

Fagkveld hos BEKK – Scala

.....
Introduksjon til scala
31. mars 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 \equiv 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; }
}
```

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

Closure


```
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");

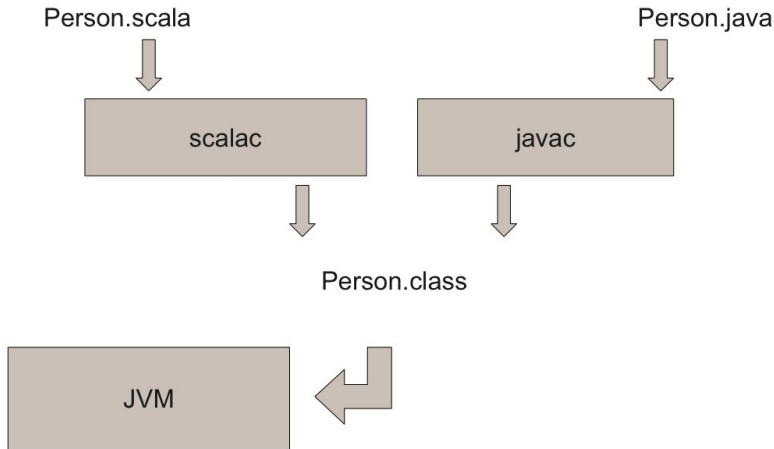
    List<String> valgte = new ArrayList<String>();
    for(String interesse : interesser){
        if(interesse.startsWith(starterMed)){
            valgte.add(interesse);
        }
    }

    return valgte;
}
```

```
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 alltid
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

- ▶ Last ned
- ▶ Gå til katalogen og kjør
 - ▶ sbt
- ▶ I sbt kjør
 - ▶ update
 - ▶ ~test-quick

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

$$3 + 5$$

Operator overload

```
//Scala listeoperator ::  
val tog = new Vogn :: new Vogn :: new Vogn :: Nil  
  
class Vogn {  
  def +(vogn: Vogn) = List(this, vogn)  
}  
  
// 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) = List(this, vogn)
```

```
  def ::(vogn: Vogn) = List(this, vogn)
```

```
}
```

```
// 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
```

Programmeringsstiler

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  
}))
```

Programmeringsstiler

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

Traits

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

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 ekstraustyr

- ▶ Genererer getters og setters
- ▶ 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{
  //companion 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 motorsykkkel")
      //verdi pattern
      case Bil(_, 4) => println("helt vanelig bil")
      //sjekking i refererte objekter, 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

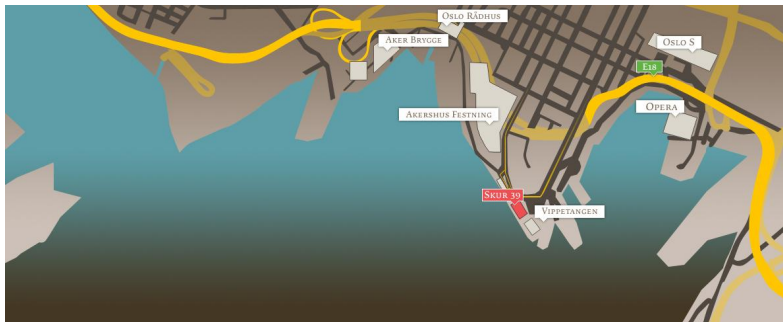
```
trait HealthCheckable{ //interface
  def isOk: Boolean
}
trait Logger { //egenskap
  def log(message: String):Unit = println(message)
}
trait LoggProcessing extends FooService{ //stackable trait
  def log(message:String):Unit

  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

BEKK CONSULTING AS
SKUR 39, VIPPETANGEN. P.O. BOX 134 SENTRUM, 0102 OSLO, NORWAY. WWW.BEKK.NO