Objektorientering: Lagring, räckvidd och livstid

Tre sorters variabler,
två sorters metoder…

Räckvidd och livstid

- Variabler (lokala och medlemsvariabler/fält) har:

Räckvidd (”scope”)	Livstid
Från vilken <u>del</u> av koden kan man *nå* variabeln?	**Hur länge finns variabeln (”lådan”)? När slängs den bort? När ”återskapas” den?**

Livstid och räckvidd:
Lokala variabler

Lokala variabler (1)

jonkv@ida

Lokala variabler x och y, deklarerade i en metod

```
public class GameGUI {
   public void paint() {
      int x = calcX();
      int y = calcY();
      paintImageAt(x, y);
      System.out.println(x);
      …
   }

   public void paintImageAt(posX, posY) {
      int square = x * x;
   }
}
```

Livstid: I detta fall, till metoden returnerar / avslutas

// Här skapas variabeln/lådan "x" och ett värde läggs i

// Variabeln x finns kvar… ända tills paint() avslutas

Räckvidd: Bara inom samma metod

// Fel: Variabeln x "lever och finns kvar", men *syns inte*,
// kan inte nås / "ses" i den anropade metoden

- Flera anrop:
  Anropa paint(); ➔ anropet får ett eget "x" ➔ metoden returnerar ➔ "x" försvinner
  Anropa paint(); ➔ anropet får ett eget "x" ➔ metoden returnerar ➔ "x" försvinner

Lokala variabler (2)

Lokala variabler, deklarerade i en metod

```
public class GameGUI {
   public void paint() {
      int x = calcX();
      int y = calcY();
      if (x == y) {
         int square = x*x;
         …
      }
      paintImageAt(x, y);
      System.out.println(x); …
   }
}
```

Deklarerar square inuti ett { block }
Räckvidd: Inom { blocket }
Livstid: Till vi går ur { blocket }

Här har square ”försvunnit” – skillnad från Python!

- **Generell tumregel:**
  - I varje situation vill vi **bara** ha tillgång till det vi **faktiskt behöver**
  - **Mindre räckvidd** → **mindre att hålla reda på** i andra delar av koden

Livstid och räckvidd: Fält (medlemsvariabler)

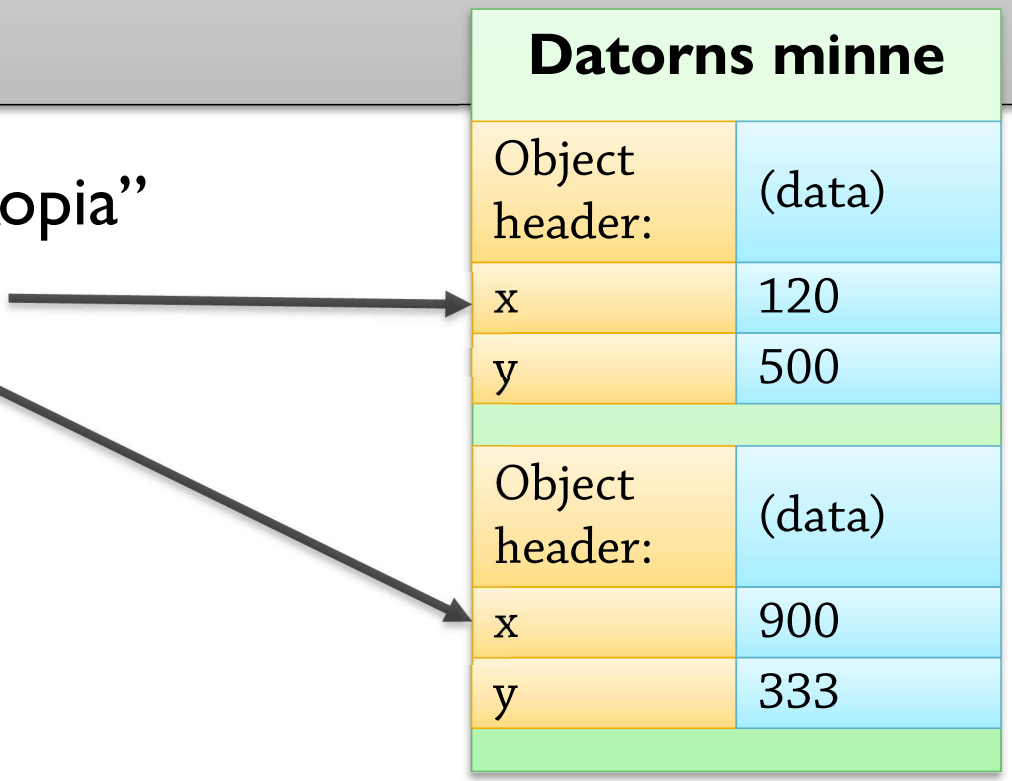
Fält (1)

Fält = medlemsvariabel = instansvariabel

```
public class Player {
   private int x, y;
   public Player(int x, int y) {
      this.x = x;
      this.y = y;
   }
}
```

Livstid: Variabler (x,y) skapas vid ”new Player()”, försvinner inte förrän *spelarobjektet* slängs bort

- Varje **instans/objekt** får sin "kopia”
  - Player p1 = **new** Player(120, 500);
  - Player p2 = **new** Player(900, 333);
    Nya objekt, *samma* fältnamn (x/y)



Fält (2)

jonkv@ida

Fält = medlemsvariabel = instansvariabel

```
public class Player {
   private int x, y;
   public Player(int x, int y) {
      this.x = x;
      this.y = y;
   }
   public void setX(int newX) {
      this.x = newX;
   }
   public void paint() {
      paintImageAt(x, y);
   }
```

Livstid: Skapas x och y vid varje ”new Player()”, försvinner inte förrän detta *spelarobjekt* slängs bort

Datorns minne

Object header:	(data)
x	12
y	50

```
public class Game {
   public static void main(String[] args) {
      Player minSpelare = new Player(12,50);
      minSpelare.setX(10);
      minSpelare.paint();
   }
}
```

Anrop till minSpelare.setX() → minSpelare.x finns, ”lever fortfarande” → *ändra* värdet

Anrop till minSpelare.paint() → minSpelare.x finns, ”lever fortfarande” → *använd* värdet

Fält eller lokal variabel?

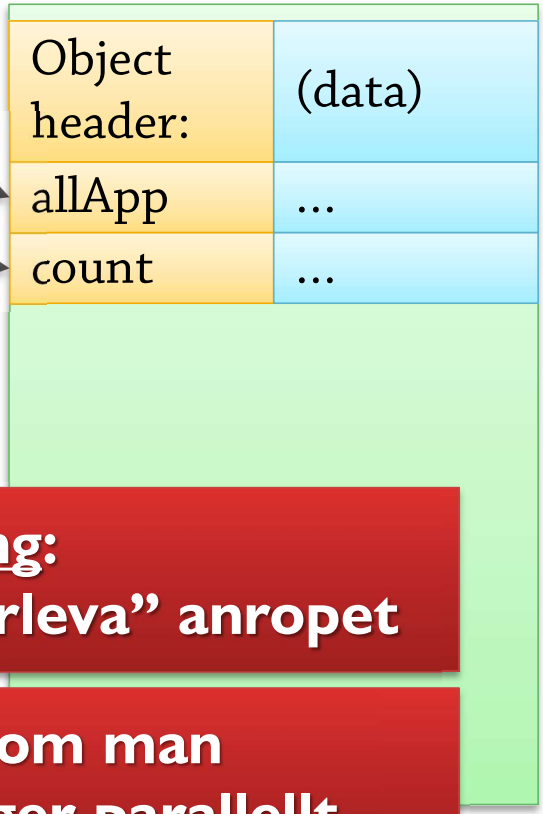
Som i åtkomsträttigheter:

Minska om du kan!

Fält: Inte i onödan – modellering, minne

- **Varning**: Använd inte fält när lokala variabler räcker!
  - Fält:
    - **Objektets tillstånd/info**, som ska **bevaras så länge objektet lever**
  - Lokala variabler:
    - **Metodens tillstånd/info**, som ska **bevaras till metoden returnerar**

```
public class Calendar {
   private List<Appointment> allAppointments;
   private int count;
   public int countAppointmentsFor(Date date) {
      count = 0;
      for (Appointment app: allAppointments) {
         if (app.hasDate(date)) count += 1;
      }
      return count;
   }
}
```



Fel modellering:
Räknaren borde inte ”överleva” anropet

Kan bli fel resultat, om man anropar metoden 2 gånger parallellt

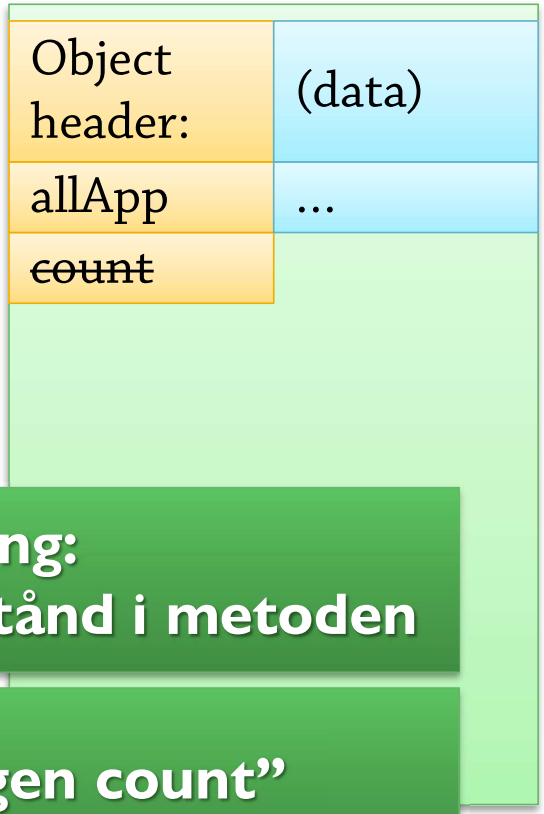
Fält: Inte i onödan – modellering, minne (2)

jonkv@ida

- Använd lokala variabler utom för **persistent information**
  - Finns inget att förlora
  - Tar inte mer tid att ”skapa variabeln varje gång”

```
public class Calendar {
   private List<Appointment> allAppointments;

   public int countAppointmentsFor(Date date) {
      int count = 0;
      for (Appointment app: allAppointments) {
         if (app.hasDate(date)) count += 1;
      }
      return count;
   }
}
```



Rätt modellering:
Räknaren är ett lokalt tillstånd i metoden

Varje anrop får ”sin egen count”

Fält: Inte i onödan – samtidiga anrop

jonkv@ida

- Exempel på problem:
  Flera **samtidiga anrop** kan använda sig av **samma variabel**

```
private List<Person> list;

public List<Person> getAllMembers() {
   list = new ArrayList<>();
   for (City city: allCities) {
      list.addAll(getMembersFrom(city));
   }
   return list;
}
```

```
public List<Person> getAllMembersFrom(City city) {
   list = new ArrayList<>();
   // Lägg till ett antal objekt i list…
   return list;
}
```

Byter ut **list** – **samma** variabel som getAllMembersFrom() använder!

Livstid och räckvidd:
Statiska fält

Statiska fält: Globala variabler

Statiskt fält = klassvariabel

- Deklareras i en klass, ***static***
  - Inte som vanliga fält, som allokeras ”dynamiskt” varje gång ett objekt skapas med ***new***
  - Vid **programkörning** skapas **en** variabel, i **klassen** (inte ett av dess objekt), som **finns kvar** under hela körningen

Class Circle	
circlesCreated	0
getCircum	...code...
getArea	...code...
setRadius	...code...

```
public class Circle {
   private static int circlesCreated = 0;
   public double x, y, r;

   public Circle(double x, double y, double r) {
      this.x = x;
      this.y = y;
      this.r = r;
      Circle.circlesCreated++;
   }
}
```

Håll reda på antal cirklar som skapats:
Måste ligga i **klassen**, inte varje cirkelobjekt

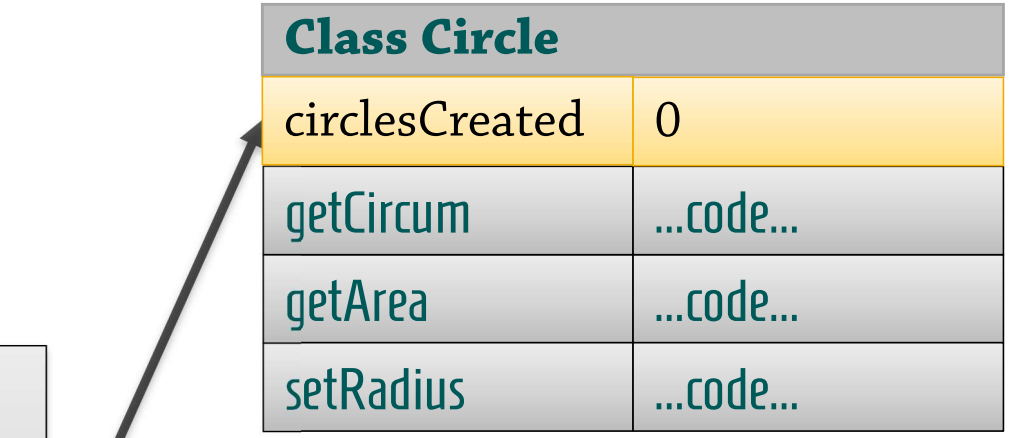
Behöver inget objekt för att komma åt värdet:
klassnamn.fältnamn

Statiska fält 2: Innan objekt skapas

- Innan några objekt skapas:
  - Statiska fält får defaultvärden (**0, 0.0, false, null**)

- Kan redan komma åt fältet:

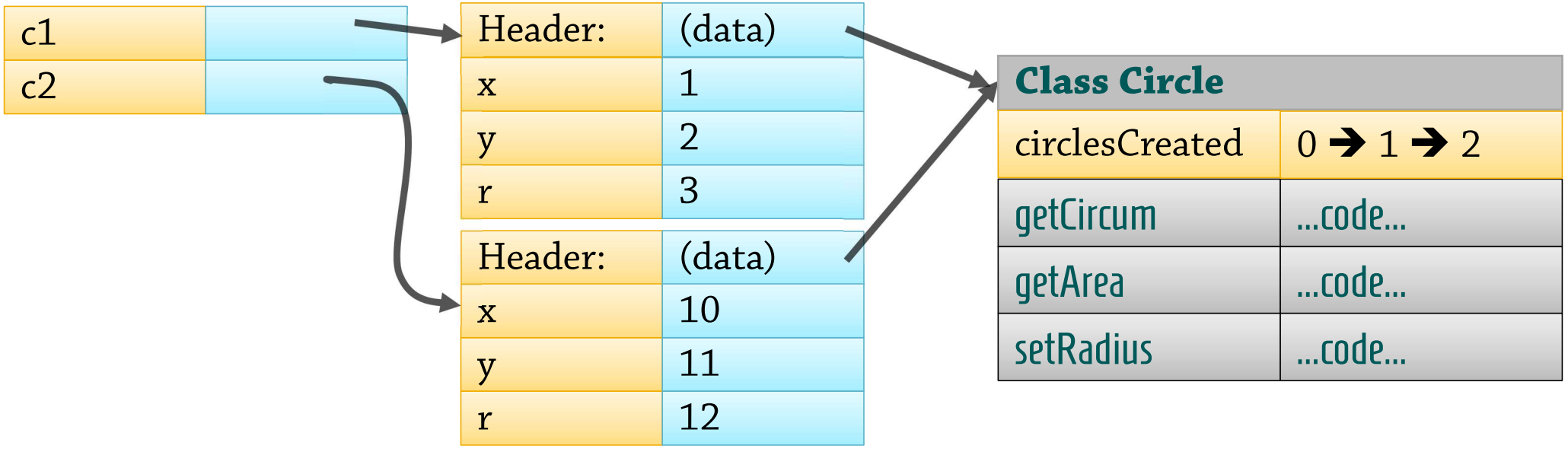
```
public class Circle {
   private static int circlesCreated = 0;
   …
   public static void main(String[] args) {
      System.out.println(Circle.circlesCreated);
   }
}
```



Statiska fält 3: När objekt skapas

jonkv@ida

- När objekt skapas:
  - Circle c1 = **new** Circle(1, 2, 3);
  - Circle c2 = **new** Circle(10,11,12);



Statiska fält:
Några acceptabla användningsfall

Statiska fält: Global information

jonkv@ida

Acceptabel användning:

Information som verkligen **måste** vara **global**, handlar om **hela klassen**
(inte så vanligt!)

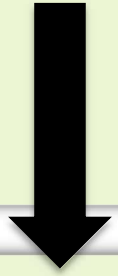
```
public class Circle {
   private static int circlesCreated = 0;
   public double x, y, r;

   public Circle(double x, double y, double r) {
      this.x = x;
      this.y = y;
      this.r = r;
      Circle.circlesCreated++;
   }
}
```

Statiska fält: Namngivna konstanter

- Konstanter bör (nästan) alltid **namnges** för läsbarhet

```
public class CardGame {
   public void shuffle() {
      for (int i = 0; i < 52; i++) {
         // ...
      }
   }
}
```



```
public class CardGame2 {
   private static final int DECK_SIZE = 52;

   public void shuffle() {
      for (int i = 0; i < DECK_SIZE; i++) {
         // ...
      }
   }
}
```

Namngivna konstanter:
static – i klassen
final – ändras inte
namn – stora bokstäver (WITH_UNDERSCORE)

Lagras en gång → slösar inte minne

Utan namn → ofta komplettering!

Statiska fält: Initialisering

jonkv@ida

- Hur sätter man ett mer ”komplext” värde? Konstruktor?

```
public class PictureViewer1
{
   private final static List<String> KNOWN_EXTENSIONS = new ArrayList<>();

   public PictureViewer1() {
      KNOWN_EXTENSIONS.add("png");
      KNOWN_EXTENSIONS.add("jpg");
      KNOWN_EXTENSIONS.add("gif");
   }
}
```

Problem: Fler element adderas för varje PictureViewer vi skapar

new PV1() → [”png”, ”jpg”, ”gif”]
new PV1() igen → [”png”, ”jpg”, ”gif”, ”png”, ”jpg”, ”gif”] … …

Fel även om vi bara har 1 PictureViewer:
Blandar ihop objektnivå / klassnivå

Statiska fält: Initialisering (2)

jonkv@ida

- **Bättre**
  - Skapa datastrukturer direkt, med List.of, Set.of, Map.of
  - (Blir ”oföränderliga” datastrukturer; kan inte lägga till något senare)

```
public class PictureViewer2
{
   private final static List<String> KNOWN_EXTENSIONS =
      List.of("png", "jpg", "gif");
}
```

Statiska fält: Initialisering (3)

jonkv@ida

- Mer generellt: Anropa **statisk metod** som ger korrekt värde
  - Korrekt dataflöde:
    Initialisering direkt, genom att *fråga getExtensions() efter värdet*
  - Ger möjlighet att göra mer komplexa beräkningar,
    även om vi inte behövde det just här

```
public class PictureViewer3
{
   private final static List<String> KNOWN_EXTENSIONS = getExtensions();

   private static List<String> getExtensions() {
      final List<String> extensions = new ArrayList<>();
      extensions.add("png");
      extensions.add("jpg");
      extensions.add("gif");
      return extensions;
   }
}
```

Statiska fält: Finns alternativ

- Konstanter kan istället namnges **lokalt i en metod**

```
public class CardGame3 {
   public void shuffle() {
      final int deckSize = 52;
      for (int i = 0; i < deckSize; i++) {
         // ...
      }
   }
}
```

Enkel konstant,
används i en enda metod
➔ sprid inte ut koden i onödan,
skapa inte globala namn i onödan,
överbelasta inte programmeraren!

```
public class CardGame4 {
   // ... IMAGE takes time to read...
   private static final Icon IMAGE = new ImageIcon("/resources/card.png");

   public void paint() {
      // ... use IMAGE every time ...
   }
}
```

Men om det tar tid
att beräkna värdet (ladda in en bild):
Statiskt fält ➔ beräknas EN gång!

Metoder på klassnivå
Statiska hjälpklasser

Exempel: Absolutvärde

- Vi vill räkna ut **absolutvärdet** för ett tal

Python

```
def abs(num):
    if num < 0:
        return –num
    else:
        return num
```

Java

```
public class Calculator {
    public double abs(double num) {
        if (num < 0)
            return –num;
        else
            return num;
    }
}
```

```
Calculator calc = new Calculator();
double x = calc.abs(y);
```

Måste vi skapa ett **räknarobjekt** som räknar åt oss?

Känns onaturligt, och räknaren behöver ingen *info* från objektet

Två sorters metoder

jonkv@ida

Instansmetoder

- Anropas för ett specifikt objekt:
  myCircle.paint()
- Kan **använda** "samma objekt": **this**
  - Anropa andra metoder i objektet
  - Använda fält – position, radie, färg

Statiska metoder (klassmetoder)

- Anropas i en **klass**:
  Math.abs(), **Circle.getPI()**
- Det närmaste man kan komma till en "global funktion“ (och till Pythons funktioner)

Statiska metoder: Exempel 1

- Absolutvärde som statisk metod

Java

```
public class Calculator {
   public static double abs(double num) {
      if (num < 0)
         return –num;
      else
         return num;
   }
}
```

Överanvänd inte!
Används när det är *onaturligt att behöva skapa ett objekt* för att anropa koden

```
double x = Calculator.abs(y);
```

```
double x = abs(y);
```

Inget objekt skapas

Fungerar ungefär som Python-funktioner, men inte ”på toppnivå” utan inuti en klass

Inuti **samma klass**
är klassnamnet underförstått

Statiska metoder: Exempel 2

jonkv@ida

- Att anropa metoder i samma klass:

```
public class StaticTest {
   public static void test(String str) {
      staticMethod();
      instanceMethod();
      int len = str.length();
   }
   public static void staticMethod() {
      System.out.println("Here");
   }
   public void instanceMethod() {
      System.out.println("Here");
   }
}
```

Metoden är statisk ➔
StaticTest.staticMethod()
Fungerar bra!

Metoden är inte static ➔
Samma som
this.instanceMethod()

Men **this** = "objektet som den här metoden, **main**(), anropades i"!

Finns inte ➔ kompileringsfel!

Metoden length() är inte static,
men vi *anger* ett objekt (str)
➔ fungerar bra!

Statiska metoder: Main

- När du **startar ett Python-program…**
  - …körs kod på toppnivån i filen
  - Men Java har ingen ”körbar” kod på toppnivån!

- När du **startar ett Java-program**...
  - Du bestämmer vilken **klass** som ska "startas" (CircleTest)
  - Java skapar ***inte*** ett objekt av den typen
    - Laddar in själva ***klassen***
    - Letar efter en ***speciell*** statisk metod: **public static** void main(String[] args)
    - Om den existerar: Anropar den

```
public class CircleTest {
    public static void main(String[] args) {
        ...
    }
}
```

args = kommandoradsparametrarna

Hjälpklasser

jonkv@ida

- **Bara** statiska metoder ➔ en ”**hjälpklass**” (utility class)
  - Används i **undantagsfall**
    - Flera associerade funktioner
      som inte ”hör ihop” med andra klasser – som java.lang.Math
    - Kod som är komplex eller används av flera klasser
      (annars kan det placeras i användande klassen!)