Vecka 5. Klasser och datamodellering

Programmering, grundkurs (pgk)

Björn Regnell

Datavetenskap, LTH, Lunds universitet
https://lunduniversity.github.io/pgk

EDAA45, Lp1-2, HT2025

Kompilerad den 22 augusti 2025

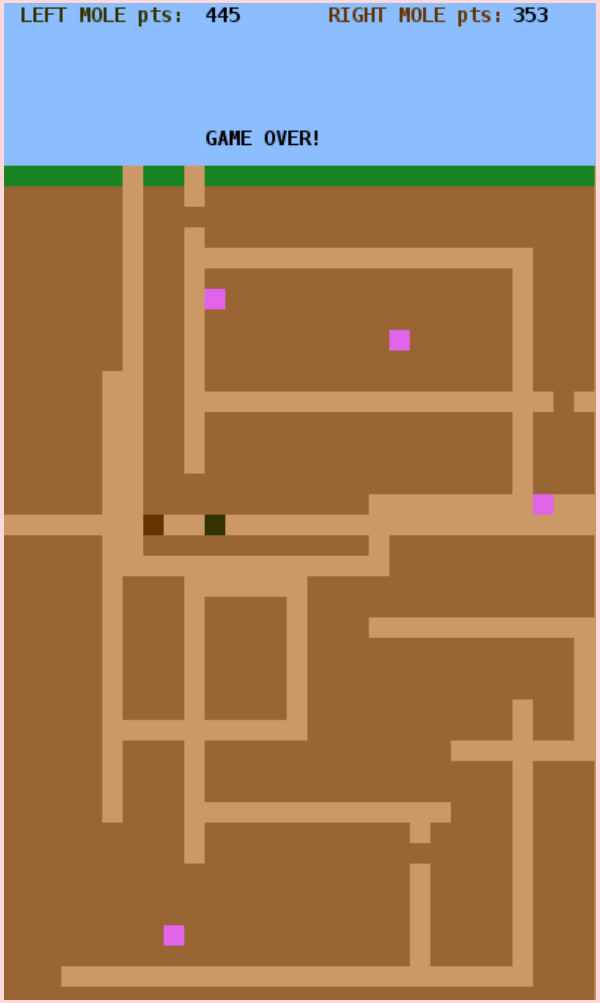
5 Klasser och datamodellering

- Vad är en klass?
- Olika sätt att skapa instanser
- Case-klasser och fördelen med oföränderlighet
- Konstruktor
- Referens saknas: `null`
- Initialisering av attribut
- Referensen `this`
- Getters och setters
- Likhet
- Implementation saknas: ???

Övning: `classes`

- Kunna deklarera klasser med klassparametrar.
- Kunna skapa instanser med och utan **`new`**.
- Kunna ge argument vid instansiering.
- Förstå innebörden av referensvariabler och värdet **`null`**.
- Kunna använda nyckelordet **`private`** för att styra synlighet av attribut och konstruktorparametrar.
- Förstå syftet med getters och setters.
- Kunna förklara accessregler för kompanjonsobjekt.
- Kunna skapa fabriksmetod i kompanjonsobjekt.
- Känna till nyttan med en privat konstruktor.
- Förstå skillnaden mellan referenslikhet och strukturlikhet.
- Känna till skillnaden mellan == och eq, samt != och ne.
- Kunna förklara hur case-klasser hanterar instansiering.
- Känna till hur case-klasser hanterar likhet.

Lab `blockbattle` redovisas NÄSTA läsvecka 6



- Ett arkadspel för TVÅ spelare.
- Nu med TVÅ blockmullvader!
- Programmet `blockbattle` är **mer omfattande** jmf m tidigare labbar.
- Labben omfattar TVÅ veckors förel. + övn.: klasser (w05) & mönster (w06).
- Det normala är registrering ”kompletteras” i SAM på `blockbattle0` under w05.
- Läs igenom labben innan du gör övningarna.
- Kod från övningarna behövs till labben.
- OBS! Labbtider är **schemalagda som vanligt** under läsvecka 5. Om du mot förmodan hinner redovisa redan denna vecka så ska du **ändå närvara** och t.ex. jobba med **extrauppgifter**.
- Dra nytta av undervisningen så att du lär dig så mycket som möjligt!

Vad är en klass?

En metafor för klass: Stämpel

En klass liknar en **stämpel**.

En metafor för klass: Stämpel

En klass liknar en **stämpel**.

- En stämpel kan **tillverkas** – motsvarar **deklaration** av klassen.
- Det händer inget förrän man **stämplar** – motsvarar **instansiering**.
- Då skapas **avbildningar** av stämpeln – motsvarar **allokering av ett objekt** som är en **instans** av klassen.
- Allokering kallas också **konstruktion** och funktionen/koden som gör själva allokeringen kallas **konstruktor**.

Vad är en klass?

- En klass är en mall (eng. *template*) för att skapa objekt.
- Objekt kan skapas med **`new`** `Klassnamn(parametrar)`, vilket kallas **instansiering**.
- I Scala 3 är **`new`** valfritt, det räcker med `Klassnamn(parametrar)`.
- Ett objekt som skapats med klassen `Klassnamn` som mall kallas för en **instans** av klassen `Klassnamn`.
- En klass innehåller **medlemmar** (eng. *members*), som bl.a. kan vara:
  - **attribut**, kallas även fält (eng. *field*): **`val`**, **`lazy val`**, **`var`**
  - **metoder**, kallas även operationer: **`def`**
- Varje instans har sin **egen** uppsättning värden på attributen, som tillsammans utgör instansens **tillstånd**.

Datamodellering

Varför behövs klasser?

- I en viss **applikationsdomän** (eng. *application domain*), tex. skatteverkets deklarationssystem, behövs en **modell av domänspecifik data**, t.ex. personer, personnummer, adresser, inkomster, avdrag, fastigheter, etc.
- Med klasser kan du skapa **nya** typer (utöver `Int`, `String` ...) som bättre representerar domänens data.
- Med klasser implementerar du modeller som representerar väsentliga **attribut** ur applikationsdomänen.
- Med **metoder** (funktioner i klasser) kan du skapa och behandla domänens data.
- Datamodellering i Scala görs ofta med s.k. **`case`**-klasser och **oföränderliga** instanser.

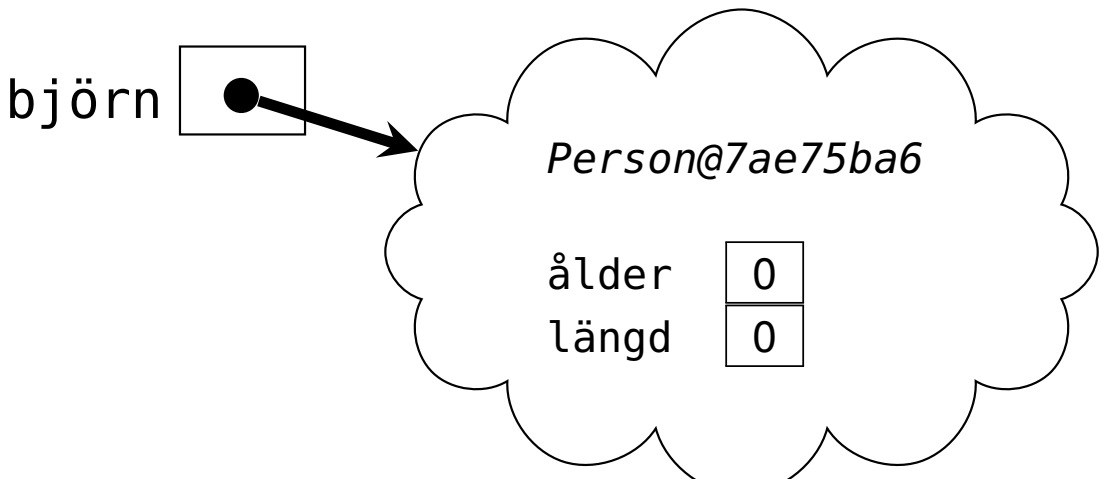
Singelobjekt jämfört med klass

Vi har tidigare deklarerat **singelobjekt** som bara finns i **en** enda upplaga:

```
scala> object Björn { var ålder = 54; val längd = 178 }
```

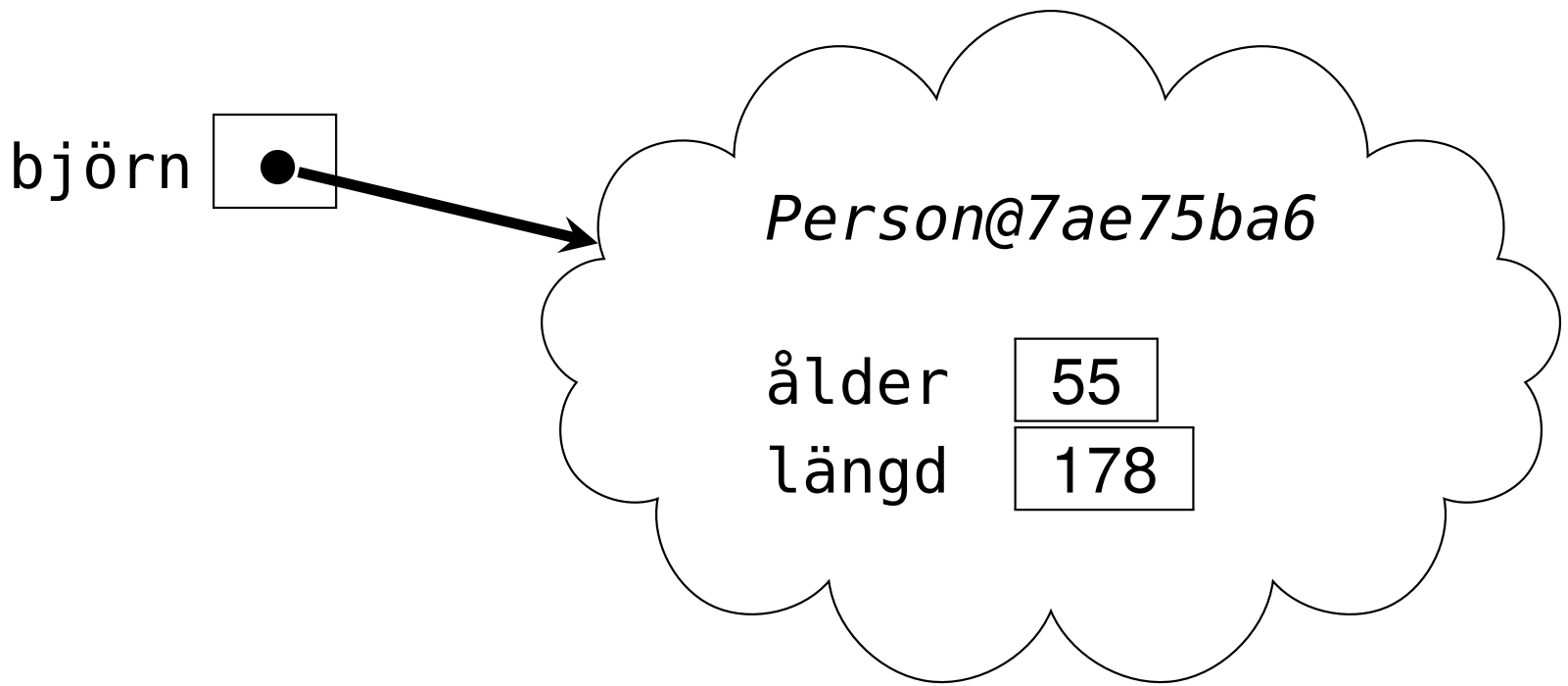
Med en **klass** kan man skapa **godtyckligt många instanser av klassen** med hjälp av nyckelordet **new** följt av klassens namn:

```
scala> class Person { var ålder = 0; var längd = 0 }

scala> val björn = new Person   // allokera plats i minnet
björn: Person = Person@7ae75ba6  // unikt id för instansen
```



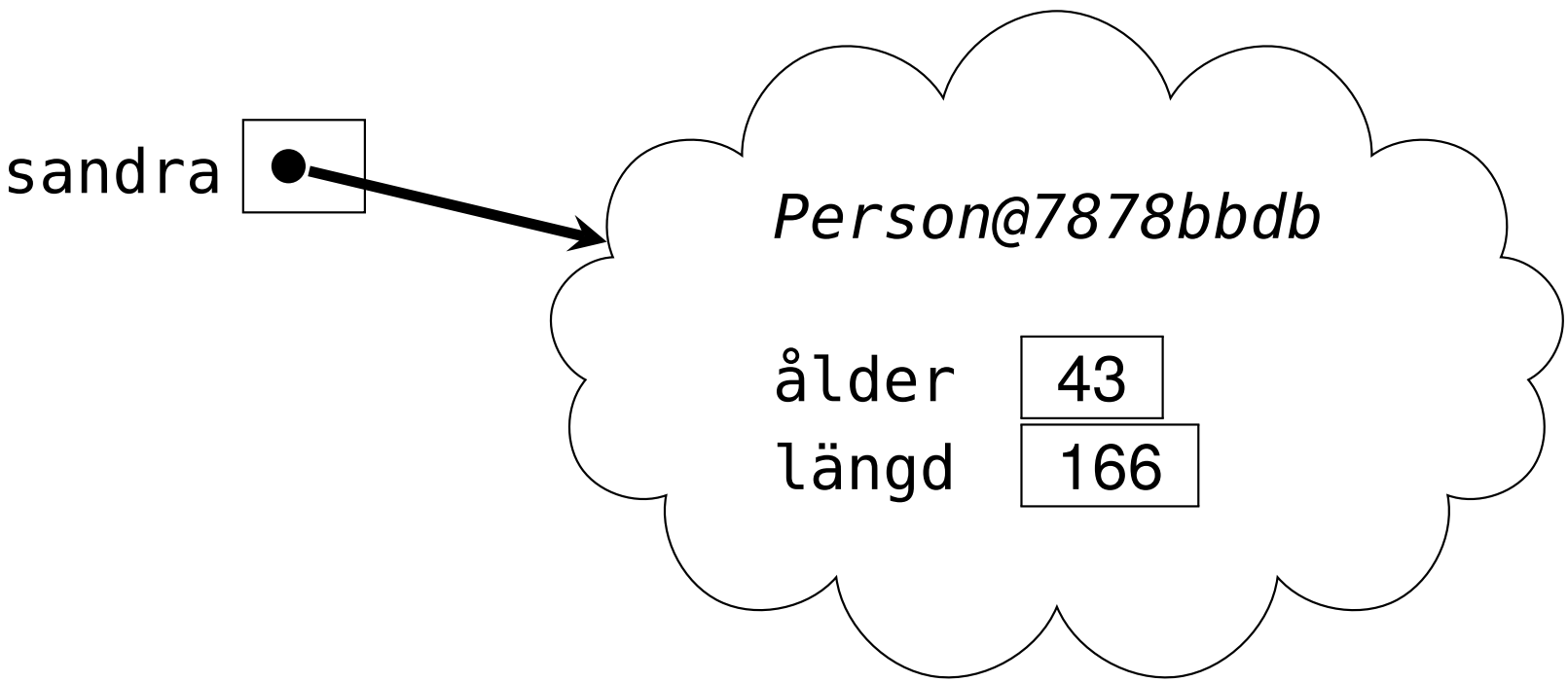
Förändring av objektets tillstånd

```
scala> björn.ålder = 55

scala> björn.längd = 178
```



Bättre att initialisera med hjälp av klassparametrar

```
scala> class Person(var ålder: Int, var längd: Int)

scala> val sandra = new Person(43, 166)
sandra: Person = Person@7878bbdb
```



Klassdeklarationer och instansiering

- Syntax för deklaration av klass:

```
class Klassnamn(parametrar){ medlemmar }
```

- Exempel: **deklaration**

```
class Klassnamn(val attribut1: Int, attribut2: String):  //klassparametrar
  val attribut3: Double = 42.0                 //publikt oföränderligt attribut
  private var attribut4: Boolean = false       //privat medlem syns inte utåt
  def metod(parameter: Int) = attribut1 + 1 //funktion i objekt kallas metod
  lazy val attr5 = Vector.fill(100000)(42.0)     //fördröjd initialisering
```

- **Klass-parametrar** blir **attribut** som initialiseras med de argument som ges vid **new**. (Kompilatorn skapar **primärkonstruktor**: kod som allokerar & initialiserar alla attribut.)
- Exempel: **instansiering** med argument för initialisering av klassparametrar

```
val instansReferens = new Klassnamn(42, "hej")  // new är valfritt i Scala 3
```

- Parametrar som inte föregås av modifierare (t.ex. **private val**, **val**, **var**) blir **attribut** som bara är synliga i **denna** instans, de kallas då **instansprivata**.
- Attribut i klasskroppen är **publika** (alltså synliga utåt) om de inte deklareras **private** (eller **protected** som begränsar synlighet till subtyper som vi ska se senare).

Övning: en klass som representerar en person

1. Deklarera en klass `Person` med dessa publika attribut:
   - oföränderligt förnamn
   - oföränderligt efternamn
   - förändringsbar ålder med defaultargument `0`
2. lägg till en metod i klasskroppen med explicit returtyp som ger en 2-tupel med förnamn och efternamn
3. skriv en deklaration som deklarerar en variabel `p` som initialiseras med värdet av ett uttryck som instansierar klassen `Person` med ditt namn och din ålder som nyfödd.
4. skriv en sats som skriver ut ditt förnamn genom att referera attribut med punktnotation
5. skriv en tilldelningssats som ändrar tillståndet för den instans som referensen `p` refererar till så att åldersattributets värde blir din nuvarande ålder

Lösning: klassen Person

```
class Person(
  val givenName: String,
  val familyName: String,
  var age: Int = 0
):
  def name: (String, String) = (givenName, familyName)
```

```
scala> val p = Person("Björn", "Regnell")
val p: Person = Person@783dc0e7

scala> println(p.name._1)
Björn

scala> p.age = 50
```

Kan vi få se något som är finare än `Person@783dc0e7` ?

Skapa egen najs toString

```
class Person(
  val givenName: String,
  val familyName: String,
  var age: Int = 0
):
  def name: (String, String) = (givenName, familyName)
  override def toString = "najs toString"
```

```
scala> val p = Person("Björn", "Regnell")
val p: Person = najs toString

scala> println(p.name._1)
Björn

scala> p.age = 55
```

Vad vill du se i stället för `"najs toString"`?
Övning: Visa instansens tillstånd med stränginterpolatorn s`"?"`

Instansprivata klassparametrar

- Parametrar som **inte** föregås av någon modifierare alls (t.ex. `val`, `var` etc.) blir medlemmar som är bara är synliga i **denna** instans.
- Exempel på konsekvensen av **instansprivata** parametrar:

```
1  scala> class C(a: Int){ def add(other: C): Int = a + other.a }
2  -- Error:
3  1 |class C(a: Int){ def add(other: C): Int = a + other.a }
4    |                                            ^^^^^^^
5    | value a cannot be accessed as a member of (other : C) from class C.
```

- Men detta fungerar fint:

```
1  scala> class D(private val a: Int){ def add(other: D): Int = a + other.a }
2
3  scala> D(42).add(D(43))
4  res0: Int = 85
```

...eftersom modifieraren `private val` ger en medlem som ”bara” är **klassprivat** och ger därmed synlighet i **alla** `D`-instanser (men bara där; medlemmen är inte ens synlig i subtyper till `D`).

Case-klasser är som vanliga klasser med extra godis

Med **`case`** framför **`class`** får du en massa **godis** på köpet, bland annat detta:

- En najs `toString`-metod med klassens namn och dess attributvärden.

```
scala> case class Person(name: String, age: Int)

scala> val p = Person("Björn", 55)

scala> p.toString
val res0: String = Person(Björn,55)
```

- Parameter till case-klass blir automatiskt ett **publikt oföränderligt attribut**, alltså en **`val`**-medlem utan att du behöver skriva något.

```
scala> p.age
val res1: Int = 55
```

- En `copy`-metod med alla attribut som parametrar och instansens attributvärden som default-argument: det blir då **smidigt att skapa delvis förändrade kopior** där några attribut ändrats med namngivna argument och andra förblir som innan.

```
scala> p.copy(age = p.age + 1)
val res2: Person = Person(Björn,56)
```

Fördjupning: Styra synlighet med `private[X]`

Med hjälp av **`private`**`[X]` kan du begränsa synlighet till X, där X kan vara ett singelobjekt, en typ eller ett paket:

```
scala> object  X:
     |   object Y { private[X] var y = 42 }
     |   def visaHemlis = Y.y  // y syns i X
     |
// defined object X

scala> X.Y.y
-- Error:
1 |X.Y.y
  |^^^^^
  |variable y cannot be accessed as a member of X.Y.type from module class rs$line$26.

scala> X.visaHemlis
val res0: Int = 42
```

Styra användningen av infixa alfanumeriska operatorer

Metoder som har **alfanumeriska namn**, alltså namn med bokstäver och ev. siffror ger en **varning** vid operatornotation om de **inte** är deklarerade med nyckelordet `infix`.

Styra användningen av infixa alfanumeriska operatorer

Metoder som har **alfanumeriska namn**, alltså namn med bokstäver och ev. siffror ger en **varning** vid operatornotation om de **inte** är deklarerade med nyckelordet **`infix`**.

```
case class Box(x: Int):
  def +(other: Box): Box = Box(x + other.x)    // utan varning
  def plus(other: Box) = Box(x + other.x)      // ger varning
  infix def add(other: Box) = Box(x + other.x) // utan varning
```

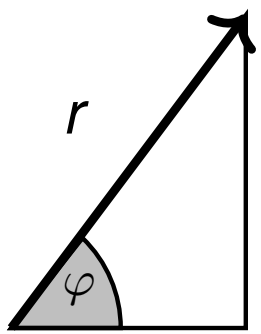
```
scala> Box(41) plus Box(1)
1 warning found
-- Warning: ------------------------------------------------------------------
1 |Box(41) plus Box(1)
  |        ^^^^
  |Alphanumeric method plus is not declared infix; it should not be used as infix operator.
  |Instead, use method syntax .plus(...) or backticked identifier `plus`.
val res0: Box = Box(42)

scala> Box(41) add Box(1)
val res1: Box = Box(41)
```

Övning: Klassen Complex

Implementera klassen `Complex` nedan som representerar komplexa tal:

```
class Complex(val re: Double, val im: Double):
  def r  = ???  // absolutbeloppet
  def fi = ???  // vinkeln i radianer
  def +(other: Complex): Complex = ???  // resultatet av addition
  var imSymbol = 'i'  // symbol för imaginärdel, används i toString
  override def toString = ???  // en strängrepresentation av talet
```

Exempel:
$z = 3 + 4i$



imaginär-delen är 4

real-delen är 3

Exempel: Klassen Complex

```
class Complex(val re: Double, val im: Double):
  def r  = math.hypot(re, im)
  def fi = math.atan2(im, re) // motstående sida först
  def +(other: Complex) = new Complex(re + other.re, im + other.im)
  var imSymbol = 'i'
  override def toString = s"$re + $im$imSymbol"
```

```
1  scala> val z1 = new Complex(3, 4)  // konstruktion av instans av Complex
2  z: Complex = 3.0 + 4.0i
3
4  scala> val polärForm = (z1.r, z1.fi)
5  polärForm: (Double, Double) = (5.0,0.6435011087932844)
6
7  scala> val z2 = Complex(1, 2)   // new behövs inte i Scala 3
8  z2: Complex = 1.0 + 2.0i
9
10 scala> z1 + z2
11 res0: Complex = 4.0 + 6.0i
```

https://scala-lang.org/api/3.x/scala/math.html#atan2-44b

Exempel: Principen om enhetlig access

```
class Complex(val re: Double, val im: Double):
  val r  = math.hypot(re, im)
  val fi = math.atan2(im, re)
  def +(other: Complex) = new Complex(re + other.re, im + other.im)
  var imSymbol = 'i'
  override def toString = s"$re + $im$imSymbol"
```

Exempel: Principen om enhetlig access

```
class Complex(val re: Double, val im: Double):
  val r  = math.hypot(re, im)
  val fi = math.atan2(im, re)
  def +(other: Complex) = new Complex(re + other.re, im + other.im)
  var imSymbol = 'i'
  override def toString = s"$re + $im$imSymbol"
```

- Efter som attributen `re` och `im` är oföränderliga, kan vi lika gärna ändra i klass-implementationen och göra om metoderna `r` och `fi` till **`val`**-variabler utan att klientkoden påverkas.
- Då anropas `math.hypot` och `math.atan2` bara en gång vid initialisering (och inte varje gång som med **`def`**).
- Vi skulle även kunna använda **`lazy val`** och då bara räkna ut `r` och `fi` om och när de verkligen refereras av klientkoden, annars inte.
- Eftersom klientkoden inte ser skillnad på metoder och variabler, kallas detta **principen om enhetlig access**. (Många andra språk har **inte** denna möjlighet, tex Java där metoder *måste* ha parenteser.)

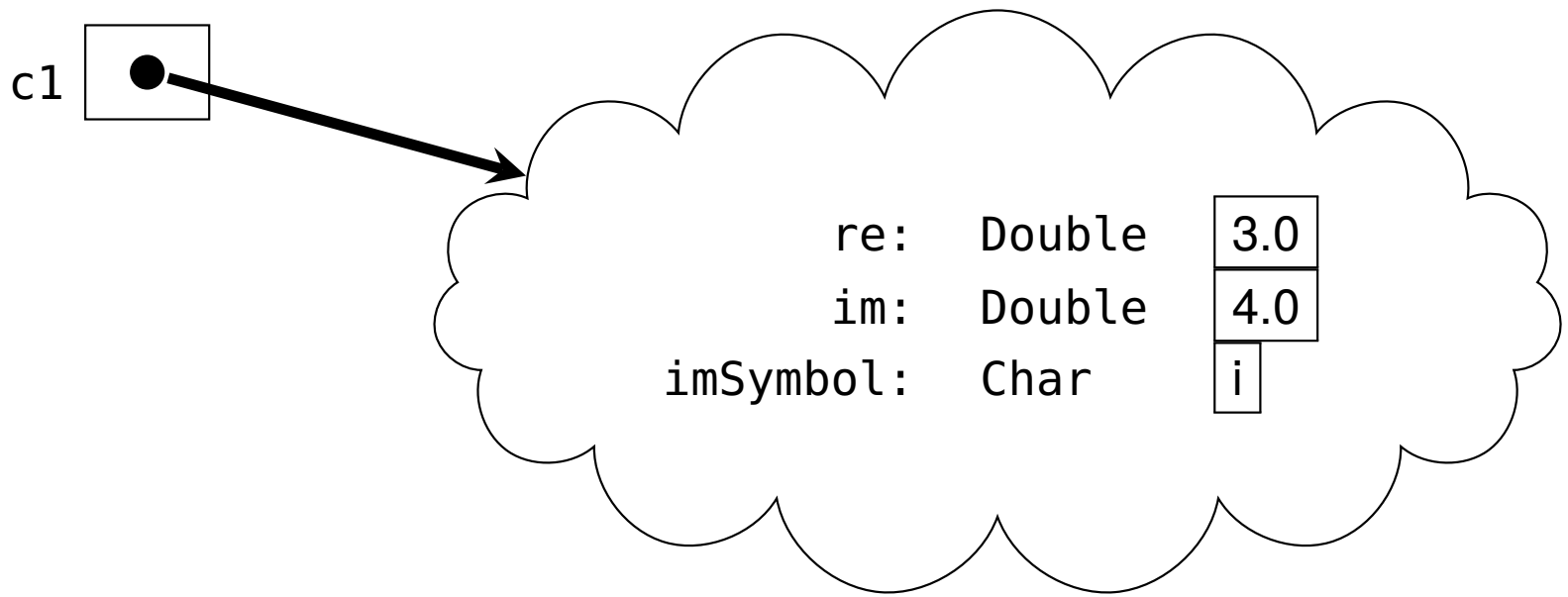
Olika sätt att skapa instanser

Instansiering med direkt användning av new

Instansiering genom **direkt användning** av **new**

(här första varianten av Complex med `r` och `fi` som metoder)

```
scala> val c1 = new Complex(3, 4)
```

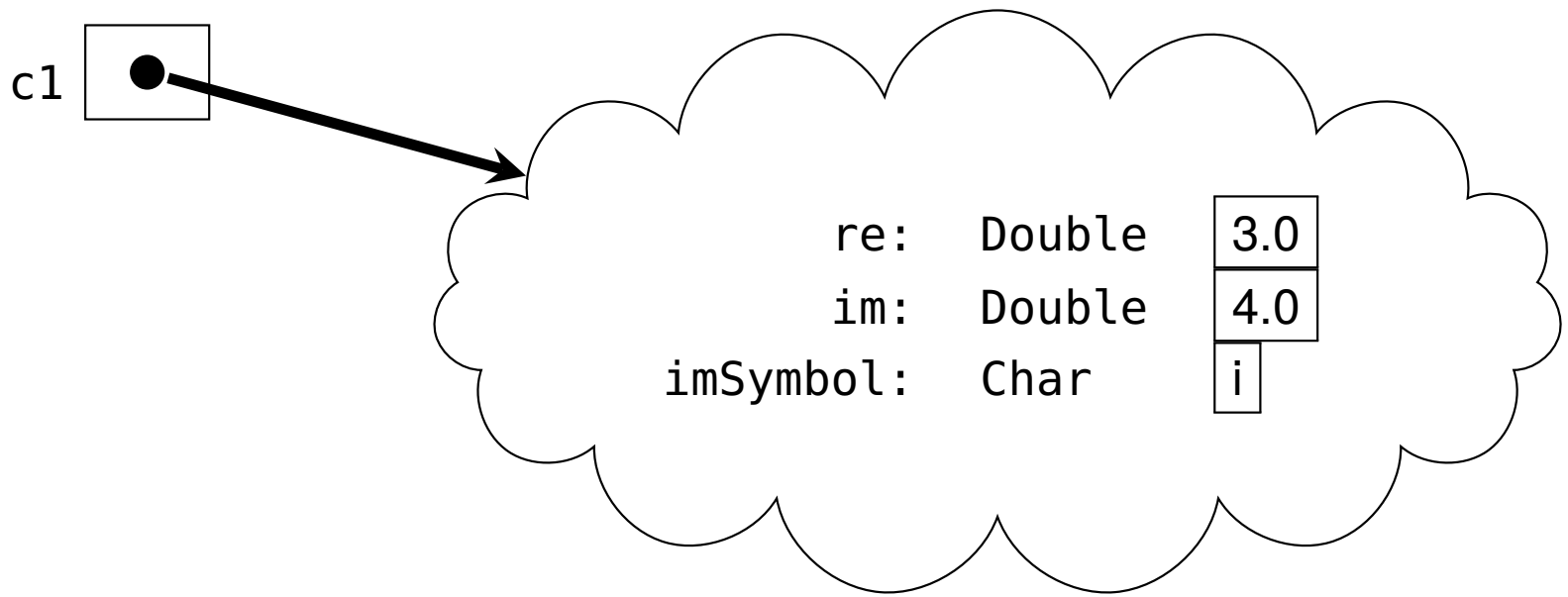


Instansiering med direkt användning av new

Instansiering genom **direkt användning** av **new**

(här första varianten av Complex med `r` och `fi` som metoder)

```
scala> val c1 = new Complex(3, 4)
```



Ofta vill man göra **indirekt** instansiering så att vi senare har friheten att ändra hur instansiering sker.

Indirekt instansiering med fabriksmetoder

En **fabriksmetod** är en metod som används för att instansiera objekt.

```
object MyFactory {
  def createComplex(re: Double, im: Double) = new Complex(re, im)
  def createReal(re: Double)                = new Complex(re, 0)
  def createImaginary(im: Double)           = new Complex(0, im)
}
```

Indirekt instansiering med fabriksmetoder

En **fabriksmetod** är en metod som används för att instansiera objekt.

```
object MyFactory {
  def createComplex(re: Double, im: Double) = new Complex(re, im)
  def createReal(re: Double)                = new Complex(re, 0)
  def createImaginary(im: Double)           = new Complex(0, im)
}
```

Instansiera **inte direkt**, utan **indirekt** genom användning av **fabriksmetoder**:

```
1  scala> import MyFactory.*
2
3  scala> createComplex(3, 4)
4  res0: Complex = 3.0 + 4.0i
5
6  scala> createReal(42)
7  res1: Complex = 42.0 + 0.0i
8
9  scala> createImaginary(-1)
10 res2: Complex = 0.0 + -1.0i
```

Hur förhindra direkt instansiering?

Om vi vill **förhindra direkt instansiering** kan vi göra primärkonstruktorn **privat**:

```
class Complex private (val re: Double, val im: Double):
  def r  = math.hypot(re, im)
  def fi = math.atan2(im, re)
  def +(other: Complex) = new Complex(re + other.re, im + other.im)
  var imSymbol = 'i'
  override def toString = s"$re + $im$imSymbol"
```

MEN... då går det ju **inte** längre att instansiera något alls! : (

```
scala> new Complex(3,4)
error:
 constructor Complex in class Complex cannot be accessed
```

Kompanjonsobjekt med indirekt instansiering

- Ett **kompanjonsobjekt** (eng. *companion object*) är ett singelobjekt som ligger i **samma kodfil** som en klass, och som har **samma namn** som klassen.
- Medlemmar i ett kompanjonsobjekt **får accessa privata** medlemmar i kompanjonsklassen (och vice versa) och kompanjonsobjektet får därför accessa privat konstruktor och kan göra **new**.
- Fabriksmetod + privat konstruktor: tillåt **enbart indirekt instansiering**.

```
class Complex private (val re: Double, val im: Double):
  def r  = math.hypot(re, im)
  def fi = math.atan2(im, re)
  def +(other: Complex) = new Complex(re + other.re, im + other.im)
  var imSymbol = 'i'
  override def toString = s"$re + $im$imSymbol"

object Complex:
  def apply(re: Double, im: Double) = new Complex(re, im) // new behövs här
  def real(re: Double)              = new Complex(re, 0)
  def imag(im: Double)              = new Complex(0, im)
```

- **new** behövs för att förhindra rekursivt anrop av `apply` och stack overflow

Användning av kompanjonsobjekt med fabriksmetoder

Nu kan vi **bara** instansiera **indirekt**! :)

```
scala> Complex.real(42.0)
res0: Complex = 42.0 + 0.0i

scala> Complex.imag(-1)
res1: Complex = 0.0 + -1.0i

scala> Complex.apply(3,4)
res2: Complex = 3.0 + 4.0i

scala> Complex(3,4)
res3: Complex = 3.0 + 4.0i

scala> new Complex(3, 4)
error:
     constructor Complex in class Complex cannot be accessed
```

Alternativa direktinstansieringar med default-argument

Med **default-argument** kan vi erbjuda **alternativa** sätt att direktinstansiera.

```
class Complex(val re: Double = 0, val im: Double = 0):
  def r  = math.hypot(re, im)
  def fi = math.atan2(im, re)
  def +(other: Complex) = new Complex(re + other.re, im + other.im)
  var imSymbol = 'i'
  override def toString = s"$re + $im$imSymbol"
```

```
1  scala> new Complex()
2  res0: Complex = 0.0 + 0.0i
3
4  scala> new Complex(re = 42)  //anrop med namngivet argument
5  res1: Complex = 42.0 + 0.0i
6
7  scala> new Complex(im = -1)
8  res2: Complex = 0.0 + -1.0i
9
10 scala> new Complex(1)
11 res3: Complex = 1.0 + 0.0i
```

Alternativa sätt att instansiera med fabriksmetod

Vi kan också erbjuda **alternativa** sätt att instansiera **indirekt** med fabriksmetoden `apply` i ett kompanjonsobjekt genom default-argument:

```
class Complex private (val re: Double, val im: Double):
  def r  = math.hypot(re, im)
  def fi = math.atan2(im, re)
  def +(other: Complex) = new Complex(re + other.re, im + other.im)
  var imSymbol = 'i'
  override def toString = s"$re + $im$imSymbol"

object Complex:
  def apply(re: Double = 0, im: Double = 0) = new Complex(re, im)
  def real(r: Double) = apply(re = r)
  def imag(i: Double) = apply(im = i)
  val zero = apply()
```

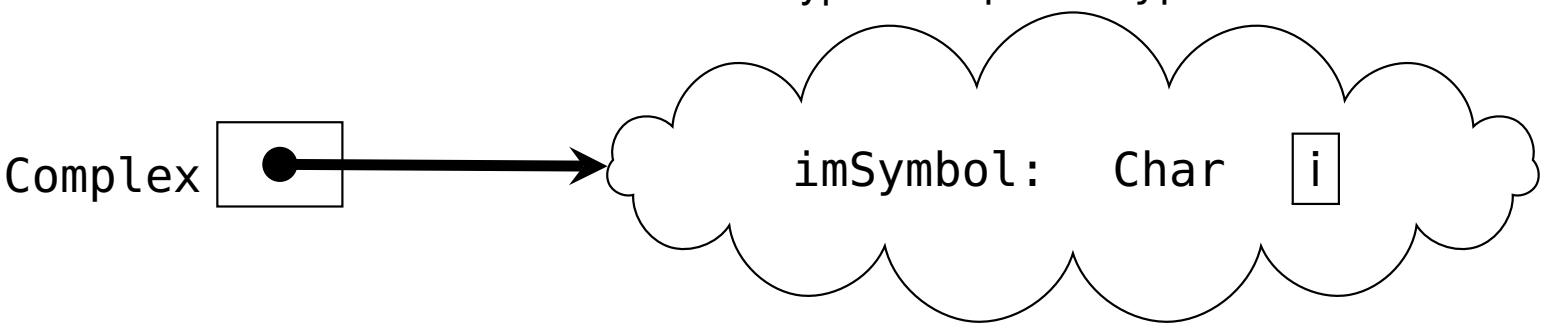
Medlemmar som bara behövs i en enda upplaga

Attributet `imSymbol` passar bättre att ha i **kompanjonsobjektet**, eftersom det räcker att ha **en enda upplaga**, som kan vara gemensam för alla objekt:

```
class Complex private (val re: Double, val im: Double):
  def r  = math.hypot(re, im)
  def fi = math.atan2(im, re)
  def +(other: Complex) = new Complex(re + other.re, im + other.im)
  override def toString = s"$re + $im${Complex.imSymbol}"

object Complex:
  var imSymbol = 'i'
  def apply(re: Double = 0, im: Double = 0) = new Complex(re, im)
  def real(r: Double) = apply(re = r)
  def imag(i: Double) = apply(im = i)
  val zero = apply()
```

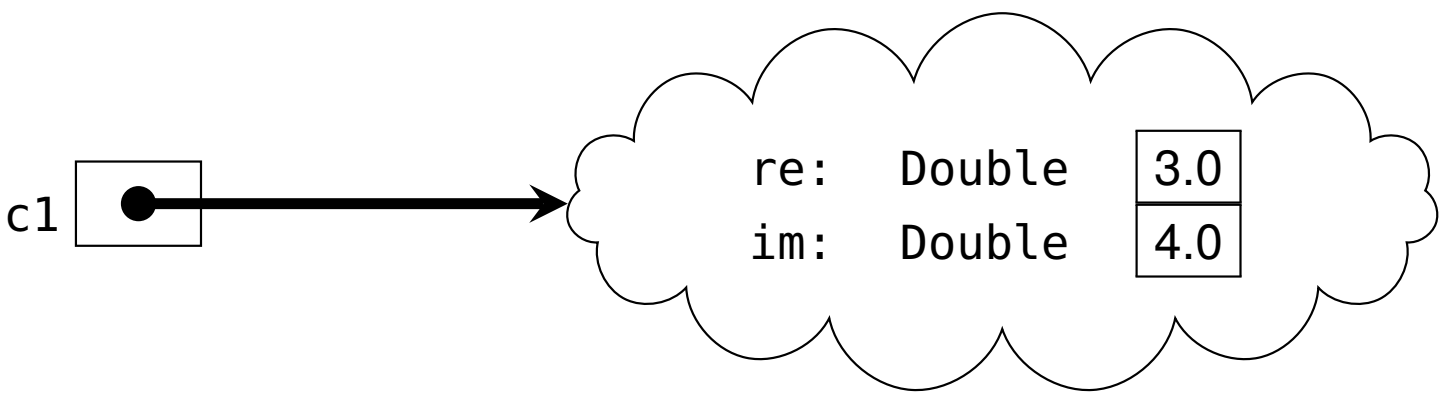
Medlemmar i singelobjekt är statiskt allokerade

Minnesplatsen för **attribut i singelobjekt** allokeras automatiskt en gång för alla, och kallas därför **statiskt** allokerad. Singelobjektets namn `Complex` utgör en statisk referens till den enda instansen och är av typen `Complex.type`.



Nu bereder vi inte plats för `imSymbol` i varenda **dynamiskt** allokerade instans:

```
scala> val c1 = Complex(3, 4)
```

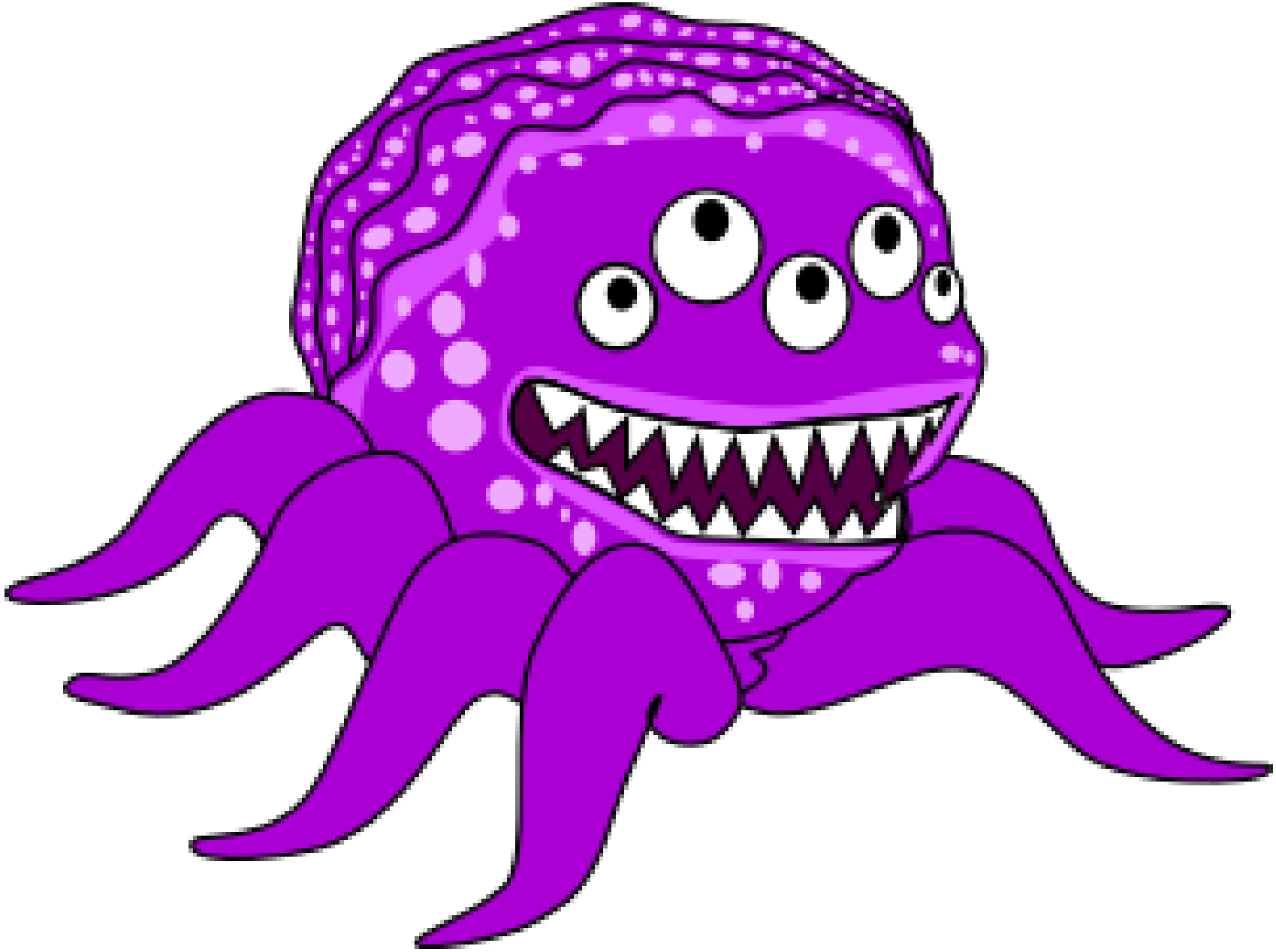


Attribut i kompanjonsobjekt användas för sådant som är gemensamt för alla instanser

Om vi ändrar på statiska `imSymbol` så ändras `toString` för **alla** dynamiskt allokerade instanser.

```
scala> val c1 = Complex(3, 4)
c1: Complex = 3.0 + 4.0i

scala> Complex.imSymbol = 'j'
Complex.imSymbol: Char = j

scala> val c2 = Complex(5, 6)
c2: Complex = 5.0 + 6.0j

scala> c1
res0: Complex = 3.0 + 4.0j
```

Case-klasser och fördelen med oföränderlighet



Övning: en läskig mutant

1. Skapa en klass med namnet `Mutant` som har ett förändringsbart attribut som klassparameter med namnet `i` av typen `Int` med default-argumentet `5`.

2. Deklarera två `val`-variabler som kallas `fem1` och `fem2` och som båda refererar till **samma** `Mutant`-instans.

En `Mutant`-instans där `i` kanske är fem.

3. Skriv kod som ändrar tillstånd via den ena mutantreferensen.
4. Syns ändringen via den andra mutantreferensen?

Case-klasser

Case-klasser är ett smidigt sätt att skapa **oföränderliga** datastrukturer. Med nyckelordet **`case`** framför **`class`** får du mycket ”**godis på köpet**”:

- Klassparametrar blir automatiskt publika[1] oföränderliga attribut och du slipper alltså skriva **`val`**.
- Du får en automatisk **toString** med klassens namn och värdet av alla **`val`**-attribut som ges av klassparametrarna
- och en **copy**-metod för att skapa nya, delvis förändrade instanser, med attributvärdena som defaultargument.
- Du får ett automatiskt kompanjonsobjekt med en fabriksmetod `apply` för indirekt instansiering där alla klassparametrarnas **`val`**-attribut initialiseras.

[1] alltså **inte** instansprivata som i vanliga klasser.

Case-klasser

Case-klasser är ett smidigt sätt att skapa **oföränderliga** datastrukturer. Med nyckelordet **`case`** framför **`class`** får du mycket ”**godis på köpet**”:

- Klassparametrar blir automatiskt publika[1] oföränderliga attribut och du slipper alltså skriva **`val`**.
- Du får en automatisk **toString** med klassens namn och värdet av alla **`val`**-attribut som ges av klassparametrarna
- och en **copy**-metod för att skapa nya, delvis förändrade instanser, med attributvärdena som defaultargument.
- Du får ett automatiskt kompanjonsobjekt med en fabriksmetod `apply` för indirekt instansiering där alla klassparametrarnas **`val`**-attribut initialiseras.
- ... och **mer därtill** men mer om det senare...

[1] alltså **inte** instansprivata som i vanliga klasser.

Exempel: oföränderliga case-klassen `Point`

```
case class Point(x: Double, y: Double)
```

```
scala> val p1 = Point(3, 4)
p1: Point = Point(3.0,4.0)

scala> val p2 = p1
p2: Point = Point(3.0,4.0)

scala> p1.x = 42
error: reassignment to val
```

Vi kan utan risk dela med oss av en referens till en oföränderlig klass – ingen kan ändra dess innehåll. (Jämför läskiga mutanten i tidigare exempel.)

Konstruktor

Vad är en konstruktor?

- En **konstruktor** är den kod som exekveras när klasser instansieras.
- Konstruktorn skapar ett nytt objekt i minnet vid varje anrop.
- I Scala **genererar kompilatorn** en **primärkonstruktor** åt dig med maskinkod som initialiserar alla attribut baserat på klassparametrarna som du deklarerat.

Vad är en konstruktor?

- En **konstruktor** är den kod som exekveras när klasser instansieras.
- Konstruktorn skapar ett nytt objekt i minnet vid varje anrop.
- I Scala **genererar kompilatorn** en **primärkonstruktor** åt dig med maskinkod som initialiserar alla attribut baserat på klassparametrarna som du deklarerat.
- I Scala **kan** man också skriva egna alternativa s.k. **hjälpkonstruktorer** (eng. *auxilliary constructor*), men det är **ovanligt**, eftersom man har möjligheten med fabriksmetoder i kompanjonsobjekt och default-argument.

Fördjupning: Hjälpkonstruktorer i Scala (ovanliga)

Fördjupning för kännedom:

- I Scala kan man skapa ett alternativ till primärkonstruktorn, en så kallad **hjälpkonstruktor** (eng. *auxilliary constructor*) genom att deklarera en metod med det speciella namnet `this`.
- Hjälpkonstruktorer **måste** börja med att anropa en **annan** konstruktor som står **före** i koden, till exempel primärkonstruktorn.

```
class Point(val x: Int, val y: Int, val z: Int): // primärkonstruktor
  def this(x: Int, y: Int) = this(x, y, 0) // anropa primärkonstruktorn
  def this(x: Int) = this(x, 0) // anropa hjälpkonstruktor
```

- Varför? Enklare att använda från **Java**-kod jämfört med apply i kompanjonsobjekt. (Men om din Scala-kod inte ska användas från Java så är detta onödigt: använd då hellre kompanjonsobjekt med fabriksmetod.)

Fördjupning: Användning av hjälpkonstruktor

```
scala> val p1 = Point(1)
p1: Point = Point@21312342

scala> val p2 = Point(1, 2)
p2: Point = Point@43254325

scala> val p3 = Point(1, 2, 3)
p3: Point = Point@346654
```

Fördjupning: Användning av hjälpkonstruktor

```
scala> val p1 = Point(1)
p1: Point = Point@21312342

scala> val p2 = Point(1, 2)
p2: Point = Point@43254325

scala> val p3 = Point(1, 2, 3)
p3: Point = Point@346654
```

Men man gör **mycket oftare** så här i Scala:

```
case class Point(x: Int, y: Int = 0, z: Int = 0)
```

Använd alltså hellre

- defaultargument i klassparametrar, eller
- fabriksmetoder i kompanjonsobjekt, antingen med default-argument eller överlagrade.

Referens saknas: `null`

Referens saknas: `null`

- I Java och många andra språk använder man ofta nyckelordet **`null`** för att representera att ett **värde saknas**.
- En referens som är **`null`** refererar inte till någon instans.
- Om du försöker referera till instansmedlemmar med punktnotation genom en referens som är **`null`** kastas ett **undantag** `NullPointerException`.
- Oförsiktig användning av **`null`** är en vanlig källa till **buggar**, som kan vara svåra att hitta och fixa.

Exempel: `null`

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

scala> var g: Gurka = null        // ingen instans allokerad än
var g: Gurka = null

scala> g.vikt
java.lang.NullPointerException

scala> g = Gurka(42)          // instansen allokeras
g: Gurka = Gurka@1ec7d8b3

scala> g.vikt
val res0: Int = 42

scala> g = null         // instansen kommer att destrueras av skräpsamlaren
```

- Scala har **`null`** av kompatibilitetsskäl, men det är brukligt att **endast** använda **`null`** om man anropar Java-kod.
- Scala erbjuder smidiga `Option`, `Some` och `None` för säker hantering av saknade värden; mer om detta kommande vecka.

Initialisering av attribut

Defaultvärden under pågående konstruktion

`Int`	`0`
`Double`	`0.0`
`Float`	`0.0F`
`Long`	`0L`
`Short`	`0.toShort`
`Byte`	`0.toByte`
`Char`	`0.toChar`
`Boolean`	`false`
Alla referenstyper, tex. `String`	**`null`**

Problem med initialisering av attribut vid konstruktion

```
class InitBug1:
  val HEJ = hej.toUpperCase
  val hej = "hej"

class InitBug2:
  val b = a
  val a = 10

class InitBug3:
  val hej2 = hej1
  val hej1 = "hej"
```

```
1 scala> val ib1 = new InitBug1
2 scala> val ib2 = new InitBug2
3 scala> ib2.b
4 scala> val ib3 = new InitBug3
5 scala> ib3.hej2
```

Vad händer?

Vilka värden har attribut medan konstruktion pågår?

```
class InitBug1:
  val HEJ = hej.toUpperCase
  val hej = "hej"

class InitBug2:
  var b = a
  var a = 10

class InitBug3:
  val hej2 = hej1
  val hej1 = "hej"
```

```
1 scala> val ib1 = new InitBug1   // java.lang.NullPointerException
2 scala> val ib2 = new InitBug2
3 scala> ib2.b                    // val res0: Int = 0  // WHAT????
4 scala> val ib3 = new InitBug3
5 scala> ib3.hej2                 // val res1: String = null  //WHAT???
```

Varför? Vad finns det för lösningar?

Hur undvika initialiseringsproblem vid konstruktion?

Några tips för att undvika initialiseringsproblem av attribut:

- Ändra om möjligt ordningen på attribut-deklarationer
- Använd om möjligt i stället **`lazy val`** (init sker senare)
- Använd om möjligt i stället **`def`** (evaluering vid varje anrop)
- Använd denna kompilatoroption för att få hjälp med varningar vid risk för initialiseringsproblem: `-Wsafe-init`

Hur undvika initialiseringsproblem vid konstruktion?

Några tips för att undvika initialiseringsproblem av attribut:

- Ändra om möjligt ordningen på attribut-deklarationer
- Använd om möjligt i stället **`lazy val`** (init sker senare)
- Använd om möjligt i stället **`def`** (evaluering vid varje anrop)
- Använd denna kompilatoroption för att få hjälp med varningar vid risk för initialiseringsproblem: `-Wsafe-init`

Om du **verkligen** behöver ha ett oinitialiserat värde:

```
class Box:
  private var x: String = scala.compiletime.uninitialized // tydliggör null-risk
  def get: String = if x != null then x else ""           // glöm ej kolla null
  def getOrElse(alt: String): String = if x != null then x else alt
  def set(value: String): Unit = x = value
```

Försök att **undvika `null`** om det går eftersom det ger stor risk för buggar!

(I ovan fiktiva exempel hade vi kunnat undvika detta enkelt genom att ge x startvärdet `""` i stället för null. En sådan lösing förutsätter att det finns en rimlig representation av ett saknat värde. Mer om hantering av saknade värden senare...)

Referensen `this`

Referensen `this`

- Nyckelordet `this` ger en referens till den aktuella instansen.

```
scala> class Gurka(var vikt: Int){def jagSjälv = this}

scala> val g = Gurka(42)
val g: Gurka = Gurka@5ae9a829

scala> g.jagSjälv
val res0: Gurka = Gurka@5ae9a829

scala> g.jagSjälv.vikt
val res1: Int = 42

scala> g.jagSjälv.jagSjälv.vikt
val res2: Int = 42
```

- Referensen `this` används ofta för att komma runt ”namnkrockar” där variabler med samma namn gör så att den ena variabeln inte syns.

Getters och setters

Getters och setters

- I många språk (t.ex. Java, Python) finns inget motsvarande nyckelord **`val`** som garanterar oföränderliga attributreferenser. [2]
- Därför gör man i dessa språk nästan alltid alla attribut **privata** för att förhindra att de ändras på ett okontrollerat sätt.
- Därför är det normalt att införa metoder som kallas **getters** och **setters**, som används för att **indirekt** läsa och uppdatera **attribut**.
- Dessa metoder känns i många språk igen genom konventionen att de heter något som börjar med **get** respektive **set**. (Men **ej** vanligt i Scala.)
- Med **indirekt access** av attribut kan man åstadkomma **flexibilitet**, så att implementationen kan ändras utan att ändra i klientkoden:
  - man kan t.ex. i efterhand ändra representation av de privata attributen eftersom all access sker genom getters och setters.
- Man kan åstadkomma **oföränderliga** datastrukturer där attributreferenserna inte förändras efter allokering om klassen **inte** erbjuder en **setter** för privata attribut.

[2] Java har visserligen **`final`** men det är annorlunda som vi ska se senare.

Java-exempel: Klassen JPerson

Indirekt access av **privata** attribut:

```
public class JPerson {
    private String name;
    private int age = 0;

    public JPerson(String name){
        //namnkrock fixas med this
        this.name = name;
    }

    public String getName(){
        return name;
    }

    public int getAge(){
        return age;
    }

    public void setAge(int age){
        this.age = age;
    }
}
```

```
> scala repl .
scala> val p = JPerson("Björn")
val p: JPerson = JPerson@7e774085

scala> p.getAge
val res0: Int = 0

scala> p.setAge(42)

scala> p.getAge
val res1: Int = 42

scala> p.age
-- Error:
p.age
^^^^^
value age is not a member of JPerson
```

Motsvarande JPerson i Scala

Så här brukar man åstadkomma ungefär motsvarande i Scala:

```
class Person(val name: String):
  var age = 0
```

Notera att alla attribut här är **publika**.

Förhindra felaktiga attributvärden med setters

Med hjälp av **setters** kan vi förhindra **felaktig** uppdatering av attributvärden, till exempel **negativ ålder** i klassen `JPerson` i Java:

```
    public void setAge(int age){
        if (age >= 0) {
            this.age = age;
        } else {
            this.age = 0;
        }
    }
```

Hur kan vi åstadkomma **motsvarande i Scala**?

Förhindra felaktiga attributvärden med setters

Med hjälp av **setters** kan vi förhindra **felaktig** uppdatering av attributvärden, till exempel **negativ ålder** i klassen `JPerson` i Java:

```
    public void setAge(int age){
        if (age >= 0) {
            this.age = age;
        } else {
            this.age = 0;
        }
    }
```

Hur kan vi åstadkomma **motsvarande i Scala**?
Antag att vi började med nedan variant, men **ångrar** oss och sedan vill införa funktionalitet som förhindrat negativ ålder **utan att ändra i klientkod**:

```
class Person(val name: String):
  var age = 0
```

Om vi inför en ny metod `setAge` och gör attributet age privat så funkar det **inte** längre att skriva `p.age = 42` och vi ”kvaddar” klientkoden! `:(`

Getters och setters i Scala

- Principen om **enhetlig access** tillsammans med **specialsyntax** för **setters** kommer till vår räddning!
- En **setter** kan i Scala skapas med **procedur vars namn slutar med** _=

Getters och setters i Scala

- Principen om **enhetlig access** tillsammans med **specialsyntax** för **setters** kommer till vår räddning!
- En **setter** kan i Scala skapas med **procedur vars namn slutar med** `_=`
- I Scala kan man utan att kvadda klientkod införa getter+setter så här:

```
class Person(val name: String): // ändrad implementation men samma access
  private var myPrivateAge = 0
  def age = myPrivateAge          // getter
  def age_=(a: Int): Unit =       // setter
    if a >= 0 then myPrivateAge = a else myPrivateAge = 0
```

Getters och setters i Scala

- Principen om **enhetlig access** tillsammans med **specialsyntax** för **setters** kommer till vår räddning!
- En **setter** kan i Scala skapas med **procedur vars namn slutar med** `_=`
- I Scala kan man utan att kvadda klientkod införa getter+setter så här:

```
class Person(val name: String): // ändrad implementation men samma access
  private var myPrivateAge = 0
  def age = myPrivateAge          // getter
  def age_=(a: Int): Unit =       // setter
    if a >= 0 then myPrivateAge = a else myPrivateAge = 0
```

```
1  scala> val p = Person("Björn")
2  val p: Person = Person@28ac3dc3
3
4  scala> p.age = 42      // najs syntax om getter parad med setter enl ovan
5  val p.age: Int = 42
6
7  scala> p.age = -1      // nu förhindras negativ ålder
8  val p.age: Int = 0
```

Likhet

Referenslikhet eller innehållslikhet?

Det finns två **principiellt olika** sorters **likhet**:

- **Referenslikhet** (eng. *reference equality*): två referenser anses lika om de refererar till **samma instans** i minnet.
- **Innehållslikhet**, ä.k. strukturlikhet (eng. *structural equality*): två referenser anses lika om de refererar till objekt med **samma innehåll**.

Referenslikhet eller innehållslikhet?

Det finns två **principiellt olika** sorters **likhet**:

- **Referenslikhet** (eng. *reference equality*): två referenser anses lika om de refererar till **samma instans** i minnet.
- **Innehållslikhet**, ä.k. strukturlikhet (eng. *structural equality*): två referenser anses lika om de refererar till objekt med **samma innehåll**.
- I Scala finns flera metoder som testar likhet:
  - metoden `eq` testar **referenslikhet** och `r1.eq(r2)` ger **`true`** om `r1` och `r2` refererar till **samma** instans.
  - metoden `ne` testar referens**o**likhet och `r1.ne(r2)` ger **`true`** om `r1` och `r2` refererar till **olika** instanser.
  - metoden == som anropar metoden `equals` som default testar referenslikhet med `eq` men som **kan överskuggas** om man **själv vill bestämma** om det ska vara referenslikhet eller strukturlikhet.

Referenslikhet eller innehållslikhet?

Det finns två **principiellt olika** sorters **likhet**:

- **Referenslikhet** (eng. *reference equality*): två referenser anses lika om de refererar till **samma instans** i minnet.
- **Innehållslikhet**, ä.k. strukturlikhet (eng. *structural equality*): två referenser anses lika om de refererar till objekt med **samma innehåll**.
- I Scala finns flera metoder som testar likhet:
  - metoden `eq` testar **referenslikhet** och `r1.eq(r2)` ger `true` om `r1` och `r2` refererar till **samma** instans.
  - metoden `ne` testar referens**o**likhet och `r1.ne(r2)` ger `true` om `r1` och `r2` refererar till **olika** instanser.
  - metoden == som anropar metoden `equals` som default testar referenslikhet med `eq` men som **kan överskuggas** om man **själv vill bestämma** om det ska vara referenslikhet eller strukturlikhet.
- Scalas **standardbibliotek** och **grundtyperna** `Int`, `String` etc. testar **innehållslikhet** genom metoden ==

Exempel: referenslikhet och innehållslikhet

I Scalas standardbibliotek har man överskuggat `equals` så att metoden == ger test av **innehållslikhet** mellan instanser:

```
1  scala> val v1 = Vector(1,2,3)
2  v1: scala.collection.immutable.Vector[Int] = Vector(1, 2, 3)
3
4  scala> val v2 = Vector(1,2,3)
5  v2: scala.collection.immutable.Vector[Int] = Vector(1, 2, 3)
6
7  scala> v1 eq v2                //referenslikhetstest: olika instanser
8  res0: Boolean = false
9
10 scala> v1 ne v2
11 res1: Boolean = true
12
13 scala> v1 == v2                //innehållslikhetstest: samma innehåll
14 res2: Boolean = true
15
16 scala> v1 != v2
17 res3: Boolean = false
```

Referenslikhet och egna klasser

Om du inte gör något speciellt med dina egna klasser så ger metoden == test av **referenslikhet** mellan instanser:

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

scala> val g1 = new Gurka(42)
g1: Gurka = Gurka@2cc61b3b

scala> val g2 = new Gurka(42)
g2: Gurka = Gurka@163df259

scala> g1 == g2       // samma innehåll men olika instanser
res0: Boolean = false

scala> g1.vikt == g2.vikt
res1: Boolean = true
```

Case-klasser ger innehållslikhet

Förutom annat ”godis på köpet” får du med `case class` även detta:

- Metoden == ger **innehållslikhet** (och inte referenslikhet).

Likhet och case-klasser

Metoden `equals` är i case-klasser automatiskt överskuggad så att metoden == ger test av strukturlikhet.

```
1  scala> case class Gurka(vikt: Int)
2
3  scala> val g1 = Gurka(42)
4  g1: Gurka = Gurka(42)
5
6  scala> val g2 = Gurka(42)
7  g2: Gurka = Gurka(42)
8
9  scala> g1 eq g2          // olika instanser
10 res0: Boolean = false
11
12 scala> g1 == g2          // samma innehåll!
13 res1: Boolean = true
```

Sammanfattning case-klass-godis

Kom-ihåg-lista med ”godis” i **`case`**-klasser så här långt:

1. klassparametrar blir **`val`**-attribut
2. najs toString
3. automatisk fabriksmetod `apply` i kompanjonsobjekt
4. == ger innehållslikhet (eng. *structural equality*)

Sammanfattning case-klass-godis

Kom-ihåg-lista med ”godis” i **`case`**-klasser så här långt:

1. klassparametrar blir **`val`**-attribut
2. najs toString
3. automatisk fabriksmetod `apply` i kompanjonsobjekt
4. == ger innehållslikhet (eng. *structural equality*)

   ...

Men vi har inte sett allt godis än:
Mönstermatchning (mer om det senare).

Implementation saknas: ???

Implementation saknas: ???

- Ofta vill man bygga kod iterativt och steg för steg lägga till olika funktionalitet.
- Standardfunktionen ??? ger vid anrop undantaget `NotImplementedError` och kan användas på platser i koden där man ännu inte är färdig.
- ??? tillåter **kompilering av ofärdig kod**.

Implementation saknas: ???

- Ofta vill man bygga kod iterativt och steg för steg lägga till olika funktionalitet.
- Standardfunktionen ??? ger vid anrop undantaget **`NotImplementedError`** och kan användas på platser i koden där man ännu inte är färdig.
- ??? tillåter **kompilering av ofärdig kod**.
- Undantag har bottentypen `Nothing` som är subtyp till *alla* typer och kan därmed tilldelas referenser av godtycklig typ.

```
scala> lazy val sprängsSnart: Int = ???

scala> sprängsSnart + 42
scala.NotImplementedError: an implementation is missing
```

Exempel: ofärdig kod

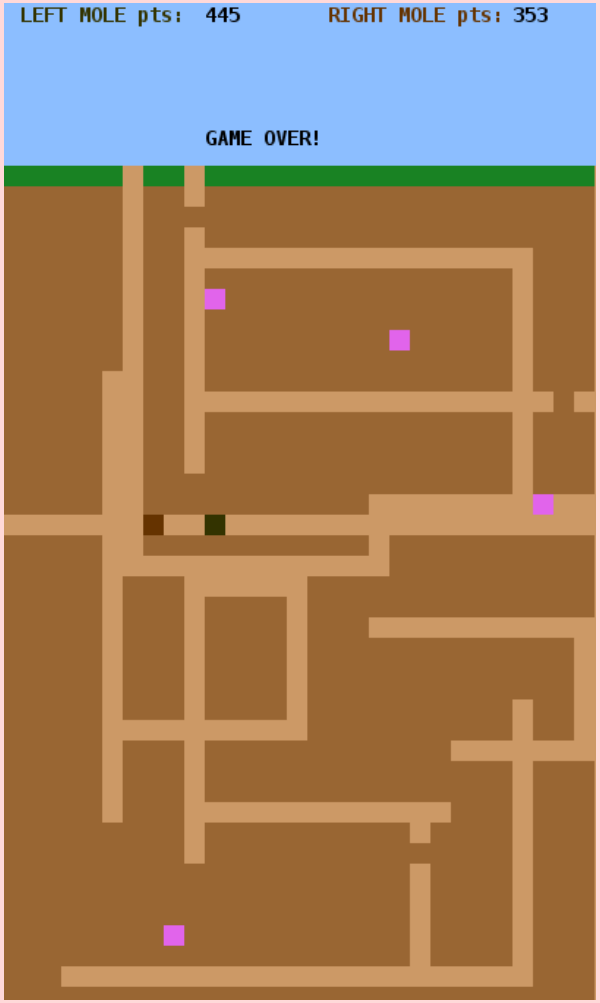
```
case class Person(name: String, age: Int):
  def ärTonåring = age >= 13 && age <= 19
  def ärUng = !ärGammal
  def ärGammal: Boolean = ???   //implementation ännu ej klar
```

```
scala> Person("Björn", 51).ärTonåring
res23: Boolean = false

scala> Person("Sandra", 39).ärUng
scala.NotImplementedError: an implementation is missing
```

Övning: `classes`

- Kunna deklarera klasser med klassparametrar.
- Kunna skapa instanser med och utan **`new`**.
- Kunna ge argument vid instansiering.
- Förstå innebörden av referensvariabler och värdet **`null`**.
- Kunna använda nyckelordet **`private`** för att styra synlighet av attribut och konstruktorparametrar.
- Förstå syftet med getters och setters.
- Kunna förklara accessregler för kompanjonsobjekt.
- Kunna skapa fabriksmetod i kompanjonsobjekt.
- Känna till nyttan med en privat konstruktor.
- Förstå skillnaden mellan referenslikhet och strukturlikhet.
- Känna till skillnaden mellan == och eq, samt != och ne.
- Kunna förklara hur case-klasser hanterar instansiering.
- Känna till hur case-klasser hanterar likhet.

Lab `blockbattle` redovisas NÄSTA läsvecka 6



- Ett arkadspel för TVÅ spelare.
- Nu med TVÅ blockmullvader!
- Programmet `blockbattle` är **mer omfattande** jmf m tidigare labbar.
- Labben omfattar TVÅ veckors förel. + övn.: klasser (w05) & mönster (w06).
- Det normala är registrering ”kompletteras” i SAM på `blockbattle0` under w05.
- Läs igenom labben innan du gör övningarna.
- Kod från övningarna behövs till labben.
- OBS! Labbtider är **schemalagda som vanligt** under läsvecka 5. Om du mot förmodan hinner redovisa redan denna vecka så ska du **ändå närvara** och t.ex. jobba med **extrauppgifter**.
- Dra nytta av undervisningen så att du lär dig så mycket som möjligt!