Objektorientering: Lagring, räckvidd och livstid

Tre sorters variabler,
två sorters metoder…

Räckvidd och livstid

jonkv@ida

- Variabler (lokala och medlemsvariabler) har:

Scope (räckvidd)	Livstid
Från vilken del av koden kan man *nå* variabeln?	**Hur länge finns variabeln(s värde)? 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 "finns kvar", men *inte här*,
// 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)

jonkv@ida

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)

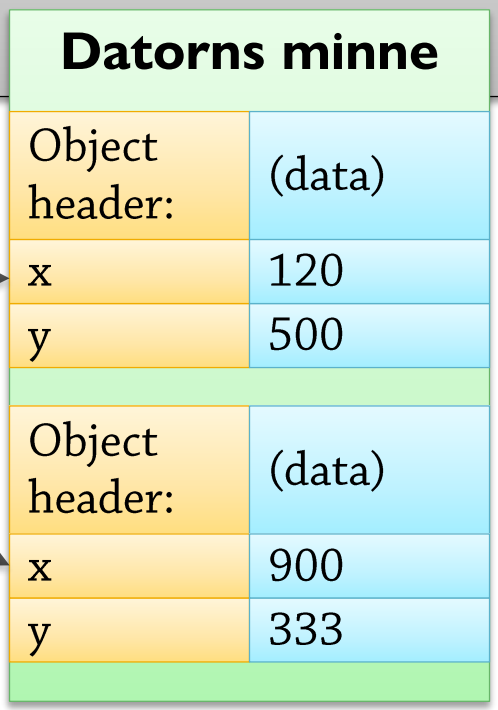
Fält = medlemsvariabel = instansvariabel

```
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 variabler, *samma* namn



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	120
y	500

Anrop till minSpelare.setX() ➔ minSpelare.x finns redan, ”lever fortfarande” ➔ ändra värdet

Anrop till minSpelare.setX() ➔ minSpelare.x finns redan, ”lever fortfarande” ➔ använd värdet

Fält eller lokal variabel?

Som i åtkomsträttigheter:

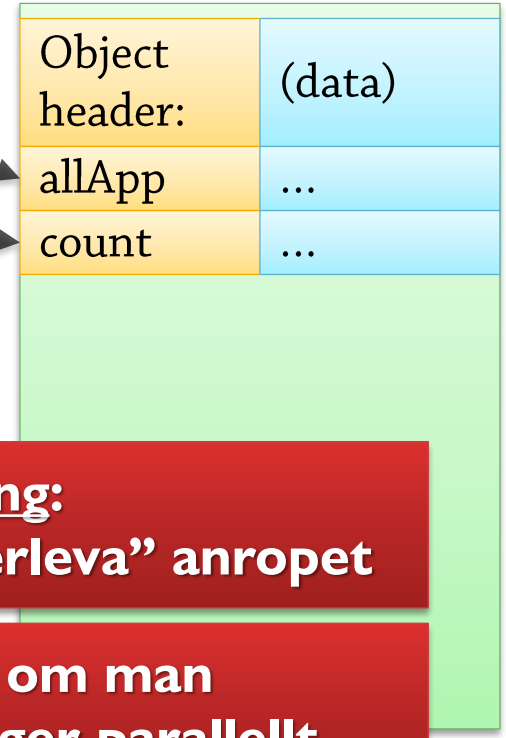
Minska om du kan!

Fält: Inte i onödan – modellering, minne

jonkv@ida

- **Varning**: Använd inte fält när lokala variabler räcker!
  - Fält:
    - **Objektets tillstånd**, som ska **bevaras så länge objektet lever**
  - Lokala variabler:
    - **Metodens tillstånd**, 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 tillstånd**
  - Tar inte mer tid eller minne

```
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;
   }
}
```

Object header:	(data)
allApp	…
~~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:
  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!

…och i multitrådade program kan flera trådar anropa samma metod på samma gång – så det behövs inte *flera* metoder för att det ska gå fel!

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

jonkv@ida

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

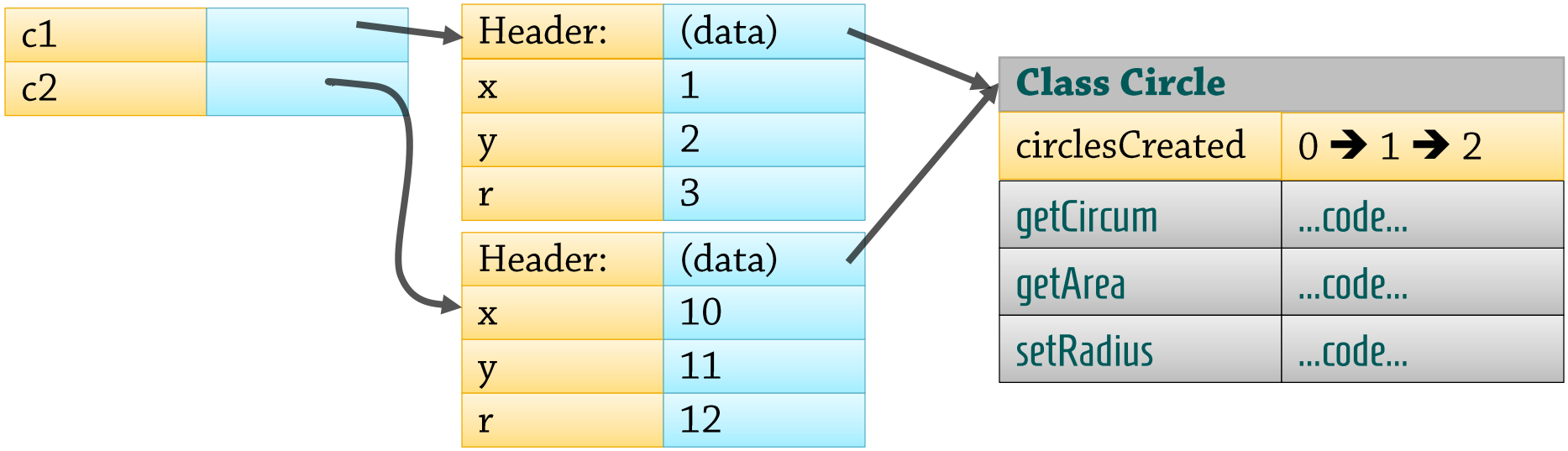
Class Circle	
circlesCreated	0
getCircum	...code...
getArea	...code...
setRadius	...code...

- Kan redan komma åt fältet:
  - System.out.println(Circle.circlesCreated); **// (Om man har rätt access)**

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!)

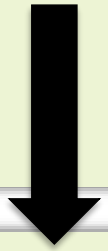
```
public class Circle {
   private static int circlesCreated = 0;
   public double x, y, r;

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

Statiska fält: Namngivna konstanter

jonkv@ida

- 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: Rätt indelning?

jonkv@ida

- Mönster: **Samla alla konstanter**”
  - ```
    public class Constants {
        public static final double PI = 3.14159;
        public static final int BUTTON_POS = 200;
        public static final String PLAYER_NAME_1
                = "Main Player";
    }
    ```

Ofta dålig indelning, *antimönster*: ”Här är allt som är konstant i hela projektet”

- ***Konstanterna hör ofta till någon annan*** – placera dem där!
  - GameGUI hanterar gränssnittet [och vet/äger allt om det, inklusive positioner]
  - ```
    public class GameGUI {
        public final int BUTTON_POS = 200;
        …
    }
    ```

Bättre: ”Här är allt som har med grafiska gränssnittet att göra”

Statiska fält: Finns alternativ

jonkv@ida

- Konstanter kan också 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

```
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 ...
   }
}
```

Ett undantag:

Om det tar tid att beräkna värdet

Statiska fält / globala variabler: Fall inte för frestelsen!

Exempel: Poänglista i Tetris (1)

- Exempel: **Poänglista i Tetris**
  - **Just nu** ska det bara finnas **en** lista:
    Den är i någon mening *global*
  - Några delar av koden (GUI, spara-på-fil, …)
    behöver tillgång till den

- Hmmm. När en variabel är **static**
  behöver man inte hitta ”rätt objekt”…
  - Och är den **public**
    kan vem som helst komma åt den…
  - *”Trevligt! Mindre data att skicka runt!”*
    …eller?

Exempel: Poänglista i Tetris (2)

- **Varning 1A:** Lagra all information *direkt* som statiska variabler

```
public class HighscoreList
{
   public static List<Score> scores;
   public static int maxScore;
}
```

Alla fält är statiska och publika ➔ vem som helst kommer åt dem

Inte objektorienterat:
Det finns inget *objekt* av HighscoreList-typ
➔ kan inte skicka listan som parameter,
kan inte skapa flera listor,
kan inte använda ärvning [senare], …

KOMPLETTERING!

Exempel: Poänglista i Tetris (3)

- **Varning 1B:** Privata statiska variabler

```
public class HighscoreList
{
   private static List<Score> scores;
   private static int maxScore;
   public static void addScore(Score newScore) {
      scores.add(newScore);
      scores.sort(...);
   }
   public static Score getHighestScore() {
      return scores.get(0);
   }
}
```

Alla fält är statiska men privata

Användning: Anropa ”global funktion”
HighscoreList.addScore(theScore);

Samma problem som tidigare
Informationen är fortfarande global

KOMPLETTERING!

Exempel: Poänglista i Tetris (4)

- **Varning 2**, **Singleton**: Börja som en vanlig klass…

```
public class HighscoreList {
   public List<Score> scores;
   public void addScore(Score newScore) { … }
   public Score getHighestScore() { … }
```

  - …men se till att bara **en** instans skapas, och ge **global** tillgång till den

```
   // Privat konstruktor, så bara klassen själv kan skapa instanser.
   private HighscoreList() { … }

   // Klassen skapar ETT objekt av denna typ
   public static final HighscoreList
      INSTANCE = new HighscoreList();
}
```

(1) Kan fortfarande komma åt listan utan att ha "fått" den
➔
Svårare att förstå dataflödet

(2): Fortfarande ett fundamentalt designbeslut att det bara kan finnas 1 global highscorelista –
får liknande konsekvenser!

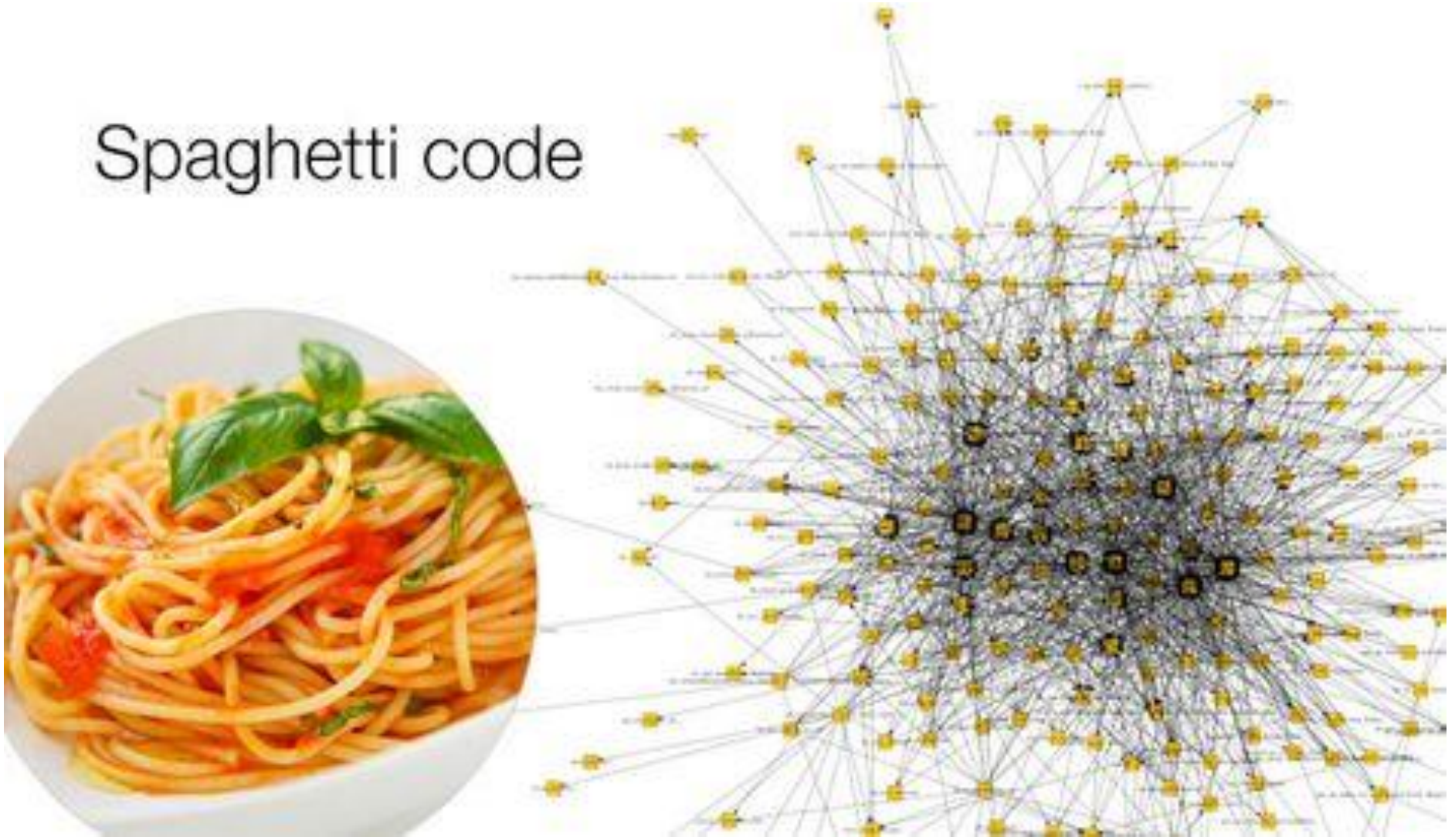
  - …och använd sedan denna instans

```
HighscoreList hs = HighscoreList.INSTANCE;
hs.addScore(theScore);
```

Static: Röriga beroenden

jonkv@ida

- Global tillgång ger **röriga beroenden**, en form av **spaghettikod**



Exempel: Poänglista i Tetris (5)

jonkv@ida

- Bäst: Låt HighscoreList och Board vara **helt vanliga** klasser…

```
public class HighscoreList {
   public HighscoreList() { … }

   public List<Score> scores;
   public void addScore(Score newScore) { … }
   public Score getHighestScore() {  }
}
```

- …Men hur ser vi till att det bara blir **en** highscorelista, **ett** Board?

Exempel: Poänglista i Tetris (6)

jonkv@ida

- **Skapa** helt enkelt bara **en** av varje
  - Du har ju kontrollen över din kod!

```
public class Tetris {
   public static void main(String[] args) {
      HighscoreList highscores = new HighscoreList();
      Board board = new Board(highscores, ...);
      HighscoreViewer viewer = new HighscoreViewer(highscores, ...);
      Controller ctrl = new Controller(board, ...);
   }
}
```

Skicka info till de som behöver!

Finns bara 1 lista: Här skapas den

Finns bara 1 spelbräde: Här skapas det

Vill du någon gång ha 2 listor är allt förberett!

Metoder på klassnivå
Statiska hjälpklasser

Exempel: Absolutvärde

jonkv@ida

- 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 *infor* 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

jonkv@ida

- 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 ➔
CircleTest.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!)

Igen: Varning för överanvändning av static

- Även om det bara finns *ett* objekt av en typ,
  är det normalt *objektet* och inte *klassen* som *gör* något