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
- Minneshantering
- Referens saknas: `null`
- Klasser i Java
- Referensen `this`
- Getters och setters
- Likhet
- Case-klasser och likhet
- Implementation saknas: ???
- Klass-specifikationer
- 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?

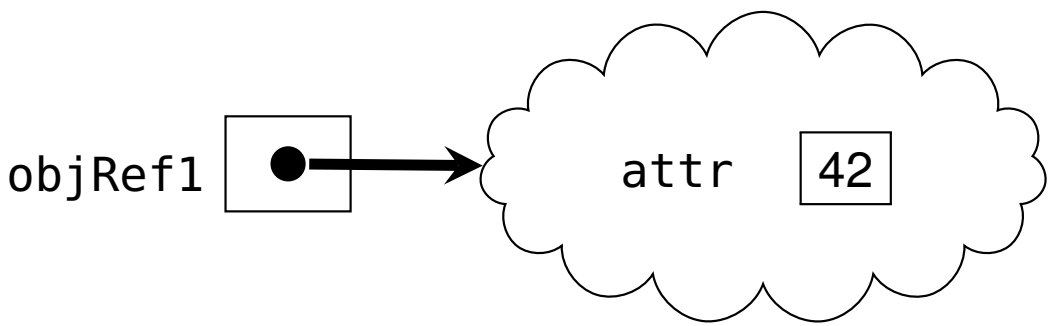
Metafor: En klass liknar en **stämpel**

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

Klass och instans

```
scala> class C { var attr = 42 }

scala> val objRef1 = new C
```

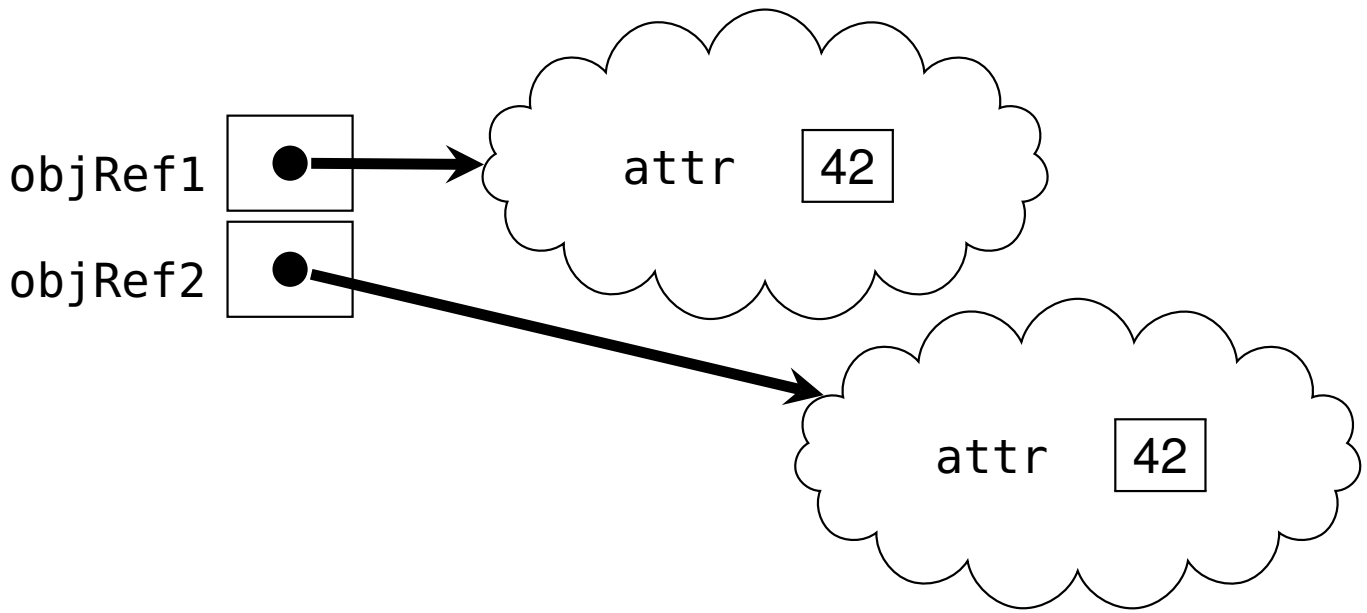


Klass och instans

```
scala> class C { var attr = 42 }

scala> val objRef1 = new C

scala> val objRef2 = new C
```



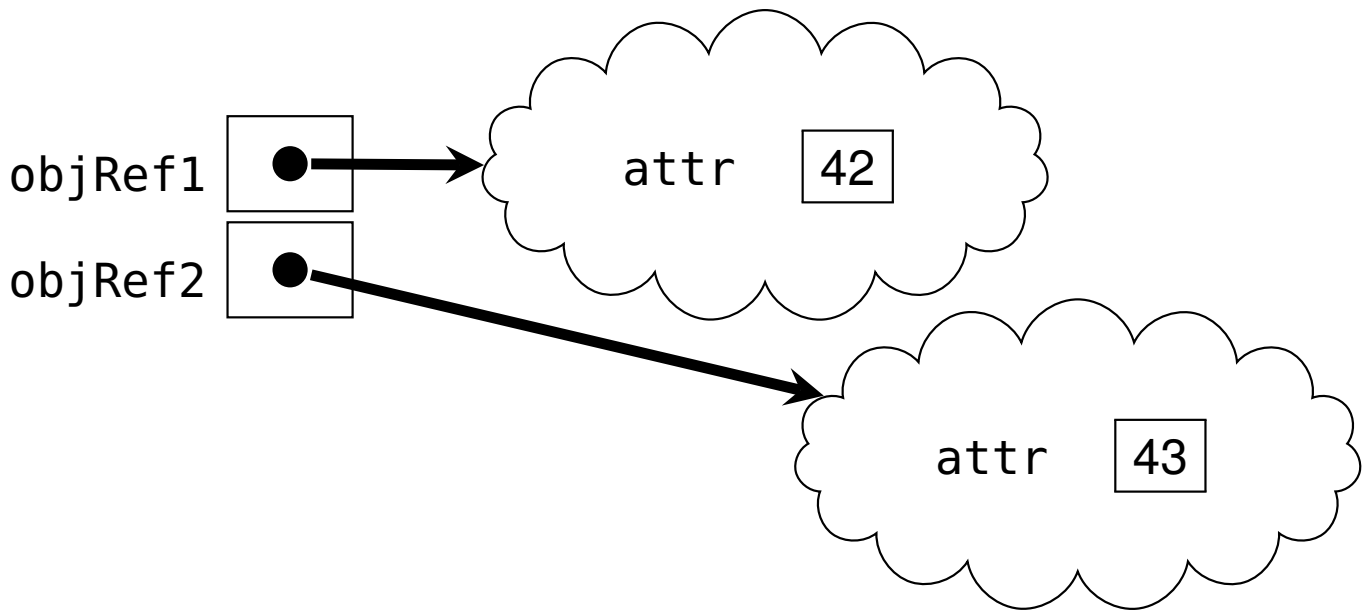
Klass och instans

```
scala> class C { var attr = 42 }

scala> val objRef1 = new C

scala> val objRef2 = new C

scala> objRef2.attr = 43
```



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 attribut4: 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

```
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 där metoder *måste* ha parenteser.)

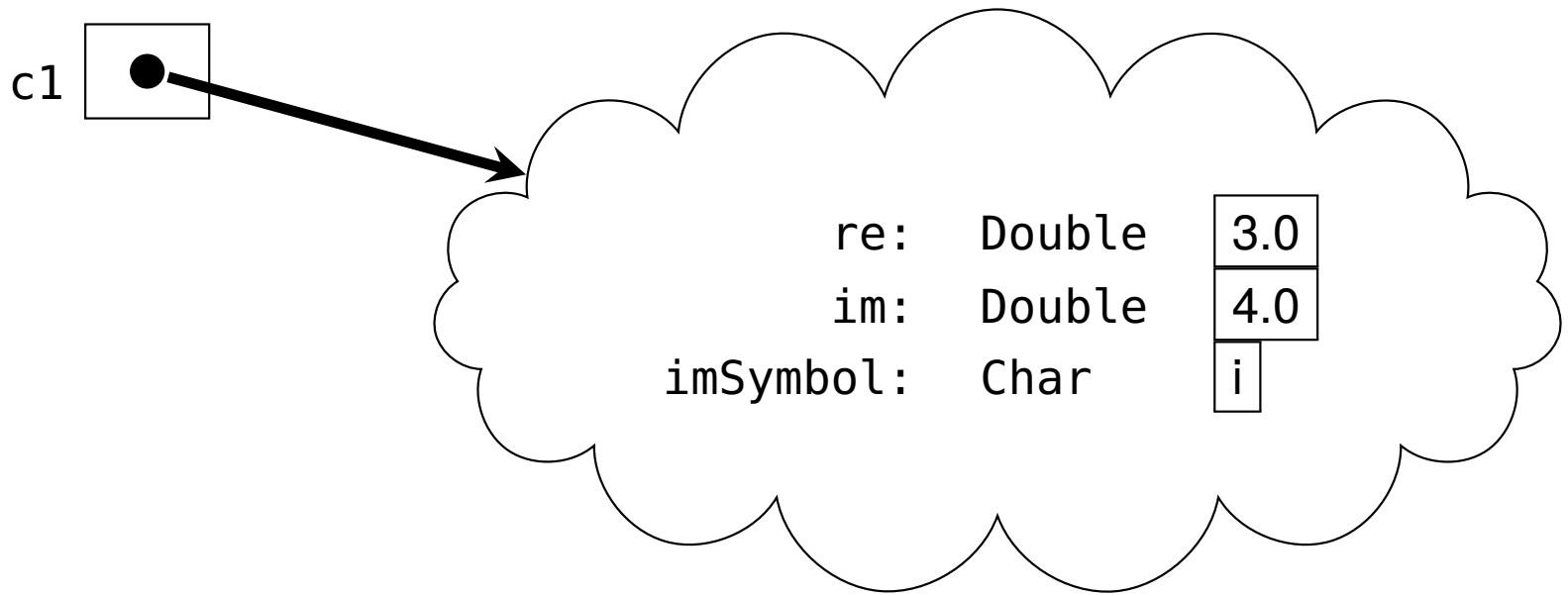
Olika sätt att skapa instanser

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)
```

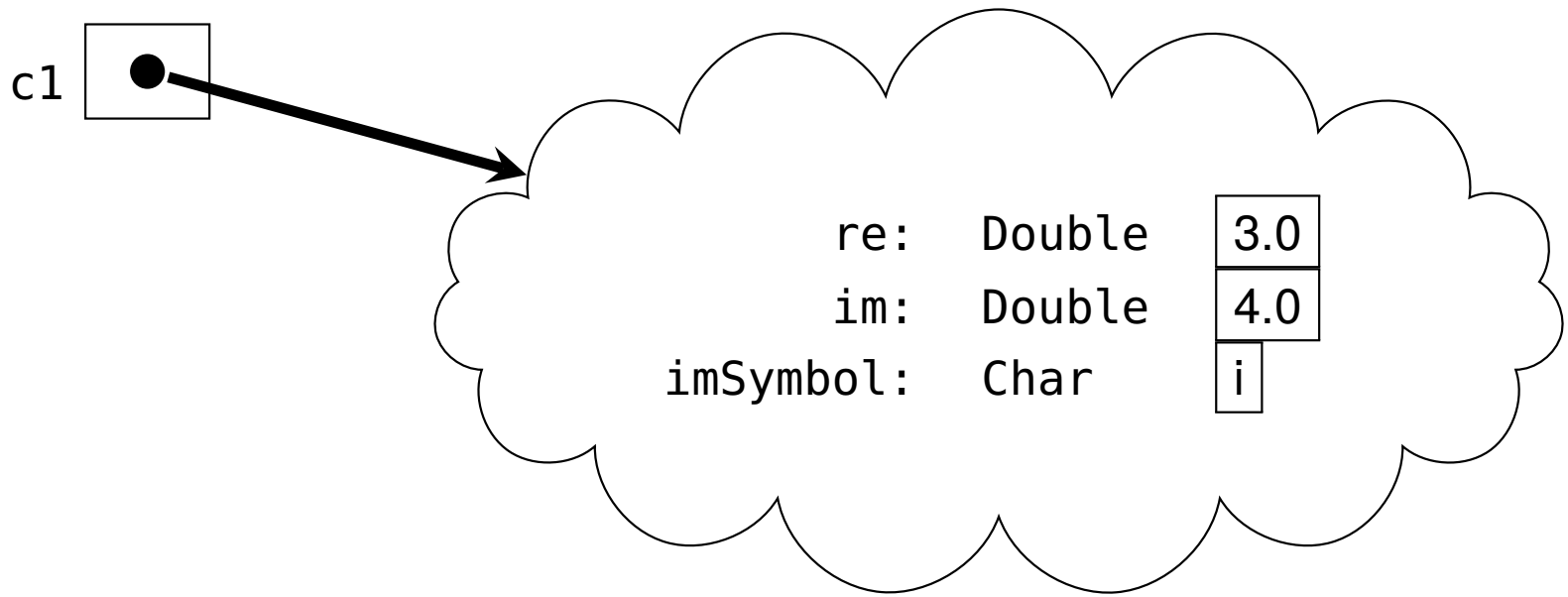


Instansiering med direkt användning av new

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(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) och kompanjonsobjektet får därför accessa privat konstruktor och kan göra **new**.

```
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)
}
```

- Fabriksmetoder i kompanjonsobjektet ovan och privat kontstruktor gör att vi **enbart** tillåter **indirekt instansiering**.

Användning av kompanjonsobjekt med fabriksmetoder för indirekt instansiering

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(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=r)
  def imag(i: Double) = apply(im=i)
  val zero = apply()
}
```

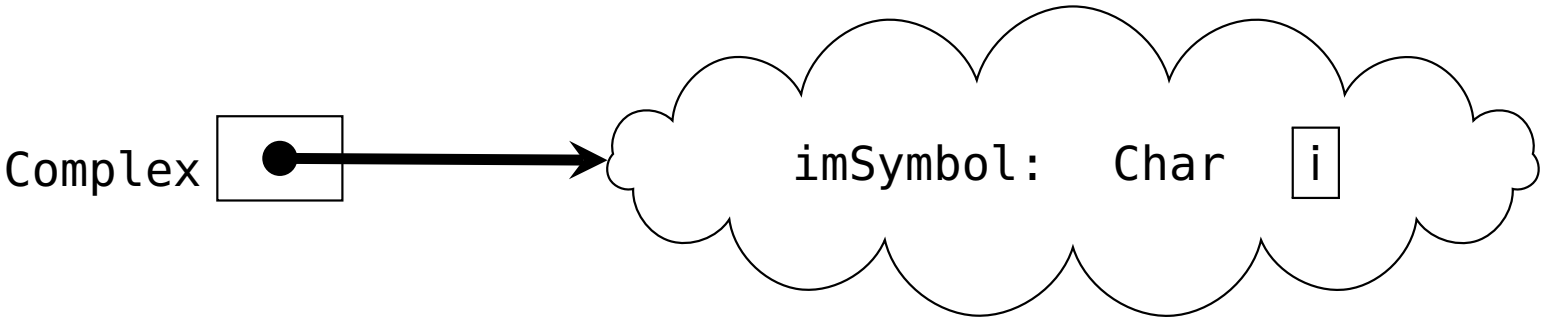
Medlemmar som bara behövs i en enda upplaga

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(re, im)
  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()
}
```

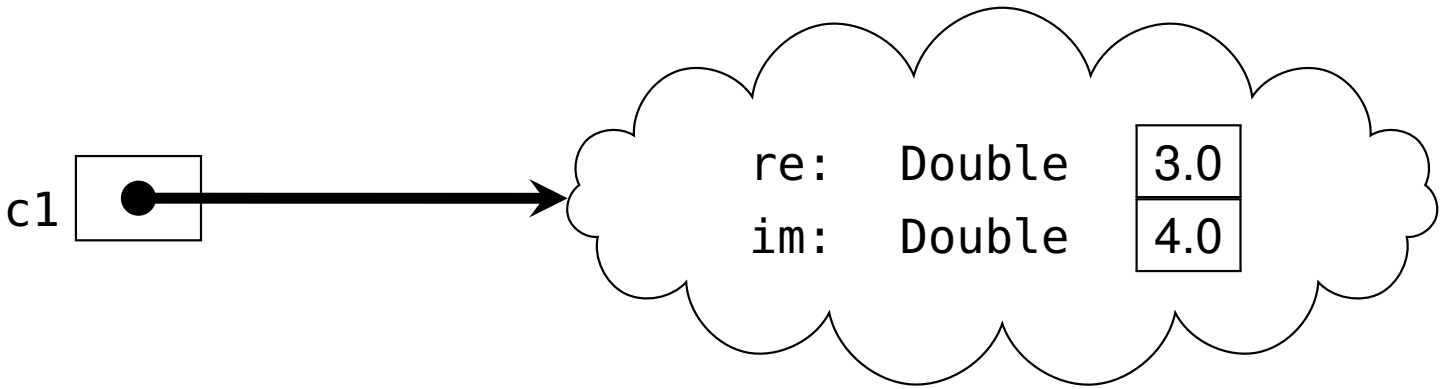
Medlemmar i singelobjekt är statiskt allokerade

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** allokerad instans:

```
scala> val c1 = Complex(3, 4)   // dynamisk allokering på heapen som växer
```



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
```

Minneshantering

Vad är en konstruktor?

- En **konstruktor** är den kod som exekveras när klasser instansieras med `new`.
- Konstruktorn skapar nya objekt i minnet under körning när den anropas.
- 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 med speciell syntax och göra alla initialiseringar själv. Man kan ha många olika alternativa konstruktorer i Java.
- I Scala **kan** man också skriva egna **alternativa konstrukturer** med speciell syntax, men det är **ovanligt**, eftersom man har möjligheten med fabriksmetoder i kompanjonsobjekt och default-argument (saknas i Java).

Hjälpkonstruktorer i Scala

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){
  def this(x: Int, y: Int) = this(x, y, 0)   //anrop av primärkonstruktorn
  def this(x: Int) = this(x, 0)              //anrop av hjälpkonstruktor
  override def toString =
    if (z == 0) s"Point($x,$y)" else s"Point($x,$y,$z)"
}
```

Genom att känna till hur hjälpkonstruktorer fungerar i Scala, blir det lättare att begripa konstruktorer i Java.

Användning av hjälpkonstruktor

```
scala> val p1 = new Point(1)
p1: Point = Point(1,0)

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

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

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

```
class Point(val x: Int, val y: Int = 0, val z: Int = 0){
  override def toString =
    if (z == 0) s"Point($x,$y)" else s"Point($x,$y,$z)"
}
```

Eller använder en fabriksmetod i kompanjonsobjekt.
Eller ännu hellre en case-klass...

Vad gör skräpsamlaren?

- Scala och Java är båda programmeringsspråk som förutsätter en körmiljö med **automatisk skräpsamling** (eng. *garbage collection*).
- **Skräpsamlaren** (eng. *the garbage colector*) är ett program som automatiskt körs i bakgrunden då och då och **städar minnet** genom att frigöra den plats som upptas av **objekt som inte längre används**.
- JVM:en bestämmer själv när skräpsamlaren ska jobba och programmeraren har ingen kontroll över detta.
- Den stora **fördelen** med automatisk skärpsamling är att man slipper bry sig om det svåra och felbenägna arbetet att **avallokera** minne.
- **Nackdelen** är att man inte kan styra exakt hur och när skräpsamlingen ska ske och man kan därmed inte bestämma när processorn ska belastas med minneshanteringen. Detta är normalt inget problem, utom i vissa tidskritiska realtidssystem med hårda minnesbegränsningar och svarstidskrav.
- I språk utan automatisk skräpsamling, t.ex. C++, måste man ta hand om destruktion av objekt och skriva egna s.k. **destruktorer**.

Referens saknas: `null`

Referens saknas: `null`

- I Java och många andra språk använder man ofta literalen **`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

```
1  scala> class Gurka(val vikt: Int)
2  defined class Gurka
3
4  scala> var g: Gurka = null        // ingen instans allokerad än
5  g: Gurka = null
6
7  scala> g.vikt
8  java.lang.NullPointerException
9
10 scala> g = new Gurka(42)          // instansen allokeras
11 g: Gurka = Gurka@1ec7d8b3
12
13 scala> g.vikt
14 res0: Int = 42
15
16 scala> g = null         // instansen kommer att destrueras av skräpsamlaren
```

- Scala har **`null`** av kompabilitetsskäl, 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 i vecka 8.

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

Java-exempel: Klassen JComplex

```
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, så metoder blir minst 3 rader

    public double getR(){
        return Math.hypot(re, im);       // Math med stort M i Java
    }

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

    public JComplex add(JComplex other){ // Javametodnamn får ej ha operatortecken t.ex. +, därav namnet add
        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
```

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));
    }
}
```

- I Java måste man skriva **tomma parentes-par** efter metodnamnet vid **anrop av parameterlösa metoder**.
- **Tupler finns inte** i Java, så det går inte på ett enkelt sätt att 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)`

Statiska medlemmar i Java

- Man kan **inte** deklarera explicita singelobjekt i Java och det finns inget nyckelord **`object`**.
- I stället kan man deklarera **statiska medlemmar** i en klass med Java-nyckelordet **`static`**.
- Exempel på hur vi kan göra detta inuti klassen `JComplex`:

```
public static char imSymbol = 'i';
```

- Effekten blir den samma som ett singelobjekt i Scala:
  - Alla statiska medlemmar i en Java-klass allokeras automatisk och hamnar i en egen singulär ”klassinstans” som existerar oberoende av de dynamiska instanserna.
  - De statiska medlemmarna accessas med punktnotation genom klassnamnet:

```
JComplex.imSymbol = 'j';
```

Referensen `this`

Referensen this

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

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

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

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

scala> g.jagSjälv.vikt
res1: Int = 42

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

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

Getters och setters

Getters och setters i Java

- I Java finns inget motsvarande nyckelord **`val`** som garanterar oföränderliga attributreferenser.[1]
- Därför gör man i Java nästan alltid attribut **privata** för att förhindra att de ändras på ett okontrollerat sätt.
- Java följer inte principen om enhetlig access: åtkomst av metoder och variabler sker med olika syntax.
- Därför är det normala i Java att införa metoder som kallas **getters** och **setters**, som används för att **indirekt** läsa och uppdatera **attribut**.
- Dessa metoder känns igen genom Java-konventionen att de heter något som börjar med **get** respektive **set**.
- Med indirekt access av attribut kan man i Java åstadkomma **flexibilitet**, så att klassimplementationen 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.
- Om klassen **inte** erbjuder en **setter** för privata attribut kan man åstadkomma **oföränderliga** datastrukturer där attributreferenserna inte förändras efter allokering.

[1] Det finns 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;

    public JPerson(String name){
        //namnkrock fixas med this
        this.name = name;
        age = 0;
    }

    public String getName(){
        return name;
    }

    public int getAge(){
        return age;
    }

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

```
$ javac JPerson.java
$ scala
Welcome to Scala 2.11.8 (Java HotSpot
Type in expressions for evaluation. O

scala> val p = new JPerson("Björn")
p: JPerson = JPerson@7e774085

scala> p.getAge
res0: Int = 0

scala> p.setAge(42)

scala> p.getAge
res1: Int = 42

scala> p.age
error:
value age is not a member of JPerson
```

Motsvarande JPerson men 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** i Scala är en **procedur som har ett namn som slutar med** _=

Getters och setters i Scala

- Principen om **enhetlig access** tillsammans med **specialsyntax** för **setters** kommer till vår räddning!
- En **setter** i Scala är en **procedur som har ett namn som 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) 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** i Scala är en **procedur som har ett namn som 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) myPrivateAge = a else myPrivateAge = 0
}
```

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

Likhet

Referenslikhet eller strukturlikhet?

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

- **Referenslikhet** (eng. *reference equality*) där två referenser anses lika om de refererar till **samma instans** i minnet.
- **Strukturlikhet** (eng. *structural equality*) där två referenser anses lika om de refererar till instanser med **samma innehåll**.

Referenslikhet eller strukturlikhet?

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

- **Referenslikhet** (eng. *reference equality*) där två referenser anses lika om de refererar till **samma instans** i minnet.
- **Strukturlikhet** (eng. *structural equality*) där två referenser anses lika om de refererar till instanser 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 men som **kan överskuggas** om man **själv vill bestämma** om det ska vara referenslikhet eller strukturlikhet.

Referenslikhet eller strukturlikhet?

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

- **Referenslikhet** (eng. *reference equality*) där två referenser anses lika om de refererar till **samma instans** i minnet.
- **Strukturlikhet** (eng. *structural equality*) där två referenser anses lika om de refererar till instanser 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 men som **kan överskuggas** om man **själv vill bestämma** om det ska vara referenslikhet eller strukturlikhet.
- Scalas **standardbibliotek** och **grundtyperna** `Int`, `String` etc. testar **strukturlikhet** genom metoden ==

Referenslikhet eller strukturlikhet?

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

- **Referenslikhet** (eng. *reference equality*) där två referenser anses lika om de refererar till **samma instans** i minnet.
- **Strukturlikhet** (eng. *structural equality*) där två referenser anses lika om de refererar till instanser 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 men som **kan överskuggas** om man **själv vill bestämma** om det ska vara referenslikhet eller strukturlikhet.
- Scalas **standardbibliotek** och **grundtyperna** `Int`, `String` etc. testar **strukturlikhet** genom metoden ==
- I Java är det annorlunda: symbolen == är ingen metod i Java utan specialsyntax som testar referenslikhet mellan instanser, medan metoden `equals` kan överskuggas med valfri likhetstest.

Exempel: referenslikhet och strukturlikhet

I Scalas standardbibliotek har man överskuggat `equals` så att metoden == ger test av **strukturlikhet** 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                 //strukturlikhetstest: 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 och likhet

Varför case-klass?

Med case-klasser får du mycket ”godis på köpet”:

- Skapa **oföränderlig datastruktur** med få kodrader.
- Klassparametrar blir automatiskt publika **`val`**-attribut (inte `private[this]` som i vanliga klasser).
- Du får en automatisk **toString** som ger klassens namn och värdet av alla **`val`**-attribut som ges av klassparametrarna.
- Du slipper skriva **`new`** eftersom du får ett automatiskt kompanjonsobjekt med en fabriksmetod `apply` för indirekt instansiering där alla klassparametrarnas **`val`**-attribut initialiseras.
- Metoden == ger **strukturlikhet** (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

Minnesschecklista med ”godis” i **`case`**-klasser så här långt:

1. klassparametrar blir **`val`**-attribut
2. najs toString
3. slipper skriva **`new`**
4. == ger strukturlikhet

Sammanfattning case-klass-godis

Minnesschecklista med ”godis” i **`case`**-klasser så här långt:

1. klassparametrar blir **`val`**-attribut
2. najs toString
3. slipper skriva **`new`**
4. == ger strukturlikhet

   ...

Men vi har inte sett allt godis än...
Vecka 8: Mönstermatchning.

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
  at scala.Predef$.$qmark$qmark$qmark(Predef.scala:230)
  at .sprängsSnart$lzycompute(<console>:11)
  at .sprängsSnart(<console>:11)
```

Exempel: ofärdig kod

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

```
scala> Person("Björn", 49).ärTonåring
res23: Boolean = false

scala> Person("Sandra", 35).ärUng
scala.NotImplementedError: an implementation is missing
  at scala.Predef$.$qmark$qmark$qmark(Predef.scala:230)
  at Person.ärGammal(<console>:12)
  at Person.ärUng(<console>:13)
```

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 ???
- **Dokumentationskommentarer** 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 **saknar implementationer** och 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;
```

Grumligt-lådan

Grumligt-lådan

Veckans skörd av lappar i ”grumligtlådan”:

12 case class
8 Map och map
8 private, public
5 override
3 toString
3 kompanjonsobjekt
2 typparametrar [Int]
2 Specialfall, sekvensalgoritmer

1 lab pirates
1 Hur ska jag träna datastrukturer?
1 underscore i olika sammanhang
1 Stränginterpolator s"\$x"
1 heap
1 Assume
1 Mutable / immutable
1 Vad är en typ och hur kan klass bli en typ?
1 tomma parenteser ()
1 skillnad mellan argument och parameter
1 pseudokod
1 Hur hitta buggar?
1 w04 datastrukturer
1 skillnad på olika parenteser \{[(
1 terminologi allmänt
1 uppdatering av variabler som refererar till varandra
1 val, lazy val, var
1 när använda terminal, editor, IDE?
1 Formattering/upplägg av kod, indrag, var ska objekt vara?
— Bättre instruktioner på labbarna
— bra med sammanfattning på slutet av föreläsningarna

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.

Grupplaboration w07

- Läs kapitel 0.1.2 på sidan 13
- Skumläs laboration `turtlerace-team`
- Träffas i din samarbetsgrupp på en fika och börja planera arbetet.