EDAA45 Programmering, grundkurs

Läsvecka 7: Arv

Björn Regnell

Datavetenskap, LTH

Lp1-2, HT 2016

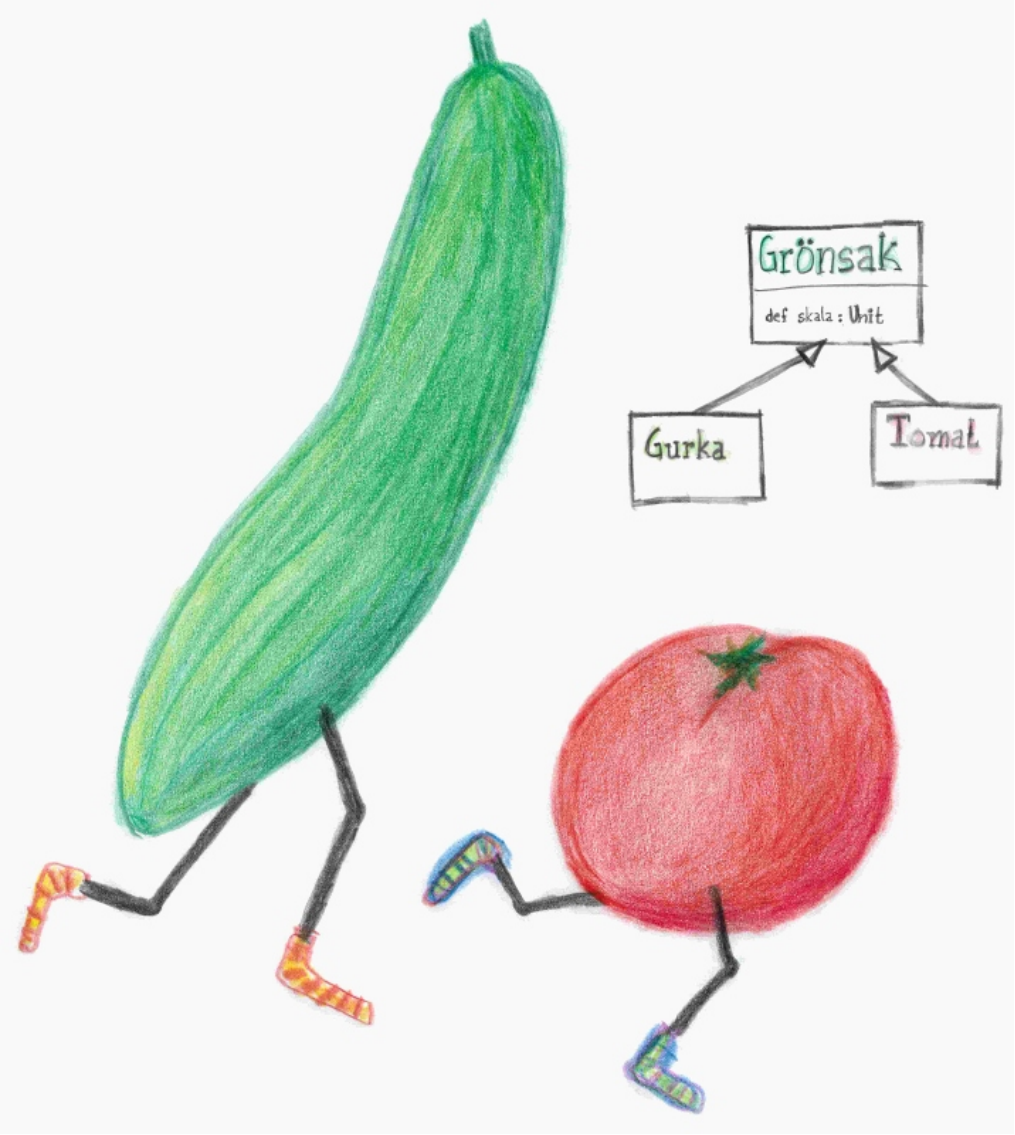
7 Arv

- Vad är arv?
- Överskuggingsregler
- `super`
- Trait eller abstrakt klass?

Vad är arv?

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

```
1  scala> class Gurka(val vikt: Int)
2
3  scala> class Tomat(val vikt: Int)
4
5  scala> val gurkor = Vector(new Gurka(200), new Gurka(300))
6  gurkor: scala.collection.immutable.Vector[Gurka] =
7    Vector(Gurka@60856961, Gurka@2fd953a6)
8
9  scala> gurkor.map(_.vikt)
10 res0: scala.collection.immutable.Vector[Int] = Vector(200, 300)
11
12 scala> val grönsaker = Vector(new Gurka(200), new Tomat(42))
13 grönsaker: scala.collection.immutable.Vector[Object] =
14   Vector(Gurka@669253b7, Tomat@5305c37d)
15
16 scala> grönsaker.map(_.vikt)
17 <console>:15: error: value vikt is not a member of Object
18        grönsaker.map(_.vikt)
```

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

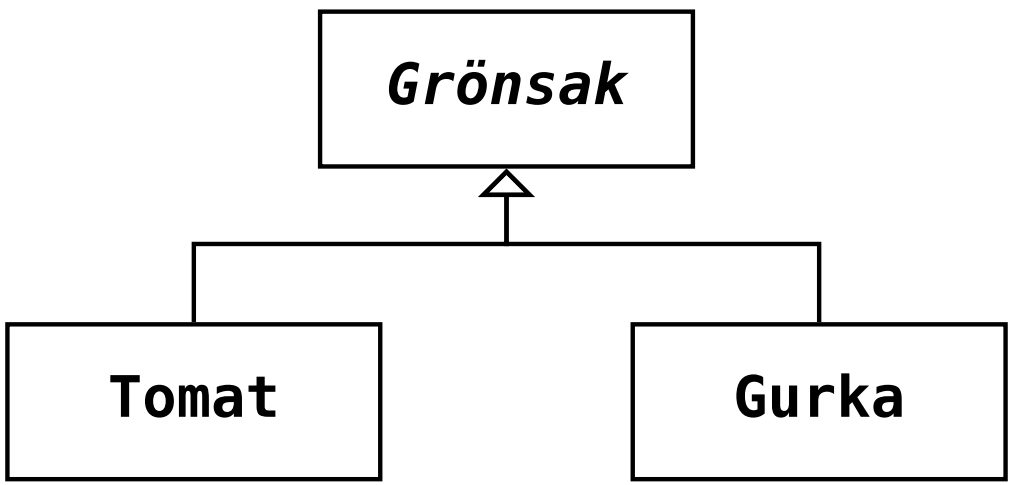
Behovet av gemensam bastyp

```
1  scala> class Gurka(val vikt: Int)
2
3  scala> class Tomat(val vikt: Int)
4
5  scala> val gurkor = Vector(new Gurka(200), new Gurka(300))
6  gurkor: scala.collection.immutable.Vector[Gurka] =
7    Vector(Gurka@60856961, Gurka@2fd953a6)
8
9  scala> gurkor.map(_.vikt)
10 res0: scala.collection.immutable.Vector[Int] = Vector(200, 300)
11
12 scala> val grönsaker = Vector(new Gurka(200), new Tomat(42))
13 grönsaker: scala.collection.immutable.Vector[Object] =
14   Vector(Gurka@669253b7, Tomat@5305c37d)
15
16 scala> grönsaker.map(_.vikt)
17 <console>:15: error: value vikt is not a member of Object
18        grönsaker.map(_.vikt)
```

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

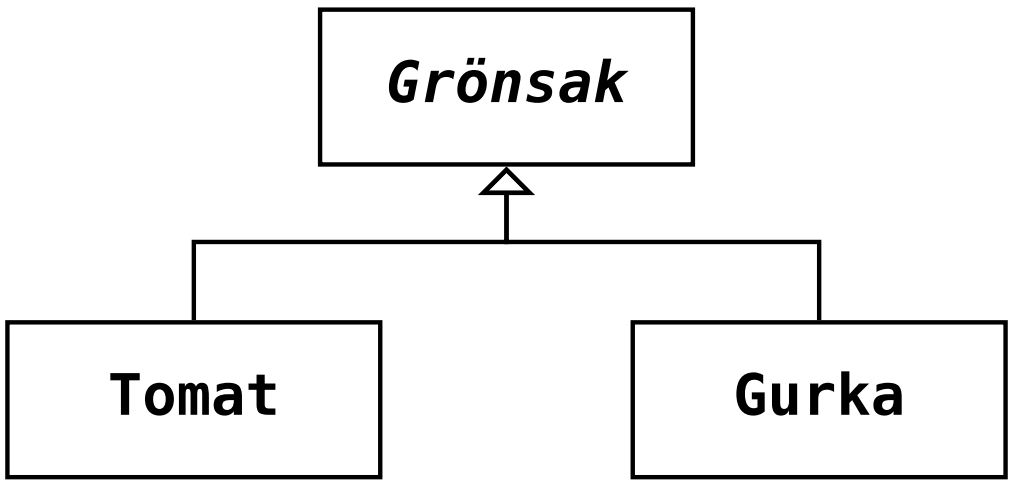
Skapa en gemensam bastyp

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



Skapa en gemensam bastyp

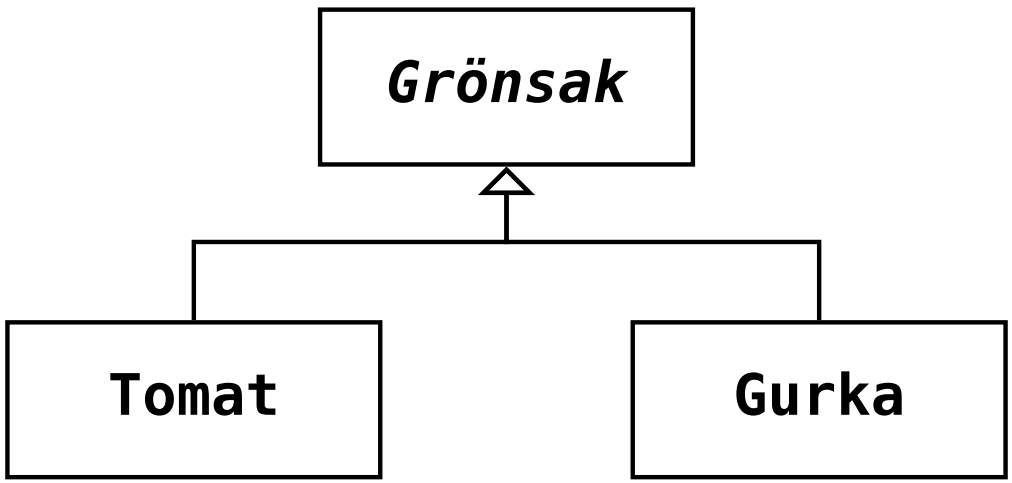
Typen ***Grönsak*** är en **bastyp** i nedan arvshierarki:



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`:

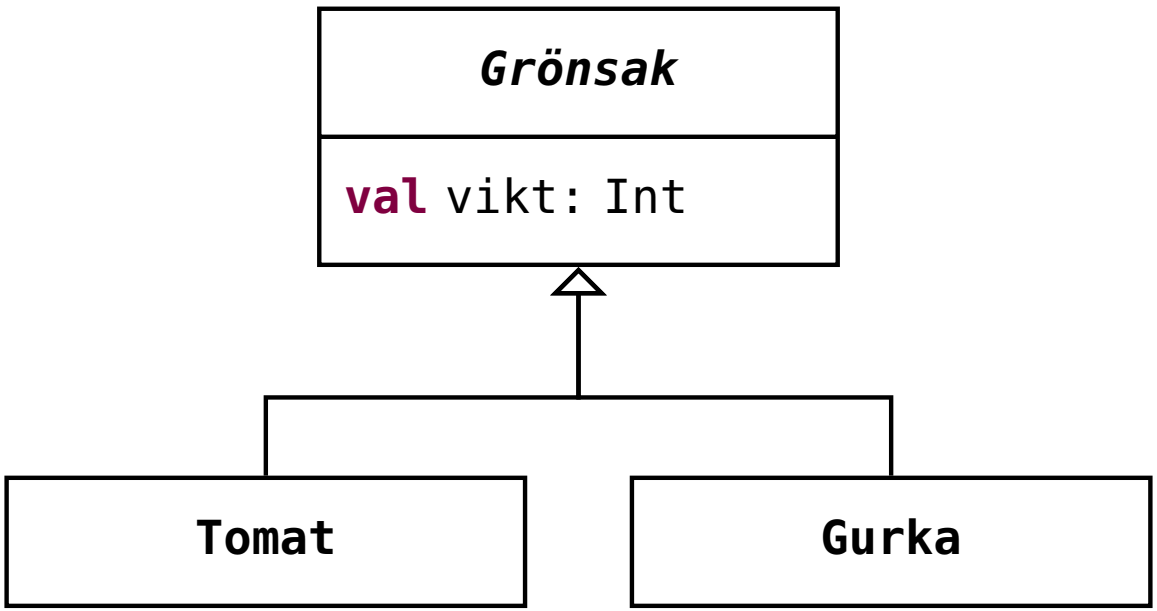
```
1  scala> trait Grönsak
2
3  scala> class Gurka(val vikt: Int) extends Grönsak
4
5  scala> class Tomat(val vikt: Int) extends Grönsak
6
7  scala> val grönsaker = Vector(new Gurka(200), new Tomat(42))
8  grönsaker: scala.collection.immutable.Vector[Grönsak] =
9    Vector(Gurka@3dc4ed6f, Tomat@2823b7c5)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

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:

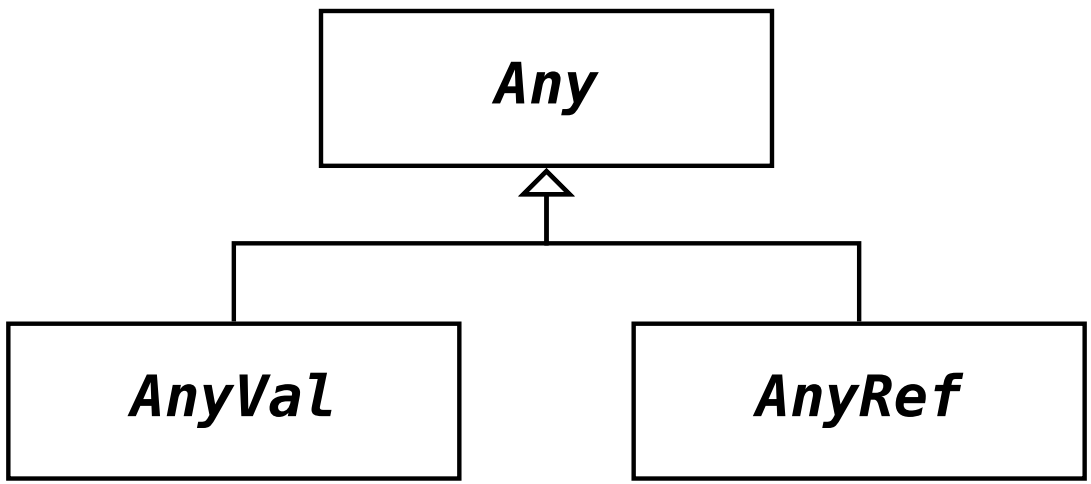
```
trait Grönsak { val vikt: Int }

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`

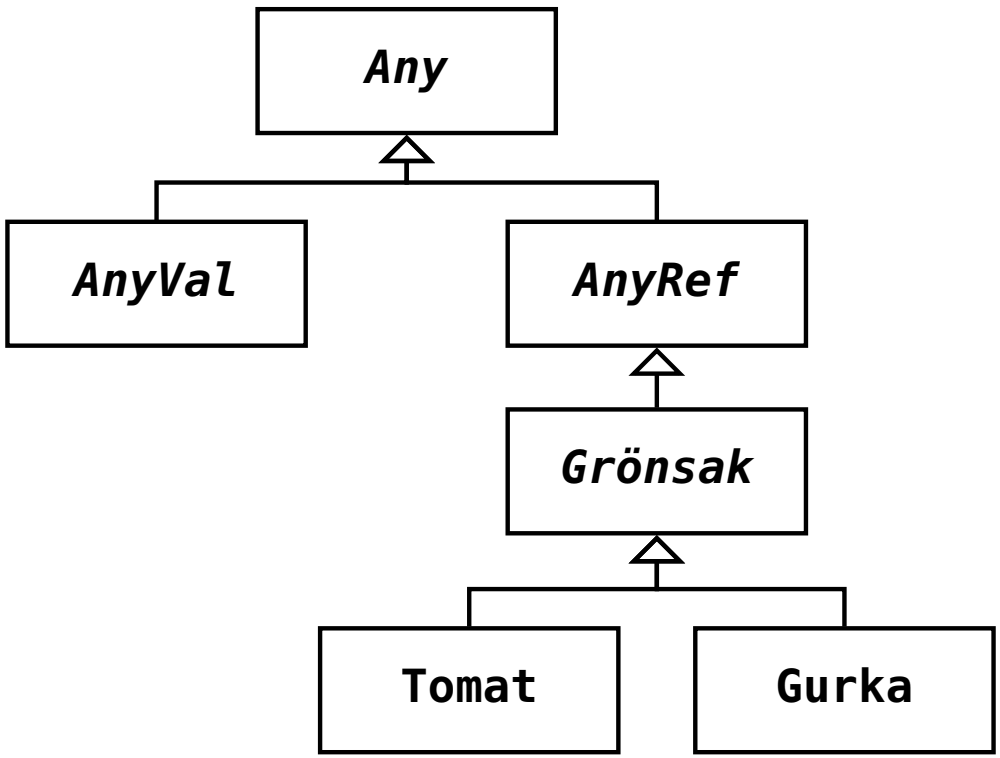
Den översta delen av typhierarkin i Scala:



- 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:



Vad är en trait?

- Trait betyder **egenskap** på engelska.
- En trait liknar en klass, **men** speciella regler gäller:
  - den **kan** innehålla delar som **saknar implementation**
  - den **kan mixas** med flera andra traits så att olika koddelar kan återanvändas på flexibla sätt.
  - den **kan inte** instansieras direkt som den är; den måste återanvändas genom arv.
  - den **kan inte** ha klassparametrar eller konstruktorer

Vad används en trait till?

En **`trait`** används för att skapa en bastyp som kan vara hemvist för gemensamma delar hos subtyper:

```
trait Bastyp { val x = 42 }                    // Bastyp har medlemmen x
class Subtyp1 extends Bastyp { val y = 43 } // Subtyp1 ärver x, har även y
class Subtyp2 extends Bastyp { val z = 44 } // Subtyp2 ärver x, har även z
```

Vad används en trait till?

En **`trait`** används för att skapa en bastyp som kan vara hemvist för gemensamma delar hos subtyper:

```
trait Bastyp { val x = 42 }                    // Bastyp har medlemmen x
class Subtyp1 extends Bastyp { val y = 43 } // Subtyp1 ärver x, har även y
class Subtyp2 extends Bastyp { val z = 44 } // Subtyp2 ärver x, har även z
```

```
1  scala> val a = new Subtyp1
2  a: Subtyp1 = Subtyp1@51016012
3
4  scala> a.x
5  res0: Int = 42
6
7  scala> a.y
8  res1: Int = 43
9
10 scala> a.z
11 <console>:15: error: value z is not a member of Subtyp1
12
13 scala> new Bastyp
14 <console>:13: error: trait Bastyp is abstract; cannot be instantiated
```

En trait kan ha abstrakta medlemmar

```
trait X { val x: Int }   // x är abstrakt, d.v.s. saknar implementation
class A extends X { val x = 42 }   // x ges en implementation
class B extends X { val x = 43 }   // x ges en annan implementation
```

En trait kan ha abstrakta medlemmar

```
trait X { val x: Int }   // x är abstrakt, d.v.s. saknar implementation
class A extends X { val x = 42 }   // x ges en implementation
class B extends X { val x = 43 }   // x ges en annan implementation
```

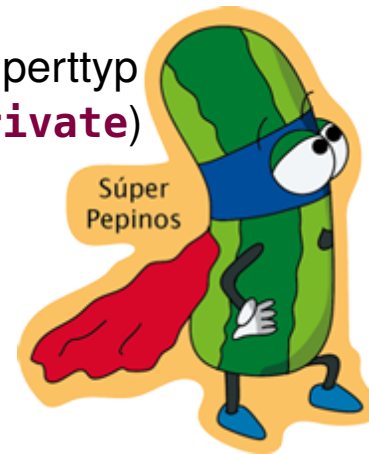
```
scala> val a = new A
a: A = A@5faeada1

scala> val b = new B
b: B = B@cb51256

scala> val xs = Vector(a,b)
xs: scala.collection.immutable.Vector[X] = Vector(A@5faeada1, B@cb51256)

scala> xs.map(_.x)
res0: scala.collection.immutable.Vector[Int] = Vector(42, 43)

scala> class Y { val y: Int }
  error: class Y needs to be abstract, since value y is not defined

scala> trait Z(x: Int)
  error: traits or objects may not have parameters
```

Terminologi och nyckelord

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`	gör klassen final: förhindrar vidare subtypning
`super.gurka`	refererar till supertypens medlem `gurka` (jmf `this`)

Terminologi och nyckelord

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`	gör klassen final: förhindrar vidare subtypning
`super.gurka`	refererar till supertypens medlem `gurka` (jmf `this`)



Abstrakta och konkreta medlemmar

```
1  object exempelVego1 {
2
3    trait Grönsak {
4      def skala(): Unit
5      var vikt: Double
6      val namn: String
7      var ärSkalad: Boolean = false
8      override def toString = s"$namn ${if (!ärSkalad) "o" else ""}skalad $vikt g"
9    }
10
11   class Gurka(var vikt: Double) extends Grönsak {
12     val namn = "gurka"
13     def skala(): Unit = if (!ärSkalad) {
14       println("Gurkan skalas med skalare.")
15       vikt = 0.99 * vikt
16       ärSkalad = true
17     }
18   }
19
20   class Tomat(var vikt: Double) extends Grönsak {
21     val namn = "tomat"
22     def skala(): Unit = if (!ärSkalad) {
23       println("Tomaten skalas genom skållning.")
24       vikt = 0.99 * vikt
25       ärSkalad = true
26     }
27   }
28 }
```

Undvika kodduplicering med hjälp av arv

```
1  object exempelVego2 {
2
3    trait Grönsak {  // innehåller alla gemensamma delar; hjälper oss undvika upprepning
4      val skalningsmetod: String
5      val skalfaktor = 0.99
6      def skala(): Unit = if (!ärSkalad) {
7        println(skalningsmetod)
8        vikt = skalfaktor * vikt
9        ärSkalad = true
10     }
11     var vikt: Double
12     val namn: String
13     var ärSkalad: Boolean = false
14     override def toString = s"$namn ${if (!ärSkalad) "o" else ""}skalad $vikt g"
15   }
16
17   class Gurka(var vikt: Double) extends Grönsak { // bara det som är speciellt för gurkor
18     val namn = "gurka"
19     val skalningsmetod = "Gurkan skalas med skalare."
20   }
21
22   class Tomat(var vikt: Double) extends Grönsak { // bara det som är speciellt för tomater
23     val namn = "tomat"
24     val skalningsmetod = "Tomaten skalas genom skållning."
25   }
26 }
```

Överskuggning

```
1  object exempelVego3 {
2
3    trait Grönsak {
4      val skalningsmetod: String
5      val skalfaktor = 0.99
6      def skala(): Unit = if (!ärSkalad) {
7        println(skalningsmetod)
8        vikt = skalfaktor * vikt
9        ärSkalad = true
10     }
11     var vikt: Double
12     val namn: String
13     var ärSkalad: Boolean = false
14     override def toString = s"$namn ${if (!ärSkalad) "o" else ""}skalad $vikt g"
15   }
16
17   class Gurka(var vikt: Double) extends Grönsak {
18     val namn = "gurka"
19     val skalningsmetod = "Gurkan skalas med skalare."
20   }
21
22   class Tomat(var vikt: Double) extends Grönsak {
23     val namn = "tomat"
24     val skalningsmetod = "Tomaten skalas genom skållning."
25     override val skalfaktor = 0.95                    // överskuggning
26   }
27 }
```

Varför kan kodduplicering orsaka problem?

- Mer att skriva (inte jättestort problem)

Varför kan kodduplicering orsaka problem?

- Mer att skriva (inte jättestort problem)
- Fler kodrader att läsa och förstå

Varför kan kodduplicering orsaka problem?

- Mer att skriva (inte jättestort problem)
- Fler kodrader att läsa och förstå
- Fler kodrader som påverkas vid tillägg

Varför kan kodduplicering orsaka problem?

- Mer att skriva (inte jättestort problem)
- Fler kodrader att läsa och förstå
- Fler kodrader som påverkas vid tillägg
- Fler kodrader att underhålla:
  - Om man rättar en bug på ett ställe måste man komma ihåg att göra **exakt samma ändring** på alla de ställen där kodduplicering förekommer $\rightarrow$ **risk för nya buggar**

Varför kan kodduplicering orsaka problem?

- Mer att skriva (inte jättestort problem)
- Fler kodrader att läsa och förstå
- Fler kodrader som påverkas vid tillägg
- Fler kodrader att underhålla:
  - Om man rättar en bug på ett ställe måste man komma ihåg att göra **exakt samma ändring** på alla de ställen där kodduplicering förekommer → **risk för nya buggar**
- Principen på engelska:
  DRY == `"Don't Repeat Yourself!"`

Varför kan kodduplicering orsaka problem?

- Mer att skriva (inte jättestort problem)
- Fler kodrader att läsa och förstå
- Fler kodrader som påverkas vid tillägg
- Fler kodrader att underhålla:
  - Om man rättar en bug på ett ställe måste man komma ihåg att göra **exakt samma ändring** på alla de ställen där kodduplicering förekommer → **risk för nya buggar**
- Principen på engelska:
  `DRY == "Don't Repeat Yourself!"`
- Men det kan finnas tillfällen när kodduplicering faktiskt är att föredra: t.ex. om man vill att olika delar av koden ska vara helt oberoende av varandra.

Filnamnsregler och -konventioner

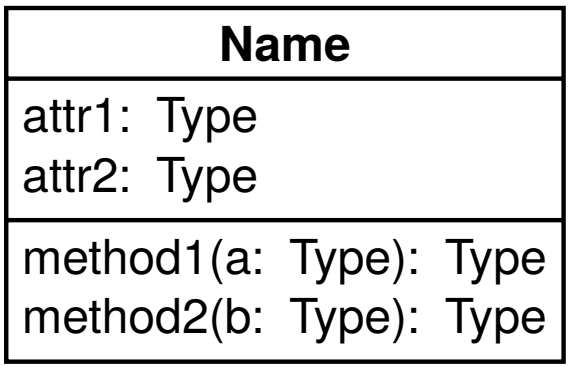
- Java
  - I Java får man bara ha **en enda** publik klass per kodfil.
  - I Java måste kodfilen ha **samma namn** som den publika klassen, t.ex. `KlassensNamn.java`
- Scala
  - I Scala får man ha **många** klasser/traits/singelobjekt i samma kodfil.
  - I Scala får man döpa kodfilerna **oberoende** av deras innehåll.

Filnamnsregler och -konventioner

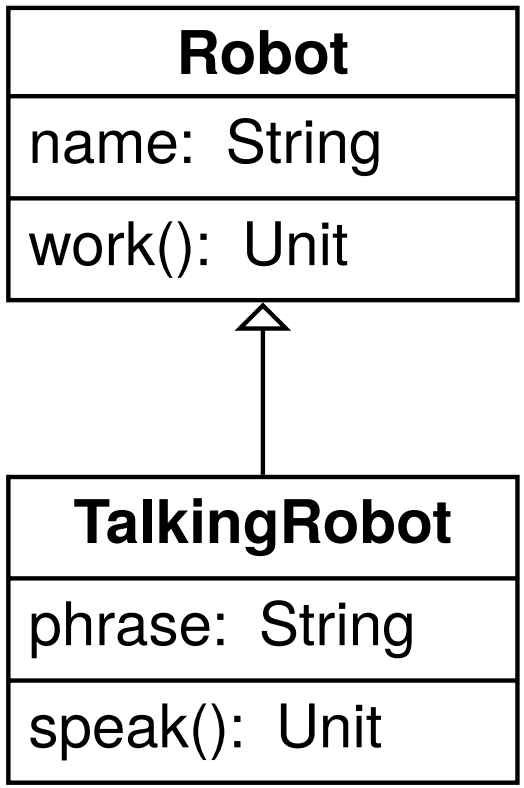
- Java
  - I Java får man bara ha **en enda** publik klass per kodfil.
  - I Java måste kodfilen ha **samma namn** som den publika klassen, t.ex. `KlassensNamn.java`
- Scala
  - I Scala får man ha **många** klasser/traits/singelobjekt i samma kodfil.
  - I Scala får man döpa kodfilerna **oberoende** av deras innehåll. Dessa **konventioner** används:
    - Om en kodfil bara innehåller **en enda** klass/trait/singelobjekt ge filen samma namn som innehållet, t.ex. `KlassensNamn.scala`
    - Om en kodfil innehåller **flera** saker, döp filen till något som återspeglar hela innehållet och använd **liten begynnelsebokstav**, t.ex. `bastypensNamn.scala`

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

Överskuggingsregler

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
```

super

Att skilja på mitt och ditt med `super`

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

scala> class Y extends X {
         override val gurka = ":("
         val sg = super.gurka
       }

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

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

Super Pepinos to the rescue:

Att skilja på mitt och ditt med `super`

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

scala> class Y extends X {
         override val gurka = ":("
         val sg = super.gurka
       }

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

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

Super Pepinos to the rescue:

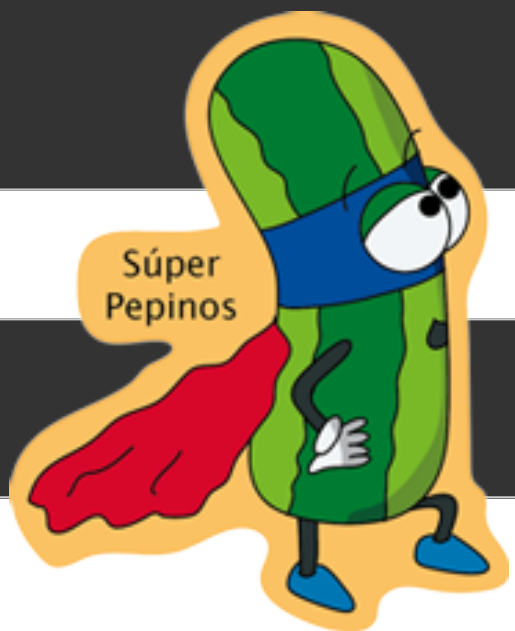
```
scala> y.sg
res1: String = super pepinos
```

Att skilja på mitt och ditt med `super`

```
1  scala> class X { val gurka = "super pepinos" }
2
3  scala> class Y extends X {
4           override val gurka = ":("
5           val sg = super.gurka
6         }
7
8  scala> val y = new Y
9  y: Y = Y@26ba2a48
10
11 scala> y.gurka
12 res0: String = :(
```

Super Pepinos to the rescue:

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



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.