En Scala Tutorial
for Javaprogrammerere

Version 1.3
July 1, 2010

Michel Schinz,
Philipp Haller,
Ivar Grimstad (Norsk oversettelse)

PROGRAMMING METHODS LABORATORY
EPFL
SWITZERLAND

1 Innledning

Dette dokumentet gir en kort oversikt over programmeringsspråket Scala og Scala kompilatoren. Det er ment for folk som allerede har litt erfaring med programmering og ønsker en oversikt over hva de kan gjøre med Scala. Det antas at leseren har grunnleggende kjennskap til objekt-orientert programmering generelt, og Java spesielt.

2 Et første eksempel

Som et første eksempel vil vi bruke det vanlige *Hello World*. Det er kanskje ikke så veldig interessant, men gjør det lett å demonstrere bruk av Scala verktøy uten å kunne så mye om språket. Her ser du hvordan det ser ut:

```
object HelloWorld {
  def main(args: Array[String]) {
    println("Hello, world!")
  }
}
```

Strukturen i programmet burde være kjent for de fleste Java-programmerere. Det består av en metode `main`, som tar en liste av strenger som kommandolinjeargumenter. Selve metoden består av et enkelt kall til den forhåndsdefinerte metoden `println` med den vennlige hilsningen som argument. Metoden `main` returnerer ingenting (det en prosedyre). Derfor er det ikke nødvendig å deklarer returtype.

Det som kanskje er mindre kjent for Java-programmerere er **`object`**-deklarasjonen som inneholder `main`-metoden. En slik deklarasjon introduserer det som vanligvis er kjent som et *singleton* objekt. Altså en klasse som bare har en instans. Deklarasjonen ovenfor deklarerer altså både klassen `HelloWorld` og en instans av den klassen. Instansen heter også `HelloWorld` og skapes ved behov første gang den blir brukt.

Den observante leseren legger sikkert merke til at `main`-metoden ikke er deklarert som `static`. Dette er fordi statiske medlemmer (metoder eller felter) ikke finnes i Scala. I stedet for å definere statiske medlemmer, deklareres disse som singleton objekter i Scala.

2.1 Kompilere eksempelet

For å kompilere eksempelet bruker vi Scala kompilatoren `scalac`. `scalac` fungerer som de fleste andre kompilatorer. Den tar en kildekodefil som argument, eventuelt noen argumenter og produserer en eller flere objektfiler. Objektfilene som `scalac` produserer er standard Java class-filer.

Om vi sparer programmet over i en fil som heter `HelloWorld.scala` kan vi kompilere den med følgende kommando (større-en '>' representerer kommandolinje-

prompten og skall derfor ikke skrives inn):

```
> scalac HelloWorld.scala
```

Dette vil generere noen class filer i den gjeldende katalogen. En av dem heter `HelloWorld.class` og inneholder en klasse som kan kjøres direkte ved å bruke scala kommandoen. Dette beskrives i neste avsnitt.

2.2 Kjøre eksempelet

Når programmet er kompilert kan det kjøres ved å bruke `scala` kommandoen. Måten den brukes på er veldig lik den måten java kommandoen brukes til å kjøre Java-programmer. Den aksepterer til og med de samme argumentene. HelloWorld-eksempelet kan kjøres ved følgende kommando:

```
> scala -classpath . HelloWorld

Hello, world!
```

3 Samspillet med Java

En av styrkene til Scala er at det er veldig lett å samspille med Java-kode. Alle klasser i `java.lang` pakken er importert som standard, mens andre må importeres eksplisitt. La oss se på et eksempel som viser dette. Vi vil hente dagens dato og formatere den utifra konvensjonen i et spesifikt land, for eksempel Frankrike[1].

Java sine biblioteker inneholder kraftige hjelpeklasser, som `Date` og `DateFormat`. Siden Scala interagerer sømløst med Java er det ikke nødvendig å implementere ekvivalente klasser i Scala. Vi kan helt enkelt importere klassene fra Java-pakkene:

```
import java.util.{Date, Locale}
import java.text.DateFormat
import java.text.DateFormat._

object FrenchDate {
  def main(args: Array[String]) {
    val now = new Date
    val df = getDateInstance(LONG, Locale.FRANCE)
    println(df format now)
  }
}
```

[1] Andre områder, som for eksempel den fransktalende delen av Sveits bruker samme konvensjoner.

Scala sin importfunksjonalitet er veldig lik den som finnes i Java, bare mer kraftfull. Flere klasser kan importeres fra samme pakke ved å liste dem mellom krøllparenteser. Dette er vist på den første linjen.

En annen forskjell er at når man importerer alt i en pakke eller klasse bruker man understrek (_) i stedet for stjerne (*). Grunnen til det er at stjernen er en gyldig Scala identifikator (for eksempel som metodenavn).

Den tredje importraden i eksempelet importerer derfor alle medlemmene av `DateFormat`-klassen. Dette gjør at den statiske metoden `getDateInstance` og det statiske feltet `LONG` er direkte tilgjengelig.

Inne i `main` metoden skaper vi først en instanse av Java sin `Date` klasse, som inneholder dagens dato. Deretter definerer vi et datoformat ved å bruke den statiske metoden `getDateInstance`.

Til slutt skriver vi ut dagens dato formatert etter den lokaliserte `DateFormat` instansen. Den siste linjen viser en interessant egenskap med Scala sin syntaks. Metoder som bare tar ett argument kan bli kalt uten infiks syntaks. Det betyr at uttrykket

```
df format now
```

bare er en annen, litt enklere måte å skrive

```
df.format(now)
```

Dette kan kanskje virke som en liten syntaktisk detalj, men har viktige konsekvenser. En av dem undersøker vi dypere i neste avsnitt.

For å avslutte dette avsnittet om integrasjon med Java, kan det være verdt å nevne at det også er mulig å arve fra Java-klasser og implementere Java-grensesnitt (interface) direkte i Scala.

4 Alt er objekter

Scala er et rent objekt-orientert språk i den betydelse at alt er et objekt. Dette inkluderer tall og funksjoner. På dette området er det litt annerledes enn Java, siden Java skiller primitive typer (som `boolean` og `int`) fra riktige objekter. Java tillater heller ikke å behandle funksjoner som verdier.

4.1 Tall er objekter

Siden tall er objekter har de også metoder. Faktisk består et aritmetisk uttrykk som:

```
1 + 2 * 3 / x
```

bare av metodekall. Dette er fordi det, som vi så i forrige avsnitt, er ekvivalent med:

```
(1).+(((2).*(3))./(x))
```

Dette betyr også at `+`, `*`, osv. er gyldige identifikatorer i Scala.

Parentesene rundt tallene i den andre versjonen er nødvendige siden Scala sin lexer bruker lengste treff regelen for tokens. Derfor vil den bryte ned:

```
1.+(2)
```

til `1.`, `+` og `2`. Grunnen til at denne tokenisering er valgt er fordi `1.` er en lengre treff enn `1`. Tokenet `1.` er tolket som literalen code1.0, som gjør den til en `Double` i stedet for en `Int`. Ved å skrive:

```
(1).+(2)
```

forhindrer man at `1` tolkes som `Double`.

4.2 Funksjoner er objekter

Antagelig er den største overraskelsen for Java-programmerere at også funksjoner er objekter i Scala. Det er derfor mulig å sende funksjoner som argumenter, spare dem i variable og returnere dem fra andre funksjoner. Denne muligheten til å manipulere funksjoner som verdier er et av fundamentene i en veldig interessant programmeringsparadigme som heter *funksjonell programmering*.

Som et veldig enkelt eksempel på hvorfor det kan være nyttig å bruke funksjoner som verdier kan vi tenke oss en timer funksjon som skal utføre en oppgave hvert sekund. Hvordan skal vi sende inn hvilken oppgave den skal gjøre? Ganske logisk, som en funksjon. Denne måten av funksjons-sending burde være kjent av de fleste programmere da den brukes ofte i brukergrensesnitt-kode ved å registrere såkalte call-back funksjoner som blir kalt når en hendelse inntreffer.

I følgende program er timer funksjonen kalt `oncePerSecond`, og den får en call-back funksjon som argument. Denne typen funksjon skrives som `() => Unit`, og er den typen alle funksjoner som ikke har noen argumenter og ikke returnerer noe har (`Unit` er liknende `void` i Java). main funksjonen i dette programmet kaller denne timer funksjonen med en call-back, som skriver ut en setning i terminalvinduet. Med andre ord så skriver dette programmet ut setningen "time flies like an arrow" hvert sekund i evighet.

```
object Timer {
  def oncePerSecond(callback: () => Unit) {
    while (true) { callback(); Thread sleep 1000 }
  }
  def timeFlies() {
    println("time flies like an arrow...")
  }
  def main(args: Array[String]) {
```

```
    oncePerSecond(timeFlies)
  }
}
```

4.2.1 Anonyme funksjoner

Selv om dette programmet er lett å forstå, kan det forfines ytterligere. Legg merke til at funksjonen `timeFlies` bare er definert for å senere bli sendt som parameter til oncePerSecond funksjonen. Det kjennes unødvendig å måtte navngi en funksjon som bare blir brukt en gang, så det ville vært fint om man kunne skape funksjonen idet den sendes til `oncePerSecond`.

Dette er mulig i Scala ved å bruke anonyme funksjoner. Anonyme funksjoner er funksjoner uten et navn. Den reviderte versjonen av timer programmet, som bruker en anonym funksjon i stedet for for `timeFlies` ser ut som dette:

```
object TimerAnonymous {
  def oncePerSecond(callback: () => Unit) {
    while (true) { callback(); Thread sleep 1000 }
  }
  def main(args: Array[String]) {
    oncePerSecond(() =>
      println("time flies like an arrow..."))
  }
}
```

En anonym funksjon kjennetegnes ved pilen '=>'. Denne skiller funksjonens liste av argumenter fra kroppen. I eksempelet er argumentlisten tom, som man kan se av de tomme parentesene på venstre side av pilen. Kroppen til funksjonen er den samme som for `timeFiles` over.

5 Klasser

Som vi har sett over er Scala et objekt-orientert språk, og har derfor et klassekonseptet[2]. Klasser i Scala deklareres med en syntaks som er lik den i Java. En viktig forskjell er at klasser i Scala kan ha parametre. Dette er illustrert av den følgende definisjonen av komplekse tall:

```
class Complex(real: Double, imaginary: Double) {
  def re() = real
  def im() = imaginary
}
```

[2] For ordens skyld skal det noteres at visse objekt-orienterte språk ikke har klassekonseptet, men Scala er ikke ett av dem.

Denne klassen tar to argumenter, den reelle og imaginære delen av et komplekst tall. Disse argumentene må bli sendt med når man skaper en instans av klassen `Complex`. Dette gjøres slik: `new Complex(1.5, 2.3)`. Klassen inneholder to metoder, `re` og `im` som gir aksess til disse to delene.

Legg merke til at returtypen til disse to metodene er gitt implisitt. Den vil bli antydet av kompilatoren, som ser at høyre siden av disse to metodene returnerer verdier av type `Double`. Kompilatoren kan ikke alltid antyde typen som den kan her, og det er dessverre ingen enkel regel som sier når dette er tilfelle og når det ikke er tilfelle. I praksis er ikke dette noe problem da kompilatoren klager når den ikke klarer det. Som en enkel regel, kan nybegynnere med Scala prøve å la være å eksplisitt angi type som virker enkle å utlede fra konteksten og se om kompilatoren er enig. Etter en stund burde man få en bra følelse for når man kan la være å bruke type definisjoner og når man må spesifisere dem eksplisitt.

5.1 Metoder uten argument

Et lite problem med metodene `re` og `im` er at man må legge til et tomt parentes-par etter navnet når man kaller dem. Dette er vist i følgende eksempel:

```
object ComplexNumbers {
  def main(args: Array[String]) {
    val c = new Complex(1.2, 3.4)
    println("imaginary part: " + c.im())
  }
}
```

Et bedre alternativ ville vært om man kunne aksessere den reelle og imaginære delen som om de var felter, uten de tomme parentesene. Dette er fullt mulig i Scala. Det eneste man behøver å gjøre er å definere dem som metoder uten argumenter. Slike metoder skiller seg fra metoder med null argumenter ved at de ikke har parenteser etter seg, hverken når de defineres eller når de brukes. Vår `Complex` klasse kan da bli skrevet om slik:

```
class Complex(real: Double, imaginary: Double) {
  def re = real
  def im = imaginary
}
```

5.2 Arv og overstyring

Alle klasser i Scala arver fra en superklasse. Om ingen superklasse er spesifisert, som i `Complex` eksempelet i forrige avsnitt, brukes `scala.AnyRef` implisitt. Det er mulig å overstyre metoder som arves fra en superklasse. Merk at man må spesifisere at en metode overstyrer en annen ved å bruke override modifikatoren. Dette er for å unngå at man overstyrer en metode uten å egentlig ville det. Som et eksempel, vår

`Complex` klasse kan bli utvidet til å inneholde en omdefinert `toString` metode som den arver fra `Object`.

```
class Complex(real: Double, imaginary: Double) {
  def re = real
  def im = imaginary
  override def toString() =
    "" + re + (if (im < 0) "" else "+") + im + "i"
}
```

6 Case klasser og mønstergjenkjenning

En type datastruktur som ofte brukes i programmer er trestrukturer. For eksempel representerer tolkere og kompilatorer vanligvis programmer internt som trær. XML-dokumenter er trær og mange containere er basert på trær.

Vi vill nå se nærmere på hvordan slike trestrukturer er representert og manipulert i Scala. Dette gjør vi ved å bruke et lite kalkulatorprogram. Målet til programmet er å manipulere veldig enkle aritmetiske uttrykk sammensatt av summer, konstanter og variable. To eksempler på slike uttrykk er `1 + 2` og `(x+x) + (7 + y)`.

Først må vi definere en måte å representere slike uttrykk. Den mest naturlige måten er ved å bruke en trestruktur hvor nodene er operasjoner (i dette tilfellet addisjon) og løvene er verdier (i dette tilfellet konstanter eller variable).

I Java kan et slikt tre representeres ved å bruke en abstrakt superklasse for trærne og en konkret subklasse per node eller løv. I et funksjonelt programmeringsspråk ville man brukt en algebraisk type for samme formål.

Scala tilbyr konseptet *case klasser* som ligger et sted mellom de to. Her er et eksempel på hvordan case klasser kan bli brukt til å definere typen av trær for vår oppgave:

```
abstract class Tree
case class Sum(l: Tree, r: Tree) extends Tree
case class Var(n: String) extends Tree
case class Const(v: Int) extends Tree
```

Det faktum at klassene `Sum`, `Var` og `Const` er deklarert som case klasser betyr at de skiller seg fra vanlige klasser på flere måter:

- nøkkelordet **`new`** er ikke nødvendig når man skaper instanser av disse klassene (Det vil si at man kan man skrive `Const(5)` i stedet for **`new`** `Const(5)`),
- getter funksjoner er automatisk definert for konstruktor parametre (for eksempel er det mulig å får verdien til konstruktorparameteren v til en instans c av klassen Const ved å skrive `c.v`),

- standard definisjoner av metodene `equals` og `hashCode` som bygger på strukturen av instansen og ikke på identiteten er forhåndsdefinert,
- en standard definisjon av metoden `toString` er forhåndsdefinert. Denne skriver verdien i kildekodeform (for eksempel skrives treet for uttrykket $x+1$ ut som `Sum(Var(x),Const(1))`),
- instanser av disse klasser kan bli dekomponert gjennom mønstergjenkjenning, noe som vi vil se nedenfor.

Nå som vi har definert datatypene til å representere våre aritmetiske uttrykk, kan vi begynne å definere operasjoner til å manipulere dem. Vi starter med en funksjon som evaluerer et uttrykk i et eller annet miljø. Målet til miljøet er å gi variablene verdier. For eksempel uttrykket $x+1$ evaluert i et miljø som assosierer verdien 5 med variabelen x, skrevet som $\{x \rightarrow 5\}$ gir 6 som resultat.

Vi må derfor finne en måte å representere miljøet. Selvfølgelig kunne vi brukt en eller annen assosierende datastruktur, som en hashtable, men vi kan også bruke funksjoner direkte. Et miljø er egentlig ikke annet enn en funksjon som assosierer en verdi til et (variabel)navn. Miljøet $\{x \rightarrow 5\}$ ovenfor kan helt enkelt skrives som følgende i Scala:

```
{ case "x" => 5 }
```

Denne notasjonen definerer en funksjon som, når den får strengen `"x"` som argument, returnerer heltallet 5 og kaster en exception ellers.

Før vi skriver evalueringsfunksjonen, gir vi et navn til miljøet. Vi kan selvfølgelig bruke typen `String => Int` for miljøer, men det forenkler programmet om vi introduser et navn for denne typen. Dette gjør også senere endringer lettere. I Scala gjøres dette med følgende notasjon:

```
type Environment = String => Int
```

Videre kan typen `Environment` brukes som et alias for funksjoner av typen fra `String` til `Int`.

Nå kan vi definere evalueringsfunksjonen. Konseptuelt er det veldig enkelt: Verdien av en sum av to uttrykk er helt enkelt summen av verdiene til disse uttrykkene. Verdien til variabelen fås direkte fra miljøet og verdien av en konstant er konstanten selv. å skrive det i Scala er like enkelt:

```
def eval(t: Tree, env: Environment): Int = t match {
  case Sum(l, r) => eval(l, env) + eval(r, env)
  case Var(n)    => env(n)
  case Const(v)  => v
}
```

Denne evalueringsfunksjonen virker på den måten at den gjør mønstergjenkjenning på trestrukturen `t`. Intuitivt burde meningen av definisjonen være ganske klar:

1. Først sjekkes det om treet `t` er en `Sum`. Om det er det binder den det venstre undertreet til en variabel `l` og det høyre undertreet til en variabel `r`. Deretter fortsetter evalueringen av uttrykket etter pilen. Dette uttrykket kan (og vil) bruke variablene bundet til mønsteret på venstre siden av pilen, dvs `l` og `r`.
2. Om den første sjekken ikke går gjennom, dvs om treet ikke er en `Sum`, sjekker programmet om det en `Var`. Om det er det, binder den navnet i `Var`-noden til variabelen `n` og fortsetter med høyre siden.
3. Om den andre sjekken heller ikke går gjennom, dvs. om `t` hverken er en `Sum` eller en `Var`, sjekker programmet om den er en `Const`. Dersom det er tilfelle binder den verdien i `Const`-noden til variabelen `v` og fortsetter med høyre siden.
4. Til slutt, om alle sjekker feiler, kastes en exception for å signalere at mønstergjenkjenningen ikke gikk gjennom. Dette kan hende om flere subklasser av `Tree` var deklarert.

Av dette ser vi at den grunnlegende ideen i mønstergjenkjenning er å prøve å matche en verdi med en serie av mønster. Deretter ekstrahere navn og ulike deler av verdien så snart et mønster kjennes igjen, for å til slutt kunne evaluere noe kode som bruker disse navngitte delene.

En erfaren objekt-orientert programmerer undrer kanskje hvorfor vi ikke definerte `eval` som en metode på `Tree` klassen og dens subklasser. Dette kunne vi fint gjort siden Scala tillater metodedefinisjoner i case klasser på samme måte som vanlige klasser. Valget å bruke mønstergjenkjenning eller metoder er derfor egentlig bare et spørsmål om hva man foretrekker, men det har også viktige implikasjoner på utvidbarhet:

- Når man bruker metoder er det lett å legge til en ny type node siden det helt enkelt gjøres ved å definere en ny subklasse av Tree. På den andre siden er det ganske omstendelig å legge til en ny operasjon som manipulerer treet da man må implementere metoden i alle subklassene til `Tree`.
- Når man bruker mønstergjenkjenning er situasjonen motsatt. å legge til en ny node medfører at man må modifisere alle funksjonene som gjør gjenkjenning på treet, mens det å legge til en ny operasjon er bare å definere en ny uavhengig funksjon.

La oss se på en annen operasjon på aritmetiske uttrykk for å utforske mønstergjenkjenning videre, nemlig symbolsk derivasjon. Leseren husker kanskje de følgende reglene for denne operasjonen:

1. Den deriverte av en sum er er summen av alle de deriverte

2. Den deriverte av en variabel v er en hvis v er variabel relativt til der derivasjonen skjer, ellers er den null
3. Den deriverte av en konstant er null

Disse regler kan oversettes nesten bokstavelig til Scala kode:

```
def derive(t: Tree, v: String): Tree = t match {
  case Sum(l, r) => Sum(derive(l, v), derive(r, v))
  case Var(n) if (v == n) => Const(1)
  case _ => Const(0)
}
```

Denne funksjonen introduserer to nye konsepter relatert til mønstergjenkjenning. Først, **case** uttrykket for variable har en *guard*, dvs et uttrykk etter nøkkelordet **if**. Denne *quard* forhindrer mønstergjenkjenningen i å lykkes om ikke dets uttrykk er sant. Her brukes det for å forsikre oss om at vi returnerer konstanten 1 bare hvis navnet til variabelen som deriveres er det samme som v. Den andre nye egenskapen av mønstergjenkjenning vi bruker her er *wildcard*: _. Dette er et mønster som matcher alle verdier uten at vi behøver å gi det et navn.

Vi har ikke gått gjennom alt om mønstergjenkjenning her, men stopper nå for å holde lengden på dette dokumentet nede. Men vi vil likevel se på hvordan de to funksjonene over brukes i et virkelig eksempel. Vi skriver derfor en main funksjon som utfører en rekke operasjoner på uttrykket $(x + x) + (7 + y)$. Først beregner den verdien i miljøet $\{x \to 5, y \to 7\}$, deretter beregnes den deriverte relativt x og så y.

```
def main(args: Array[String]) {
  val exp: Tree = Sum(Sum(Var("x"),Var("x")),Sum(Const(7),Var("y")))
  val env: Environment = { case "x" => 5 case "y" => 7 }
  println("Expression: " + exp)
  println("Evaluation with x=5, y=7: " + eval(exp, env))
  println("Derivative relative to x:\n " + derive(exp, "x"))
  println("Derivative relative to y:\n " + derive(exp, "y"))
}
```

Når vi kjører dette programmet får vi følgende forventede resultat:

```
Expression: Sum(Sum(Var(x),Var(x)),Sum(Const(7),Var(y)))
Evaluation with x=5, y=7: 24
Derivative relative to x:
 Sum(Sum(Const(1),Const(1)),Sum(Const(0),Const(0)))
Derivative relative to y:
 Sum(Sum(Const(0),Const(0)),Sum(Const(0),Const(1)))
```

Når vi ser på utskriften, ser vi at resultatet av den deriverte burde bli forenklet før det vises for brukeren. å definere en forenklende funksjon ved hjelp av mønstergjenkjen-

ning er et interessant (og overraskende komplisert) problem som overlates til den interesserte leseren.

7 Traits

Ved siden av å arve kode fra en superklasse, kan Scala også importere kode fra en eller flere *traits*.

Kanskje er den beste måten for en Java-programmerer å forstå hva *traits* er, å se på dem som interfaces som også kan inneholde kode. Når en klasse arver fra en trait i Scala, implementer den *traitets* interface og arver all koden som er med i *traitet*.

La oss se på et klassisk eksempel for å demonstrere styrken med *traits*: sorterte objekter. Det er ofte greit å kunne sammenligne objekter av en gitt klasse med hverandre, for eksempel for å kunne sortere dem. I Java gjøres det ved at klasser som skal sammenlignes implementerer interfacet `Comparable`. I Scala kan vi gjøre det enda bedre enn i Java ved å definere en ekvivalent til `Comparable` som en *trait*. Denne kaller vi `Ord`.

Når vi sammenligner objekter er det seks forskjellige predikater som er interessante: mindre enn, mindre enn eller lik, lik, ulik, større enn eller lik og større enn. Men å definere alle sammen er unødvendig siden fire av disse seks kan bli utledet av de andre to. Det vil si at om vi for eksempel har predikatene lik og mindre enn, kan man utlede de andre fra disse. I Scala kan alle disse observasjonene fanges i den følgende definisjonen av en *trait*:

```
trait Ord {
  def < (that: Any): Boolean
  def <=(that: Any): Boolean = (this < that) || (this == that)
  def > (that: Any): Boolean = !(this <= that)
  def >=(that: Any): Boolean = !(this < that)
}
```

Denne definisjonen inneholder to elementer. Først av alt skaper den en ny type, `Ord`. Denne typen fungerer på samme måte som Java sitt `Comparable` interface. I tillegg definerer den default implementasjoner av tre predikater som er utledet av den fjerde abstrakte. Predikatene for lik og ulik er ikke med siden de alltid gjelder for alle objekter.

Typen `Any` er supertypen til alle andre typer i Scala. Den kan ses på som en mer generell versjon av Java sin `Object` type siden den også er en supertype for bastypene `Int`, `Float` osv.

For å gjøre objekter av en klasse sammenlignbare er det derfor nok å definere predikatene som tester for likhet og ulikhet, samt mikse inn `Ord` klassen. Som et eksempel kan vi definere en `Date` klasse som representerer datoene i den gregorianske kalender. Slike datoer er satt sammen av en dag, en måned og et år. Alle disse representert

som heltall. Vi kan derfor starte definisjonen av `Date` klassen slik:

```
class Date(y: Int, m: Int, d: Int) extends Ord {
  def year = y
  def month = m
  def day = d

  override def toString(): String = year + "-" + month + "-" + day
```

Det viktige her er deklarasjonen extends Ord som følger etter klassenavnet og parametrene. Det deklarerer at Date klassen arver fra *traitet* `Ord`.

Deretter definerer vi `equals` metoden som arves fra `Object`, slik at den sammenligner datoer ved å sammenligne ett og ett felt. Default implementasjonen av `equals` kan ikke brukes fordi den, på samme måte som i Java, sammenligner objekter fysisk. Dermed får vi følgende definisjon av equals:

```
  override def equals(that: Any): Boolean =
    that.isInstanceOf[Date] && {
      val o = that.asInstanceOf[Date]
      o.day == day && o.month == month && o.year == year
    }
```

Denne metoden bruker de forhåndsdefinerte metodene `isInstanceOf` og `asInstanceOf`. Den første, `isInstanceOf`, kan sammenlignes med Java sin `instanceof` operator, og returnerer true dersom objektet som sammenlignes er en instans av den angitte typen.

Den andre, `asInstanceOf`, kan sammenlignes med Java sin `cast` operator. Dersom objektet er av den angitte typen, blir det behandlet deretter. Ellers kastes en `ClassCastException`.

Den siste metoden som må defineres er den som sjekker for mindre enn. Den bruker en annen forhåndsdefinert metode, `error`, som kaster en exception med den gitte feilmeldingen.

```
  def <(that: Any): Boolean = {
    if (!that.isInstanceOf[Date])
      error("cannot compare " + that + " and a Date")

    val o = that.asInstanceOf[Date]
    (year < o.year) ||
    (year == o.year && (month < o.month ||
                        (month == o.month && day < o.day)))
  }
```

Med dette er er definisjonen av `Date` klassen komplett. Instanser av denne klassen kan enten ses på som datoer eller sammenlignbare objekter. Videre, definerer de

alle seks sammenligningspredikater nevnt ovenfor. `equals` og `<` fordi de er med direkte i definisjonen av `Date` klassen, og de andre siden de arves fra `Ord` *traitet*.

Traits kan brukes i mange andre situasjoner enn den som ble vist her, men å ta opp alt her er utenfor omfanget av dette dokumentet.

8 Generiskhet

Den siste karakteristikken av Scala vi undersøker i denne introduksjonen er generiskhet. Java-programmerere kjenner nok godt til problemene mangelen på generiskhet medførte i versjonene før Java 1.5.

På samme måte som i Javaversjonene fra 1.5 og nyere, gjør Scala det mulig å definere generiske klasser (og metoder). La oss se på dette ved et enkelt eksempel. En referanse kan enten være tom eller peke på et objekt av en eller annen type.

```
class Reference[T] {
  private var contents: T = _

  def set(value: T) { contents = value }
  def get: T = contents
}
```

Klassen `Reference` er parametrisert med typen `T`, som er typen til elementene klassen refererer til. Denne typen blir brukt i klassen som typen til `contents` variabelen, dvs. argumentet til set-metoden og returtypen til get-metoden.

Koden over introduserer variabler i Scala, noe som ikke burde kreve videre forklaring. Det er likevel interessant å se at den initielle verdien som gis til variabelen er `_`, som representerer en default verdi. Default verdien er 0 for en numerisk type, **`false`** for en *boolsk* type, `()` for en `Unit` type og **`null`** for alle objektreferenser.

For å bruke `Reference` klassen, behøver man å spesifisere hvilken type som skal brukes for parameteren `T`, dvs. typen til elementet som holdes i cellen. For å skape, og bruke, en celle som inneholder et heltall, kan man skrive følgende:

```
object IntegerReference {
  def main(args: Array[String]) {
    val cell = new Reference[Int]
    cell.set(13)
    println("Reference contains the half of " + (cell.get * 2))
  }
}
```

Som det fremgår av dette eksempelet er det ikke nødvendig å *caste* verdien som returneres av get metoden før man bruker den som et heltall. Det er heller ikke mulig å spare noe annet enn ett heltall i den aktuelle cellen, siden den var deklarert

til å skulle inneholde et heltall.

9 Konklusjon

Dette dokumentet gav en kort oversikt over programmeringsspråket Scala og presenterte noen enkle eksempler. Om du er interessert i å lese mer kan du fortsette med følgedokumentet *Scala By Example* som inneholder mer avanserte eksempler. Ved behov kan *Scala Language Specification* konsulteres.