

# EDAA45 Programmering, grundkurs

## Läsvecka 7: Arv

Björn Regnell

Datavetenskap, LTH

Lp1-2, HT 2016

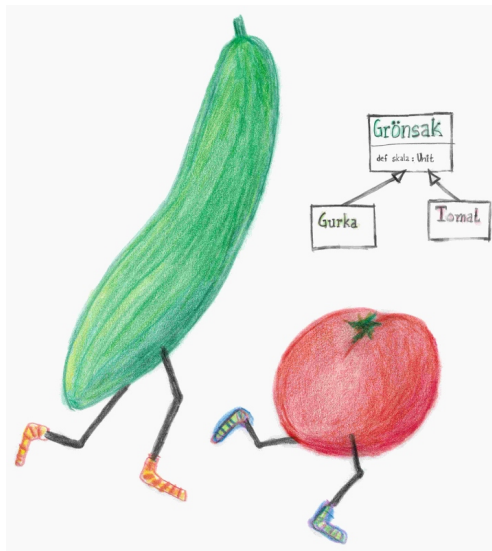
## 7 Arv

- Vad är arv?
- Uppräknade värden
- Exempel: Shape
- Överskuggningsregler
- super
- Trait eller abstrakt klass?
- Nästa vecka: kontrollskrivning
- Veckans uppgifter

# Vad är arv?

# Vad är arv?

Med arv kan man  
beskriva relationen  
 $X \text{ är en } Y$



# 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 bas typ

```
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]`?

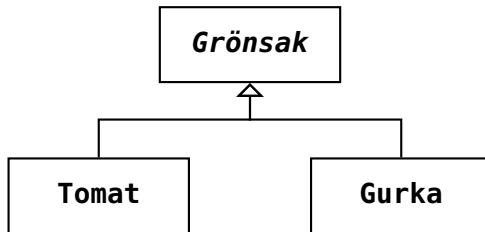
# Behovet av gemensam bastyp

```
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]`? → Skapa en **bastyp**!

# Skapa en gemensam basotyp

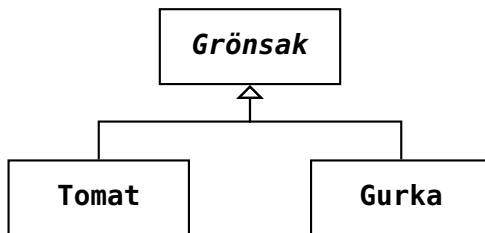
Typen **Grönsak** är en **basotyp** i nedan arvshierarki:





# Skapa en gemensam bas typ

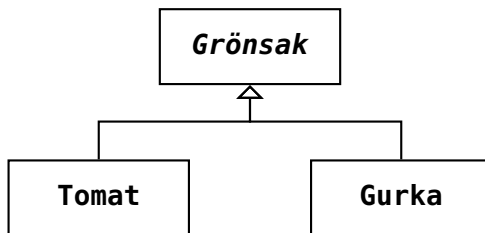
Typen **Grönsak** är en **bas typ** i nedan arvshierarki:



Pilen  betecknar **arv** och utläses "**är en**"

# Skapa en gemensam bas typ

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 bas typ med trait och extends

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

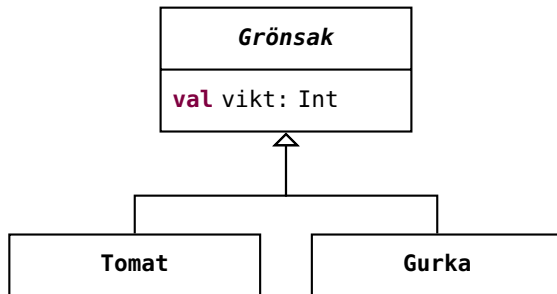
```
1 scala> trait Grönsak
2
3 scala> class Gurka(val vikt: Int) extends Grönsak
4
5 scala> class Tomat(val vikt: Int) extends Grönsak
6
7 scala> val grönsaker = Vector(new Gurka(200), new Tomat(42))
8 grönsaker: scala.collection.immutable.Vector[Grönsak] =
9   Vector(Gurka@3dc4ed6f, Tomat@2823b7c5)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

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 bas typ med gemensamma delar

Placera gemensamma medlemmar i bas typen:



- Alla grönsaker har attributet **val** vikt.
- Det specifika värdet på vikten definieras **inte** i bas typen.
- Medlemmen 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:

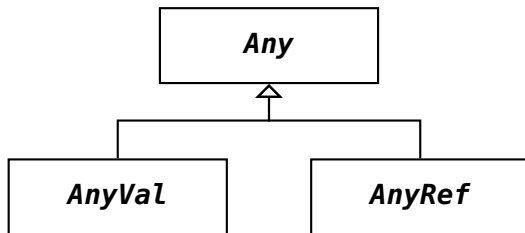
```
trait Grönsak { val vikt: Int }  
  
class Gurka(val vikt: Int) extends Grönsak  
  
class Tomat(val vikt: Int) extends Grönsak
```

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

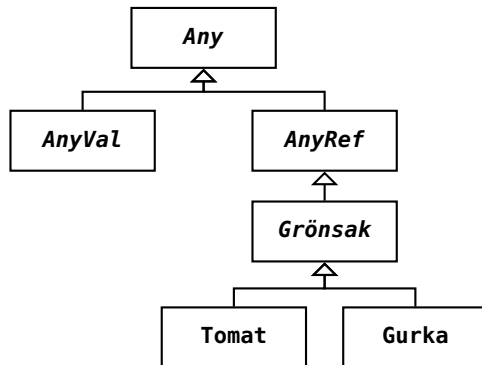
Den översta delen av typhierarkin i Scala:



- 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 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
- Jämförelse med Java:
  - En Scala-trait liknar det som i Java kallas **interface**, men man kan göra mer med Scala-traits: färre begränsningar, fler abstraktionsmöjligheter.
  - En Scala-trait med enbart abstrakta medlemmar kompileras till bytekod i JVM:en som kan användas från Java-kod precis som ett Java-interface.

# 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 bas typ 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
```

```
1  scala> val a = new A
2  a: A = A@5faeada1
3
4  scala> val b = new B
5  b: B = B@cb51256
6
7  scala> val xs = Vector(a,b)
8  xs: scala.collection.immutable.Vector[X] = Vector(A@5faeada1, B@cb51256)
9
10 scala> xs.map(_._1)
11 res0: scala.collection.immutable.Vector[Int] = Vector(42, 43)
12
13 scala> class Y { val y: Int }
14     error: class Y needs to be abstract, since value y is not defined
15
16 scala> trait Z(x: Int)
17     error: traits or objects may not have parameters
```

# Terminologi och nyckelord

|                        |  |
|------------------------|--|
| <b>subtyp</b>          | en typ som ärver en supertyp   |
| <b>supertyp</b>        | en typ som ärvs av en subtyp   |
| <b>bastyp</b>          | en typ som är rot i ett arvsträd   |
| <b>abstrakt medlem</b> | en medlem som saknar implementation  |
| <b>konkret medlem</b>  | en medlem som ej saknar implementation                                     |
| <b>abstrakt typ</b>    | en typ som kan ha abstrakta medlemmar; kan ej instansieras                 |
| <b>konkret typ</b>     | en typ som ej har abstrakta medlemmar; kan instansieras                    |
| <b>class</b>           | en klass är en konkret typ som <b>ej kan ha abstrakta medlemmar</b>        |
| <b>abstract class</b>  | en klass är en abstrakt typ som <b>kan ha parametrar</b>                   |
| <b>trait</b>           | är en abstrakt typ som <b>ej kan ha parametrar</b> men <b>kan mixas in</b> |
| <b>extends</b>         | står före en supertyp, medför arv av supertypens medlemmar                 |
| <b>override</b>        | en medlem överskuggar (byter ut) en medlem i en supertyp                   |
| <b>protected</b>       | gör en medlem synlig i subtyper till denna typ (jmf <b>private</b> )       |
| <b>final</b> gurka     | gör medlemen gurka final: förhindrar överskuggning                         |
| <b>final class</b>     | gör klassen final: förhindrar vidare subtypning                            |
| <b>sealed trait</b>    | förseglad trait: bara de direkta subtyperna i denna kodfil                 |
| <b>super</b> . gurka   | refererar till supertypens medlem gurka (jmf <b>this</b> )                 |

# 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 som 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, medför arv av supertypens medlemmar

**override**

en medlem överskuggar (byter ut) en medlem i en supertyp

**protected**

gör en medlem synlig i subtyper till denna typ (jmf **private**)

**final** gurka

gör medlemmen gurka final: förhindrar överskuggning

**final class**

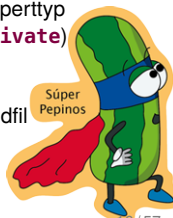
gör klassen final: förhindrar vidare subtypning

**sealed trait**

förseglad trait: bara de direkta subtyperna i denna kodfil

**super**.gurka

refererar till supertypens medlem gurka (jmf **this**)





└ Vecka 7: Arv

└ Vad är arv?

# Abstrakta och konkreta medlemmar

```
1  object exempelVegol {
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 = "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 = "Skållas."
25     }
26 }
```

## 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 → **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.



└ Vecka 7: Arv

└ Vad är arv?

# 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 = "Skalas med skalare."
20     }
21
22     class Tomat(var vikt: Double) extends Grönsak {
23         val namn = "tomat" //nyckelordet override behövs ej vid abstrakt medlem, men tillåtet:
24         override val skalningsmetod = "Skållas."
25         // override val skalningsmetod = "Skållas." //kompilatorn hittar felet (stavfel, s saknas)
26         override val skalfaktor = 0.95 //överskuggning: override måste anges vid ny impl.
27     }
28 }
```

# En final medlem kan ej överskuggas

```
1  object exempelVego4 {
2
3      trait Grönsak {
4          val skalningsmetod: String
5          final val skalfaktor = 0.99                // en final medlem kan ej överskuggas
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 = "Skalas med skalare."
20     }
21
22     class Tomat(var vikt: Double) extends Grönsak {
23         val namn = "tomat"
24         val skalningsmetod = "Skållas."
25         // override val skalfaktor = 0.95 // KOMPILERINGSFEL: "cannot override final member"
26     }
27 }
```

# Protected ger synlighet begränsad till subtyper

```
1 scala> trait Super {  
2     private val minHemlis = 42  
3     protected val vårHemlis = 42  
4 }  
5  
6 scala> class Sub extends Super { def avslöjad = minHemlis }  
7 error: not found: value minHemlis  
8  
9 scala> class Sub extends Super { def avslöjad = vårHemlis }  
10  
11 scala> val s = new Sub  
12 s: Sub = Sub@2eeee9593  
13  
14 scala> s.avslöjad  
15 res0: Int = 42  
16  
17 scala> s.minHemlis  
18 error: value minHemlis is not a member of Sub  
19  
20 scala> s.vårHemlis  
21 error: Access to protected value vårHemlis not permitted
```

# 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. `drawing.scala` eller `bastypensNamn.scala`

# Klasser, arv och klassparametrar

Klasser kan ärva klasser. Om superklassen har klassparametrar måste primärkonstruktor ges argument efter **extends**.

```
1 object personExample1 {  
2  
3   class Person(val namn: String)  
4  
5   class Akademiker(namn: String,  
6                     val universitet: String) extends Person(namn)  
7  
8   class Student(namn: String,  
9                 universitet: String,  
10                val program: String) extends Akademiker(namn, universitet)  
11  
12  class Forskare(namn: String,  
13                universitet: String,  
14                val titel: String) extends Akademiker(namn, universitet)  
15  
16  def main(args: Array[String]): Unit = {  
17    val kim = new Student("Kim Robinsson", "Lund", "Data")  
18    println(s"${kim.namn} ${kim.universitet} ${kim.program}")  
19  }  
20  
21 }
```

# Inmixning

Man kan ärva flera traits. Detta kallas inmixning (eng. *mix-in*) och görs med **with**.

```
1 object personExample2 {
2
3   trait Person { val namn: String }
4
5   trait Akademiker { val universitet: String }
6
7   trait Examinerad { val titel: String }
8
9   class Student(val namn: String,
10                val universitet: String,
11                val program: String) extends Person with Akademiker
12
13   class Forskare(val namn: String,
14                 val universitet: String,
15                 val titel: String) extends Person with Akademiker with Examinerad
16
17   def main(args: Array[String]): Unit = {
18     var p: Person = new Forskare("Robin Smith", "Lund", "Professor Dr")
19     println(s"${p.namn}") // staiska typen Person har inget universitet el. titel
20     if (p.isInstanceOf[Akademiker]) println(p.namn)
21   }
22 }
```

# Statisk och dynamisk typ

```
var p: Person = new Forskare("Robin Smith", "Lund", "Professor Dr")
```

- Den **statiska typen** för p är Person vilket gör att vi sedan kan låta p referera till andra instanser som är av typen Person.

```
p = new Student("Kim Robinson", "Lund", "Data")
```



# Statisk och dynamisk typ

```
var p: Person = new Forskare("Robin Smith", "Lund", "Professor Dr")
```

- Den **statiska typen** för p är Person vilket gör att vi sedan kan låta p referera till andra instanser som är av typen Person.

```
p = new Student("Kim Robinson", "Lund", "Data")
```

- Med "statisk typ" menas den typinformation som kompilatorn känner till vid kompileringstid.

# Statisk och dynamisk typ

```
var p: Person = new Forskare("Robin Smith", "Lund", "Professor Dr")
```

- Den **statiska typen** för p är Person vilket gör att vi sedan kan låta p referera till andra instanser som är av typen Person.

```
p = new Student("Kim Robinson", "Lund", "Data")
```

- Med "statisk typ" menas den typinformation som kompilatorn känner till vid kompileringstid.
- Den **dynamiska typen**, även kallad **körtidstypen**, som gäller under körning är här mer specifik och mångfaceterad: p är efter tilldelning nu Student, Person och Akademiker (men inte Examinerad).

# Statisk och dynamisk typ

```
var p: Person = new Forskare("Robin Smith", "Lund", "Professor Dr")
```

- Den **statiska typen** för p är Person vilket gör att vi sedan kan låta p referera till andra instanser som är av typen Person.

```
p = new Student("Kim Robinson", "Lund", "Data")
```

- Med "statisk typ" menas den typinformation som kompilatorn känner till vid kompileringstid.
- Den **dynamiska typen**, även kallad **körtidstypen**, som gäller under körning är här mer specifik och mångfaceterad: p är efter tilldelning nu Student, Person och Akademiker (men inte Examinerad).
- Man kan undersöka om den dynamiska typen för p är EnVisstTyp med `p.isInstanceOf[EnVisstTyp]`

# Statisk och dynamisk typ

```
var p: Person = new Forskare("Robin Smith", "Lund", "Professor Dr")
```

- Den **statiska typen** för p är Person vilket gör att vi sedan kan låta p referera till andra instanser som är av typen Person.

```
p = new Student("Kim Robinson", "Lund", "Data")
```

- Med "statisk typ" menas den typinformation som kompilatorn känner till vid kompileringstid.
- Den **dynamiska typen**, även kallad **körtidstypen**, som gäller under körning är här mer specifik och mångfaceterad: p är efter tilldelning nu Student, Person och Akademiker (men inte Examinerad).
- Man kan undersöka om den dynamiska typen för p är EnVissTyp med `p.isInstanceOf[EnVissTyp]`
- Man kan säga åt kompilatorn: *"jag garanterar att p är av typen EnVissTyp så du kan omforma den till EnVissTyp"* med `p.asInstanceOf[EnVissTyp]`

# Statisk och dynamisk typ

```
var p: Person = new Forskare("Robin Smith", "Lund", "Professor Dr")
```

- Den **statiska typen** för p är Person vilket gör att vi sedan kan låta p referera till andra instanser som är av typen Person.

```
p = new Student("Kim Robinson", "Lund", "Data")
```

- Med "statisk typ" menas den typinformation som kompilatorn känner till vid kompileringstid.
- Den **dynamiska typen**, även kallad **körtidstypen**, som gäller under körning är här mer specifik och mångfaceterad: p är efter tilldelning nu Student, Person och Akademiker (men inte Examinerad).
- Man kan undersöka om den dynamiska typen för p är EnVissTyp med `p.isInstanceOf[EnVissTyp]`
- Man kan säga åt kompilatorn: *"jag garanterar att p är av typen EnVissTyp så du kan omforma den till EnVissTyp"* med `p.asInstanceOf[EnVissTyp]` (detta är inte så vanligt i normal Scala-kod)

# isInstanceOf och asInstanceOf

Testa körtidstyp med `isInstanceOf[Typ]`. Lova kompilatorn (och ta själv ansvar för) att det är en viss körtidstyp med `asInstanceOf[Typ]`.

```
1 object personExample3 {
2
3   trait Person { val namn: String }
4
5   trait Akademiker { val universitet: String }
6
7   trait Examinerad { val titel: String }
8
9   class Student(val namn: String,
10                val universitet: String,
11                val program: String) extends Person with Akademiker
12
13   class Forskare(val namn: String,
14                 val universitet: String,
15                 val titel: String) extends Person with Akademiker with Examinerad
16
17   def main(args: Array[String]): Unit = {
18     var p: Person = new Forskare("Robin Smith", "Lund", "Professor Dr")
19     if (p.isInstanceOf[Akademiker]) println(p.namn + " är akademiker")
20     p = new Student("Kim Robinson", "Lund", "Data")
21     if (p.isInstanceOf[Student]) println(p.asInstanceOf[Student].program)
22   }
23 }
```

# Polymorfism och dynamisk bindning

```
trait Robot { def work(): Unit }

case class CleaningBot(name: String) extends Robot {
  override def work(): Unit = println(" Städa Städa")
}

case class TalkingBot(name: String) extends Robot {
  override def work(): Unit = println(" Prata Prata")
}
```

**Polymorfism** betyder "många former". Referenserna `r` och `bot` nedan kan ha olika "former", d.v.s de kan referera till olika sorters robotar.

**Dynamisk bindning** innebär att körtidstypen avgör vilken metod som körs.

```
1 scala> def robotDoWork(bot: Robot) = { print(bot); bot.work }
2
3 scala> var r: Robot = new CleaningBot("Wall-E")
4
5 scala> robotDoWork(r)
6 CleaningBot(Wall-E) Städa Städa
7
8 scala> r = new TalkingBot("C3P0")
9
10 scala> robotDoWork(r)
11 TalkingBot(C3P0) Prata Prata
```

# Uppräknade värden



# Uppräknade värden med heltal

Vi kan använda heltalskonstanter för att representera olika färger.

```
object Färg {  
  val Spader = 1  
  val Hjärter = 2  
  val Ruter = 3  
  val Klöver = 4  
}
```

```
case class Kort(färg: Int, valör: Int)
```

Vi kan nu använda våra uppräknade färgvärden så här:

```
1 scala> import Färg._  
2 scala> Kort(Ruter, 7)
```

# Uppräknade värden med heltal

Vi kan använda heltalskonstanter för att representera olika färger.

```
object Färg {  
  val Spader = 1  
  val Hjärter = 2  
  val Ruter = 3  
  val Klöver = 4  
}
```

```
case class Kort(färg: Int, valör: Int)
```

Vi kan nu använda våra uppräknade färgvärden så här:

```
1 scala> import Färg._  
2 scala> Kort(Ruter, 7)
```

Men kompilatorn **kan inte hindra** denna bugg:

```
1 scala> Kort(42, 7)
```

# Uppräknade värden med case-objekt

Vi kan använda case-objekt för att representera olika färger.

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

case class Kort(färg: Färg, valör: Int)
```

Vi kan nu använda våra uppräknade färgvärden så här:

```
1 scala> Kort(Ruter, 7)
2
3 scala> Kort(Spader, 1)
```

- Kompilatorn **garanterar** att vi bara använder exakt dessa färger.
- Nyckelordet **sealed** förhindrar fler subtyper förutom de som finns här.
- **case** före **object** ger en najs toString och möjliggör matchning (mer om matchning i w08).

# Uppräknade värden i samling

Vi kan placera case-objekten i en samling som kan användas i loopar. Ett lämpligt ställe för en sådan samling är i kompanjonsobjektet till Färg.

```
sealed trait Färg
object Färg {
  val values = Vector(Spader, Hjärter, Ruter, Klöver)
}
case object Spader extends Färg
case object Hjärter extends Färg
case object Ruter extends Färg
case object Klöver extends Färg
```

```
1 scala> val allaEss = for (f <- Färg.values) yield Kort(f, 1)
```

# Uppräknade värden med heltalsomvandling

Med en **sealed abstract class** och ett heltalsattribut `toInt` som klassparameter kan vi erbjuda omvandling till heltal.

```
sealed abstract class Färg(final val toInt: Int)
object Färg {
  val values = Vector(Spader, Hjärter, Ruter, Klöver)
}
case object Spader extends Färg(0)
case object Hjärter extends Färg(1)
case object Ruter extends Färg(2)
case object Klöver extends Färg(3)
```

```
1 scala> Kort(Ruter, 1).färg.toInt
2 res0: Int = 2
```

Nyckelordet **abstract** förhindrar instansiering av `Färg`.

Nyckelordet **final** förhindrar överskuggning av attributet `toInt`.

Nyckelordet **sealed** förhindrar vidare subtypning av `Färg`.

# Exempel: Shape

# Exempel: shapes1.scala

Typisk scala-kod: En trait som bastyp åt flera case-klasser.

```
1  object shapes1 {
2      type Pt = (Double, Double)
3
4      trait Shape {
5          def pos: Pt
6          def move(dx: Double, dy: Double): Shape
7      }
8
9      case class Rectangle(pos: Pt, dxy: Pt) extends Shape {
10         def move(dx: Double, dy: Double): Rectangle =
11             Rectangle((pos._1 + dx, pos._2 + dy), dxy)
12     }
13
14     case class Circle(pos: Pt, radius: Double) extends Shape {
15         def move(dx: Double, dy: Double): Circle =
16             Circle((pos._1 + dx, pos._2 + dy), radius)
17     }
18
19     case class Triangle(pos: Pt, dxy1: Pt, dxy2: Pt) extends Shape {
20         override def move(dx: Double, dy: Double): Triangle =
21             Triangle((pos._1 + dx, pos._2 + dy), dxy1, dxy2)
22     }
23 }
```

# Exempel: shapesTest1.scala

Test av konkreta subclasser till bastypen Shape.

```
1  import shapes1._
2
3  object shapesTest1 {
4      def main(args: Array[String]): Unit = {
5          val r = Rectangle(pos= (100, 100), dxy= (75, 120))
6          println(r)
7          val r2 = r.move(dx= 42, dy= 84).move(dx= -1, dy= -1)
8          println(r2)
9          val t = Triangle((0,0),(4,0), (4,3))
10         println(t)
11         println(t.move(1,1))
12         println(t)
13     }
14 }
```

```
1  Rectangle((100.0,100.0),(75.0,120.0))
2  Rectangle((141.0,183.0),(75.0,120.0))
3  Triangle((0.0,0.0),(4.0,0.0),(4.0,3.0))
4  Triangle((1.0,1.0),(4.0,0.0),(4.0,3.0))
5  Triangle((0.0,0.0),(4.0,0.0),(4.0,3.0))
```



## Exempel: draw.scala

Två traits som kan användas för att "koppla ihop" kod och minimera ändringar av befintlig kod:

```
1 trait CanDraw {  
2     def draw(dw: DrawingWindow): Unit  
3 }  
4  
5 trait DrawingWindow {  
6     def penTo(pt: (Double, Double)): Unit  
7     def drawTo(pt: (Double, Double)): Unit  
8 }
```

# Exempel: draw.scala

Två traits som kan användas för att ”koppla ihop” kod och minimera ändringar av befintlig kod:

```
1 trait CanDraw {  
2     def draw(dw: DrawingWindow): Unit  
3 }  
4  
5 trait DrawingWindow {  
6     def penTo(pt: (Double, Double)): Unit  
7     def drawTo(pt: (Double, Double)): Unit  
8 }
```

- Traits som användas för att abstrahera implementation och möjliggöra uppfyllandet av ett slags ”kontrakt” om vad som ska finnas kallas **gränssnitt** (eng. *interface*) och är grunden för skapandet av ett flexibelt api.
- Implementationen av de delar vi vill kunna ändra senare placeras i subtyper som inte används direkt av klientkoden.
- Vi visar bara information om vad som erbjuds men inte hur det ser ut ”inuti”.

# Exempel: shapes2.scala

Genom att **mixa in** vår **trait** CanDraw kan en rektangel nu även ritas ut:

```
1  object shapes2 {
2      type Pt = (Double, Double)
3
4      trait Shape {
5          def pos: Pt
6          def move(dx: Double, dy: Double): Shape
7      }
8
9      case class Rectangle(pos: Pt, dxy: Pt) extends Shape with CanDraw { // inmixning
10         override def move(dx: Double, dy: Double): Rectangle =
11             Rectangle((pos._1 + dx, pos._2 + dy), dxy)
12
13         override def draw(dw: DrawingWindow): Unit = { // implementation av draw
14             dw.penTo(pos)
15             dw.drawTo((pos._1 + dxy._1, pos._2))
16             dw.drawTo((pos._1 + dxy._1, pos._2 + dxy._2))
17             dw.drawTo((pos._1, pos._2 + dxy._2))
18             dw.drawTo(pos)
19         }
20     }
21 }
```

# Exempel: shapes2.scala

Genom att **mixa in** vår **trait** CanDraw kan en rektangel nu även ritas ut:

```
1  object shapes2 {
2      type Pt = (Double, Double)
3
4      trait Shape {
5          def pos: Pt
6          def move(dx: Double, dy: Double): Shape
7      }
8
9      case class Rectangle(pos: Pt, dxy: Pt) extends Shape with CanDraw { // inmixning
10         override def move(dx: Double, dy: Double): Rectangle =
11             Rectangle((pos._1 + dx, pos._2 + dy), dxy)
12
13         override def draw(dw: DrawingWindow): Unit = { // implementation av draw
14             dw.penTo(pos)
15             dw.drawTo((pos._1 + dxy._1, pos._2))
16             dw.drawTo((pos._1 + dxy._1, pos._2 + dxy._2))
17             dw.drawTo((pos._1, pos._2 + dxy._2))
18             dw.drawTo(pos)
19         }
20     }
21 }
```

Notera: ingen ändring i Shape! Vi behöver nu bara ett **DrawingWindow...**

# Exempel: SimpleDrawingWindow.scala

Vi skapar en ny klass som ärver SimpleWindow, som **dessutom** även **är ett** DrawingWindow, tack vare **inmixning** med nyckelordet **with**.  
Observera att vi måste skicka vidare klassparametrarna till superklassens konstruktor.

```
1  class SimpleDrawingWindow(title: String = "Untitled", size: (Int, Int) = (640, 400))
2    extends cslib.window.SimpleWindow(size._1, size._2, title)
3      with DrawingWindow {
4
5      def xPos(pt: (Double, Double)): Int = pt._1.round.toInt
6      def yPos(pt: (Double, Double)): Int = pt._2.round.toInt
7
8      override def penTo(pt: (Double, Double)): Unit = moveTo(xPos(pt), yPos(pt))
9      override def drawTo(pt: (Double, Double)): Unit = lineTo(xPos(pt), yPos(pt))
10 }
```

# Exempel: SimpleDrawingWindow.scala

Vi skapar en ny klass som ärver SimpleWindow, som **dessutom** även **är ett** DrawingWindow, tack vare **inmixning** med nyckelordet **with**.  
Observera att vi måste skicka vidare klassparametrarna till superklassens konstruktor.

```

1  class SimpleDrawingWindow(title: String = "Untitled", size: (Int, Int) = (640, 400))
2    extends cslib.window.SimpleWindow(size._1, size._2, title)
3      with DrawingWindow {
4
5      def xPos(pt: (Double, Double)): Int = pt._1.round.toInt
6      def yPos(pt: (Double, Double)): Int = pt._2.round.toInt
7
8      override def penTo(pt: (Double, Double)): Unit = moveTo(xPos(pt), yPos(pt))
9      override def drawTo(pt: (Double, Double)): Unit = lineTo(xPos(pt), yPos(pt))
10 }

```

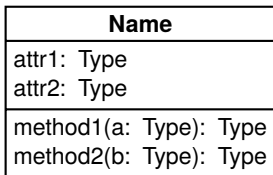
```

1  import shapes2._
2
3  object shapesTest2 {
4    def main(args: Array[String]): Unit = {
5      val sdw = new SimpleDrawingWindow(title="Shapes")
6      val r = Rectangle(pos=(100, 100), dxy=(75, 120))
7      r.draw(sdw)
8      r.move(dx=42, dy=84).draw(sdw)
9    }
10 }

```

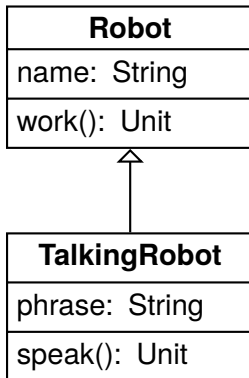
# Attribut och metoder i UML-diagram

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



Ibland utelämnar man typerna.

[en.wikipedia.org/wiki/Class\\_diagram](http://en.wikipedia.org/wiki/Class_diagram)



# Överskuggningsregler



# 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

```
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:

# 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
```



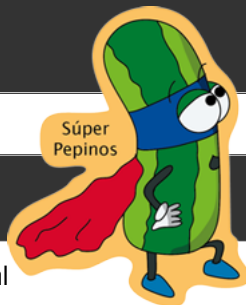
# Att skilja på mitt och ditt med super

```
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
```

[www.pepinadas.com/tirassuper1.html](http://www.pepinadas.com/tirassuper1.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.

# Nästa vecka: kontrollskrivning

# Obl. kontrollskrivning: 25/10 kl 14:00-19 Gasquesalen

Kontrollskrivningen motsvarar i omfång en **halv** ordinarie tentamen och är uppdelad i två delar; del A och del B.

- Del A omfattar 20% av den maximala poängsumman och innehåller uppgifter med korta svar (likt övningarna): "ange typ och värde".
- Del B omfattar 80% av den maximala poängsumman och innehåller uppgifter med svar i form av kod.
- Maximal poäng på kontrollskrivningen är 50p. (Ordinarie tenta 100p)
- Om du erhåller p poäng på kontrollskrivningen bidrar du med  $(p / 10.0) . \text{round}$  i individuell bonuspoäng inför sammanräkningen av samarbetsbonus.
- Din samarbetsbonus är medelvärde av poängen från dig och de av dina gruppmedlemmar som skriver kontrollskrivningen enligt denna beräkning:

```
def collaborationBonus(points: Seq[Int]): Int =  
    (points.sum / points.size.toDouble).round.toInt
```

- Samarbetsbonus motsvarar max 5% av totala ordinarie tentapoäng.
- Samarbetsbonus påverkar inte om du blir godkänd på tentan, men kan påverka vilket betyg du får.

# Obligatorisk kontrollskrivning: instruktioner

## Medtag:

legitimation, penna blyerts, penna i avvikande färg helst röd, ev. förtäring och **Scala Quickref/Java Snabbref**.

- Moment 1, ca 2,5h: **Lösning av uppgifterna**.
  - Du löser uppgifterna individuellt med blyertspenna.
  - När du är klar lämnar du in alla dina svar.
- Moment 2: **Parvis kamraträttning**.
- Moment 3: **Bedömning av rättning**.

# Obligatorisk kontrollskrivning: instruktioner

- Moment 1, ca 2,5h: **Lösning av uppgifterna.**
- Moment 2: **Parvis kamraträttning.**
  - Ni sätter er parvis och får ut rättningsmallen som ni läser.
  - Efter ca 10 minuter får ni ut två andra personers skrivningar som ni rättar enligt anvisningarna i rättningsmallen.
  - Medtag och använd penna med avvikande färg, helst röd.
  - När rättningstiden är slut samlar vi in alla rättade skrivningarna.

# Obligatorisk kontrollskrivning: instruktioner

- Moment 1, ca 2,5h: **Lösning av uppgifterna.**
- Moment 2: **Parvis kamraträttning.**
  - Ni sätter er parvis och får ut rättningsmallen som ni läser.
  - Efter ca 10 minuter får ni ut två andra personers skrivningar som ni rättar enligt anvisningarna i rättningsmallen.
  - Medtag och använd penna med avvikande färg, helst röd.
  - När rättningstiden är slut samlar vi in alla rättade skrivningarna.
- Moment 3: **Bedömning av rättning.**
  - Du får hämta din egen skrivning och titta på rättningen.
  - Är du nöjd med rättningen lämnar du bara tillbaka skrivningen igen.
  - Är du inte nöjd med rättningen kontaktar du skrivningsansvarig genom handuppräckning.



# Plugga på kontrollskrivning

- Träffas och plugga i samlarbetsgrupperna.
- Hjälp varandra med det som är svårt.
- Träna på att skriva kod på papper.
- Gör övningarna.
- Repetera laborationerna.
- Läs föreläsningssanteckningarna.
- Studera Scala Quickref MYCKET NOGA så att du vet vad som är givet och var det står så att du kan hitta det du behöver snabbt.
- Se sidan 329 i kompendiet (tips inför ordinarie tenta)

# Veckans uppgifter

# Övning: traits, ev. prioritera uppg. 1, 2, 3, 6, 10, 11

- Förstå följande begrepp: supertyp, subtyp, bastyp, abstrakt typ, polymorfism.
- Kunna deklarerar och använda en arvshierarki i flera nivåer med nyckelordet **extends**.
- Kunna deklarerar och använda inmixning med flera traits och nyckelordet **with**.
- Kunna deklarerar och känna till nyttan med finala klasser och finala attribut och nyckelordet **final**.
- Känna till synlighetsregler vid arv och nyttan med privata och skyddade attribut.
- Kunna deklarerar och använda skyddade attribute med nyckelordet **protected**.
- Känna till hur typtester och typkonvertering vid arv kan göras med metoderna `isInstanceOf` och `asInstanceOf` och känna till att detta görs bättre med **match**.
- Känna till begreppet anonym klass.
- Kunna deklarerar och använda överskuggade metoder med nyckelordet **override**.
- Känna till reglerna som gäller vid överskuggning av olika sorters medlemmar.
- Kunna deklarerar och använda hierarkier av klasser där konstruktorparametrar överförs till superklasser.
- Kunna deklarerar och använda uppräknade värden med case-objekt och gemensam bastyp.

# Instruktioner Grupplaboration

- Diskutera i din samarbetsgrupp hur ni ska dela upp koden mellan er i flera olika delar, som ni kan arbeta med var för sig. En sådan del kan vara en klass, en trait, ett objekt, ett paket, eller en funktion.
- Varje del ska ha en *huvudansvarig* individ.
- Arbetsfördelningen ska vara någorlunda jämt fördelad mellan gruppmedlemmarna.
- När ni redovisar er lösning ska ni börja med att redogöra för handledaren hur ni delat upp koden och vem som är huvudansvarig för vad.
- Den som är huvudansvarig för en viss del redovisar den delen.
- Grupplaborationer görs i huvudsak som hemuppgift. Salstiden används primärt för redovisning.

## Grupplaboration: turtlerace-team

- Kunna skapa och använda arvshierarkier och förstå dynamisk bindning.
- Kunna skapa använda en trait som bastyp i en arvshierarki.