EDAA45 Programmering, grundkurs

Läsvecka 6: Klasser

Björn Regnell

Datavetenskap, LTH

Lp1-2, HT 2016

6 Klasser

- Vad är en klass?
- Olika sätt att skapa instanser
- Referens saknas: `null`
- Synlighet
- Klasser i Java
- Getters och setters
- Implementation saknas: ???
- Klass-specifikationer
- Likhet
- Case-klasser
- Grumligt-lådan
- Veckans uppgifter

Vad är en klass?

Vad är en klass?

- En klass är en mall för att skapa objekt.
- Objekt skapas med `new Klassnamn` och kallas för **instanser** av klassen `Klassnamn`.
- En klass innehåller medlemmar (eng. *members*):
  - **attribut**, kallas även fält (eng. *field*): `val`, `lazy val`, `var`
  - **metoder**, kallas även operationer: `def`
- Varje instans har sin uppsättning värden på attributen (fälten).

Vad är en klass?

Metafor: En klass liknar en **stämpel**

- En stämpel kan tillverkas – motsvarar deklaration av klassen.
- Det händer inget förrän man stämplar – motsvarar **new**.
- Då skapas avbildningar – motsvarar instanser av klassen.

Klassdeklarationer och instansiering

- Syntax för deklaration av klass:

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

- Exempel **deklaration**:

```
class Klassnamn(val attribut1: Int, attribut2: String){
  val attribut3: Double = 42.0              //publikt oföränderligt attribut
  private var attribut3: Boolean = false    //privat medlem syns inte utåt
  def metod(parameter: Int) = parameter + 1 //funktion i klass kallas metod
  lazy val attr4 = Vector.fill(100000)(42.0)     //fördröjd initialisering
}
```

- Parametrar initialiseras med de argument som ges vid **new**.
- Exempel **instansiering** med argument för initialisering av klassparametrar:

```
val instansReferens = new Klassnamn(42, "hej")
```

- Attribut blir **publika** (alltså synliga utåt) om inte modifieraren **private** anges.
- Parametrar som inte föregås av modifierare (t.ex. private val, val, var) blir **attribut** som är: **private[this] val** och bara synliga i **denna** instans.

Exempel: Klassen Complex i Scala

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

```
1  scala> val c1 = new Complex(3, 4)
2  c1: Complex = 3.0 + 4.0i
3
4  scala> val polarForm = (c1.r, c1.fi)
5  polarForm: (Double, Double) = (5.0,0.6435011087932844)
6
7  scala> val c2 = new Complex(1, 2)
8  c2: Complex = 1.0 + 2.0i
9
10 scala> c1 + c2
11 res0: Complex = 4.0 + 6.0i
```

Exempel: Principen om enhetlig access

```
class Complex(val re: Double, val im: Double){
  val r  = math.hypot(re, im)
  val fi = math.atan2(re, im)
  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(re, im)
  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.)

Exempel: Motsvarande klass JComplex i Java

```
public class JComplex {            // man kan ej deklarera klassparametrar i Java
    private double re;             // initialiseras i konstruktorn nedan
    private double im;             // initialiseras i konstruktorn nedan
    public  char   imSymbol = 'i'; // publikt attribut (inte vanligt i Java)

    public JComplex(double real, double imag){  // konstruktor, anropas vid new
        re = real;
        im = imag;
    }

    public double getRe(){  // en så kallad "getter" som ger attributvärdet, förhindra förändring av re
        return re;
    }

    public double getIm(){ return im; }  // ej bruklig formattering i Java

    public double getR(){
        return Math.hypot(re, im);
    }

    public double getFi(){
        return Math.atan2(re, im);
    }

    public JComplex add(JComplex other){
        return new JComplex(re + other.getRe(), im + other.getIm());
    }

    @Override public String toString(){
        return re + " + " + im + imSymbol;
    }
}
```

Exempel: Använda JComplex i Scala-kod

```
$ javac JComplex.java
$ scala
Welcome to Scala 2.11.8 (Java HotSpot(TM) 64-Bit Server VM, Java 1.8.0_66).
Type in expressions for evaluation. Or try :help.

scala> val jc1 = new JComplex(3, 4)
jc1: JComplex = 3.0 + 4.0i

scala> val polarForm = (jc1.getR, jc1.getFi)
polarForm: (Double, Double) = (5.0,0.6435011087932844)

scala> val jc2 = new JComplex(1, 2)
jc2: JComplex = 1.0 + 2.0i

scala> jc1 add jc2
res0: JComplex = 4.0 + 6.0i
```

- Metoder kan inte heta + i Java så vi döper metoden till add.

Exempel: Använda JComplex i Java-kod

```
public class JComplexTest {
    public static void main(String[] args){
        JComplex jc1 = new JComplex(3,4);
        String polar = "(" + jc1.getR() + ", " + jc1.getFi() + ")";
        System.out.println("Polär form: " + polar);
        JComplex jc2 = new JComplex(1,2);
        System.out.println(jc1.add(jc2));
    }
}
```

- Tupler finns inte i Java, så det går inte på ett enkelt sätt att i Java skapa par av värden som i Scala; ovan görs polär form till en sträng för utskrift.
- Operatornotation för metoder finns inte i Java, så man måste i Java använda punktnotation och skriva: `jc1.add(jc2)`

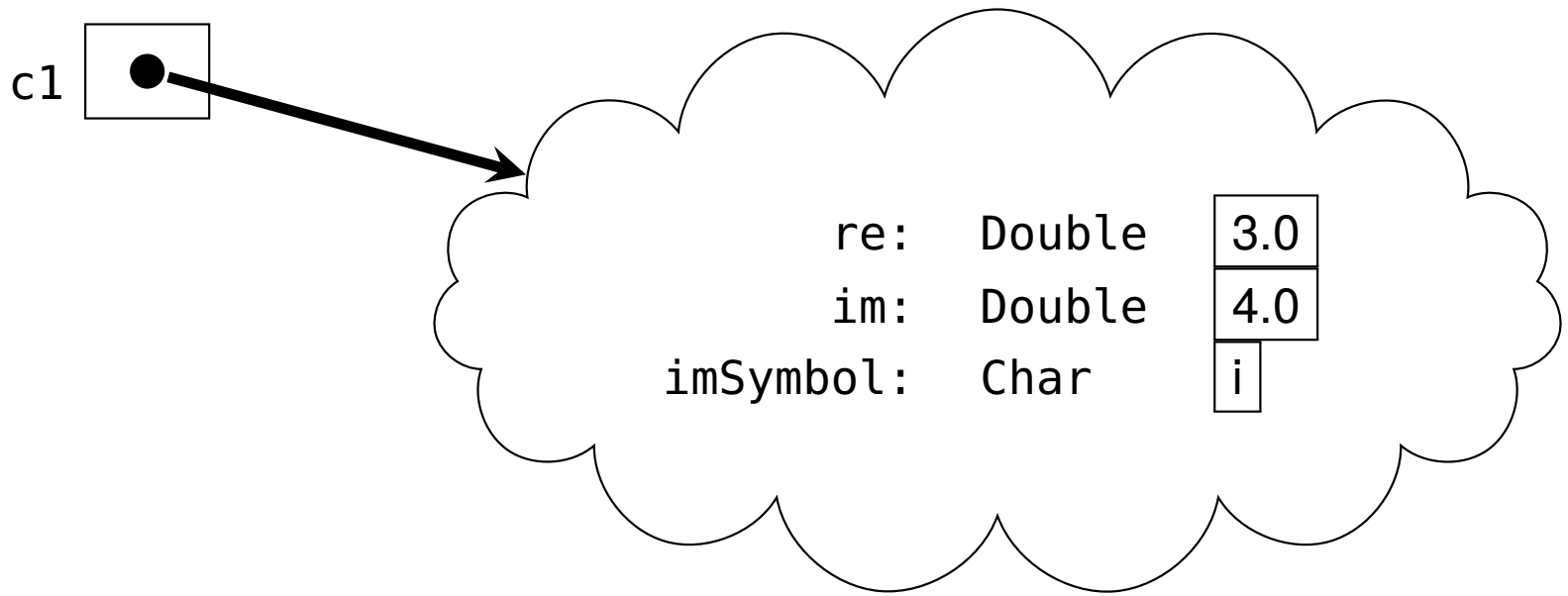
Olika sätt att skapa instanser

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)
```

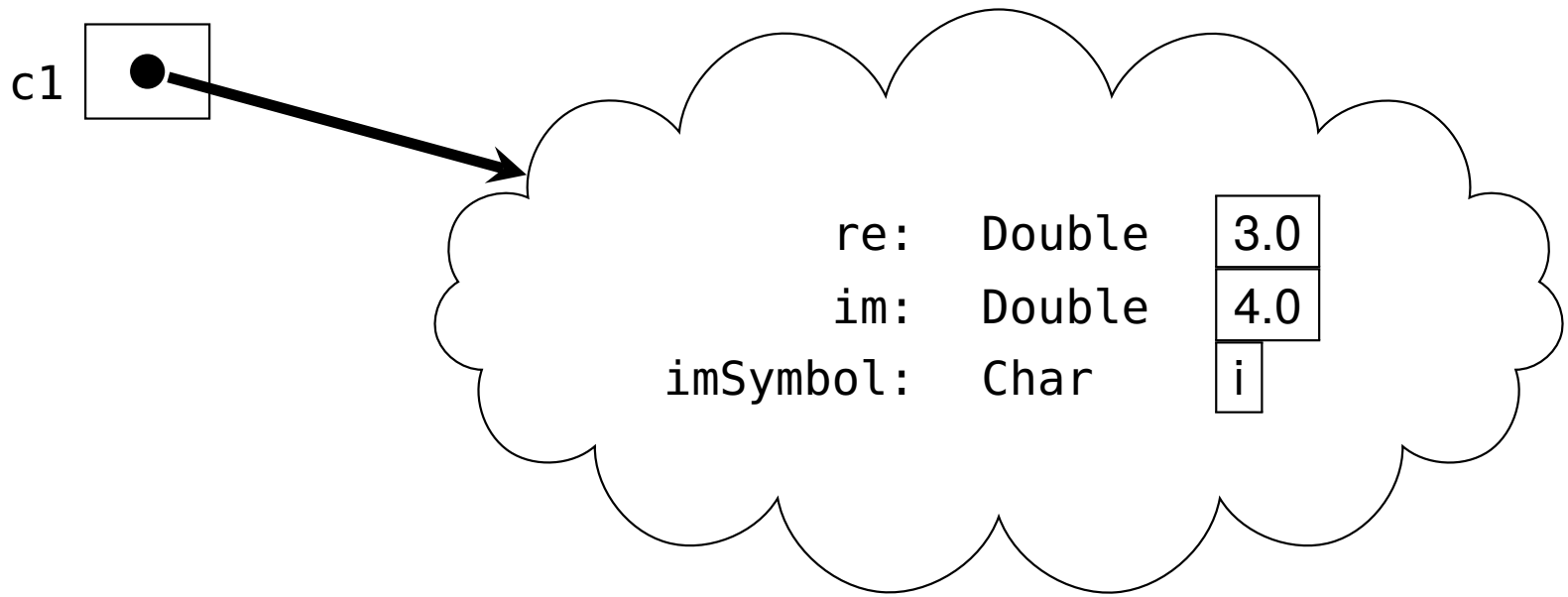


Instansiering med direkt användning av new

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(re, im)
  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 kan förhindra direkt instansiering

- Ett **kompanjonsobjekt** är ett objekt som ligger i samma kodfil som en klass och har samma namn som klassen.
- Medlemmar i ett kompanjonsobjekt **får accessa privata** medlemmar i kompanjonsklassen (och vice versa).

```
class Complex private (val re: Double, val im: Double){
  def r  = math.hypot(re, im)
  def fi = math.atan2(re, im)
  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)
  def real(re: Double)              = new Complex(re, 0)
  def imag(im: Double)              = new Complex(0, im)
}
```

Kompanjonsobjekt som fabriksmetod

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

```
scala> Complex.real(42.0)

scala> Complex.imag(-1)

scala> Complex.apply(3,4)

scala> Complex(3,4)
```

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(re, im)
  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(re, im)
  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=a)
  def imag(i: Double) = apply(im=b)
  val zero = apply()
}
```

Medlemmar som bara finns i en upplaga

Medlemmen `imSymbol` passar bättre att ha i kompanjonsobjektet, eftersom det räcker att ha en enda upplaga av denna medlem.

```
class Complex private (val re: Double, val im: Double){
  def r  = math.hypot(re, im)
  def fi = math.atan2(re, im)
  def +(other: Complex) = new Complex(re + other.re, im + other.im)
  override def toString = s"$re + $im$imSymbol"
}
object Complex {
  var imSymbol = 'i'
  def apply(re: Double = 0: im: Double = 0) = new Complex(re, im)
  def real(r: Double) = apply(re=a)
  def imag(i: Double) = apply(im=b)
  val zero = apply()
}
```

Konstruktor

- En **konstruktor** är den kod som exekveras när objekt skapas med **`new`**.
- I Scala **genererar** kompilatorn en **primärkonstruktor** med maskinkod som initialiserar alla attribut baserat på klassparamtetrar och **`val`**- och **`var`**-deklarationer.
- I Java **måste** man **själv** skriva alla konstruktorer och med speciell syntax. Man kan ha många olika alternativa konstruktorer.
- I Scala **kan** man skriva egna (alternativa) konstrukturer med speciell syntax, men det är inte vanligt, eftersom man har möjligheten med fabriksmetoder i kompanjonsobjekt och default-argument (vilka saknas i Java).

Skräpsamling

Destruktor

Referens saknas: `null`

Referens saknas: `null`

Synlighet

Synlighet

definiera/förklara: private private[this]

Synlighet av klassparametrar i klasser & case-klasser

`private[this]` är ännu mer privat än `private`

```
class Hemlis(private val hemlis: Int) {
  def ärSammaSom(annan: Hemlis) = hemlis == annan.hemlis   // Funkar!
}

class Hemligare(private[this] val hemlis: Int) {
  def ärSammaSom(annan: Hemligare) = hemlis == annan.hemlis //KOMPILERINGSFEL
}
```

Vad händer om man inte skriver något? Olika för klass och case-klass:

```
class Hemligare(hemlis: Int) { // motsvarar private[this] val
  def ärSammaSom(annan: Hemligare) = hemlis == annan.hemlis //KOMPILERINGSFEL
}

case class InteHemlig(seMenInteRöra: Int) { // blir automatiskt val
  def ärSammaSom(annan: InteHemlig): Boolean =
    seMenInteRöra == annan.seMenInteRöra
}
```

Klasser i Java

Klasser i Java

Typisk utformning av Java-klass

Typisk ”anatomi” av en Java-klass:

```
class Klassnamn {
    attribut, normalt privata
    konstruktorer, normalt publika
    metoder: publika getters, och vid förändringsbara objekt även setters
    metoder: privata abstraktioner för internt bruk
    metoder: publika abstraktioner tänkta att användas av klientkoden
}
```

www.oracle.com/technetwork/java/codeconventions-141855.html#1852

Statiska medlemmar

Getters och setters

Getters och setters i Java

Getters och setters i Scala

Ändra attributrepresentation utan att påverka existerande kod

Antag att vi vill ädnra representation av vårt komplexa tal till att ha ett attribut Polar som är en punkt med polära koordinater.
Complex som polära koordinater i Java med privat attribut
Complex som polära koordinater med publika attribut om man har enhetlig access

Implementation saknas: ???

Implementation saknas: ???

Klass-specifikationer

Specifikationer av klasser i Scala

- Specifikationer av klasser innehåller information som *den som ska implementera* klassen behöver veta.
- Specifikationer innehåller liknande information som dokumentationen av klassen (scaladoc), som beskriver vad *användaren* av klassen behöver veta.

Specification **Person**

```
/** Encapsulate immutable data about a Person: name and age. */
case class Person(name: String, age: Int = 0){
  /** Tests whether this Person is more than 17 years old. */
  def isAdult: Boolean = ???
}
```

- Specifikationer av Scala-klasser utgör i denna kurs ofullständig kod som kan kompileras utan fel.
- Saknade implementationer markeras med ???
- Kommentarer utgör krav på implementationen.

Specifikationer av klasser och objekt

Specification **MutablePerson**

```
/** Encapsulates mutable data about a person. */
class MutablePerson(initName: String, initAge: Int){
  /** The name of the person. */
  def getName: String = ???

  /** Update the name of the Person */
  def setName(name: String): Unit = ???

  /** The age of this person. */
  def getAge: Int = ???

  /** Update the age of this Person */
  def setAge(age: Int): Unit = ???

  /** Tests whether this Person is more than 17 years old. */
  def isAdult: Boolean = ???

  /** A string representation of this Person, e.g.: Person(Robin, 25) */
  override def toString: String = ???
}
object MutablePerson {
  /** Creates a new MutablePerson with default age. */
  def apply(name: String): MutablePerson = ???
}
```

Specifikationer av Java-klasser

- Specificerar signaturer för konstruktorer och metoder.
- Kommentarerna utgör krav på implementationen.
- Används flitigt på extentor i EDA016, EDA011, EDA017...
- Javaklass-specifikationerna behöver kompletteras med metodkroppar och klassrubriker innan de kan kompileras.

`class Person`

```
/** Skapar en person med namnet name och åldern age. */
Person(String name, int age);

/** Ger en sträng med denna persons namn. */
String getName();

/** Ändrar denna persons ålder. */
void setAge(int age);

/** Anger åldersgränsen för när man blir myndig. */
static int adultLimit = 18;
```

Likhet

Referenslikhet eller strukturlikhet?

- eq
- ==

Case-klasser

Case-klasser erbjuder strukturlikhet

Case-klass-godis:

- najs toString
- slipper skriva new
- == ger strukturlikhet

Grumligt-lådan

Grumligt-lådan

Veckans uppgifter

Övning: `classes`

- Kunna deklarera klasser med klassparametrar.
- Kunna skapa objekt med **`new`** och konstruktorargument.
- Förstå innebörden av referensvariabler och värdet **`null`**.
- Förstå innebörden av begreppen instans och referenslikhet.
- Kunna använda nyckelordet **`private`** för att styra synlighet i primärkonstruktor.
- Förstå i vilka sammanhang man kan ha nytta av en privat konstruktor.
- Kunna implementera en klass utifrån en specikation.
- Förstå skillnaden mellan referenslikhet och strukturlikhet.
- Känna till hur case-klasser hanterar likhet.
- Förstå nyttan med att möjliggöra framtida förändring av attributrepresentation.
- Känna till begreppen getters och setters.
- Känna till accessregler för kompanjonsobjekt.
- Känna till skillnaden mellan == och eq, samt != versus ne.

Laboration: `turtlegraphics`

- Kunna skapa egna klasser.
- Förstå skillnaden mellan klasser och objekt.
- Förstå skillnaden mellan muterbara och omuterbara objekt.
- Förstå hur ett objekt kan innehålla referenser till objekt av andra klasser, och varför detta kan vara användbart.
- Träna på att fatta beslut om vilka datatyper som bäst passar en viss tillämpning.