F3 – Klasser och objekt

ID1004 Objektorienterad programmering

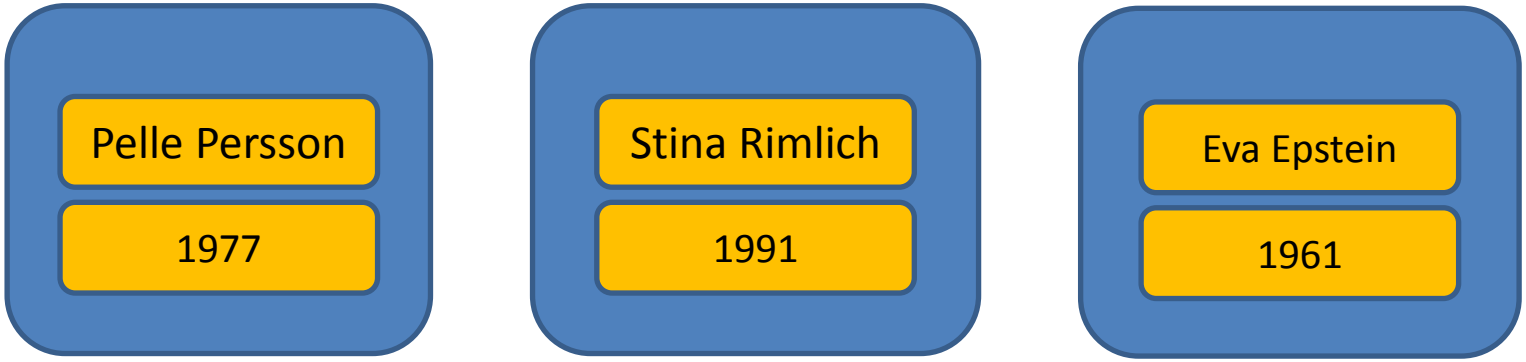
Fredrik Kilander fki@kth.se

KLASSER OCH OBJEKTORIENTERAD PROGRAMMERING (OOP)

Allmänt om klasser och OOP

- Klasser och deras instanser framställs ibland som **simpla databehållare**
- Klasser **sammanför** data med metoder för hantering av data
- Detta är styrkan med objektorienterad programmering

Simpla databehållare




```
public class Person {
  public String name;
  public int birthYear;
}
```

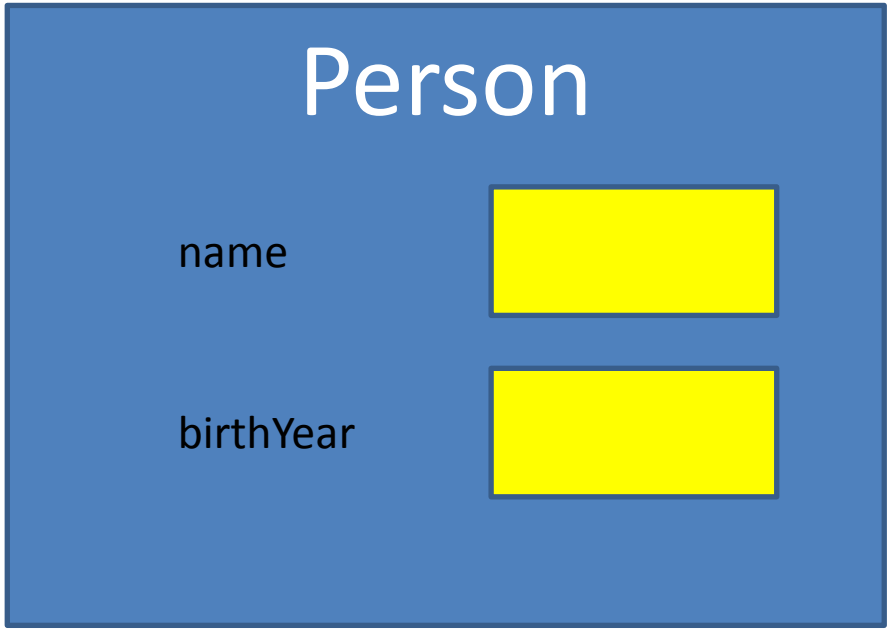
Simpla databehållare

Fälten name och birthYear är oskyddade och kan manipuleras fritt av alla programmerare som arbetar med dem.

```
public class Person {
  public String name;
  public int birthYear;
}
```

Klassen Person

Med hjälp av metoder (kod) i klassen Person, kan vi bättre skydda data och garantera dess kvalité.




```
Person(String name)
Person(String name,
       int birthYear)
String getName()
void setName(String name)
int getBirthYear()
int getAge()
String toString()
```

Se filen Person.java

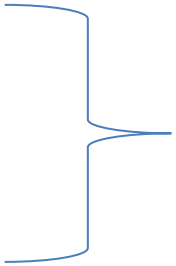
Allmänt om klasser och OOP

- Inte sällan används klasser och instanser för att följa en viss modell, till exempel:
  - naturliga element i ett problem
  - komponenter i grafiska gränssnitt
  - databasstruktur
  - händelsestyrd programmering

KLASSENS VARIABLER OCH METODER

Instansvariabler och metoder

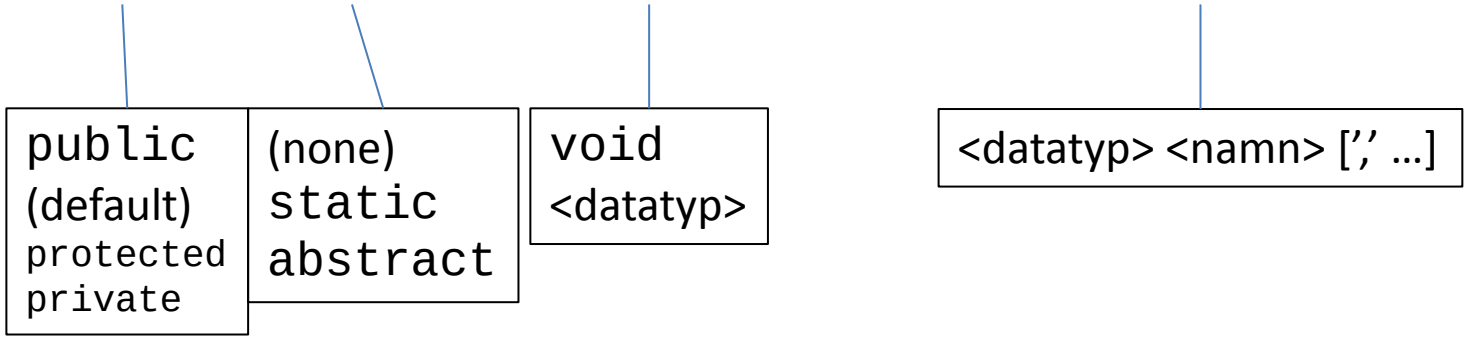
```
public class Example {

   int x;
   int y;

   String title;

}
```



Instansvariabler skapas med objektet. De är tillgängliga så länge objektet är tillgängligt.

Metoddeklaration

<åtkomst> <egenskap> <returdatatyp> <namn> ’**(**’ <parameterlista> ’**)**’ * ’**{**’ <kropp> ’**}**’




```
int getCount() {…}

private double bar(double a, double b) {…}

public static void main (String [] args) {…}
```

(*) <exception-declaration-list>

Metodens kropp

- Metodens kropp är ett s k block.
- Ett block inleds med { och avslutas med }.

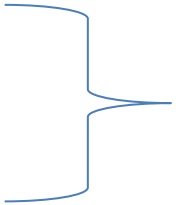
- Inuti blocket finns en sekvens av:
  - Lokala variabler
  - Programsatser:
    - Beräkning av uttryck och tilldelning av variabel
    - Konstruktioner för iteration och selektion

Instansvariabler och metoder

```
public class Example {

  int x;                          Instansvariabel

  String title;                   Instansvariabel

  void swapXY() {                 Metoddeklaration
    int temp;                     Lokal variabel
    temp = x;                     uttryck och tilldelning
    x = y;                        uttryck och tilldelning
    y = x;                        uttryck och tilldelning
  }

  void setTitle(String t){        Metoddeklaration, lokal variabel(t)
    title = t;                      uttryck och tilldelning
  }
}
```

Instansvariabler och metoder

```
public class Example {

  int x, y;

  String title;

  void swapXY() {
    int temp = x;
    x = y;
    y = x;
  }

  void setTitle(String t){
    title = t;
  }
}
```



Instansvariabler är på samma nivå som metoder i klassen.
De skapas med objektet.

Instansvariabler och metoder

```
public class Example {

  int x, y;

  String title;

  void swapXY() {
    int temp = x;
    x = y;
    y = temp;
  }

  void setTitle(String t){
    title = t;
  }
}
```

Instansvariabler är på samma nivå som metoder i klassen.
De skapas med objektet.

Lokala variabler (temp, t) i metoder existerar bara i blocket där de deklarerats.

Klassvariabler och –metoder (static)

- Konstanter, variabler och metoder kan placeras i klassdefinitionen med **`static`**
- Statiska variabler finns bara i en kopia i sin klass
- Statiska metoder kan anropas utan att först skapa instanser(objekt) av klassen

```
public static double cos(double angle)

public static void main(String[] args)
```

När kan man använda static?

- Metoden main(String[]) måste vara static
- Rena funktioner, t ex java.lang.Math
  - cos, sin, tan, exp, pow, sqrt ...
- Konstanter, t ex java.awt.Color
  - Color.red

Är det fel att använda static?

- Fel använt ger static knepiga problem och osnygga lösningar
- Grundläget är att undvika static

PACKAGES OCH STANDARDBIBLIOTEKEN

Packages

- Standardbiblioteken är organiserade i s k packages
- java.lang (importeras automatiskt)
- java.util (händiga verktyg)
- java.io (in- och utmatningsfunktioner)
- java.net (nätverkskommunikation)
- ... många fler

Importera ett package

- **`import java.util.*;`**
  - Gör alla klasser i java.util tillgängliga

- **`import java.util.StringTokenizer;`**
  - Gör endast klassen StringTokenizer tillgänglig.

Om man använder mindre än 20-30% av ett package är det tydligare för andra läsare om man uttryckligen importerar varje klass koden behöver.

java.util.Random

- Random()
- int nextInt(int num)

```
import java.util.Random;
...
Random rnd = new Random ();
...
int die1 = 1 + rnd.nextInt(6); // 1+(0..5)
int die2 = 1 + rnd.nextInt(6);
int dice = die1 + die2;
```

java.lang.Math

- double acos(double num)
- double asin(double num)
- double atan(double num)
- double cos(double angle)
- double sin(double angle)
- double tan(double angle)

java.lang.Math

- double ceil(double num)
- double floor(double num)
- double exp(double num)
- double pow(double angle, double power)
- double random()
- double sqrt(double num)

```
// java.lang är redan importerad
...
double v = Math.sqrt(u);
```

System.out.printf

- Klassen java.util.Formatter kan användas för formaterad utmatning av text
- Inspirerad av funktionen printf() i C
- Funktionaliteten kan enkelt nås genom t ex System.out.printf eller String.format

- **`System.out.printf(String format, Object ... args)`**

Formatsträngen anger hur argumentet skall formateras

Antalet parametrar ska stämma mot formatsträngen

System.out.printf

- **`System.out.printf(”%s:%4d%n”, name, age);`**
  - `Allan Kluring:  42`
- %s – en sträng
- %4d – ett heltal i ett fält om fyra kolumner
- %n – ny rad

- Läs mer på
  http://download.oracle.com/javase/6/docs/api/java/util/Formatter.html#syntax

System.out.printf

”However, using the printf method is not a particularly clean object-oriented solution to the problem of formatting output, so we avoid its use in this book.”
Lewis & Loftus, 6th ed, pg 164

”Bah!”

FK

enum is good for you

ENUMERATED TYPES

Enum (uppräkningstyp)

- Ibland behöver koden distinkta symboliska värden från en liten mängd
- t ex **`måndag, tisdag, onsdag, ... , söndag`**
- Dessa *kan* modelleras som t ex konstanter:
- **`static final int MONDAY=0, TUESDAY=1`**, …

Enum (uppräkningstyp)

- Att använda primitiva datatyper för litet antal värden inbjuder till problem
- `static final int MONDAY=1, TUESDAY=1,...`
- `int weekday = MONDAY; // detta är förstås ok`
- `weekday = -999; // tyvärr även detta`
- Kompilatorn vet inte om att vi vill ha ett fåtal, unika värden, och kan inte hjälpa till

enum

- Deklarationen enum skapar en ny datatyp med bara de uppräknade värdena
- **`enum Weekday {monday, tuesday, ... }`**
- Därefter kan vi deklarera variabler som vanligt
- **`Weekday today;`**
- Och tilldela dem
- **`today = Weekday.tuesday;`**
- Kompilatorn kan nu kontrollera koden bättre

omslag kring primitiva datatyper

WRAPPER CLASSES

Wrapper classes

- Primitiva datatyper (int, long, char, ...) är inte klasser
- Ibland önskar man att de vore det
- Varje primitiv datatyp har därför en motsvarande *wrapper class*
- Integer, Long, Char ... (Stor bokstav!)
- Wrapper-klassen innehåller exakt en lagringsplats för sin primitiva datatyp

Wrapper classes

- Wrapper-klasser behövs om man vill stoppa in en primitiv datatyp i t ex en hash-tabell
- java.util.HashTable erbjuder nämligen bl a metoderna:
  - put (Object key, Object value) // stoppa in data
  - Object get (Object key) // ta ut data
- Men en primitiv datatyp är inte ett Object.
- Lösningen är att använda en wrapper-klass

Wrapper classes

- Antag att nyckeln är en int och värdet en sträng.
- Vi använder Integer runt vår int-nyckel.

```
import java.util.HashTable;

HashTable<Integer,String> mytable =
                          new HashTable<Integer,String>();

mytable.put(new Integer(43), ”Alphanor”);
```

Genom att slå in 43 i en instans av Integer så fungerar det.

Wrapper classes - Autoboxing

- Kompilatorn inser många gånger när en wrapper-klass behövs
- Den kan då själv lägga till den nödvändiga koden

```
import java.util.HashTable;

HashTable<Integer,String> mytable =
                          new HashTable<Integer,String>();

mytable.put(new Integer(43), ”Alphanor”);
mytable.put(43, ”Alphanor”);                  // Autoboxing
```

Kompilatorn lägger till koden new Integer(43).

Wrapper classes - unboxing

- Kompilatorn inser också när en wrapper-klass används som en primitiv datatyp
- Den kan då själv lägga till den nödvändiga koden

```
Integer myInt = new Integer(43);

int x = myInt.intValue(); // Det traditionella sättet

int x = myInt; // unboxing
```

Kompilatorn lägger till anropet till myInt.intValue().

Wrapper classes – statiska metoder

- Wrapper-klasser har även användbara statiska metoder för den primitiva datatypen, t ex

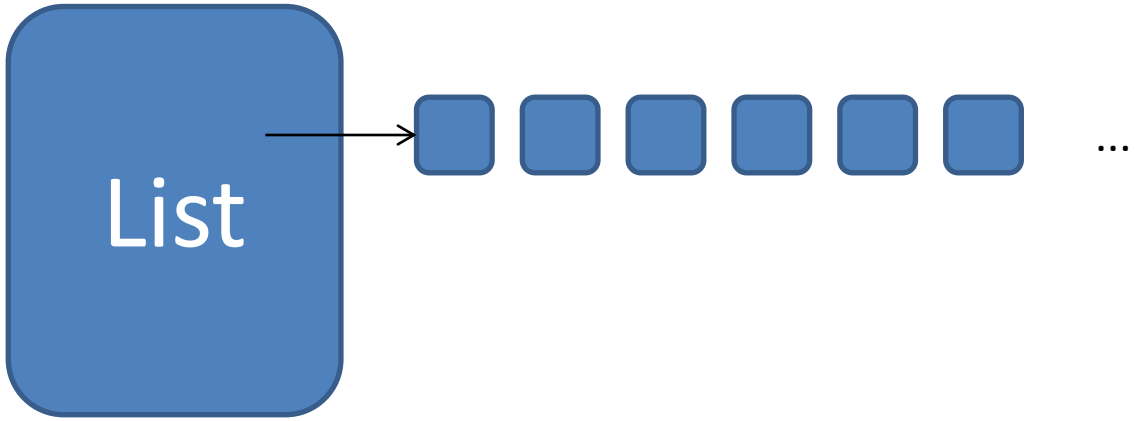
- **`static int Integer.parseInt(String s)`**
- **`static String toBinaryString(int num)`**
- **`static String toHexString(int num)`**
- **`static String toOctalString(int num)`**

- och många fler...

meh

SLUT PÅ BILDER

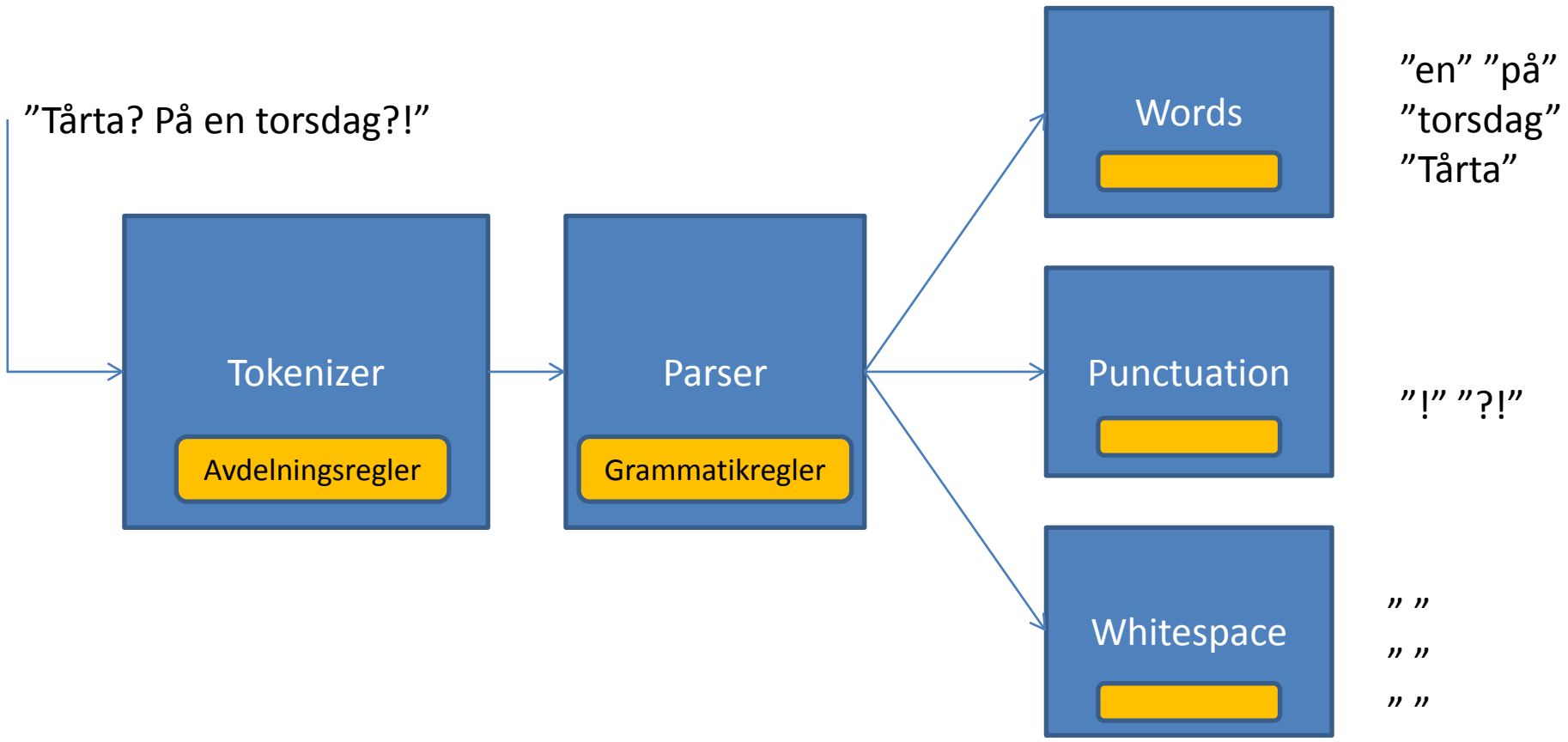
Objekt som datasamlingar



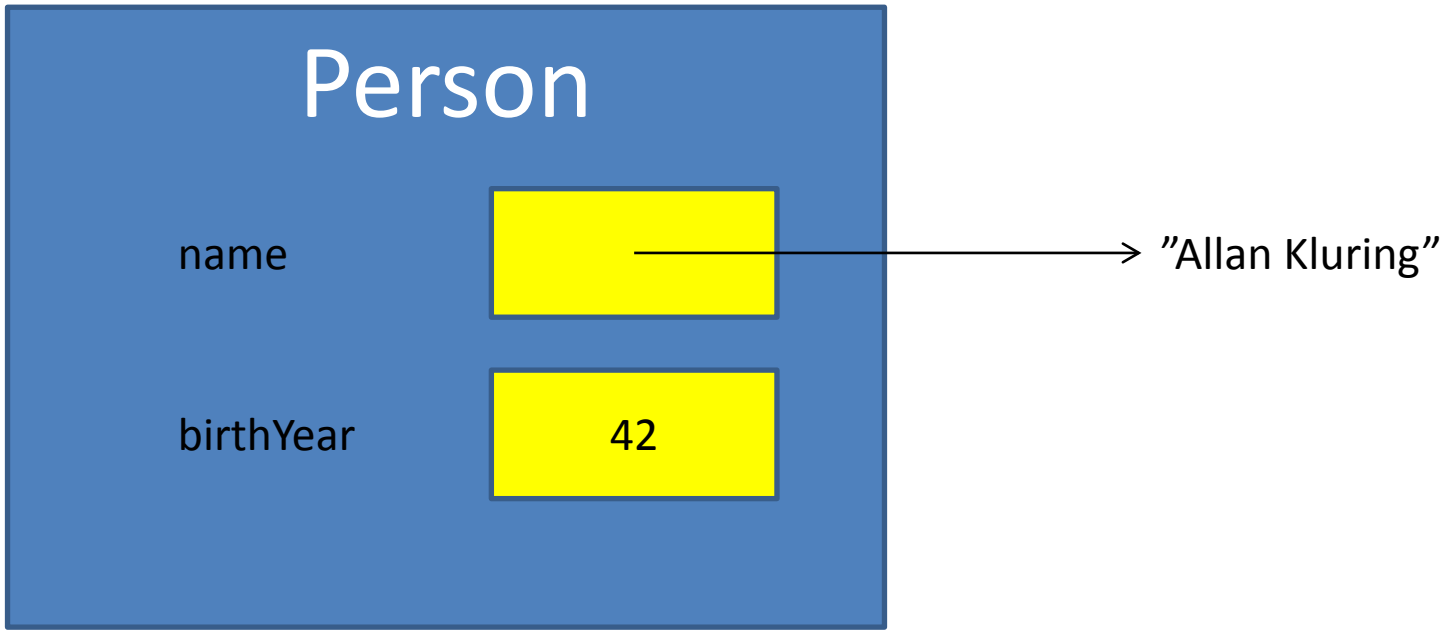

```
boolean add(Object)
void clear()
boolean contains(Object)
Object get(int index)
boolean isEmpty()
boolean remove(Object)
int size()
```

och många fler metoder...

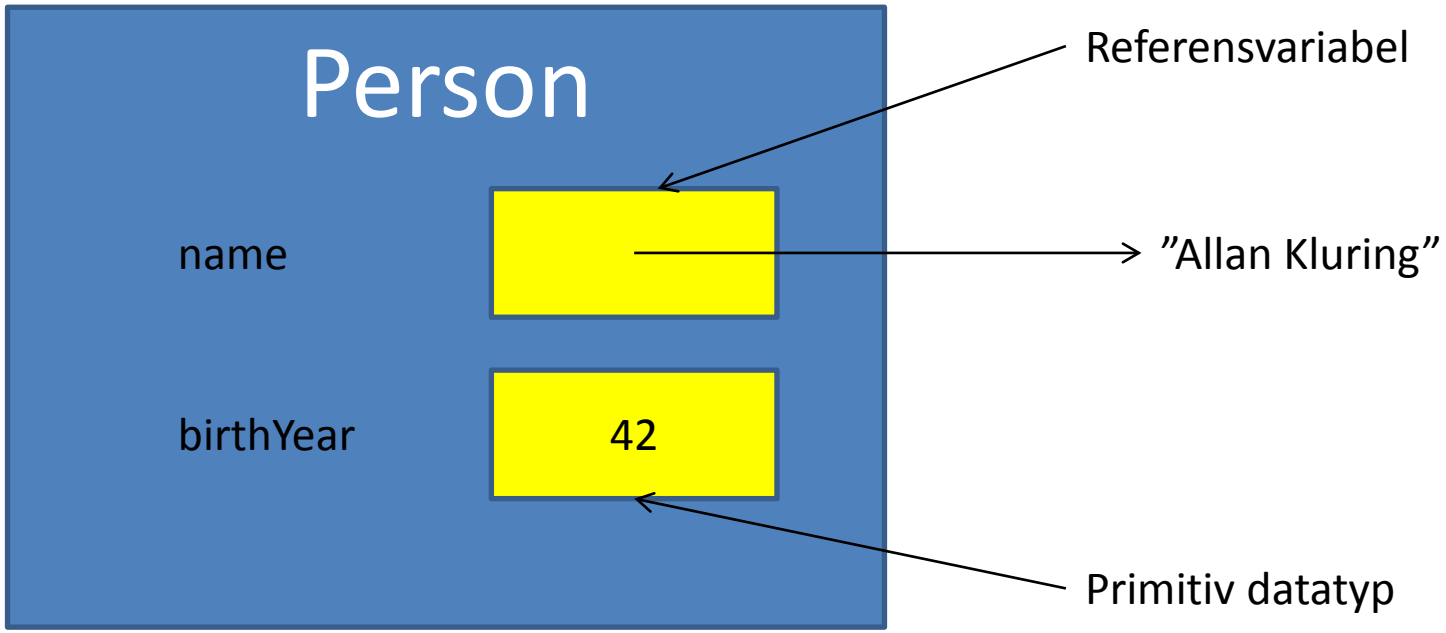
Processelement med lokala data



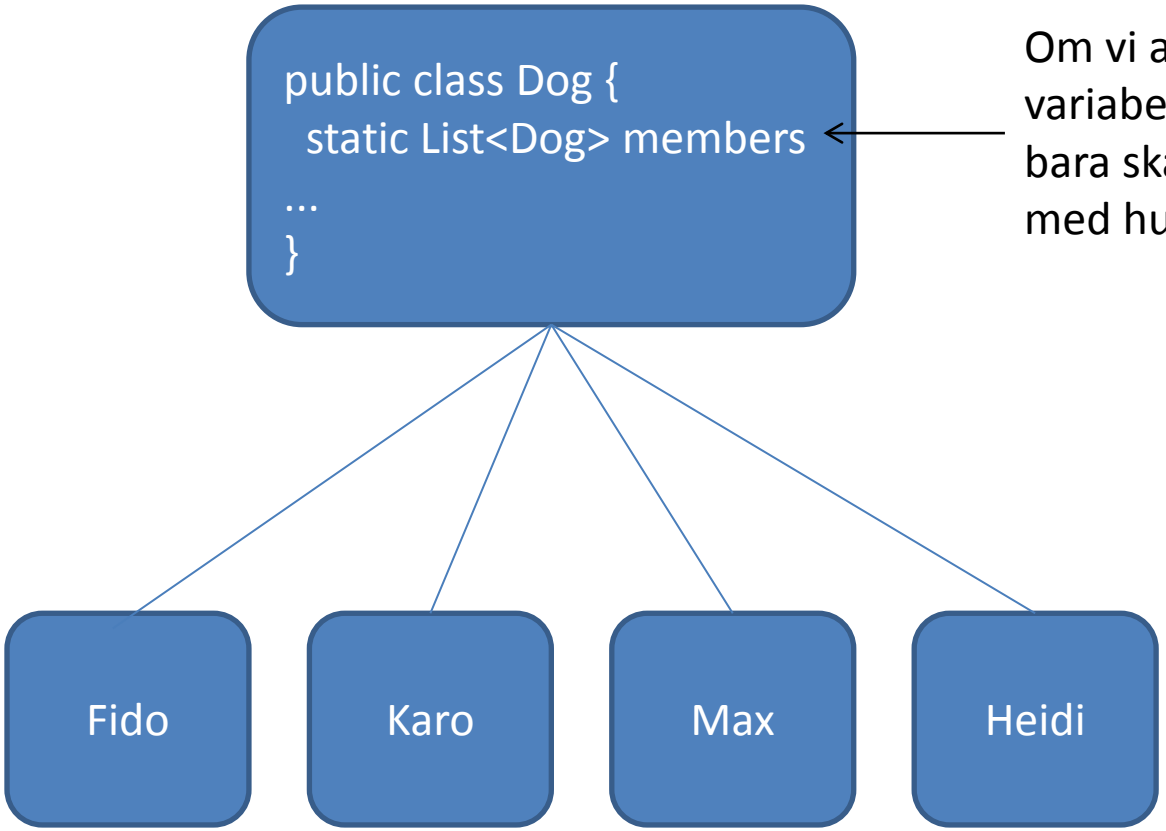
Klassen Person



Klassen Person

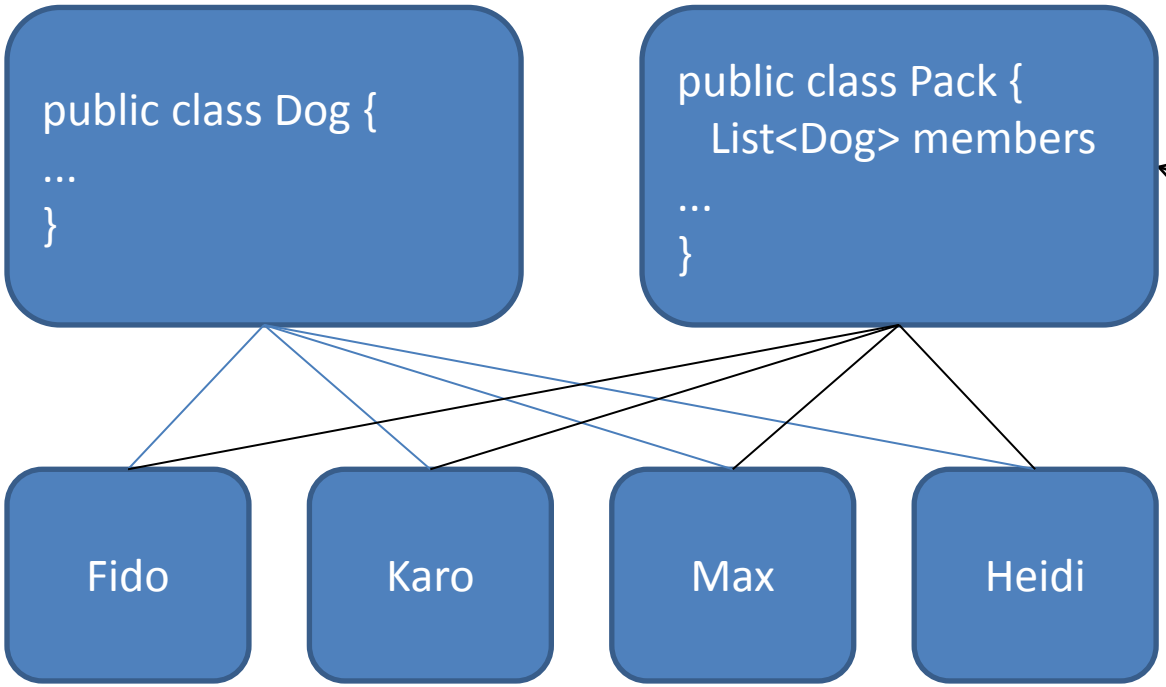


Hundar och hundflock



Om vi använder en statisk variabel i klassen så kan vi bara skapa en enda flock med hundar.

Hundar och hundflock



Med en separat klass kan vi organisera hur många olika flockar vi vill. Mer generellt!