EDAA45 Programmering, grundkurs

Läsvecka 7: Arv

Björn Regnell

Datavetenskap, LTH

Lp1-2, HT 2016

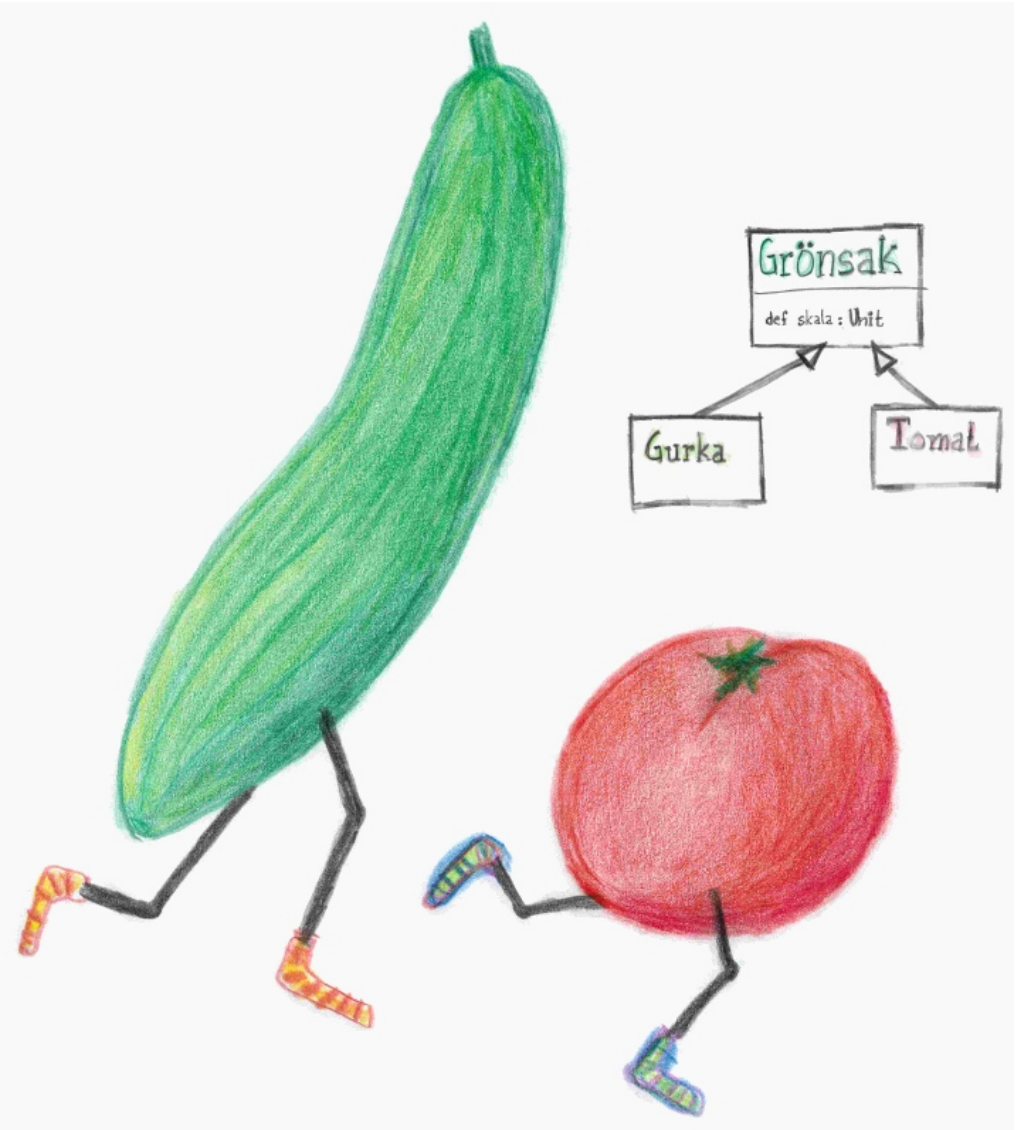
7 Arv

- Arv och nyckelordet `extends`
- Överskugging
- Trait eller abstrakt klass?
- `super`

Arv och nyckelordet `extends`

Vad är arv?

Med arv kan man beskriva relationen
X **är en** *Y*



Varför behövs arv?

- Man kan använda arv för att dela upp kod i:
  - **generella** (gemensamma) delar och
  - **specifika** (specialanpassade) delar.
- Man kan åstadkomma **kontrollerad flexibilitet**:
  - Klientkod kan **utvidga** (eng. *extend*) ett givet API med egna specifika tillägg.
- Man kan använda arv för att deklarera en gemensam **bastyp** så att generiska samlingar kan ges en mer specifik elementtyp.
  - Det räcker att man vet bastypen för att kunna anropa gemensamma metoder på alla element i samlingen.

Behovet av gemensam bastyp

```
scala> class Gurka(val vikt: Int)

scala> class Tomat(val vikt: Int)

scala> val gurkor = Vector(new Gurka(200), new Gurka(300))
gurkor: scala.collection.immutable.Vector[Gurka] =
  Vector(Gurka@60856961, Gurka@2fd953a6)

scala> gurkor.map(_.vikt)
res0: scala.collection.immutable.Vector[Int] = Vector(200, 300)

scala> val grönsaker = Vector(new Gurka(200), new Tomat(42))
grönsaker: scala.collection.immutable.Vector[Object] =
  Vector(Gurka@669253b7, Tomat@5305c37d)

scala> grönsaker.map(_.vikt)
<console>:15: error: value vikt is not a member of Object
       grönsaker.map(_.vikt)
```

Hur ordna en mer specifik typ än `Vector[Object]`?

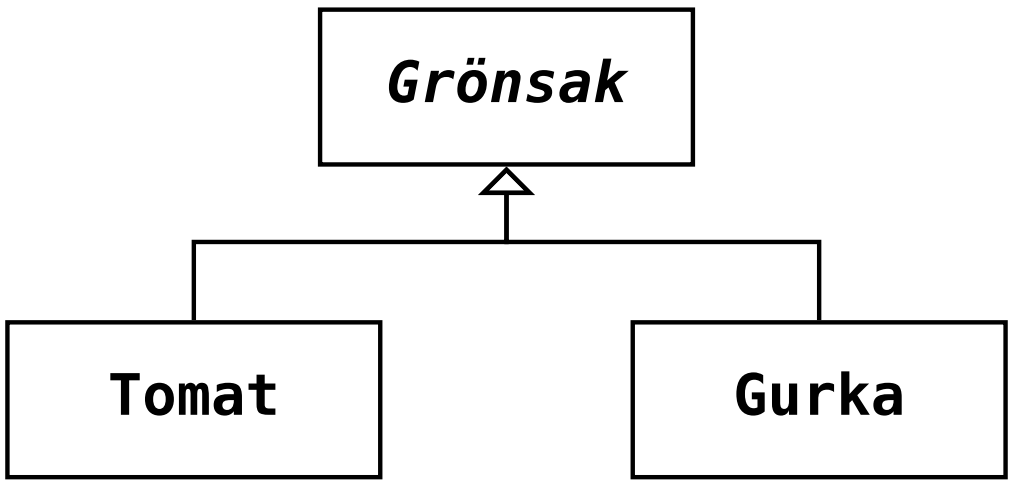
Behovet av gemensam bastyp

```
scala> class Gurka(val vikt: Int)

scala> class Tomat(val vikt: Int)

scala> val gurkor = Vector(new Gurka(200), new Gurka(300))
gurkor: scala.collection.immutable.Vector[Gurka] =
  Vector(Gurka@60856961, Gurka@2fd953a6)

scala> gurkor.map(_.vikt)
res0: scala.collection.immutable.Vector[Int] = Vector(200, 300)

scala> val grönsaker = Vector(new Gurka(200), new Tomat(42))
grönsaker: scala.collection.immutable.Vector[Object] =
  Vector(Gurka@669253b7, Tomat@5305c37d)

scala> grönsaker.map(_.vikt)
<console>:15: error: value vikt is not a member of Object
       grönsaker.map(_.vikt)
```

Hur ordna en mer specifik typ än `Vector[Object]`? → Skapa en **bastyp**!

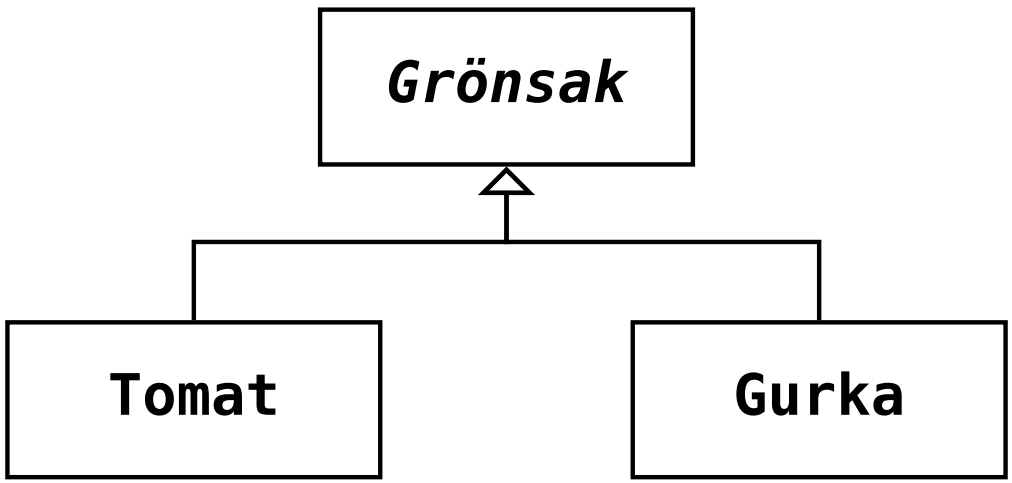
Skapa en gemensam bastyp

Typen ***`Grönsak`*** är en **bastyp** i nedan arvshierarki:



Skapa en gemensam bastyp

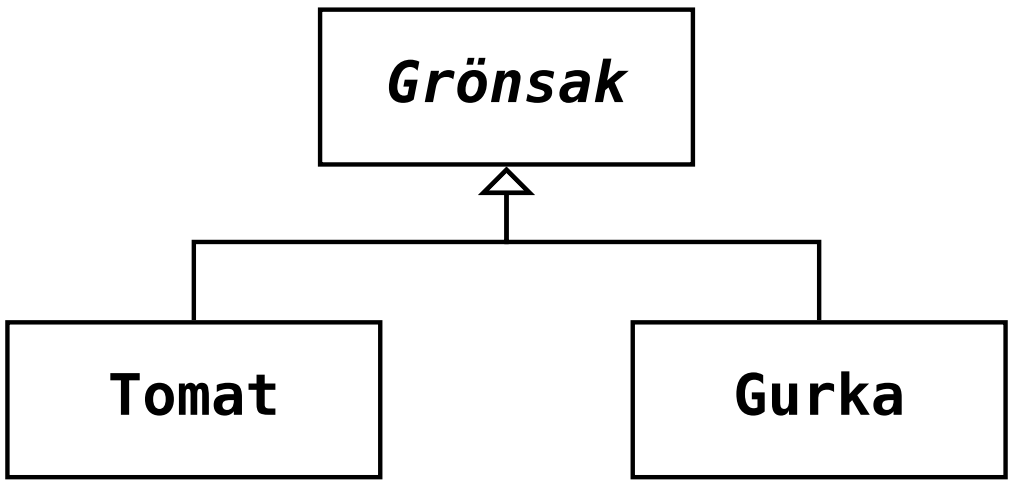
Typen ***Grönsak*** är en **bastyp** i nedan arvshierarki:



Pilen ↑ betecknar **arv** och utläses ”**är en**”

Skapa en gemensam bastyp

Typen ***Grönsak*** är en **bastyp** i nedan arvshierarki:



Pilen ⇧ betecknar **arv** och utläses ”**är en**”

Typerna `Tomat` och `Gurka` är **subtyper** till den **abstrakta** typen `Grönsak`.

Skapa en gemensam bastyp med `trait` och `extends`

Med **`trait`** `Grönsak` kan klasserna `Gurka` och `Tomat` få en gemensam **bastyp** genom att båda **subtyperna** gör **`extends`** `Grönsak`:

```
1  scala> trait Grönsak
2
3  scala> class Gurka(val vikt: Int) extends Grönsak
4
5  scala> class Tomat(val vikt: Int) extends Grönsak
6
7  scala> val grönsaker = Vector(new Gurka(200), new Tomat(42))
8  grönsaker: scala.collection.immutable.Vector[Grönsak] =
9    Vector(Gurka@3dc4ed6f, Tomat@2823b7c5)
```

Skapa en gemensam bastyp med `trait` och `extends`

Med **`trait`** `Grönsak` kan klasserna `Gurka` och `Tomat` få en gemensam **bastyp** genom att båda **subtyperna** gör **`extends`** `Grönsak`:

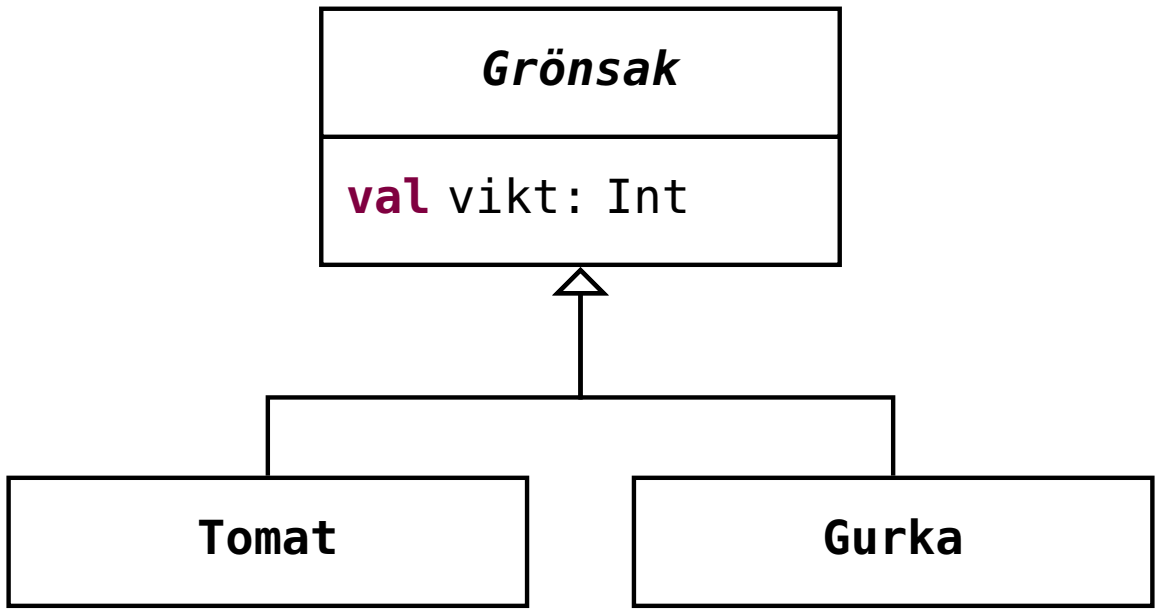
```
1  scala> trait Grönsak
2
3  scala> class Gurka(val vikt: Int) extends Grönsak
4
5  scala> class Tomat(val vikt: Int) extends Grönsak
6
7  scala> val grönsaker = Vector(new Gurka(200), new Tomat(42))
8  grönsaker: scala.collection.immutable.Vector[Grönsak] =
9    Vector(Gurka@3dc4ed6f, Tomat@2823b7c5)
```

Men det är fortfarande inte som vi vill ha det:

```
scala> grönsaker.map(_.vikt)
<console>:15: error: value vikt is not a member of Grönsak
       grönsaker.map(_.vikt)
```

En gemensam bastyp med gemensamma delar

Placera gemensamma medlemmar i bastypen:



- Alla grönsaker har attributet `val vikt`.
- Det specifika värdet på vikten definieras **inte** i bastypen.
- Medlemen `vikt` kallas **abstrakt** eftersom den **saknar implementation**.

Placera gemensamma delar i bastypen

Vi inkluderar det gemensamma attributet **`val vikt`** som en **abstrakt medlem** i bastypen:

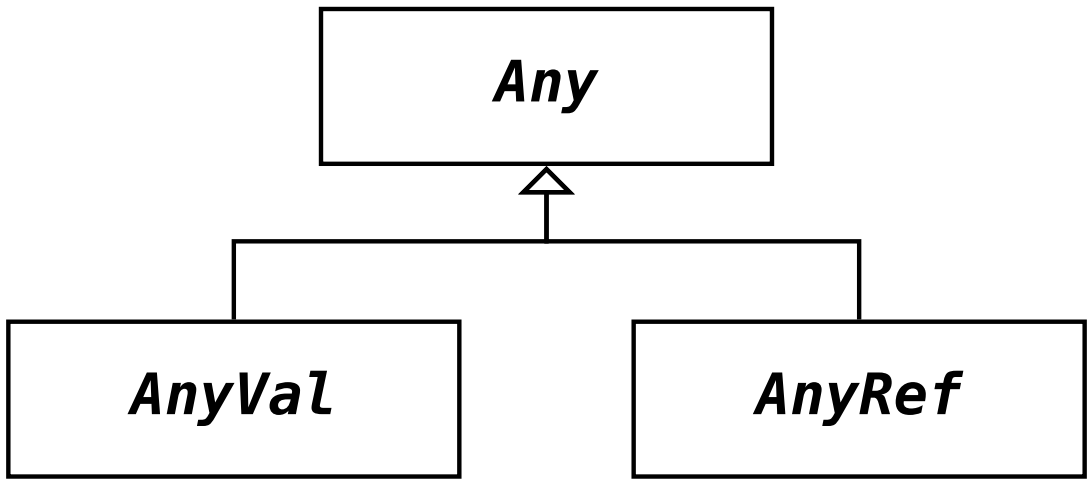
```
trait Grönsak { val vikt: Int }
class Gurka(val vikt: Int) extends Grönsak
class Tomat(val vikt: Int) extends Grönsak
```

Nu vet kompilatorn att alla grönsaker har en vikt:

```
1  scala> val grönsaker = Vector(new Gurka(200), new Tomat(42))
2  grönsaker: scala.collection.immutable.Vector[Grönsak] =
3    Vector(Gurka@3dc4ed6f, Tomat@2823b7c5)
4
5  scala> grönsaker.map(_.vikt)
6  res0: scala.collection.immutable.Vector[Int] = Vector(200, 42)
```

Scalas typhierarki och typen `Object`

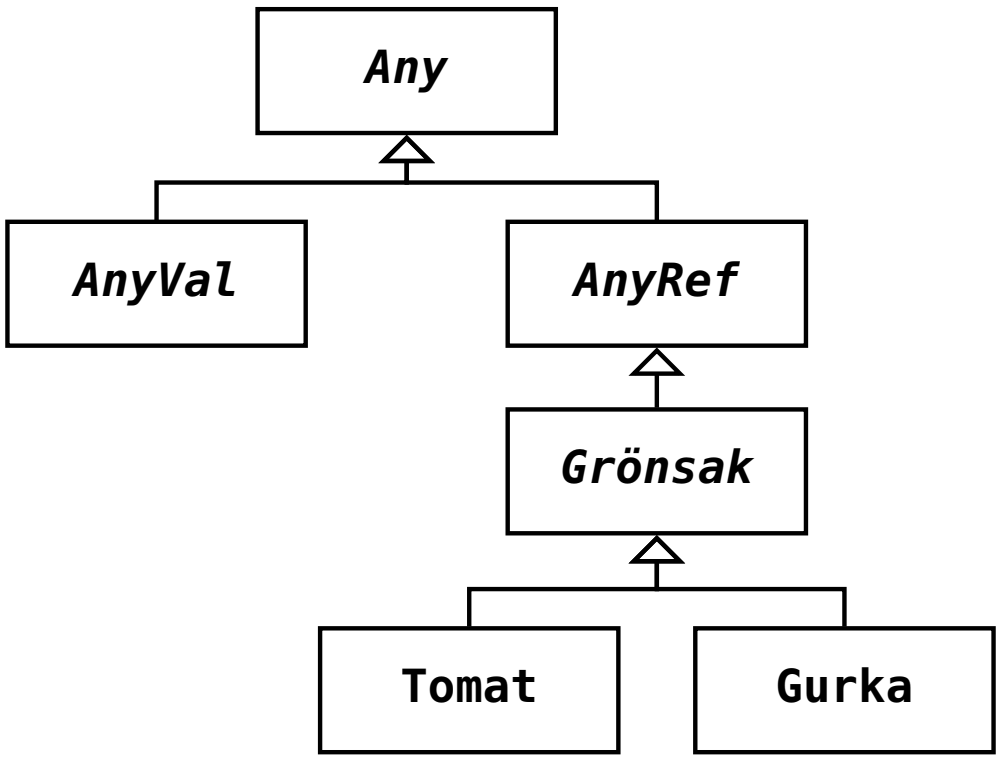
Den översta delen av typhierarkin i Scala:



- De numeriska typerna `Int`, `Double`, etc är subtyper till **`AnyVal`** och kallas **värdetyper** och lagras på ett speciellt, effektivt sätt i minnet.
- Alla dina egna klasser är subtyper till **`AnyRef`** och kallas **referenstyper** och kräver (direkt eller indirekt) konstruktion med **`new`**.
- `AnyRef` motsvaras av **`java.lang.Object`** i JVM.

Implicita supertyper till dina egna klasser

Alla dina egna typer ingår underförstått i Scalas typhierarki:

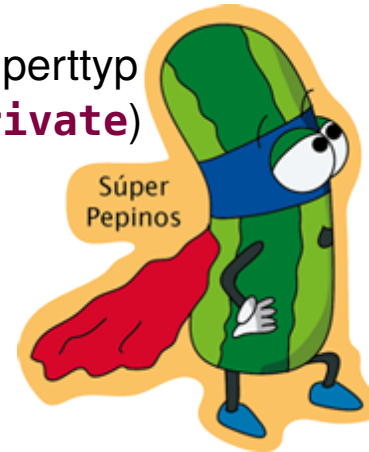


Terminologi

subtyp	en typ som ärver en supertyp
supertyp	en typ som ärvs av en subtyp
bastyp	en typ som är rot i ett arvsträd
abstrakt medlem	en medlem som saknar implementation
konkret medlem	en medlem som ej saknar implementation
abstrakt typ	en typ kan ha abstrakta medlemmar; kan ej instansieras
konkret typ	en typ som ej har abstrakta medlemmar; kan instansieras
`class`	en klass är en konkret typ som **ej kan ha abstrakta medlemmar**
`abstract class`	en klass är en abstrakt typ som **kan ha parametrar**
`trait`	är en abstrakt typ som **ej kan ha parametrar** men **kan mixas in**
`extends`	står före en supertyp, indikerar arv
`override`	en medlem överskuggar (byter ut) en medlem i en superttyp
`protected`	gör en medlem synlig i subtyper till denna typ (jmf `private`)
`final gurka`	gör medlemen gurka final (förhindrar överskuggning)
`final class`	deklarerar en final klass, förhindrar arv
`super.gurka`	refererar till supertypens medlem `gurka` (jfm `this`)

Terminologi

subtyp	en typ som ärver en supertyp
supertyp	en typ som ärvs av en subtyp
bastyp	en typ som är rot i ett arvsträd
abstrakt medlem	en medlem som saknar implementation
konkret medlem	en medlem som ej saknar implementation
abstrakt typ	en typ kan ha abstrakta medlemmar; kan ej instansieras
konkret typ	en typ som ej har abstrakta medlemmar; kan instansieras
`class`	en klass är en konkret typ som **ej kan ha abstrakta medlemmar**
`abstract class`	en klass är en abstrakt typ som **kan ha parametrar**
`trait`	är en abstrakt typ som **ej kan ha parametrar** men **kan mixas in**
`extends`	står före en supertyp, indikerar arv
`override`	en medlem överskuggar (byter ut) en medlem i en superttyp
`protected`	gör en medlem synlig i subtyper till denna typ (jmf `private`)
`final gurka`	gör medlemen gurka final (förhindrar överskuggning)
`final class`	deklarerar en final klass, förhindrar arv
`super.gurka`	refererar till supertypens medlem `gurka` (jfm `this`)

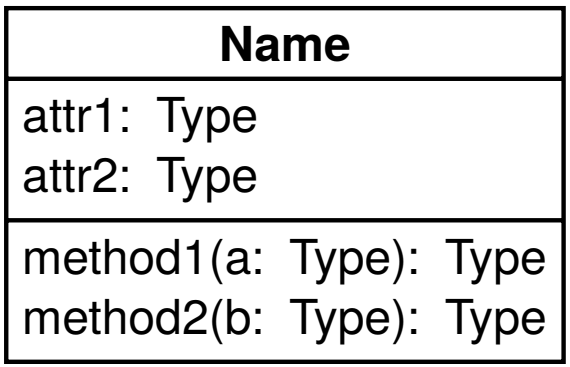


Abstrakta och konkreta medlemmar

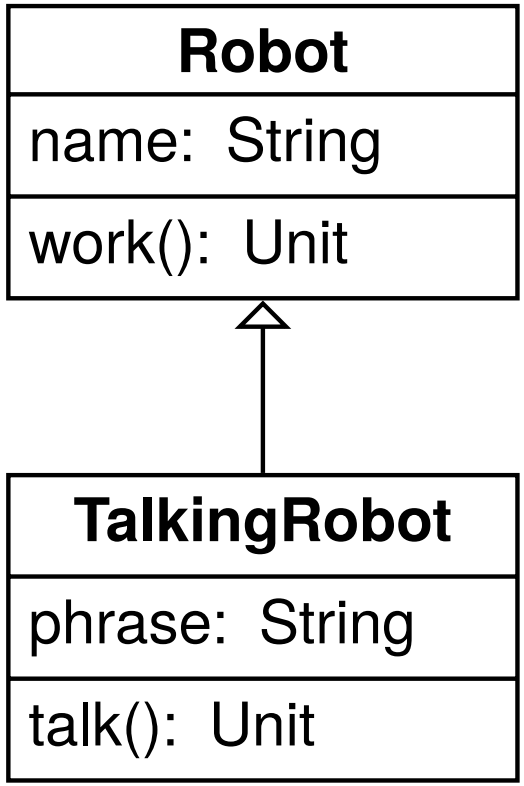
```
1  trait Grönsak {
2    def skala(): Unit
3    var vikt: Double
4    val namn: String
5    var ärSkalad: Boolean = false
6    override def toString = s"$namn ${if (!ärSkalad) "o" else ""}skalad $vikt g"
7  }
8
9  class Gurka(var vikt: Double) extends Grönsak {
10   val namn = "gurka"
11   def skala(): Unit = if (!ärSkalad) {
12     println("Gurkan skalas med skalare.")
13     vikt = 0.98 * vikt
14     ärSkalad = true
15   }
16 }
17
18 class Tomat(var vikt: Double) extends Grönsak {
19   val namn = "tomat"
20   def skala(): Unit = if (!ärSkalad) {
21     println("Tomaten skalas genom skållning.")
22     vikt = 0.99 * vikt
23     ärSkalad = true
24   }
25 }
```

Attribut och metoder i UML-diagram

En klass i ett **UML**-diagram kan ha 3 delar:



Ibland utelämnar man typerna.



en.wikipedia.org/wiki/Class_diagram

Överskugging

Medlemmar, arv och överskuggning

Olika sorters överskuggningsbara medlemmar i klasser och traits i **Scala**:

- `def`
- `val`
- `lazy val`
- `var`

Medlemmar, arv och överskuggning

Olika sorters överskuggningsbara medlemmar i klasser och traits i **Scala**:

- `def`
- `val`
- `lazy val`
- `var`

Olika sorters överskuggningsbara instansmedlemmar i **Java**:

- variabel
- metod

Medlemmar som är `static` kan ej överskuggas (men döljas) vid arv.

Medlemmar, arv och överskuggning

Olika sorters överskuggningsbara medlemmar i klasser och traits i **Scala**:

- `def`
- `val`
- `lazy val`
- `var`

Olika sorters överskuggningsbara instansmedlemmar i **Java**:

- variabel
- metod

Medlemmar som är `static` kan ej överskuggas (men döljas) vid arv.

- När man överskuggar (eng. *override*) en medlemmen med en annan medlem med samma namn i en subtyp, får denna medlem en (ny) implementation.
- När man konstruerar ett objektorienterat språk gäller det att man definierar sunda överskuggningsregler vid arv. Detta är förvånansvärt knepigt.
- Singelobjekt kan ej ärvas (och medlemmar i singelobjekt kan därmed ej överskuggas).

Fördjupning: Regler för överskuggning i Scala

En medlem M1 i en supertyp får överskuggas av en medlem M2 i en subtyp, enligt dessa regler:

1. M1 och M2 ska ha samma namn och typerna ska matcha.
2. `def` får bytas ut mot: `def`, `val`, `var`, `lazy val`
3. `val` får bytas ut mot: `val`, och om M1 är abstrakt mot en `lazy val`.
4. `var` får bara bytas ut mot en `var`.
5. `lazy val` får bara bytas ut mot en `lazy val`.
6. Om en medlem i en supertyp är abstrakt *behöver* man inte använda nyckelordet `override` i subtypen. (Men det är bra att göra det ändå så att kompilatorn hjälper dig att kolla att du verkligen överskuggar något.)
7. Om en medlem i en supertyp är konkret *måste* man använda nyckelordet `override` i subtypen, annars ges kompileringsfel.
8. M1 får inte vara `final`.
9. M1 får inte vara `private` eller `private[this]`, men kan vara `private[X]` om M2 också är `private[X]`, eller `private[Y]` om X innehåller Y.
10. Om M1 är `protected` måste även M2 vara det.

Fördjupning: Regler för överskuggning i Java

```
http://docs.oracle.com/javase/tutorial/java/
IandI/override.html
```

Trait eller abstrakt klass?

Trait eller abstrakt klass?

Använd en **trait** som supertyp om...

- ...du är osäker på vilket som är bäst. (Du kan alltid ändra till en abstrakt klass senare.)
- ...du vill kunna mixa in din trait tillsammans med andra traits.
- ...du bara har abstrakta medlemmar.

Använd en **abstract class** som supertyp om...

- ...du vill ge supertypen en parameter vid konstruktion.
- ...du vill ärva supertypen från klasser skrivna i Java.
- ...du vill minimera vad som behöver omkompileras vid ändringar.

super

super

```
1  scala> class X { def gurka = "super pepinos" }
2
3  scala> class Y extends X { override def gurka = ":("; def sup = super.gurka }
4
5  scala> val y = new Y
6  y: Y = Y@26ba2a48
7
8  scala> y.gurka
9  res0: String = :(
```

super

```
scala> class X { def gurka = "super pepinos" }

scala> class Y extends X { override def gurka = ":("; def sup = super.gurka }

scala> val y = new Y
y: Y = Y@26ba2a48

scala> y.gurka
res0: String = :(
```

```
scala> y.sup
res1: String = super pepinos
```

super

```
1  scala> class X { def gurka = "super pepinos" }
2
3  scala> class Y extends X { override def gurka = ":("; def sup = super.gurka }
4
5  scala> val y = new Y
6  y: Y = Y@26ba2a48
7
8  scala> y.gurka
9  res0: String = :(
```

```
scala> y.sup
res1: String = super pepinos
```