# TDDD78

## Objektorientering i Java, del 4

Hur vet man om två objekt är *lika*?

Hur *undviker* man objekt – och när?

jonas.kvarnstrom@liu.se – 2016

# Identitet och likhet

# Identitet och likhet



- Är **identitet** och **likhet** samma sak?

> **– Oj, vi har samma kläder på oss idag!**
> **– Nej, men likadana!**
>
> *– Besserwisser*

- **Två rutor** som:
  - Är **exakt likadana**
  - Men inte **samma** – de har **varsin identitet**

# <u>Identitet</u> i Java

(vad är "samma" objekt?)

# Identitet



- "Samma objekt" **betyder** "samma adress"
  - **Ser** jag *två saker* i minnet,
    så är de inte **samma** sak:
    Då vore de ju *en sak*
- Olika adress = olika identitet!
  - Även om *innehållet (tillståndet)*
    är lika

**Datorns minne**

| | |
|---|---|
| Object header: | (data) |
| x | 12.7 |
| y | 4.512 |
| r | 0.0002 |

| | |
|---|---|
| Object header: | (data) |
| x | 12.7 |
| y | 4.512 |
| r | 0.0002 |

# Identitet och ==



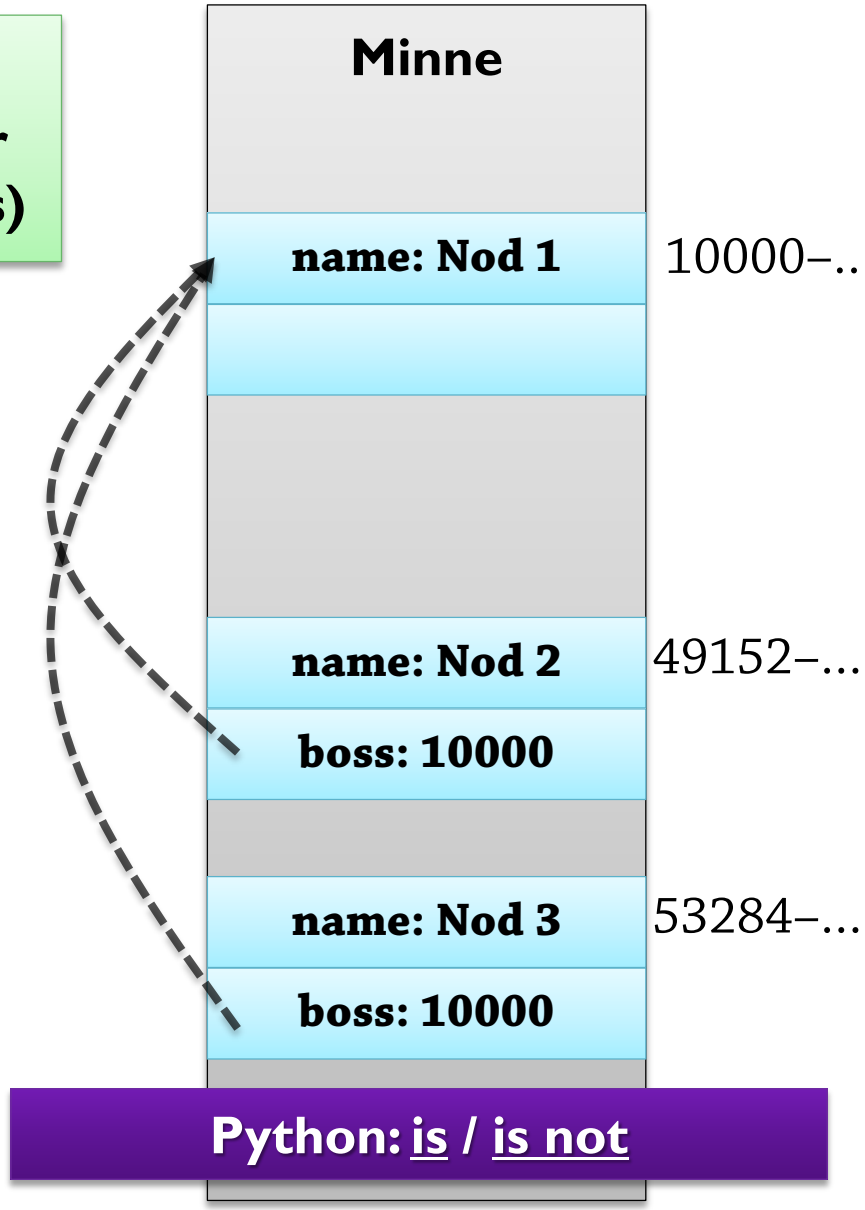
== jämför <u>värdet</u> på två variabler

Värdet av en objektvariabel är en <u>pekare</u> (adress)

== jämför pekare

== testar om två variabler pekar på <u>samma</u> objekt!

**nod2 != nod3**
**nod2.boss == nod3.boss**



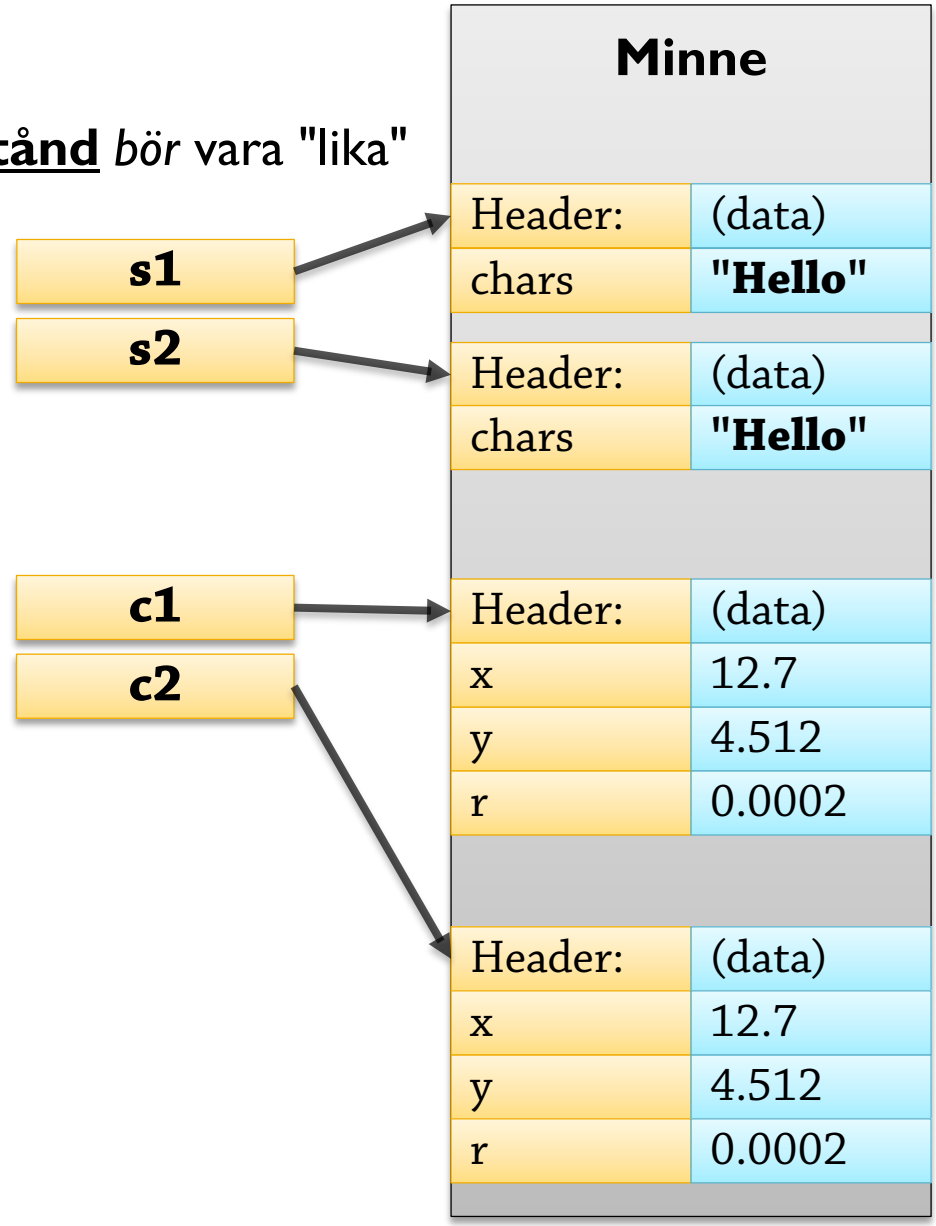
# Likhet i Java

# Likhet = identiskt tillstånd?



jonkv@ida

- Vad **<u>betyder</u>** likhet?
  - 1) Två objekt med **<u>identiskt tillstånd</u>** *bör* vara "lika"
    - Exempel: Två strängobjekt på olika plats i minnet **<u>innehåller samma tecken</u>**
    - s1 != s2, men vi tycker objekten är "lika"
    - Exempel: Två cirkelobjekt med samma position och radie
    - c1 != c2, men vi tycker objekten är "lika"



# Likhet: Även vid olika tillstånd!



- Scenario: Vi vill hitta fel i cirkelklassen
  - Lägger till **tidsstämpel** för när varje objekt skapades
  - Objekt med **olika tidsstämpel** kan anses "**lika**"!

- Scenario: Vi håller reda på datum och tider
  - Fält för datum, klockslag och **tidszon**
  - Vi vill (kanske) att
    12:00 GMT är "samma tid" som
    13:00 WET (Western European Time)

**Datorns minne**

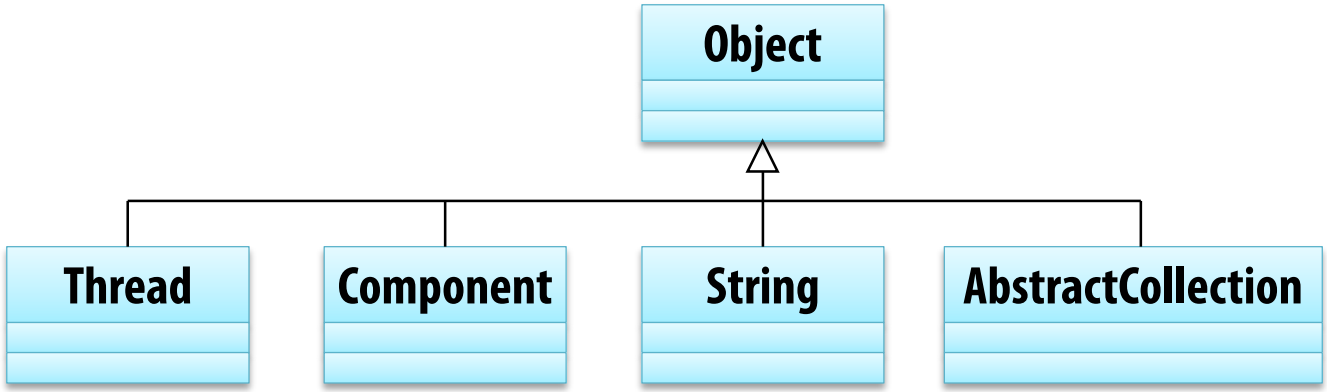
| | |
|---|---|
| Object header: | (data) |
| x | 12.7 |
| y | 4.512 |
| r | 0.0002 |
| created | 4711.31 |
| | |
| Object header: | (data) |
| x | 12.7 |
| y | 4.512 |
| r | 0.0002 |
| created | 4715.77 |

**→ "Likhet" är inte samma sak som "alla fält har samma värde"**

# Likhet: Inte fördefinierad



- **Vi** måste definiera likhet – vad menar vi egentligen?
  - Ska *finnas* för alla klasser
    - Klassen **Object** definierar metoden **equals**(), ärvs till alla andra klasser




  - Kan behöva vara *olika* för olika klasser
    - Använd overriding för att definiera om **equals**()

  - **Används** av *många* klasser och metoder
    - List.contains(element) tittar om listan har ett element som är *lika / equal*
    - List.indexOf(element) ger positionen för ett element som är *lika / equal*

# Likhet: Metodsignatur



- Måste kunna **jämföra** objekt **utan att veta typen i förväg**!
  - Lösning: equals() tar Object som parameter
    - ➔ Accepterar vilken typ av objekt som helst

**Object.java**

```
public class Object {
    public boolean equals(Object other) {
        ...
    }
}
```

**MyList.java**

```
public class MyList {
    public boolean contains(Object other) {
        for (int i = 0; i < size(); i++) {
            Object atThisPos = get(i);
            if (atThisPos.equals(other)) return true;
        }
        return false;
    }
}
```

**equals() *finns* i alla objekt – även atThisPos**
**equals() *accepterar* alla objekt – även other**

# Likhet: Standardbeteende



- Måste ha en **första implementation** i **Object**
  - Men **Object** har inga fält
  - Enda alternativen:
    - Två **Object** anses alltid **lika** (per default)
    - Två **Object** anses alltid **olika** (per default)

  - Vill inte **riskera** att returnera sant om objekten borde ses som olika:

**Object.java**

```
public class Object {
    public boolean equals(Object other) {
        return this == other;
    }
}
```

# Likhet: Standardbeteende



- I andra klasser: Overriding
  - Åtminstone *om* man behöver jämföra objekt av denna typ!

**Circle.java**

```
public class Circle {
    // Cirklar jämför man så här!
    public boolean equals(Object other) { ... }
}
```

**Parametern <u>måste</u> vara av typ Object, inte Circle: Vi måste kunna jämföra en cirkel med <u>vad som helst</u>!**

# Likhet: Några krav



- Några **krav** på likhet (en *ekvivalensrelation*):
  - Måste vara **reflexiv**: x.equals(x)
  - Måste vara **symmetrisk**: x.equals(y) ➔ y.equals(x)
  - Måste vara **transitiv**: x equals(y) och y.equals(z) ➔ x.equals(z)
  - Får inte returnera sant för null: x.equals(null) == **false**

**Ger ramar att hålla oss till**

# Implementera equals(): Circle





- En equals() för Circle:

```
public class Circle {
   protected double x, y, r;
   public boolean equals(final Object o) {
      if (this == o) return true;      // Alltid lika med sig själv
      if (o == null) return false;     // Aldrig lika med null
      if (getClass() != o.getClass()) return false; // Måste vara samma klass

      final Circle that = (Circle) o;

      if (this.x != that.x) return false;
      if (this.y != that.y) return false;
      if (this.r != that.r) return false;

      return true;
   }
}
```

**OK, det var en cirkel:**
**Se till att få rätt pekartyp**

**Testar våra egna krav:**
**Samma x, y och radie**

**Allt OK: Är lika**

# Implementera equals(): Objektfält





- När vissa **fält** är **objektpekare**: Hur jämför vi?

```
public class ColorCircle {
    protected double x, y, r;
    protected Color color;
    public boolean equals(final Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false;

        final ColorCircle that = (ColorCircle) o;
        if (this.x != that.x) return false;
        if (this.y != that.y) return false;
        if (this.r != that.r) return false;
        if (this.color != that.color) return false;

        return true;
    }
}
```

**!= testar om that.color pekar på samma färgobjekt**

```
Circle gc1 = new ColorCircle(10, 11, 12, new Color(0,0,0));
Circle gc2 = new ColorCircle(10, 11, 12, new Color(0,0,0));
```

**gc1.equals(gc2) är falskt!**

# Implementera equals(): Objektfält



- Använd equals() för **objektfält**!

```
public class ColorCircle {
    protected double x, y, r;
    protected Color color;

    public boolean equals(final Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false;

        final ColorCircle that = (ColorCircle) o;

        if (this.x != that.x) return false;
        if (this.y != that.y) return false;
        if (this.r != that.r) return false;
        if (!this.color.equals(that.color)) return false;

        return true;
    }
}
```

**equals() testar om det är samma <u>färgnyans</u>**

# Implementera equals(): Objektfält, null





- När vissa **fält** kan vara **null**:

```
public class ColorCircle {
    protected double x, y, r;
    protected Color color;      // Kan vara null

    @Override public boolean equals(final Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false;

        final ColorCircle that = (ColorCircle) o;
        if (this.x != that.x) return false;
        if (this.y != that.y) return false;
        if (this.r != that.r) return false;
        if (this.color != null && !this.color.equals(that.color)) return false;
        if (this.color == null && that.color != null) return false;

        return true;
    }
}
```

**Har vi en färg? Kolla att den är lika med det andra objektets.**

**Saknar vi färg? Kolla att den andra också saknar det.**

# Implementera equals(): Ärvning



- När vi behöver låta **superklassen** testa sin likhet:

```
public class Circle {
   protected double x, y, r;
   public boolean equals(Object other) {
      if (this == o) return true;          // Alltid lika med sig själv
      if (o == null) return false;         // Aldrig lika med null
      if (getClass() != o.getClass()) return false; // Måste vara samma klass
      … testa x, y, r …
   }
}
```

```
public class GraphicCircle extends Circle {
   protected Color outline, fill;

   public boolean equals(final Object o) {
      if (!super.equals(o)) return false;

      final GraphicCircle that = (GraphicCircle) o;
      if (!outline.equals(that.outline)) return false;
      if (!fill.equals(that.fill)) return false;
      return true;
   }
}
```

**Superklassen testar:**
**null, klass, superklassens fält**
**Vi testar:Våra nya fält**

Tre sorters variabler,
med olika lagring och livstid

# Tre sorters variabel (1): Lokal



## Lokal variabel

- Deklareras i en metod
- Varje **anrop** får sin egen "kopia":
  Två anrop till foo() ➔
  egna variabler, med samma namn 'x'
- Kort livstid: Anropet avslutas ➔ variabeln försvinner

```
public class MyGame {
   public void paint() {
      int x = calcX();
      int y = calcY();
      drawPlayerAt(x, y);
      ...
   }
}
```

***Tillfälliga* värden,**
**hör till *metodanropet***

**När vi är klara, släng bort variablerna**

# Tre sorters variabel (2): Fält



## Fält / medlemsvariabel

- Deklareras i en klass
- Varje **instans/objekt** får sin "kopia":
  Två olika objekt ➔ egna fält, med samma namn 'x'
- Två *anrop till metoder* i samma objekt ➔
  *samma* variabel x!
- Längre livstid: Objektet slängs bort ➔
  variabeln försvinner

```
public class MyProg {
   private int x, y;
   public void paint() {
      drawPlayerAt(x, y);
   }
   public void setX(int x) {
      this.x = x;
   }
}
```

**Värdena hör till *objektet*,**
**och måste finnas kvar längre**

***Används* från metoder, kanske flera**
**Kan (eventuellt) *ändras* av andra metoder**

# Varning för fält (1)



- **Varning**: Använd inte fält när lokala variabler räcker!
  - Exempel:
    Här finns listan kvar ända till nästa anrop,
    trots att ingen annan behöver den – *minnesläcka?*

```
public class FieldCollision {
    private List list;
    public List getAllMembers() {
        list = new ArrayList<>();
        // Lägg till objekt i list
        // Lägg till objekt i list
        // Lägg till objekt i list
        return list;
    }
}
```

# Varning för fält (2)



- **Varning**: Använd inte fält när lokala variabler räcker!
  - Exempel:
    Med **multitrådade** program kan man få flera *samtidiga anrop*

```
public class FieldCollision {
    private List list;
    public List getAllMembers() {
        list = new ArrayList<>();
        // Lägg till objekt i list
        // Lägg till objekt i list
        // Lägg till objekt i list
        return list;
    }
}
```

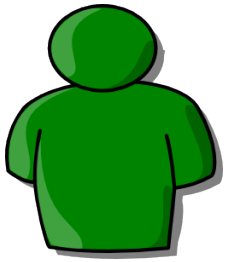
**Halvvägs genom getAllMembers() anropar någon annan samma metod**

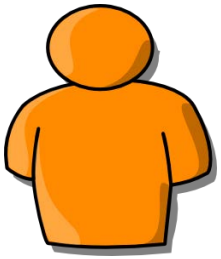
**Fältet list byts ut,**
***även för det första anropet!***

**Ena anropet ger 2 element, andra 4**

# Varning för fält (3)







```
public List getAllMembers() {
    list = new ArrayList<>();
    // Lägg till objekt i list
    // Lägg till objekt i list
    // Lägg till objekt i list
    return list;
}
```

```
public List getAllMembers() {
    list = new ArrayList<>();
    // Lägg till objekt i list
    // Lägg till objekt i list
    // Lägg till objekt i list
    return list;
}
```

```
private List list;
```

# Tre sorters variabel (3): Statisk



## Statiskt fält

- Deklareras i en klass, ***static***
- Varje **programkörning** har sin egen "kopia", allokeras *statiskt* (inte *dynamiskt* per objekt / anrop)
- Ett enda värde lagras, i **klassen**
- Ligger inte i ett objekt; behöver inte ens något objekt för att komma åt värdet

```
public class Circle {
   public 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 skapades**

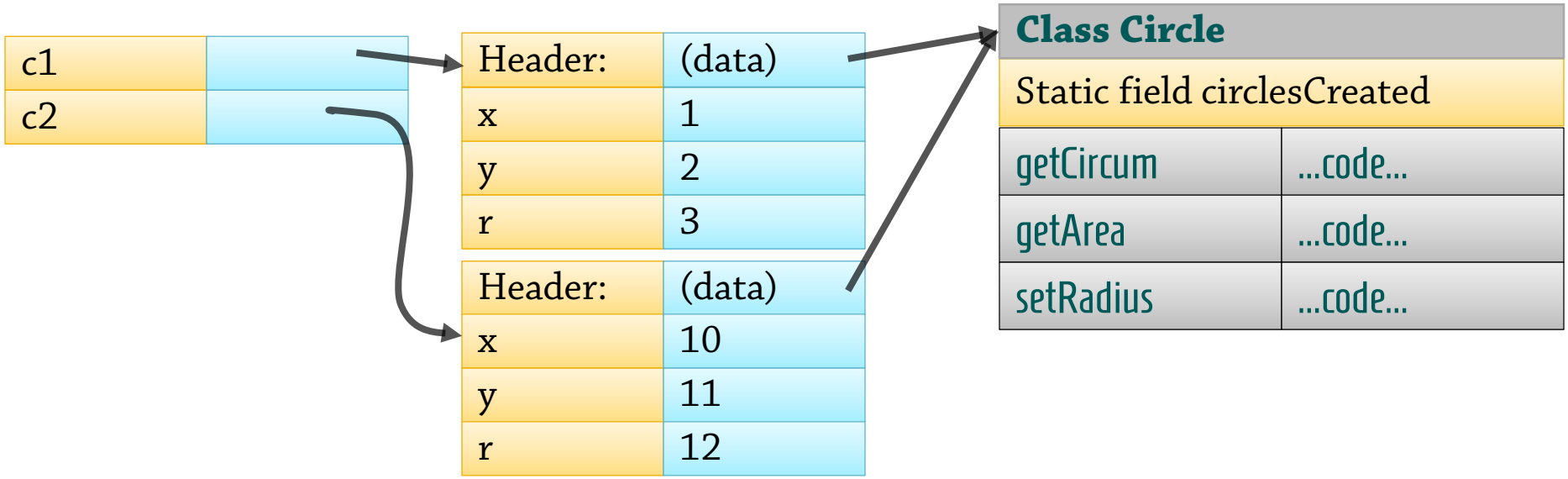
**Värdena hör till *klassen*, och måste finnas kvar efter objekten försvinner**

# Statiska fält 1: Lagring



- Illustration: Statiska fält lagras i *klassen*
  - Circle c1 = **new** Circle(1, 2, 3);
    Circle c2 = **new** Circle(10,11,12);
    System.out.println(Circle.circlesCreated);

**Åtkomst via *klassen*, inte ett objekt**



# Statiska fält 2: Som 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**

**Vanlig anledning till komplettering!**

# Statiska fält 3: "Globala" variabler



- Kan också användas till "globala" **variabler**, åtkomliga överallt

**Varning!**

**Globala variabler ger ofta problem!**

**Använd dem bara i undantagsfall!**

# Exempel: Poänglista i Tetris (1)





- Tetris:
  - **Alla** som spelar parallellt ska ha **samma** highscore-lista
    - Detta faktum ska **aldrig** ändras:
      Vi kommer **aldrig** att vilja dela upp spelare i grupper med olika listor
  - ➔ Globalt tillstånd kan **möjligen** vara OK

# Exempel: Poänglista i Tetris (2)



- Alternativ 1: Lagra all information *direkt* som statiska variabler

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

**Användning:**
**HighscoreList.addScore(theScore);**

**Det finns inget *objekt* av HighscoreList-typ**
**➔ kan inte skicka som parameter,**
**kan inte override:a metoder,**
**kan inte implementera gränssnitt, …**

# Exempel: Poänglista i Tetris (3)



- Alternativ 2, **Singleton**: Låt listan vara ett objekt
  - Men se till att bara **en** instans skapas, och ge enkel tillgång till den

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

   // Skapa bara objekt av denna typ EN gång, när klassen laddas.
   private static final HighscoreList INSTANCE = new HighscoreList();

   // Låt andra komma åt det unika objektet!
   public static HighscoreList getInstance() {
      return INSTANCE;
   }

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

**Problem: Vem som helst kan komma åt listan utan att ha "fått" den**
**➔**
**Kan bli svårare att förstå programmets struktur**
**➔**
**Oftast bra att undvika (skicka istället objektet till de som behöver det)**

**Användning:**

```
HighscoreList hs = HighscoreList.getInstance();
hs.addScore(theScore); // Vanligt objekt!
```

# Exempel 2: Spelbräde i Tetris



- Labb 4, Tetris:
  - **Just nu** spelas bara **ett spel i taget**
    - Frestande att låta Board vara en singleton
    - Alla kan direkt komma åt den, behöver inte skickas som metodparameter
  - Senare: Vill kunna spela flera spel parallellt, över nätet
    - Varje spel ska ha sitt eget Board
    - Måste göra om koden
  - ➔ Globalt tillstånd gör förändringar svårare!

**Använd bara globala tillstånd om ni är helt säkra på att man inte har någon anledning att ändra det…**

# Metoder på klassnivå
# Statiska hjälpklasser

# Två sorters metoder



## Instansmetoder

- Anropas för ett specifikt objekt: **myCircle.getRadius()**
- Kan använda "samma objekt": **this**
  - Anropa andra metoder
  - Använda fält

## Statiska metoder (klassmetoder)

- Anropas för en **klass**: **Circle.getPI()**
- Det närmaste man kan komma till en "global funktion"

# Statiska metoder: Exempel 1



- 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) {
        ...
    }
}
```

# Statiska metoder: Exempel 2





- Att anropa metoder i samma klass:

```
public class CircleTest {
    public static void main(String[] args) {
        printUsage1();
        printUsage2();
    }
    public void printUsage1() {
        System.out.println("Usage: java MyMain
    }
    public static void printUsage2() {
        System.out.println("Usage: java MyMain [args]");
    }
}
```

printUsage1 är inte static
Samma som **this**.printUsage1()
Men **this** = "objektet som den här metoden, main(), anropades i"
Finns inte → kompileringsfel!

Samma som CircleTest.printUsage2()
Fungerar bra!

# Statiska metoder: Exempel 3





- Men statiska metoder kan så klart anropa icke-statiska!
  - Bara vi anger *i vilket objekt*…

```
public class CircleTest {
    public static void main(String[] args) {
        String first = args[0];
        System.out.println(first.length());
    }
}
```

println() och length() är inte statiska
Fungerar bra:
Vi försöker inte använda **this**

# Statiska hjälpmetoder



- En metod som "roten ur":
  - Hör inte naturligt till ett specifikt objekt
  - Behöver inte lagra information som finns kvar när den returnerat

- Kan vara en **statisk hjälpmetod**
  - Exempel: java.lang.Math
    - 
```
public class Circle {
  public final double x, y, r; // center coordinates, radius

  public boolean contains(double a, double b) {
    // Is point (a,b) inside this circle?
    final double dx = x – a;
    final double dy = y – b;
    final double distance = Math.sqrt(dx * dx + dy * dy);
    if (distance < r) return true; else return false;
  }
}
```

# Hjälpklasser



- **Bara** statiska metoder ➔ en ”**hjälpklass**” (utility class)
  - Används i **undantagsfall**
    - Flera associerade funktioner
      som inte ”hör ihop” med ett specifikt objekt
    - Komplex kod eller används av flera klasser
      (annars kan det placeras i användande klassen!)
    - …
  - Att tänka på:
    - Vill **inte** kunna skapa objekt av denna klass
    - Skapa en **privat konstruktor**!

```
public final class Math {
    /**
     * Don't let anyone instantiate this class.
     */
    private Math() {}
    …
}
```