EDAA45 Programmering, grundkurs

Läsvecka 9: Mönster, Undantag

Björn Regnell

Datavetenskap, LTH

Lp1-2, HT 2016

9 Mönster, Undantag

- Bonus
- Specialundervisning
- Nyhet: Scala 2.12.0
- Veckans lab: `chords-team`
- Matchning
- Option
- Implementera `equals` med `match`
- Undantag
- `scala.util.Try`

Bonus

Grupper, antal medlemmar, bonus

Grupp	Antal	SumKS	Bonus
D01a	5	12	2
D01b	5	17	3
D02a	6	12	2
D02b	6	20	3
D03a	5	9	2
D03b	5	11	2
D04a	5	12	2
D04b	5	12	2
D05a	5	12	2
D05b	5	10	2
D06a	5	9	2
D06b	4	13	3
D07a	4	7	2
D07b	3	10	3
D08a	5	15	3
D08b	3	10	3
D09a	4	7	2
D09b	6	11	2
D10a	4	7	2
D10b	5	10	2
D11a	5	11	2
D11b	4	10	3
D12a	4	9	2
D12b	5	13	3

Grupper, antal medlemmar, bonus

Grupp	Antal	SumKS	Bonus
D01a	5	12	2
D01b	5	17	3
D02a	6	12	2
D02b	6	20	3
D03a	5	9	2
D03b	5	11	2
D04a	5	12	2
D04b	5	12	2
D05a	5	12	2
D05b	5	10	2
D06a	5	9	2
D06b	4	13	3
D07a	4	7	2
D07b	3	10	3
D08a	5	15	3
D08b	3	10	3
D09a	4	7	2
D09b	6	11	2
D10a	4	7	2
D10b	5	10	2
D11a	5	11	2
D11b	4	10	3
D12a	4	9	2
D12b	5	13	3

- Bonus gäller vid första ordinarie tentatillfälle

Grupper, antal medlemmar, bonus

Grupp	Antal	SumKS	Bonus
D01a	5	12	2
D01b	5	17	3
D02a	6	12	2
D02b	6	20	3
D03a	5	9	2
D03b	5	11	2
D04a	5	12	2
D04b	5	12	2
D05a	5	12	2
D05b	5	10	2
D06a	5	9	2
D06b	4	13	3
D07a	4	7	2
D07b	3	10	3
D08a	5	15	3
D08b	3	10	3
D09a	4	7	2
D09b	6	11	2
D10a	4	7	2
D10b	5	10	2
D11a	5	11	2
D11b	4	10	3
D12a	4	9	2
D12b	5	13	3

- Bonus gäller vid första ordinarie tentatillfälle
- D07b och D08b har 3 st; vill ni göra merge?

Specialundervisning

Specialundervisning

Under vecka w09 och w10 (till att börja med) kommer vi att organisera **specialundervisning** under dessa **resurstider**:

- **Torsdag kl 8-10** i både **Falk** och **Val** (Gustav, Valthor, Emil)
- **Torsdag kl 10-12** i **Falk** (Maj)
- OBS! Dessa rumtider är till för de som hade 0 eller 1 på kontrollskrivningen och som ansökt om specialundervisning via länk i speciell mejlinbjudan.

Undantag: Om du har mer än 1 på kontrollskrivningen men inte alls har möjlighet att gå på någon annan resurstid än ovan är du också välkommen; anmäl då din situation till handledaren på plats och så får du vara med i gruppen och kan få svar på frågor etc. som vanligt.

Nyhet: Scala 2.12.0

Nyhet: Scala 2.12.0 släpptes 3:e Nov 2016

- Nytt i Scala 2.12:
  - **Optimeringar** ”under huven” som **kräver Java 8**
  - **Snabbare, klarar sig med mindre minne, kortare bytekod, ...**
  - Väsentligt förbättrad **Scaladoc**:
    `http://www.scala-lang.org/api`
  - Du hittar gamla Scaladoc för 2.11.8 här:
    `http://www.scala-lang.org/api/2.11.8/`
- I denna kursomgång och på LTH:s datorer kommer vi att stanna kvar vid **2.11.8** (nästa kursomgång kör vi 2.12)
- Observera att 2.12 **inte är bytekodskompatibel** med 2.11 så du måste kompilera om all gammal kod om den ska funka med nykompilerad kod om du installerar 2.12.
- Med `sbt` (se appendix G) är det enkelt att ha många olika versioner av Scala-kompilatorn igång på samma maskin.

För den intresserade, läs mer här: `http://www.scala-lang.org/news/2.12.0`

Veckans lab: chords-team

Veckans lab: `chords-team`

Övergripande syfte:

- Träna på case-klasser, matchning, undantag
- Jobba med ett större program med flera delar i olika filer
- Jobba flera personer på samma program

Innehåll:

- Skapa och spara ackord på gitarr (6 strängar) och ukulele (4 strängar)
- Spela upp ackord med Javas inbyggda musikspelare inkapslad i `SimpleNotePlayer`
- Rita ackord med `SimpleWindow`

Hur mycket ni gör beror på hur många ni är i gruppen och hur stora ambitioner ni har. Diskutera detta med handledare på resurstid.

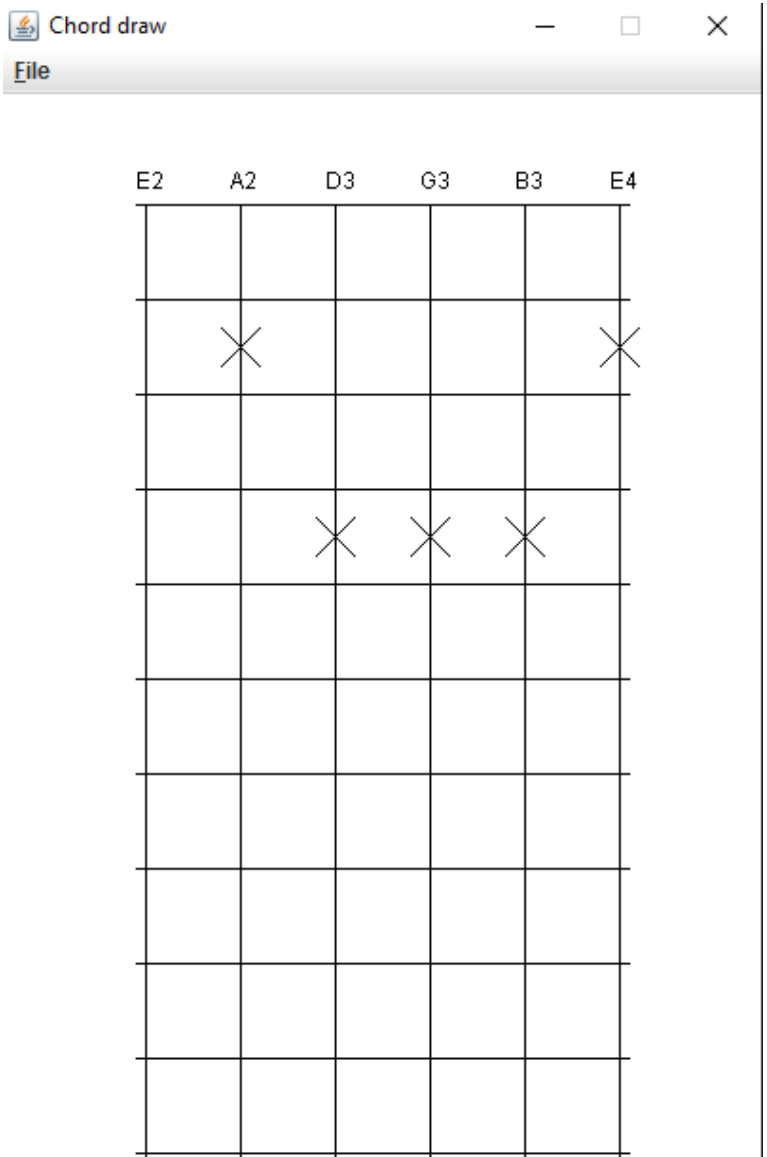
Toner, oktaver och ackord

- Det finns 12 toner som har speciella namn:
  C, C#, D, D#, etc. (uttalas: c, ciss, d, diss, etc.)
- Jämför vita och svarta tangenter på ett piano:
  avståndet mellan varje tangent är ett s.k. *halvt tonsteg*.
- Toner återkommer i oktaver, modulo 12.
- Tonen som representeras av strängen `"D2"` är tonen D i andra oktaven.
- Tonen `"D2"` motsvarar heltalet 26 på labben.
- Ett ackord består av flera toner.

```
1  scala> val notes = Vector("C", "C#", "D", "D#", "E", "F", "F#", "G", "G#", "A", "A#", "B")
2
3  scala> notes.size
4  res0: Int = 12
5
6  scala> notes(26 % 12)
7  res1: String = D
```

Toner på ett stränginstrument

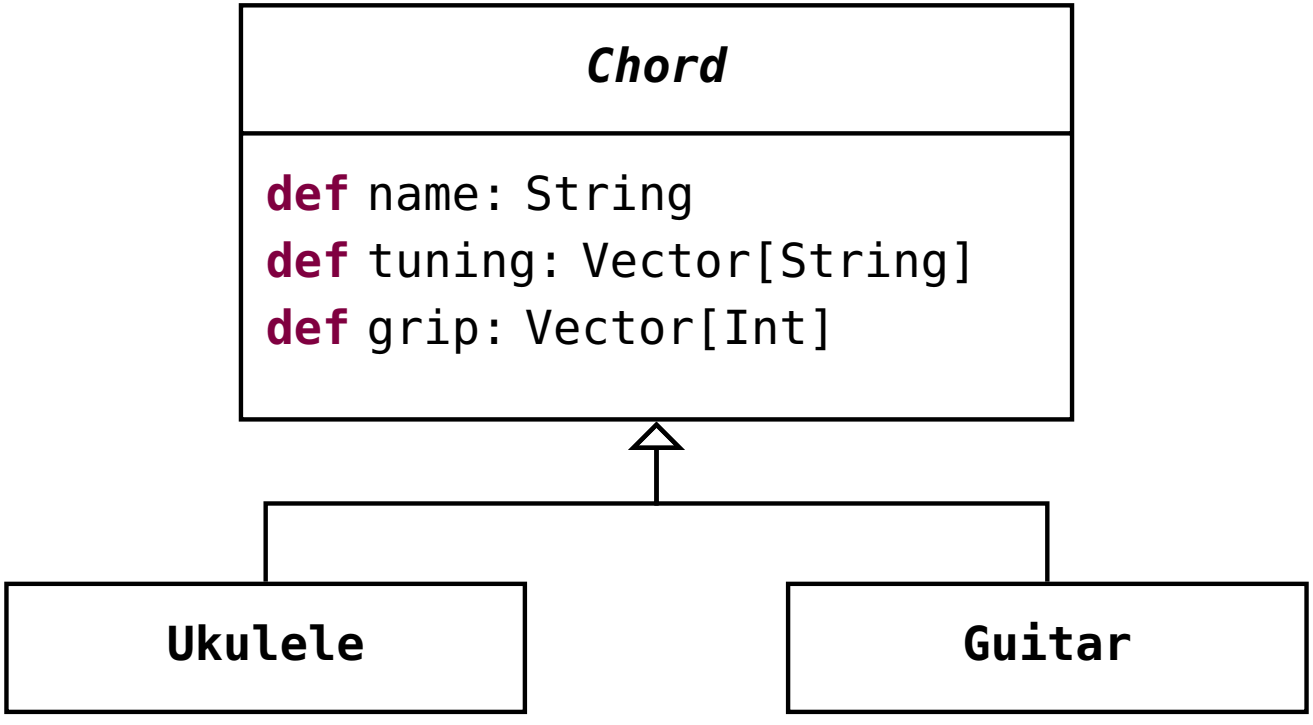
- Gitarr och ukulele har 6 resp. 4 strängar och en greppbräda med s.k. band.
- Om man trycker ned ett finger på (egentligen bakom) första bandet höjs tonen ett halvt tonsteg.
- Exempel: om en sträng är stämd i D3 blir tonen om man trycker ned fingret på fjärde bandet F#3.
- www.gitarr.org
- www.stefansukulele.com



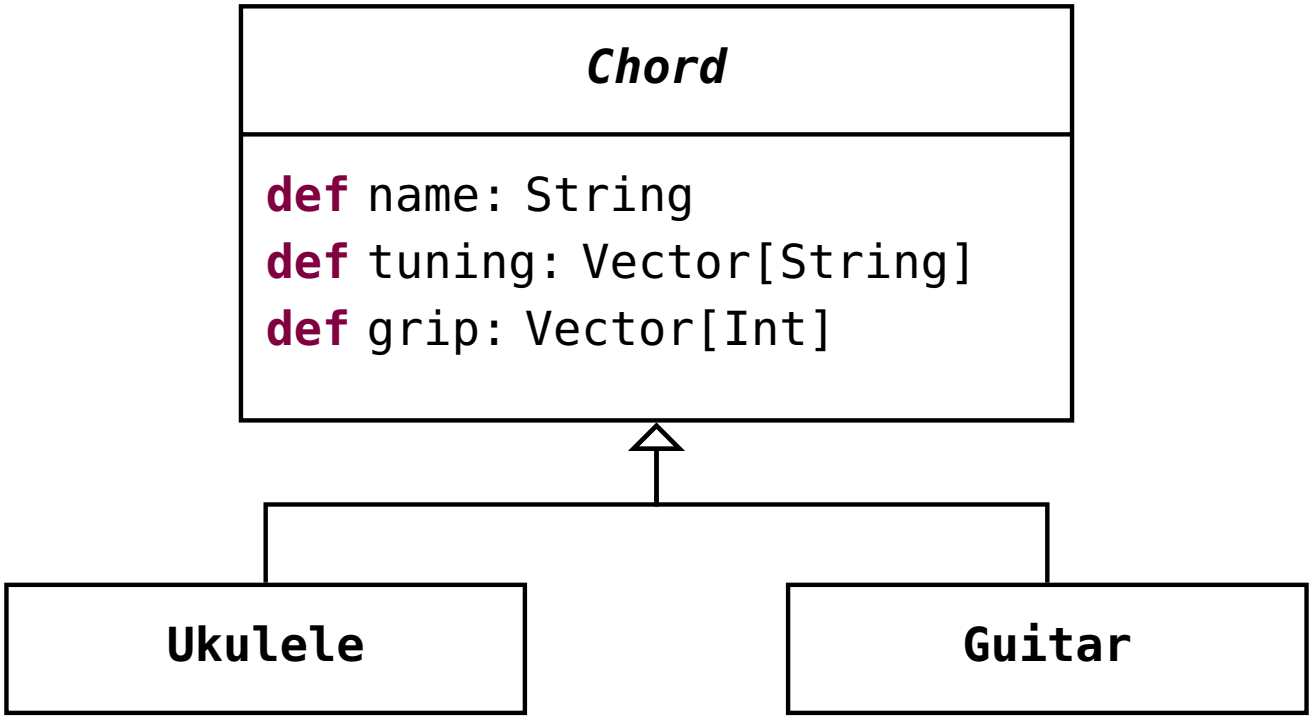
Modell av gitarr och ukulele

```
object model {

  type Tuning = Vector[String]
  type Grip = Vector[Int]

  trait Chord {
    def name: String
    def tuning: Tuning
    def grip: Grip
  }

  case class Guitar(name: String, grip: Grip) extends Chord {
    val tuning = Vector("E2", "A2", "D3", "G3", "B3", "E4")
  }

  case class Ukulele(name: String, grip: Grip) extends Chord {
    val tuning = Vector("G4", "C4", "E4", "A4")
  }

}
```

En gemensam bastyp för olika ackord



En gemensam bastyp för olika ackord




```
1  scala> import model._
2
3  scala> val uc = Ukulele("C", Vector(0, 0, 0, 3))
4  uc: model.Ukulele = Ukulele(C,Vector(0, 0, 0, 3))
5
6  scala> val ge = Guitar("E", Vector(0, 2, 2, 1, 0, 0))
7  ge: model.Guitar = Guitar(E,Vector(0, 2, 2, 1, 0, 0))
```

Grupparbete

- Förslag på arbetssätt:
  - Träffas nu på rasten och boka nästa gruppmöte
  - Förberedelser inför första gruppmötet: individuella studier av labbinstruktioner och koden som är given i workspace .../workspace/w08_chords_team/src
    OBS! numrering av labbarna enl. ”gamla” veckor
  - Träffas gärna i ett studierum med whiteboard
  - På mötet: gå igenom uppgift och given kod så att alla fattar vad det går ut på; bestäm omfattning och ansvarsuppdelning
  - När ni träffas, skissa upp den kod som just du håller på med på whiteboard och få feedback och ge feedback till andra
  - På varje gruppmöte, bestäm tid för nästa möte och vad var och en ska försöka hinna tills dess
- Ni får **lov att ändra på omfattningen** efter antalet gruppmedlemmar, ambition och förmåga: diskutera detta med handledare på resurstid
- **Minimikrav**: att med textkommando kunna skapa/spara/ladda gitarr- och ukulele-ackord och att ni tränar på matchning

Matchning

Vad är matchning?

Vad är matchning?

Matchning gör man då man vill jämföra ett värde mot andra värden och hitta överensstämmelse (eng. *match*).

Vad är matchning?

Matchning gör man då man vill jämföra ett värde mot andra värden och hitta överensstämmelse (eng. *match*).

Detta kan man t.ex. göra med nästlade if-else-satser/uttryck:

```
val g = scala.io.StdIn.readLine("grönsak:")
val smak =
  if (g == "gurka") "gott!"
  else if (g == "tomat") "jättegott!"
  else if (g == "broccoli") "ganska gott..."
  else "inte gott :("

println(g + " är " + smak)
```

Javas switch-sats

De flesta C-liknande språk (men inte Scala) har en **switch**-sats som man kan använda istället för (vissa) nästlade if-else-satser:

```
import java.util.Scanner;

public class SwitchNoBreak {
    public static void main(String[] args) {
        Scanner scan = new Scanner(System.in);
        System.out.println("Skriv grönsak:");
        String g = scan.next();
        switch (g) {
        case "gurka":
        case "tomat":
            System.out.println("gott!");
            break;
        case "broccoli":
            System.out.println("ganska gott...");
            break;
        default:
            System.out.println(" mindre gott...");
            break;
        }
    }
}
```

Funkar bara för primitiva typer och några till (t.ex. String).

Javas switch-sats utan break

Saknad **break**-sats ”faller igenom” till efterföljande gren:

```
import java.util.Scanner;

public class SwitchNoBreak {
    public static void main(String[] args) {
        Scanner scan = new Scanner(System.in);
        System.out.println("Skriv grönsak:");
        String g = scan.next();
        switch (g) {
        case "gurka":
        case "tomat":
            System.out.println("gott!");
            break;
        case "broccoli":
            System.out.println("ganska gott...");
            break;
        default:
            System.out.println(" mindre gott...");
            break;
        }
    }
}
```

En glömd **break** kan ge svårhittad bugg...

Javas switch-sats med glömd break

```
import java.util.Scanner;

public class SwitchForgotBreak {
    public static void main(String[] args) {
        Scanner scan = new Scanner(System.in);
        System.out.println("Skriv grönsak:");
        String g = scan.next();
        switch (g) {
        case "gurka":
            System.out.println("gott!");
        case "tomat":
            System.out.println("gott!");
            break;
        case "broccoli":
            System.out.println("ganska gott...");
            break;
        default:
            System.out.println("mindre gott...");
            break;
        }
    }
}
```

Javas switch-sats med glömd break

```java
import java.util.Scanner;

public class SwitchForgotBreak {
    public static void main(String[] args) {
        Scanner scan = new Scanner(System.in);
        System.out.println("Skriv grönsak:");
        String g = scan.next();
        switch (g) {
        case "gurka":
            System.out.println("gott!");
        case "tomat":
            System.out.println("gott!");
            break;
        case "broccoli":
            System.out.println("ganska gott...");
            break;
        default:
            System.out.println("mindre gott...");
            break;
        }
    }
}
```

```
$ java SwitchForgotBreak
Skriv grönsak:
gurka
gott!
gott!
```

Scalas `match`-uttryck

Scala har ingen `switch`-sats men erbjuder i stället ett **`match`-uttryck** som är kraftfullare och ger ett värde.

```
val g = scala.io.StdIn.readLine("grönsak:")
val smak = g match {
  case "gurka" => "gott!"
  case "tomat" => "jättegott!"
  case "broccoli" => "ganska gott..."
  case _ => "mindre gott..."
}
```

Och den ”faller inte igenom” som Javas `switch`-sats!

Scalas `match`-uttryck

Scala har ingen `switch`-sats men erbjuder i stället ett **`match`-uttryck** som är kraftfullare och ger ett värde.

```
val g = scala.io.StdIn.readLine("grönsak:")
val smak = g match {
  case "gurka" => "gott!"
  case "tomat" => "jättegott!"
  case "broccoli" => "ganska gott..."
  case _ => "mindre gott..."
}
```

Och den ”faller inte igenom” som Javas `switch`-sats!

- Varje **`case`**-gren testas var för sig i tur och ordning uppifrån och ned.

Scalas `match`-uttryck

Scala har ingen `switch`-sats men erbjuder i stället ett **`match`-uttryck** som är kraftfullare och ger ett värde.

```
val g = scala.io.StdIn.readLine("grönsak:")
val smak = g match {
  case "gurka" => "gott!"
  case "tomat" => "jättegott!"
  case "broccoli" => "ganska gott..."
  case _ => "mindre gott..."
}
```

Och den ”faller inte igenom” som Javas `switch`-sats!

- Varje **`case`**-gren testas var för sig i tur och ordning uppifrån och ned.
- Det som står mellan **`case`** och **`=>`** kallas ett **mönster** (eng. *pattern*)

Scalas `match`-uttryck

Scala har ingen `switch`-sats men erbjuder i stället ett **`match`-uttryck** som är kraftfullare och ger ett värde.

```
val g = scala.io.StdIn.readLine("grönsak:")
val smak = g match {
  case "gurka" => "gott!"
  case "tomat" => "jättegott!"
  case "broccoli" => "ganska gott..."
  case _ => "mindre gott..."
}
```

Och den ”faller inte igenom” som Javas `switch`-sats!

- Varje **`case`**-gren testas var för sig i tur och ordning uppifrån och ned.
- Det som står mellan **`case`** och **`=>`** kallas ett **mönster** (eng. *pattern*)
- Sista default-grenen ovan kallas **wildcard-mönster**: **`case _ =>`**

Scalas `match`-uttryck

Scala har ingen `switch`-sats men erbjuder i stället ett **`match`-uttryck** som är kraftfullare och ger ett värde.

```
val g = scala.io.StdIn.readLine("grönsak:")
val smak = g match {
  case "gurka" => "gott!"
  case "tomat" => "jättegott!"
  case "broccoli" => "ganska gott..."
  case _ => "mindre gott..."
}
```

Och den ”faller inte igenom” som Javas `switch`-sats!

- Varje **`case`**-gren testas var för sig i tur och ordning uppifrån och ned.
- Det som står mellan **`case`** och **`=>`** kallas ett **mönster** (eng. *pattern*)
- Sista default-grenen ovan kallas **wildcard-mönster**: **`case _ =>`**
- Ovan är exempel på matchning mot **konstant-mönster**, i detta fallet tre stycken strängkonstantmönster.

Scalas `match`-uttryck

Scala har ingen `switch`-sats men erbjuder i stället ett **`match`-uttryck** som är kraftfullare och ger ett värde.

```
val g = scala.io.StdIn.readLine("grönsak:")
val smak = g match {
  case "gurka" => "gott!"
  case "tomat" => "jättegott!"
  case "broccoli" => "ganska gott..."
  case _ => "mindre gott..."
}
```

Och den ”faller inte igenom” som Javas `switch`-sats!

- Varje **`case`**-gren testas var för sig i tur och ordning uppifrån och ned.
- Det som står mellan **`case`** och **`=>`** kallas ett **mönster** (eng. *pattern*)
- Sista default-grenen ovan kallas **wildcard-mönster**: **`case _ =>`**
- Ovan är exempel på matchning mot **konstant-mönster**, i detta fallet tre stycken strängkonstantmönster.
- Det finns många andra sätt att skriva mönster.

Matchning med gard

Man kan stoppa in en s.k **gard** (eng. *guard*) innan pilen **=>** för att villkora matchningen: (notera **if**, parenteser behövs ej)

```
val g = scala.io.StdIn.readLine("grönsak:")
def f = g match {
  case "gurka" if math.random > 0.5 => "gott ibland!"
  case "tomat" => "jättegott!"
  case "broccoli" => "ganska gott..."
  case _ => "mindre gott..."
}
```

case-grenen med gard ger bara en lyckad matchning om uttrycket efter **if** är sant; annars provas nästa gren, etc.

Matchning med variabelmönster

Om det finns ett namn efter **`case`** som börjar med liten begynnelsebokstav, blir detta namn en variabel som automatiskt binds till uttrycket före **`match`**:

```
val g = scala.io.StdIn.readLine("grönsak:")
def f = g match {
  case "gurka" if math.random > 0.5 => "gott ibland!"
  case "tomat" => "jättegott!"
  case "broccoli" => "ganska gott..."
  case other => "smakar bakvänt: " + other.reverse
}
```

Ett enkelt variabelmönster, så som
`case` `other =>` `...`
i exemplet ovan, matchar **allt**!
`other` får alltså värdet av g om g **inte** är `"gurka"`, `"tomat"`, `"broccoli"`.

Matchning med typade mönster

Med en typannotering efter en variabel får man ett **typat mönster** (eng. *typed pattern*). Om matchningen lyckas blir värdet **omvandlat** till den specifika typen och binds till variabeln.

```
def f = if (math.random < 0.5) 42 + math.random else "gurka" + math.random
def g = f match {
  case x: Double => x.round.toInt
  case s: String => s.length
}
```

Vad har funktionen `f` för returtyp?

Matchning med typade mönster

Med en typannotering efter en variabel får man ett **typat mönster** (eng. *typed pattern*). Om matchningen lyckas blir värdet **omvandlat** till den specifika typen och binds till variabeln.

```
def f = if (math.random < 0.5) 42 + math.random else "gurka" + math.random
def g = f match {
  case x: Double => x.round.toInt
  case s: String => s.length
}
```

Vad har funktionen `f` för returtyp?
Matchning mot specifika typer enl. ovan används i idiomatisk Scala hellre än `isInstanceOf` men man kan göra motsvarande ovan med if-uttryck:

```
def g2 = {
  val x = f
  if (x.isInstanceOf[Double]) x.asInstanceOf[Double].round.toInt
  else if (x.isInstanceOf[String]) x.asInstanceOf[String].length
}.asInstanceOf[Int]
```

Konstruktormönster med case-klasser

En basklass med gemensamma delar och två subtyper:

```
trait Grönsak {
  def vikt: Int
  def ärRutten: Boolean
}
case class Gurka(vikt: Int, ärRutten: Boolean) extends Grönsak
case class Tomat(vikt: Int, ärRutten: Boolean) extends Grönsak
```

Konstruktormönster med case-klasser

En basklass med gemensamma delar och två subtyper:

```
trait Grönsak {
  def vikt: Int
  def ärRutten: Boolean
}
case class Gurka(vikt: Int, ärRutten: Boolean) extends Grönsak
case class Tomat(vikt: Int, ärRutten: Boolean) extends Grönsak
```

Tack vare case-klasserna kan man använda **konstruktormönster** (eng. *constructor pattern*) för att kolla vad som finns **inuti** en instans:

```
def testa(g: Grönsak): String = g match {
  case Gurka(v, false) => "gott, väger " + v
  case Gurka(_, true)  => "inte gott"
  case Tomat(v, r)     => (if (r) "inte " else "") + "gott, väger " + v
  case _ => "okänd grönsak: " + g
}
```

Konstruktormönster ”**plockar isär**” det som matchas och binder variabler till de attribut som finns i case-klassens konstruktor.

Plocka i sär samlingar med djupa mönster

```
def visa(xs: Vector[Grönsak]): String = xs match {
  case Vector()                => "ingen"
  case Vector(Gurka(v, true)) => "en rutten gurka som väger " + v
  case Vector(g1, g2)          => s"två: $g1, $g2"
  case p1 +: p2 +: ps          => "minst två"
  case p +: ps                 => "många! varav första: " + p
}
```

Mönstermatchning och case-objekt

En bastyp och specifika singelobjekt av gemensam typ:

```
trait Färg
case object Spader  extends Färg
case object Hjärter extends Färg
case object Ruter   extends Färg
case object Klöver  extends Färg

def parallellFärg(f: Färg): Färg = f match {
  case Spader  => Klöver
  case Klöver  => Spader
  case Hjärter => Ruter
}
```

Vilken case-gren har vi glömt? Kan kompilatorn hjälpa oss?

Mönstermatchning och case-objekt

En bastyp och specifika singelobjekt av gemensam typ:

```
trait Färg
case object Spader  extends Färg
case object Hjärter extends Färg
case object Ruter   extends Färg
case object Klöver  extends Färg

def parallellFärg(f: Färg): Färg = f match {
  case Spader  => Klöver
  case Klöver  => Spader
  case Hjärter => Ruter
}
```

Vilken case-gren har vi glömt? Kan kompilatorn hjälpa oss?

```
1 scala> parallellFärg(Ruter)
2 scala.MatchError: Ruter (of class Ruter$)
3   at .parallellFärg(<console>:18)
```

Runtime exception :(och en bugg som kan vara svårhittad...

Mönstermatchning och förseglade typer

Med nyckelordet **sealed** får vi en kompileringsvarning :)

```
sealed trait Färg
case object Spader  extends Färg
case object Hjärter extends Färg
case object Ruter   extends Färg
case object Klöver  extends Färg

def parallellFärg(f: Färg): Färg = f match {
  case Spader  => Klöver
  case Klöver  => Spader
  case Hjärter => Ruter
}
```

```
1 // Exiting paste mode, now interpreting.
2
3 <console>:23: warning: match may not be exhaustive.
4 It would fail on the following input: Ruter
5        def parallellFärg(f: Färg): Färg = f match {
```

Stora/små begynnelsebokstäver vid matchning

Fallgrop

Mönster på andra ställen än i `match`

Mönster i deklarationer
Mönster i for-satser
Fördjupning: läs om partiella funktioner här:

www.artima.com/pins1ed/case-classes-and-pattern-matching.html#15.7

Mönstermatchning och metoden `unapply`

När du deklarerar en case-klass kommer kompilatorn att **automatiskt generera en metod** med namnet **`unapply`** som kan ”plocka isär” en instans av klassen i sina beståndsdelar, wrappat i en tupel i case-klassen `Some`.

```
1  scala> case class Gurka(vikt: Int, ärRutten: Boolean)
2
3  scala> Gurka.unapply // tryck TAB två gånger för att se metodhuvudet
4   case def unapply(x$0: Gurka): Option[(Int, Boolean)]
5
6  scala> val g = Gurka(100, false)
7
8  scala> Gurka.unapply(g)
9  res0: Option[(Int, Boolean)] = Some((100,false))
```

Typen `Option` används för att kunna hantera värden som eventuellt saknas. Metoden `unapply` anropas vid matchning.

Mönstermatchning och metoden `unapply`

När du deklarerar en case-klass kommer kompilatorn att **automatiskt generera en metod** med namnet **`unapply`** som kan ”plocka isär” en instans av klassen i sina beståndsdelar, wrappat i en tupel i case-klassen `Some`.

```
1  scala> case class Gurka(vikt: Int, ärRutten: Boolean)
2
3  scala> Gurka.unapply // tryck TAB två gånger för att se metodhuvudet
4   case def unapply(x$0: Gurka): Option[(Int, Boolean)]
5
6  scala> val g = Gurka(100, false)
7
8  scala> Gurka.unapply(g)
9  res0: Option[(Int, Boolean)] = Some((100,false))
```

Typen `Option` används för att kunna hantera värden som eventuellt saknas. Metoden `unapply` anropas vid matchning.

Fördjupning: Man kan skapa en egna s.k. extraktorer (eng. *extractors*) som funkar med match (se fördjupningsövning 22 för exempel)

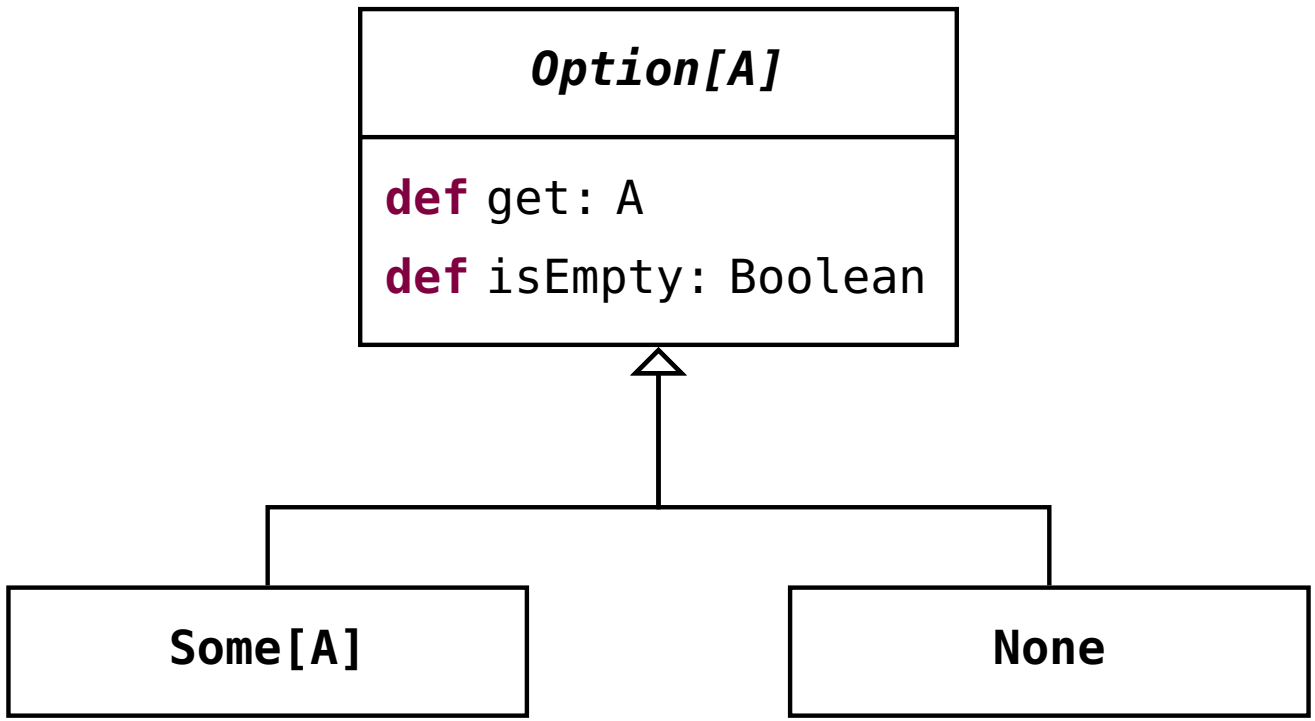
Option

Hur hantera saknade värden?

Olika sätt att hantera saknade värden:

- Hitta på ett specialvärde: exempel -1 för icke nedtryckt sträng
- `null` om `AnyRef`/`Object` (vanligt i Java, mkt ovanligt i Scala)
- Använd en samling och låt tom samling representera saknad värde:
  ```
  val grip = Vector(Vector(2), Vector(7), Vector(), Vector(3))
  ```
- `Option[T]` gemensam bastyp för:
  `None` som representerar **saknat värde**, och
  `Some[T]` som representerar att **värde finns**

En gemensam bastyp för ett värde som kanske saknas



Option för hantering av ev. saknade värden

Alla vill inte berätta för Facebook vad de har för kön.
Förbättra Facebooks kod med ett litet Scala-program:

```
trait Gender
case object Male   extends Gender
case object Female extends Gender

case class Person(name: String, gender: Option[Gender])
```

Option för hantering av ev. saknade värden

Alla vill inte berätta för Facebook vad de har för kön.
Förbättra Facebooks kod med ett litet Scala-program:

```
trait Gender
case object Male   extends Gender
case object Female extends Gender

case class Person(name: String, gender: Option[Gender])
```

```
1  scala> val p1 = Person("Björn",  Some(Male))
2  scala> val p2 = Person("Sandra", Some(Female))
3  scala> val p3 = Person("Andro",  None)
4  scala> val g2 = p2.gender
5  scala> def show(g: Option[Gender]): String = g match {
6           case Some(x) => x.toString
7           case None    => "unknown"
8         }
9  scala> show(g2)
10 scala> show(p3.gender)
11 scala> val ps = Vector(p1,p2,p3)
12 scala> ps.map(_.gender).map(show)
```

Några smidiga metoder på `Option`

getOrElse
map
foreach
isEmpty

Några samlingsmetoder som ger en `Option`

get på Map
headOption på Vector
find på Seq

Implementera `equals` med `match`

Undantag

Vad är ett undantag (eng. *exception*)?

Undantag representerar ett fel eller ett onormalt tillstånd som upptäcks under exekvering och som behöver hanteras på särskilt sätt vid sidan av det normala exekveringsflödet.

sv.wikipedia.org/wiki/Undantagshantering

Exempel på undantag:

Vad är ett undantag (eng. *exception*)?

Undantag representerar ett fel eller ett onormalt tillstånd som upptäcks under exekvering och som behöver hanteras på särskilt sätt vid sidan av det normala exekveringsflödet.

sv.wikipedia.org/wiki/Undantagshantering

Exempel på undantag:

- Indexering utanför vektorns indexgränser.
- Läsning bortom filens slut.
- Försök att öppna en fil som inte finns.
- Minnet är slut.
- Division med noll.
- `"hej".toInt` resulterar i `java.lang.NumberFormatException`

Fånga undantag med `try-catch`-uttryck

scala.util.Try

En gemensam bastyp för något som kan misslyckas