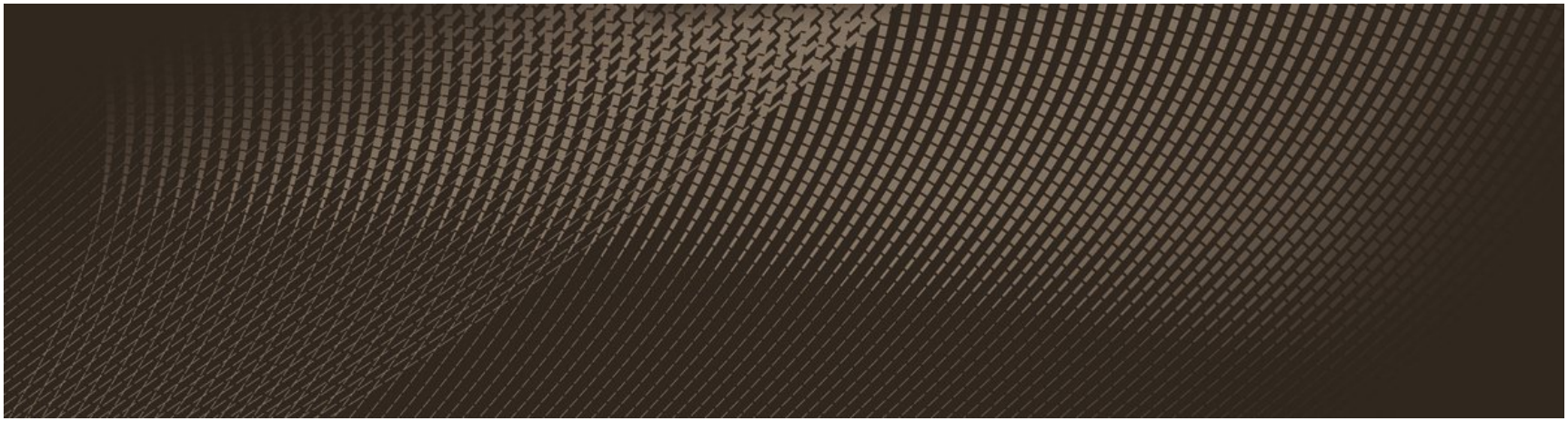


Introduksjon til scala – Scala

Introduksjon til scala
3. april 2011

BEKK

Velkommen

- Hvorfor scala?
- Kurs
  - Litt foiler
  - Litt oppgaver
  - Repeat!
- Middag

Hvorfor scala?

Hvorfor scala?

Java er gammelt
Scala er et riktig steg videre
Bedre konsepter ≡ bedre kode

Gammelt

```
public class Person{
    private String navn;
    private String etternavn;
    private Integer alder;

    public Person(String fornavn, String etternavn, Integer alder){
        this.navn = fornavn;
        this.etternavn = etternavn;
        this.alder = alder;
    }

    public String fulltNavn(){
        return navn + " " + etternavn;
    }

    public String getNavn(){ return navn; }
    public void setNavn(String navn){ this.navn = navn; }

    public String getEtternavn(){ return etternavn; }
    public void setEtternavn(String etternavn){ this.etternavn = etternavn; }

    public Integer getAlder(){ return alder; }
    public void setAlder(Integer alder){ this.alder = alder; }
}
```

Nytt

```
class Person(
  val navn:String,
  val etternavn:String,
  val alder:Int)
{
  def fulltNavn ={
    "%s %s".format(navn, etternavn)
  }
}
```

Closure

Gammelt

```
public List<String> interesse(String starterMed){
  List<String> interesser = new ArrayList<String>();
  interesser.add("Gym");
  interesser.add("Sykkel");
  interesser.add("Scala");
  interesser.add("Vask");
  interesser.add("Mat");

  List<String> valgte = new ArrayList<String>();
  for(String interesse : interesser){
    if(interesse.startsWith(starterMed)){
      valgte.add(interesse);
    }
  }

  return valgte;
}
```

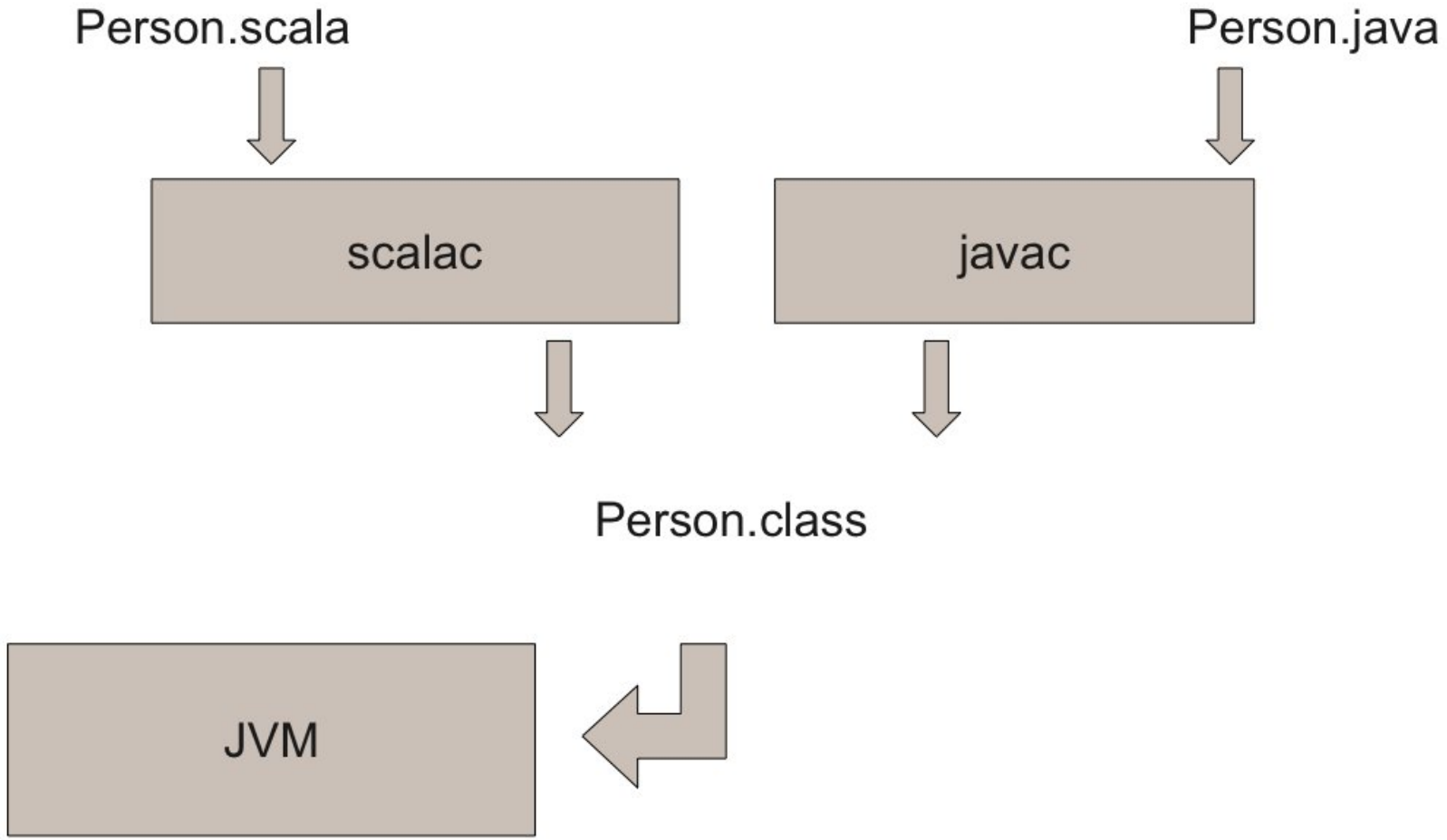
Nytt

```
val interesser = List(
   "Gym", "Sykkel", "Scala",
   "Vask", "Mat"
)

def interesse(starterMed:String) = interesser.filter(
  (s:String) => s.startsWith(starterMed)
)
```

Scala på JVM

Scala er egentlig bare java, pluss en jar.



Helt grunnleggende – Klasser

```
class Person( //Konstruktørparametere og instansevariabler
   navn:String,
   etternavn:String,
   alder:Int)
{
  //Konstruktør
  val interesser = List("ting", "og", "tang")
  val cv = "%s %s -- %s\n Liker: %s".format(
    navn, etternavn, alder, interesser.mkString(", ")
  )


  def fulltNavn ={ //Funksjon
    "%s %s".format(navn, etternavn)
  }
}
```

Helt grunnleggende – val og var

```
 //Er slik for altid
  val interesser = List("ting", "og", "tang")

  def mineInteresser ={
    var interesser = "" //Kan tilordnes på nytt
    for(interesse <- interesser){
      interesser += interesse + ", "
    }

    interesser
  }
}
```

Helt grunnleggende – Oppgaver

Koden ligger her : https://github.com/aslakjo/scala_intro_kurs

- Nett bekkguest / guest7890
- Last ned
- Gå til katalogen og kjør
  - $> sbt
  - >idea
  - Endre host + lagnavn(uten mellomrom!) i filen project/build/ScalaIntroKurs.scala
  - >reload
  - > update
  - > ~test

Oppgavene

```
describe("Variabler"){
  ignore ("definere en string variabel"){
    // -- fyll inn
    // enStreng.isInstanceOf[String] should be (true)
  }

  ignore ("definer en variabel som kan endre seg"){
    var sum = 0
    // -- fyll inn
    sum should be (10)
  }
}
```

– Les eksempel koden
– Endre testen fra ignore til it
–> Få testen til bli grønn

$$3 + 5$$

(3).+(5)

Operator overload

```
//Scala listeoperator ::
val tog = new Vogn :: new Vogn :: new Vogn :: Nil

class Vogn {
  def +(vogn: Vogn) = this :: vogn :: Nil
}

// Vår egen syntaks
val tog = new Vogn + new Vogn
```

Operator overload

```
//Scala listeoperator ::
val tog = new Vogn :: new Vogn :: new Vogn :: Nil

class Vogn {
  def +(vogn: Vogn) = this :: vogn :: Nil
  def ::(vogn: Vogn) = this :: vogn :: Nil
}

// Vår egen syntaks
val tog = new Vogn :: new Vogn :: new Vogn
```

start.oppgaver.OperatorOverload

Typer og scala

```
val liste1 = List(1,2,3)
def liste2 = Liste(4,5) //Finner ut hva typen er

val nyListe = "Streng" :: liste1 //Kopileringsfeil
```

Typer og scala

```
val liste1 = List(1,2,3)
//Kreve en spesiell type
def liste2:List[Int] = Liste(4,5)

val nyListe = List("Streng", 1,2,4)
//Ny list får nå [Any] som type
```

Kontroll strukturer

```
val en = for(i <- 1 to 3) //for
  yield i
en should be( 1 :: 2:: 3:: Nil)

val to = if(false) //If
  1
else
  2
to should be (2)

var i = 0 //While
val tre = while(i < 3){
  i = i +1
  i
}
tre.isInstanceOf[Unit] should be(true)
```

Kontroll strukturer – for

```
val en = for (i <- 1 to 3 if i % 2 == 0)
  yield i

en should be (List(2))
```

Funksjoner

```
(x: Int) => x + 1 // annonym

def annonym(x: Int): Int => { //named
  x+1 // med en kodeblokk
}

//Typisk bruk
val liste = List(1,2,3)
liste.filter((i : Int) => i % 2 == 0) should be (List(2))
```

Funksjoner – førsteklasses

```
// funksjon som variabel verdi
val funksjon = (x: Int) => x + 1
//val funksjon: Int => Int = (x:Int): Int => x+ 1

// -- som retur verdi
def giFunksjon = funksjon

funksjon(2) should be(3)
giFunksjon(3) should be(4)

// -- som parameter
def kjør(funksjon : Int => Int, i : Int) = funksjon(i)

kjør(funksjon, 5) should be(6)
kjør(giFunksjon, 10) should be (11)
```

Programeringsstiler

Imperativ

- Kan gjenbruke variable
- Forteller hvordan vi gjør noe
- Holder på tilstand

–> Vi kan fortsette med OO

```
for(i <- liste){
  sum = sum + i
}
```

Funksjonell

- Ikke kan gjenbruke variable
- Forteller hva vi skal gjøre
- Tilstand er forbudt

–> Det blir enklere å bruke flere cpu-kjerner

```
liste.foreach(i => {
  sum = sum + i
})
```

Programeringsstiler

Imperativ

```
val liste =
  List("en", "to", "tre")

var ropeList = List[String]()

for(i <- liste){
  ropeList =
    "%s!".format(i) :: ropeList
}
```

Funksjonell

```
val liste =
  List("en", "to", "tre")


def prat2Rop(tekst: String) =
  "%s!".format(tekst)


val ropeliste =
  liste.map(prat2Rop(_))
```

start.oppgaver.ProgrammeringsStil

Collections

```
val liste = List(1,2,3,4,5,6)

//foreach
liste.foreach( i => println(i))
// fra -> til
val somTekst = liste.map(i => i.toString)
// finn de vi ønsker oss
val store = liste.filter(i => i > 3)
// crazy shit!
val sum = liste.foldRight(0)((sum, neste) => sum + neste)
```

Collections

```
val liste = List(1,2)
val listeSlutt = List(3,4)

//val heleListen = liste ++ listeSlutt
// Default er alle collections immutable
// Det finnes mutable collections se
    "scala.collection.mutable"

val heleListen = liste ++ listeSlutt
heleListen should be(List(1,2,3,4))

liste should be(List(1,2))
listeSlutt should be(List(3,4))
```

advanced.Oppgaver – oppgave 2

Traits

```
object Logger{
  def error(msg:String) = println ("ÆÆÆ! : " + msg)
}

class ServiceA{
  def error(msg: String) = Logger.error(msg)
}
class ServiceB{
  def error(msg: String) = Logger.error(msg)
}
class DaoA{
  def error(msg: String) = Logger.error(msg)
}
```

Traits

```
object Logger{
  def error(msg:String) = println ("ÆÆÆ! : " + msg)
}
class Logging{
  def error(msg: String) = Logger.error(msg)
}

class ServiceA extends Logging{}
class ServiceB extends Logging{}
class DaoA extends Logging{}
```

Traits

```
object Logger{
  def error(msg:String) = println ("ÆÆÆ! : " + msg)
}
class Logging{
  def error(msg: String) = Logger.error(msg)
}

class ServiceA extends Logging{}
class ServiceB extends Logging{}
//Hva hvis denne skal arve av noe annet
class DaoA extends Logging{}
```

Traits

```
object Logger{
  def error(msg:String) = println ("ÆÆÆ! : " + msg)
}

trait Loggable{
  def error(msg: String) = Logger.error(msg)
}

class ServiceA extends Loggable{ }
class ServiceB extends Loggable{ }
class DaoA extends SuperDuperDao with Loggable{ }
```

Traits

```
object Logger{
  def error(msg:String) = println ("ÆÆÆ! : " + msg)
}

trait Loggable{
  def error(msg: String) = Logger.error(msg)
}

class ServiceA extends Loggable{ }
class ServiceB extends Loggable{ }
class DaoA extends SuperDuperDao with Loggable{ }

//Kan også mixes inn ved new
val daoB = new DaoB with Loggable
```

start.oppgaver.Traits

Option

```
def findPåLager : Plassering = {...}

findPåLager // retur: null
```

Option

```
def findPåLager: Option[Plassering] = {...}


findPåLager // retur: None
findPåLager // retur: Some(Plassering(10,20))
```

Option

```
class Option [+A]
object None extends Option[Nothing]
case class Some [+A] (x: A) extends Option[A]

val option1 : Option[String] = Some("string")
val option2 : Option[String] = None
val option3 : Option[String] = Some(1) // compile error
```

Option

```
val plassering = findPåLager

plassering match {
  case None => //har ikke
  case Some(plassering) => // hent varen
}

val varePlasering = plassering getOrElse("(tomt)")
```

start.oppgaver.OptionOppgaver

Utvidbarhet

```
val list = List(1,2,3)
list(0) == 1 //samme som list.apply(0)

 val to = 1 + 1 // sammen som 1.+(1)
```

Reglene er for alle.
Fleksibiliteten er stor.
–> Gjør det mulig å utvidescala som et scala bibliotek.

feks. Actors og continuations.

start.oppgaver.Utvidbarhet

Mer avanserte oppgaver

Case classes

En case classe er en normal klasse med masse ekstrautstyr

- Genererer geters og seters
- Hash code og equals er implementert korrekt, og på bakgrunn av verdiene
- Det companion object er laget
- Unapply / extractors er laget

Bruk av case classes

```
case class Sykkel(val farge:String, val hjul:Int) //def

object CaseClassApp extends Application{
  //compainon og new
  assert(Sykkel("rød", 2).equals(new Sykkel("rød", 2)))
}

//unapply kommer ..
```

Patternmatching

En veldig veldig kraftig variant av switch statementet

- Gir muligheten til å velge hele eller deler av et object
- Mulig å matche på typer, verdier, arv, innhold i referanser

Bruk av pattern matching

```
case class Farge(val navn:String)

case class Bil(val farge:Farge, val hjul:Int)

object PatternMatcingApp extends Application{
  def godkjenntBil(dings: AnyRef)={
    dings match {
      //Constuctor pattarn, variabel pattern, med guard og extractor
      case Bil(_, hjul) if hjul <= 2 => println("En slik bil heter gjerne motorsykkel")
      //verdi pattern
      case Bil(_, 4) => println("helt vanelig bil")
      //sjekking i refererte objecter, variable pattern og guard
      case Bil(Farge(fargePaBil), _) if fargePaBil.equals("rød") => println("Jippi en rød bil!")
      //@ binder variabelen s til det som er på høyre side
      case s@Bil(farve, _) if farve != null => println("dette er en " + farve.navn + " bil")
      // type pattern
      case s:Bil => println("dette er en bil")
      //wildcard pattern, matcher alt
      case _ => println("dette kan være hva som helst uten om en bil")
    }
  }
}
```

Traits

Et trait spesifiser egenskap.

- Kan brukes om interface(abstract traits)
- En klasse kan få flere traits "mixet inn"
- Traits kan ikke opprettes på egenhånd

Bruk av traits

```
trait HealthCheckable{ //interface
  def isOk: Boolean
}
trait Logger { //egenskap
  def log(message: String):Unit = println(message)
}
trait LoggProcessing extends FooService{ //stackable trait
  def log(message:String):Unit

  override def process:Unit={ //ny oppførsel
    log("Starting processing")
    super.process
    log("Stopped processing")
  }
}

class FooService extends HealthCheckable with Logger{
  def isOk:Boolean = true

  def process = {
    //go allot!
  }
}


object Application{ new FooService with LoggProcessing } //mix inn ved opprettelse
```

BEKK

Aslak Johannessen
Senior Consultant
982 19 249
aslakjo@bekk.no