

# EDAA45 Programmering, grundkurs

## Läsvecka 6: Klasser

Björn Regnell

Datavetenskap, LTH

Lp1-2, HT 2016

## 6 Klasser

- Vad är en klass?
- Instansiering: new
- 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**:

```
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

```
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 deklarerera 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){ // konstruktör, 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 från Scala

```
1 $ javac JComplex.java
2 $ scala
3 Welcome to Scala 2.11.8 (Java HotSpot(TM) 64-Bit Server VM, Java 1.8.0_66).
4 Type in expressions for evaluation. Or try :help.
5
6 scala> val jc1 = new JComplex(3, 4)
7 jc1: JComplex = 3.0 + 4.0i
8
9 scala> val polarForm = (jc1.getR, jc1.getFi)
10 polarForm: (Double, Double) = (5.0,0.6435011087932844)
11
12 scala> val jc2 = new JComplex(1, 2)
13 jc2: JComplex = 1.0 + 2.0i
14
15 scala> jc1 add jc2
16 res0: JComplex = 4.0 + 6.0i
```

# Exempel: Test av JComplex i Java

```
public class JComplexTest {  
    public static void main(String[] args){  
        JComplex jc1 = new JComplex(3,5);  
        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.
- Operatornotation för metoder finns inte i Java, så man måste i Java använda punktnotation och skriva:  
jc1.add(jc2)

# Instansiering: new

# Instansiering med new

Rita molbild med instanser av klassen Complex

# Instansiering med fabriksmetod

# Instansiering med default-argument



# Instansiering med alternativa fabriksmetoder

# Förändringsbar eller oföränderlig?

# Referens saknas: null

# Referens saknas: null

# Konstruktör

# Skräpsamling

Destruktor

# Synlighet

# Synlighet

definiera/förklara: `private private[this]`



# Kompanjonsobjekt

# 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

# Statiska medlemmar

# Getters och setters

# Getters och setters i Java

# Getters och setters i Scala



# Ändra attributrepresentation utan att påverka existerande kod

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 konstruktörer och metoder.
- Kommentarer utgör krav på implementationen.
- Används flitigt på extendor 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 deklarerera 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 specifikation.
- 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.