
    def ljudskiPozdrav = super[Covek].pozdrav
    def majcinskiPozdrav = super[Majka].pozdrav
    def ocinskiPozdrav = super[Otac].pozdrav
  }
  println( new Dete ljudskiPozdrav )        //> Zdravo
  println( new Dete pozdrav )               //> [veselo] Zdravo
  println( new Dete majcinskiPozdrav )      //> [toplo] Zdravo
}
```

Pozivanje konkretne nasle ene metode

- Ograni enje: konkretna klasa mora direktno da nasledi (`extends` ili `with`) sve željene tipove
- Slede i primer sadrži grešku, zato što ne nasle uje tip `Covek`

```
class Dete extends Majka with Otac {
   override def pozdrav = "[veselo] Zdravo"
   def ljudskiPozdrav = super[Covek].pozdrav
   def majcinskiPozdrav = super[Majka].pozdrav
   def ocinskiPozdrav = super[Otac].pozdrav
}
```