Funkcionalno programiranje

Klase

Motivacija

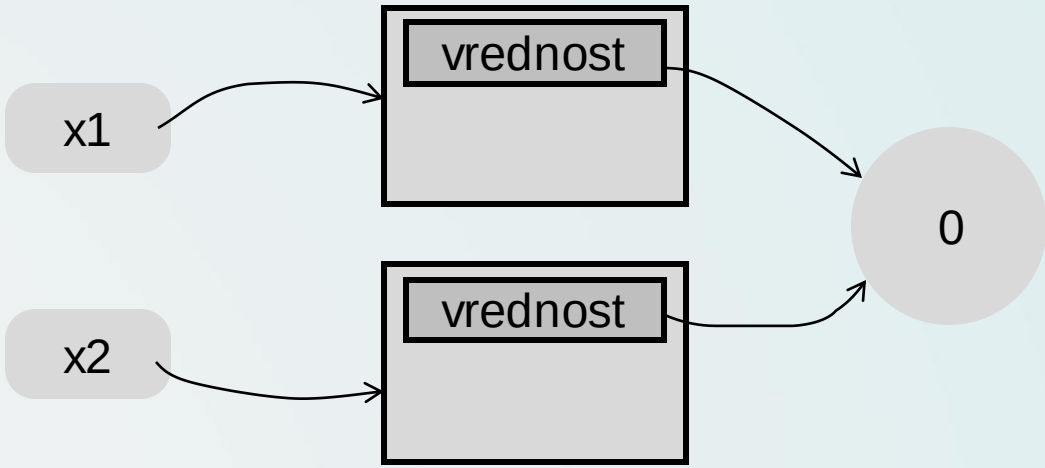
- Scala je OO jezik – potreba za klasama je evidentna
- Uloga klasa se nije promenila
  - enkapsulacija podataka
  - grupisanje srodnih funkcionalnosti
  - polimorfizam
  - ...
- Scala se snažno oslanja na Javu (mada su moguće i drugačije implementacije)
  - sve klase se implicitno izvode iz java.lang.Object
    - preciznije: iz **scala.AnyRef**, koji je alias za **java.lang.Object**
  - Scala može da koristi sve Java klase

Klase

- Osnovni elementi

```
class X {
   var vrednost = 0    // podrazumeva se public
}

val x1 = new X
val x2 = new X
```



Podsetnik: val ne znači da je objekat nepromenljiv,
    već da dato ime ne može da se odnosi na drugi objekat

```
x1.vrednost = 3        // u redu
x1 = new X             // greška
```

Klase

- Primer klase Racionalnih brojeva

```
class Rational(x: Int, y: Int) {
    def numer = x
    def denom = y
}
```

Klasa direktno prima parametre, umesto da se piše poseban konstruktor. Parametri **nisu vidljivi** van tela klase.

- Uveden je novi tip
- Implicitno definisan **primarni konstruktor**
- Objekti se stvaraju kao u Javi:

```
new Rational(1,2)
```

- Članovima se pristupa, kao u Javi, infiksim operatorom .

```
val x = new Rational(1,2) // val x: Rational = new ...
x.numer     // 1
x.denom     // 2
x.numer = 2         // greška
```

Klase

```
class Rational(x: Int, y: Int) {
    println("Racionalni broj: " + x + "/" + y)
}
```

- Sav kod koji nije deo definicije polja ili metode biće ugrađen u **primarni konstruktor**
- Ugrađivanje se vrši u redosledu pojavljivanja u kodu
- Kada je potrebna provera validnosti parametara koristi se metoda `require` (definisana u objektu `Predef`)

```
class Rational(x: Int, y: Int) {
    require(y != 0)
}
```

- Baca izuzetak `IllegalArgumentException` ako uslov nije ispunjen; u suprotnom nema efekta

Metode

- Namena je ista kao u "klasičnim" OO jezicima (C++, Java)
  - definišu ponašanje objekata
  - dozvoljeno preklapanje operatora kao i definisanje novih operatora

```
class Rational(x: Int, y: Int) {
    private def gcd(a: Int, b: Int): Int =
            if(b==0) a else gcd(b, a%b)   //razmotriti neg. vred.
    private val g = gcd(x, y)             // x.abs, y.abs
    def numer = x / g     // ili: val numer: Int = x / g
    def denom = y / g     // ili: val denom: Int = y / g
    override def toString() = numer + "/" + denom
}
```

- **def** ili **val** – isto se koriste, ali šta je bolje?
  - def se izračunava pri svakom pozivu
  - val samo jednom

Metode

- Metode često imaju samo bočni efekat, a to je promena stanja objekta
- Takve metode su po tipu `Unit` (što odgovara Javinom `void`)
- Povratni tip nije neophodno pisati, ali tada telo metode mora biti u vitičastim zagradama, a ne prethodi znak =

```
class X {
   private var vrednost = 0   // private[this] – samo this obj.
   def promeni(v: Int) { vrednost = v }
   // isto kao: def promeni(v: Int): Unit = vrednost = v
}
```

Konstruktori

- Primarni konstruktor se implicitno generiše
  - **jedino on** može da pozove konstruktor roditeljske klase
  - može biti proglašen privatnim: `class Klasa private { ... }`
- Moguće je napisati pomoćne (*auxiliary*) konstruktore
- Zovu se **`this`**

Alternativno:
`y: Int = 1`

```
class Rational(x: Int, y: Int) {
    ...
    def this(x: Int) = this(x, 1)        // poziv primarnom k.
}
```

- Svaki pomoćni konstruktor mora da pozove neki drugi konstruktor kao svoju prvu aktivnost
  - može da pozove primarni
  - može da pozove pomoćni koji je ranije definisan
  - dakle, primarni će uvek biti pozvan, nema mogućnosti ciklusa

Konstruktori

- Telo klase se uslovno može posmatrati kao telo primarnog konsturktora
  - svaki poziv metode će biti izvršen prilikom stvaranja objekta

```
class Student(ime: String, brInd: String) {
println("Stvaram studenta")
...
}

var s = new Student("Pera", "123/456")//> Stvaram studenta
```

Konstruktori

- Parametri konstruktora mogu imati modifikatore `[private] var` ili `val`
  - parametar postaje atribut klase
  - automatski se generišu metode za pristup atributu
    - `var`: generišu se inspektor i mutator
    - `val`: generiše se inspektor

```
s.ime = "Mika"          // greška
s.brInd = "122/345"     // greška

class Student2(var ime: String, val brInd: String) {
println("Stvaram studenta2")  }

val s2 = new Student2("Pera", "123/456")//> Stvaram studenta2

s2.ime = "Mika"
s2.brInd = "122/345"  // greška
println(s2.brInd)
```

Operatori

- Svaka metoda sa jednim parametrom može biti korišćena kao infiksni operator

```
class Rational(x: Int, y: Int) {
   ...
   def add(r: Rational) =
     new Rational(numer*r.denom + r.numer*denom, denom*r.denom )
   }
```

Parametri klase su dostupni samo preko **this**
x + r.x ... // greška

- Dakle, može se napisati: `r add s`
  umesto: `r.add(s)`

- 1+2 je zapravo poziv metode "+" nad `Int` objektom (1), gde je (2) parametar: 1.+(2)

- U tom smislu, ne postoji "klasično" preklapanje operatora

Operatori

- Mogu se definisati novi operatori
- Identifikatori mogu biti:
  - slovo, praćeno sekvencom slova ili cifara
  - simbol operatora, praćen drugim simbolima operatora
  - mešovit identifikator: *alfanum_operator* - primer: `unary_+`
- Prioritet operatora je određen prvim znakom
- Asocijativnost je određena poslednjim znakom

Operatori

rastući prioritet

- op. dodele (i svi drugi koji se završavaju op. =, osim <= >= == !=)
- bilo koje slovo (**ne preporučuje** se upotreba slova $)
- |
- ^
- &
- < >
- = ! // samo = nije validan identifikator, ne može se preklopiti
- : // asocijativnost s desna ulevo ako se ime operatora završava sa :
- + -
- * / %
- svi drugi specijalni znaci (**ne preporučuje** se _ jer ima specijalnu ulogu)

Operatori

- Primeri operatorskih identifikatora
  - `-` `+` `++` `:::` `<?>` `:->`
- Scala kompajler interno prevodi identifikatore operatora u validne Java identifikatore.
  - `:->` bi postao `$colon$minus$greater`
- Obratiti pažnju
  - Scala operatori mogu imati proizvoljnu dužinu
  - U Javi, izraz `x<-y` bi se interpretirao kao `x < (- y)`
  - Scala interpretacija je: `x <- y`
  - Ako se želi Java interpretacija, potreban je razmak `< -`
  - Operandi se **uvek računaju** s leva udesno, **nebitna asocijativnost**
- Primeri:
  - `x *= y+1` tumači se kao `x.*=(y+1)`
  - `a:::b` tumači se kao `b.:::(a)`, prvo se računa a, onda b

Operatori

```
class Rational(x: Int, y: Int) {
    ...
    def + (r: Rational) =
        new Rational(numer * r.denom + r.numer * denom,
                     denom * r.denom )

    def - (r: Rational) =
        new Rational(numer * r.denom - r.numer * denom,
                     denom * r.denom )

    def * (r: Rational) =
        new Rational(numer * r.numer, denom * r.denom )
}
```

```
val x = new Rational(1,2)
val y = new Rational(1,3)
x*x + y*y      // 13/36
```

Operatori

- detaljnije u poglavlju 21 -

- Nije moguće napisati

```
val r = new Rational(2,3)
val r1 = r*2
```

- Oba operanda moraju biti tipa `Rational`
- Rešenje: preklapanje operatora

```
def * (i: Int): Rational =
    new Rational(numer + i * denom, denom)
```

- Sada je moguće napisati `val r1 = r*2`
- Ali još uvek nije moguće napisati `val r1 = 2*r`
- Potrebna konverzija levog operanda
  - **implicitna** konverzija koja izraz `2 * r` prevodi u `new Rational(2) * r`

  moguće **neimplicitne** konverzije se **ne razmatraju**

```
implicit def intToRational(x: Int) = new Rational(x)
```

Operatori

- Tehnički, Scala nema preklapanje operatora
- `1+2` je zapravo poziv metode `+` nad objektom 1: `(1).+(2)`
- Elementima nizova se pristupa oblim zagradama
  - `obj(i)` je zapravo poziv `obj.apply(i)`
  - dakle pristup elementu `i` je zapravo poziv metode sa arg. `i`
- Ovo nije ograničeno samo na nizove – važi za svaki objekat (ako tip ima definisanu metodu `apply`)
- Slično: `obj(i) = neka_vrednost`
- poziva se `obj.update(i, neka_vrednost)`

Unikatni (singleton) objekti

- Scala je u nekim svojim aspektima više objektno-orijentisana od Jave
- Ne postoje statički članovi u jeziku Scala
- Umesto njih, Scala propisuje unikatne objekte
- Definicija unikatnog objekta: umesto **`class`** dolazi **`object`**
- Ne može da se instancira – ne koristi se **`new`**
- Unikatni objekat može biti nazvan isto kao i neka klasa
  - tada je on *prateći objekat* (*companion object*) toj klasi, a ona je *prateća klasa* (*companion class*) tom objektu
  - moraju oba (klasa i objekat) biti definisani u istom fajlu
  - uzajamni odnos prijateljstva – pristup čak i privatnim članovima
- U suprotnom
  - to je samostalni (standalone) objekat

Unikatni (singleton) objekti

- U unikatnim objektima se mogu pisati metode i atributi poput statičkih metoda i atributa u Javi
  - sintaksa pristupa je ista (*naziv_objekta* . *naziv_metode* ili *atributa*)

- Ali unikatni objekat je – objekat (postoji u memoriji)
  - može biti izveden iz druge klase
    i mogu mu se pridružiti osobine - crte (traits)
  - može mu se obraćati kao i drugim objektima
  - može se prosleđivati tamo gde se očekuju objekti
- Međutim
  - ne definiše tip podatka – to radi prateća klasa (ako postoji)
  - ne instancira se (ne dobija parametre prilikom stvaranja)
  - inicijalizuje se pri prvoj upotrebi

Pravljenje Scala aplikacije

- Scala program mora da poseduje najmanje jedan samostalni unikatni objekat sa `main` metodom
- Alternativno: `extends App`
- `main` metoda ima jedan parametar, tipa `Array[String]`
- Program se pokreće imenovanjem samostalnog objekta
- Scala implicitno uvozi:
  - paket `java.lang`
  - paket `scala`
  - unikatni objekat `scala.Predef`
    - `println` je zapravo `Predef.println`

Pravljenje Scala aplikacije

```
// strQuote.scala
package w02
object strQuote {
  def quote(s: String) = ">>"+s
}
// testApp.scala
package w02
import strQuote.quote
object testApp {
  def main(args: Array[String]) {
    for( arg <- args )   println(quote(arg))
  }
}
```

```
C:\> scalac strQuote.scala testApp.scala
C:\> scala testApp xyz
>>xyz
```

```
scala w02.testApp xyz
```

Apstraktne klase i nasleđivanje

- Apstraktne klase: ne mogu da se instanciraju

```
abstract class IntSet {
    def incl(x: Int): IntSet
    def contains(x: Int): Boolean
}
```

Ovde se daje primer implementacije skupa preko binarnog stabla

- Nasleđivanje

```
class Empty extends IntSet {
    def contains(x : Int): Boolean = false
    def incl(x : Int): IntSet =
        new NonEmpty(x, new Empty, new Empty)
}
```

- Primetiti: za apstraktne metode **ne mora** da se piše **override** u izvedenim klasama

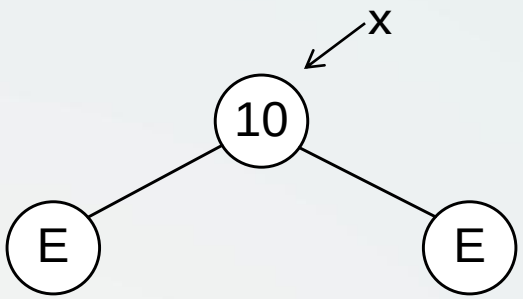
Apstraktne klase i nasleđivanje

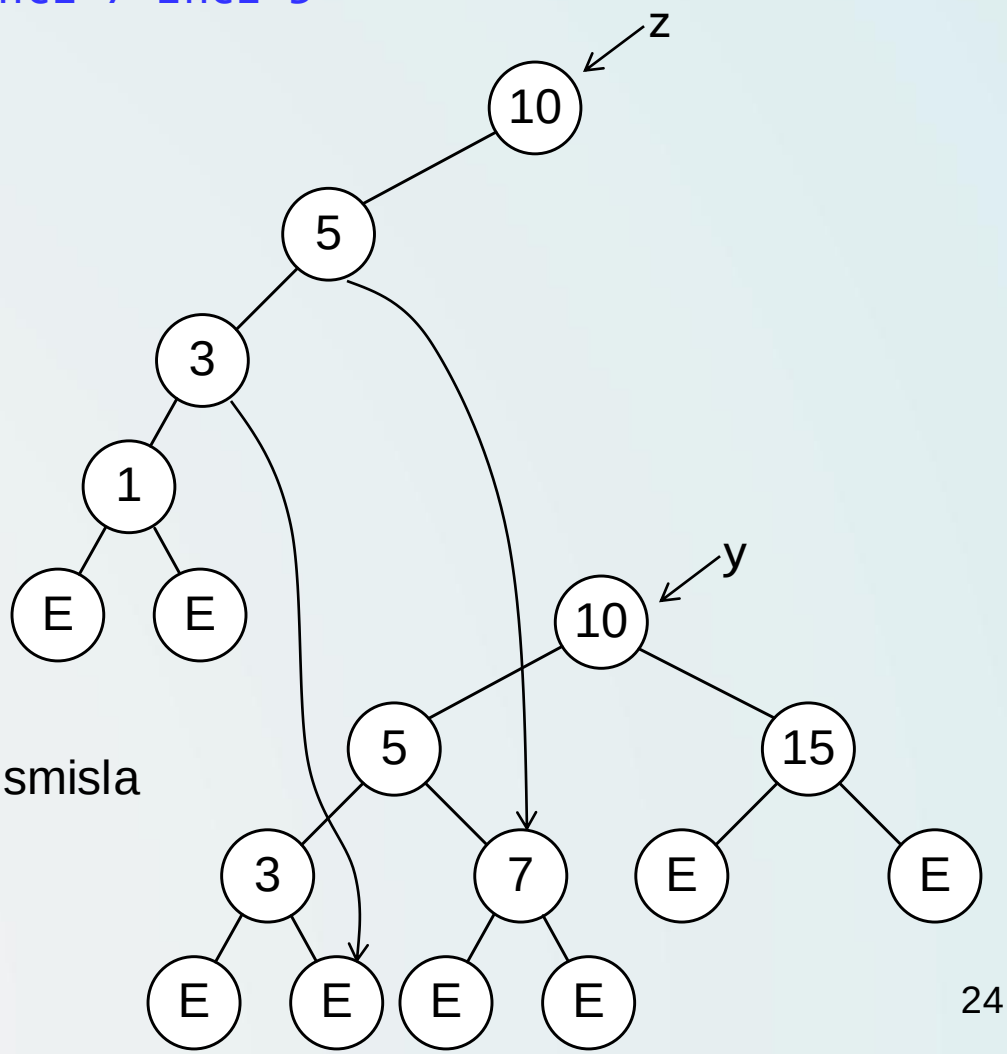
```
class NonEmpty(elem: Int, left: IntSet, right: IntSet)
                                          extends IntSet {
    def contains(x : Int): Boolean = {
      if (x < elem) left contains x
      else if (x > elem) right contains x
      else true
    }

    def incl(x : Int): IntSet = {
      if (x<elem) new NonEmpty(elem, left incl x, right)
      else if (x>elem) new NonEmpty(elem, left, right incl x)
      else this
    }
}
```

Apstraktne klase i nasleđivanje

```
val x = new NonEmpty(10, new Empty, new Empty)
val y = x incl 5 incl 15 incl 7 incl 3
val z = y incl 1
```





Primetiti: ovakva konstrukcija ima smisla zato što se pretpostavljaju nepromenljivi objekti

Apstraktne klase i nasleđivanje

```
abstract class Entity {
  def name: String
  override def toString = name
}
```

metoda **bez parametara** (*parameterless*)

razlikuje se od metoda
sa **praznim zagradama** (*empty-paren*)

```
class Constant extends Entity { override val name = "Constant" }
```

nadjačava metodu iz osnovne klase

```
class Robot(id: Int) extends Entity { def name = "T-" + id }
```

```
class Person(override val name: String) extends Entity
```

parametarsko polje: kad ima modifikator var ili val, postaje atribut klase

```
val e = new Constant
println(e.name)                    // Constant
val e2 = new Person("Pera")
println(e2.name)                   // Pera
val e3 = new Robot(800)
println(e3.name)                   // T-800
```

Apstraktne klase i nasleđivanje

- Preporučuje se upotreba metoda bez parametara
  - kada nema parametara i
  - kada je metoda **inspektor** (samo čita stanje)
- Posledica
  - klijentski kod ne zna kako je realizovan atribut (vrednost ili metoda)
  - moguće je jednostavno zameniti implementaciju po potrebi
  - to je **princip uniformnog pristupa** (*uniform access principle*)
  - vrednost zauzima više memorije, ali se brže (?) izvršava
- Scala pravi razliku između metoda bez parametara i metoda sa praznim zagradama
  - ali dozvoljava njihovo mešanje (jedna može da nadjača drugu, itd)
  - nije neophodno pisanje zagrada pri pozivu
  - ipak se preporučuje pisanje () kada poziv nije dohvatanje vrednosti

Apstraktne klase i nasleđivanje

- Parametarska polja

```
class Person(val name: String) extends Entity
```

  - istovremeno postaje parametar i (nepromenljiv) atribut klase
  - ekvivalentno:

```
class Person(n: String) extends Entity { val name = n }
```

  - umesto **val** može i **var**
  - Mogući modifikatori su
  - private ili private[this]
  - protected
  - override

```
class Cat {
val dangerous = false
}

class Tiger(
override val dangerous: Boolean,
private var age: Int
) extends Cat
```

Apstraktne klase i nasleđivanje

- Pozivanje konstruktora roditeljske klase

```
class Cat(val name: String) { val dangerous = false  }

class Tiger(   name: String,
               override val dangerous: Boolean,
               private var age: Int
           ) extends Cat(name)
```

- Može se pozvati bilo koji konstruktor roditeljske klase

Apstraktne klase i nasleđivanje

- Sprečavanje izvođenja odnosno nadjačavanja
  - modifikator **`final`**, kao u Javi
  - ispred **`class`** za sprečavanje izvođenja
  - ispred člana čije nadjačavanje treba sprečiti (atribut ili metoda)

```
final class Cat(val name: String) {  val dangerous = false  }
```

ili

```
class Cat(val name: String) { final val dangerous = false  }
```

- Modifikator **`sealed`**
  - dati tip mogu naslediti samo drugi tipovi definisani u **istom** fajlu

```
sealed abstract class Expr
class Var(name: String) extends Expr
```

Primer: singleton

- Napraviti klasu prema projektnom uzorku unikat (singleton)
- Analiza:
  - obezbediti da klasa ne bude apstraktna
  - ne može direktno da se instancira objekat date klase
  - metoda koja vrši dohvatanje jedinstvene instance
- Rešenje:
  - učiniti primarni konstruktor privatnim
  - napraviti prateći objekat sa metodom getInstance

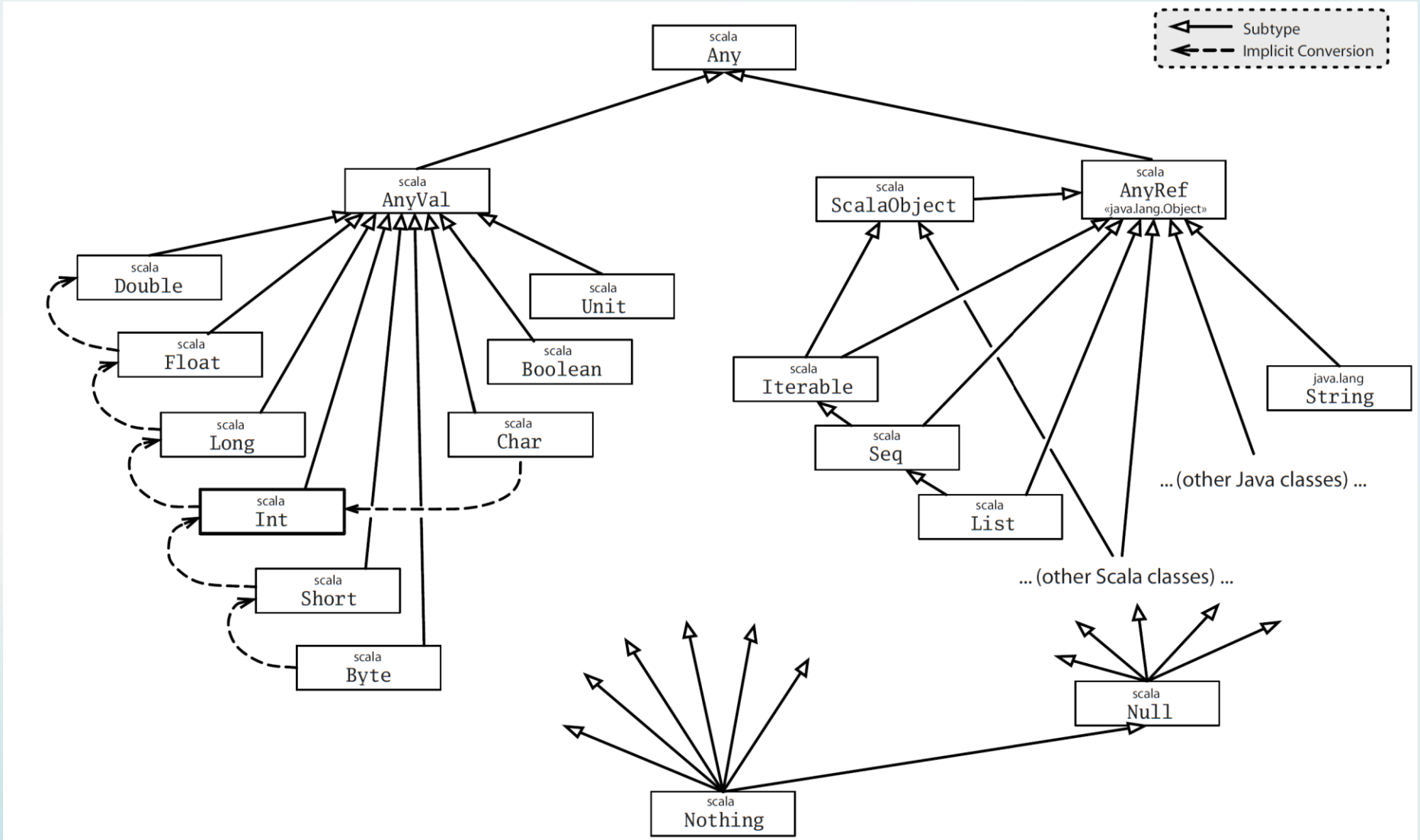
Primer: singleton

```
class NarodnaBiblioteka private {
  override def toString : String = "Narodna biblioteka"
}

object NarodnaBiblioteka {
  val instance = new NarodnaBiblioteka
  def getInstance = instance
}

val tt = NarodnaBiblioteka.getInstance
```

Hijerarhija klasa



Hijerarhija klasa

- U korenu hijerarhije je klasa Any

```
final def ==(that: Any): Boolean
final def !=(that: Any): Boolean
def equals(that: Any): Boolean
def ##: Int
def hashCode: Int
def toString: String
```

Hijerarhija klasa

- Klasa AnyVal
  - Roditeljska klasa za svaku ugrađenu *vrednosnu klasu*
  - a) Byte, Short, Char, Int, Long, Float, Double, Boolean (1:1 Java)
    - radi se automatski box-ing u odgovarajući tip – pr. `java.lang.Integer`
  - b) Unit (~ `void` u Javi)
    - rezultat dodele vrednosti, kao i nekih kontrolnih struktura (while, ...)
- Ne mogu se praviti instance (a) pomoću `new` *`tip`*
  - zato što su sve klase deklarisane kao `abstract final`
- Postoji jedna instanca (b), označava se `()`

```
scala> def greet() { println("hi") }
greet: ()Unit
scala> greet() == ()
hi
res0: Boolean = true
```

```
a = if (true) 1 else false
```

Kog tipa je a?

Hijerarhija klasa

- Klasa AnyRef
  - Roditeljska klasa za sve referentne tipove
  - Java → `java.lang.Object`
  - .NET → `system.Object`

Hijerarhija klasa

- Klasa Null
  - tip `null` reference
  - potklasa svih referentnih tipova (svih koje nasleđuju AnyRef)
  - nije kompatibilna sa vrednosnim tipovima (AnyVal)
- Klasa Nothing
  - na dnu hijerarhije Scala klasa
  - ne postoje objekti ovog tipa
  - specifični slučajevi upotrebe (izuzetak, tip elementa prazne kolek.)
- `scala.Predef`

```
def error(message: String): Nothing =
    throw new RuntimeException(message)

def divide(x: Int, y: Int): Int =
   if (y != 0) x / y
   else error("Deljenje nulom")
```

Uvoz tipova

- Automatski:
  - sve iz paketa `scala`
  - sve iz paketa `java.lang`
  - svi članovi objekta `scala.Predef`

- Sintaksa:
  - `import paket.tip`
  - `import paket.{tip1, tip2, ...}`
  - `import paket._`

Jednakost objekata

- detaljnije u poglavlju 30 -

- Razlikuje se u odnosu na pristup jezika Java
  - Smisao operatora `==` je "prirodna" jednakost
    - za vrednosne tipove, to je jednakost vrednosti
    - za referentne tipove, to je jednakost sadržaja (`equals` u Javi)
  - Za referentne tipove postoji metoda `eq`, koja odgovara `==` u Javi
    - ne koristi se mnogo, jer služi samo kada je od interesa identitet objekta
- Veza između operatora `==` i `equals` u jeziku Scala
  - nije moguće nadjačati `==` (finalna metoda u klasi Any)
  - op `==` poziva metodu `equals`
  - metoda `equals` može se nadjačati
  - za ref. tip. podrazumevano `equals`: poredi reference (kao u Javi)
  - dakle, podrazumevano `==` i `equals` ponašaju se kao `eq`

Ali podrazumevano poredi reference.

```
final def == (that: Any): Boolean =    // principijelno
  if (null eq this) {null eq that}
  else {this equals that}
```

Jednakost objekata

primer razlike Java ↔ Scala

- Java

```
boolean isEqual(Integer x, Integer y) {
   return x == y;
}
System.out.println(isEqual(421, 421));      // false
```

- Scala – uzima u obzir tip stvarnih argumenata

```
scala> def isEqual(x: Int, y: Int) = x == y
isEqual: (Int,Int)Boolean
scala> isEqual(421, 421)
res10: Boolean = true
scala> def isEqual(x: Any, y: Any) = x == y
isEqual: (Any,Any)Boolean
scala> isEqual(421, 421)
res11: Boolean = true
```

Jednakost objekata

- detaljnije u poglavlju 30 -

- U klasi Any definisana je metoda `hashCode`
- Ugovor ove metode zahteva da
  - ako `equals` nad dva objekta vraća `true`
  - onda i `hashCode` mora vratiti isti celobrojni rezultat
- Dakle, `equals` i `hashCode` se zajedno nadjačavaju
- Ova stavka ugovora je bitna zbog kolekcija čiji rad se zasniva na toj pretpostavci
- Voditi računa da se `equals` i `hashCode` ne određuju na osnovu promenljivih (atributa)
  - ako je potrebno poređenje objekata klasa koje imaju promenljive atribute, **bolje koristiti neko drugo ime**

Parametrizacija tipom

- Namena: poput generika u Javi (šablona C++)
  - reupotreba koda kada je algoritam isti
  - drugi vid polimorfizma

```
trait Crta[T]
class Klasa1[T]
class Klasa2[T](par1 : T, par2 : Klasa1[T]) extends Crta[T]
```

- Mogu se i metode parametrizovati:

```
def napravi[T](elem: T) = new Klasa3[T](elem)
napravi[Int](1)              // napravi(1)
napravi[Boolean](true)       // napravi(true)
```

Parametrizacija tipom

- Vrši se brisanje (type erasure), kao u Javi

```
val seq : Seq[String] = Seq(1,2,3) // ne prevodi se
```

```
val seq : Seq[Int] = Seq(1,2,3)
seq.isInstanceOf[Seq[Int]] // true
seq.isInstanceOf[Int] // false
seq.isInstanceOf[Seq[String]] // true
```

Parametrizacija tipom

- Moguće je navesti gornje i donje granice za parametarske tipove
- gornja granica <:
- donja granica >:

- Primer:
  - `def assertAllPositive[S <: IntSet] (r: S) : S = ...`
    S može biti svaki tip koji odgovara tipu IntSet
  - `[S >: NonEmpty]`
    S može biti NonEmpty, IntSet, AnyRef i Any
  - `[S >: NonEmpty <: IntSet]`
    bilo koji tip u intervalu `NonEmpty` i `IntSet`

Utvrđivanje tipa i konverzija

- class Any
  - `def isInstaceOf[T]: Boolean`
  - `def asInstanceOf[T]: T` // ClassCastException

```
if (x.isInstanceOf[String]) {
  val s = x.asInstanceOf[String]
  s.length
} else ...
```

- Ipak, eksplicitno utvrđivanje tipa treba izbegavati

Konverzija tipa u cilju čitanja vrednosti

- Ponekad je dovoljno dohvatiti vrednost nekog podatka bez obzira na njegov tip
- Konverzija u cilju čitanja (*view bound*): specificira da neki tip može biti "posmatran" kao neki drugi
- Koristi se oznaka <%
- Potrebna implicitna (automatska) konverzija između tipova
- Implicitne funkcije to omogućavaju
  - one dozvoljavaju svoju primenu (apply) kada to može dovesti do zadovoljenja tipova u izrazu

Konverzija tipa u cilju čitanja vrednosti

```
scala> implicit def strToInt(x: String) = x.toInt
strToInt: (x: String)Int

scala> "123"
res0: java.lang.String = 123

scala> val y: Int = "123"
y: Int = 123

scala> math.max("123", 111)
res1: Int = 123
```

Konverzija tipa u cilju čitanja vrednosti

```
scala> class Container[A <% Int] { def addIt(x: A) = 123 + x }
defined class Container
```

Ovo znači da A mora biti moguće "posmatrati" kao Int

```
scala> (new Container[String]).addIt("123")
res11: Int = 246

scala> (new Container[Int]).addIt(123)
res12: Int = 246

scala> (new Container[Float]).addIt(123.2F)
<console>:8: error: could not find implicit value for evidence
   parameter of type
   (Float) => Int (new Container[Float]).addIt(123.2)
```

Crte (traits)

- Nasleđuje se samo jedna klasa
  - kao u Javi
  - izbegavanje "dijamantske strukture"
- trait – liči na interfejs u Javi ...
  - klasa može da nasledi više crta
- ... ali trait
  - može da sadrži polja
  - može da sadrži implementacije metoda

```
trait Planar {
  def height : Int
  def width : Int
  def surface = height*width
}
```

```
class Square extends Shape
                with Planar
                with Movable
                ...
```

Funkcijske vrednosti su objekti

- Tip funkcije `A => B` je zapravo skraćeni način pisanja `scala.Function1[A, B]`

```
trait Function1[T1, R] {
  def apply(v1 : T1) : R
  def compose[A](g: A => T1): A => R
  def andThen[A](g: R => A): T1 => A
}
```

- Dakle: funkcije su objekti sa `apply` metodom
- Postoje crte `Function2`, `Function3`, ...
- Poziv funkcije je jednostavno poziv metode `apply`
- `f(a, b)` je zapravo `f.apply(a,b)`

Funkcijske vrednosti su objekti

```
(x: Int) => x * x
```

```
{
      class some_generic_name
            extends Function1[Int, Int]
      {
          def apply(x: Int) = x * x
      }
      new some_generic_name
}
```

- Sintaksa anonimne klase:

```
new Function1[Int, Int]
   { def apply(x:Int) = x * x }
```

Definicija funkcije nije funkcijska vrednost

- Definicija metode: `def f(x: Int) : Boolean = ...` nije funkcijska vrednost
- Ali, ako se f koristi tamo gde se očekuje tip `Function`, automatski se konvertuje u funkcijsku vrednost
  `(x: Int) => f(x)`
- Odnosno

```
new Function1[Int, Boolean] {
   def apply(x: Int) = f(x)
}
```

Primer: ulančane liste

- S obzirom na favorizovanje rekurzije u funkcionalnim jezicima, lista se može predstaviti iz dva gradivna bloka:
  - Nil – prazna lista
  - Kons – ćelija koja sadrži element i referencu ka ostatku liste
- Termin "cons" vodi poreklo iz jezika Lisp
- Funkcionalni jezici favorizuju nepromenljive liste

```
val testList = new Elem(1, new Elem(2, new Elem(3, new Nil)))
```

Primer: ulančane liste

```
object list {
trait List[T] {
  def prazna: Boolean
  def glava: T
  def rep: List[T]
}
class Nil[T] extends List[T] {
  def prazna = true
  def glava = throw new NoSuchElementException("Nil.glava")
  def rep = throw new NoSuchElementException("Nil.rep")
}
class Elem[T](val glava : T, val rep : List[T]) extends List[T] {
  def prazna = false
}
}
```