EDAA45 Programmering, grundkurs

Läsvecka 12: Scala och Java

Björn Regnell

Datavetenskap, LTH

Lp1-2, HT 2016

12 Scala och Java

- Veckans labb: `lthopoly-team`
- Jämförelse Scala och Java
- Variabeldeklarationer i Java
- Loopar i Java
- Huvudprogram i Java
- Array i Java
- ArrayList
- Exempel: Polygon med ArrayList
- Generisk klass
- Utökad for-sats: ”for-each”
- ”Wrapper classes” och ”auto-boxing”
- Fallgropar vid autoboxing
- Fallgrop med generiska samlingar och equals
- Fördjupning
- Grumligt- och Nyfiken-på-lådan

Veckans labb: lthopoly-team

Veckans labb: `lthopoly-team`

Förberedelse:

- Gör övning `scalajava`:
  - Övning 1: Översätt spelet Hangman från Java till Scala
  - Övning 2: Översätt Point från Scala till Java
  - Övning 3: Autoboxing
- Studera givna koden: workspace/w11_lthopoly_team

Grunduppgift:

- Implementera en förenklad variant av monopol i terminalen.

Extrauppgift:

- Implementera valfria utvidgningar t.ex. extra pengar vid ny runda

Jämförelse Scala och Java

Grundläggande likheter och skillnader

Några likheter:

- Kompilerar till bytekod som kör på JVM på många olika plattformar
- Statiskt typning: snabb maskinkod och kompilatorn hittar buggar vid kompilering

Liknande men viss skillnad:

Java

- **Objektorientering**, men inte ”äkta” (eng. *pure*) eftersom alla värden inte är objekt
- Primitivatyper är inte objekt; representeras effektivt, normalt **utan boxning**
- Visst stöd för **funktionsprogrammering**
- Typer måste alltid anges, ibland två gånger (variabeldeklaration + instansiering)

Scala

- **Äkta objektorienterat** eftersom alla värden är objekt, även funktioner
- `AnyVal`-instanser är äkta objekt men representeras ändå effektivt, normalt **utan boxning**
- Omfattande stöd för **funktionsprogrammering**
- Typinfo ska finnas vid kompileringstid men kan ofta härledas av kompilatorn

Några saker som finns i Scala men inte i Java

- `case`-klasser
- Lokala funktioner
- Metoder som operatorer
- Infix operatornotation
- Defultargument
- Namngivna argument
- Engångsinitialisering: `val`
- Fördröjd initialisering: `lazy val`
- Enhetlig access för `def`, `val`, `var`
- Egna setters med `def namn_=`
- Namnanrop, fördröjd evaluering
- Matchning, mönster och garder
- Klassparametrar, primärkonstruktor
- Singelobjekt: `object`
- Kompanjonsobjekt
- Inmixning: `trait`
- `for`-`yield`-uttryck
- Block är uttryck; slipper `return`
- Tomma värdet () av typen `Unit`
- Option, Some, None
- Try, Success, Failure
- Samlingarna i Scalas standardbibliotek, speciellt de **oföränderliga** samlingarna `Vector`, `Map`, `Set`, `List`, etc.
- Enhetlig användning av samlingar inkl. Array
- Innehållslikhet med == för oföränderliga strukturer, inkl. `< <= > >=` på strängar
- Implicita värden och klasser
- Mer precis synlighetsreglering, `private[this]`, `private[mypackage]`
- Flexibilitet och namnändring vid `import`
- Flexibel filstruktur och filnamngivning
- Flexibel nästling av klasser, objekt, traits
- Typ-alias och abstrakta typer med `type`
- Implicita värden och klasser
- ...

Några saker som finns i Java men inte i Scala

- Variabledeklaration utan initialisering
- Förändringsbara paramterar
- C-liknande prefix och postfix inkrementering och dekrementering: `i++ ++i i-- --i`
- C-liknande **`for`**-sats
- Semikolon efter alla satser
- Parenteser efter alla metoder
- Specialsyntax för indexering av array `[]` ej som i andra samlingar
- Uppräknade typer med **`enum`**
- Hoppa ut ur loop med **`break`**
  docs.oracle.com/javase/tutorial/java/nutsandbolts/branch.html
- **`switch`** ”faller igenom” utan **`break`**
- Nästan alltid snabbare kompilering
- Mer omfattande IDE-stöd
- Kontrollerade undantag (eng. *checked exceptions*) och **`throws`**
- ...

Exempel: typisk oföränderlig klass i Scala och Java

```
class Person(val name: String, val age: Int){
  def isAdult = age >= Person.AdultAge
}

object Person {
  val AdultAge = 18
}
```

Exempel: typisk oföränderlig klass i Scala och Java

```scala
class Person(val name: String, val age: Int){
  def isAdult = age >= Person.AdultAge
}

object Person {
  val AdultAge = 18
}
```

```java
public class JPerson {
    private String name;
    private int age;
    static final int ADULT_AGE = 18;

    public JPerson(String name, int age){
      this.name = name;
      this.age = age;
    }

    public String getName(){
        return name;
    }

    public int getAge(){
        return age;
    }

    public boolean isAdult(){
        return age >= ADULT_AGE;
    }
}
```

Lär dig detta mönster utantill så du snabbt får grejerna på plats!

Exempel: typisk oföränderlig klass i Scala och Java

```scala
class Person(val name: String, val age: Int){
  def isAdult = age >= Person.AdultAge
}

object Person {
  val AdultAge = 18
}
```

```java
public class JPerson {
    private String name;
    private int age;
    static final int ADULT_AGE = 18;

    public JPerson(String name, int age){
      this.name = name;
      this.age = age;
    }

    public String getName(){
        return name;
    }

    public int getAge(){
        return age;
    }

    public boolean isAdult(){
        return age >= ADULT_AGE;
    }
}
```

Övning:
Gör `Person` och `JPerson` **förändringsbara** så att namnet och åldern går att uppdatera och följande krav uppfylls:

- namnet ska ges vid konstruktion,
- åldern ska initieras till 0 vid konstr.,
- åldern ska aldrig kunna bli negativ.

Lär dig detta mönster utantill så du snabbt får grejerna på plats!

Exempel: typisk förändringsbar klass i Scala och Java

```
class MutablePerson(var name: String){
  private var _age = 0

  def age: Int = _age

  def age_=(a: Int): Unit =
    if (a >= 0) _age = a else _age = 0  //undantag?

  def isAdult: Boolean = age >= Person.AdultAge
}

object MutablePerson {
  val AdultAge = 18
}
```

Exempel: typisk förändringsbar klass i Scala och Java

```scala
class MutablePerson(var name: String){
  private var _age = 0

  def age: Int = _age

  def age_=(a: Int): Unit =
    if (a >= 0) _age = a else _age = 0  //undantag?

  def isAdult: Boolean = age >= Person.AdultAge
}

object MutablePerson {
  val AdultAge = 18
}
```

```java
public class JMutablePerson {
    private String name;
    private int age = 0;
    static final int ADULT_AGE = 18;

    public JMutablePerson(String name){
      this.name = name;
    }

    public String getName(){
        return name;
    }

    public void setName(String name){
        this.name = name;
    }

    public int getAge(){
        return age;
    }

    public void setAge(int age){
        if (age >= 0) {
          this.age = age;
        } else {
          this.age = 0;
        }
    }

    public boolean isAdult(){
        return age >= ADULT_AGE;
    }
}
```

Övning: Implementera dessa specifikationer

Specification **Vegitable**

```scala
/** Representerar en grönsak. */
class Vegitable(val name: String) {

  /** Returnerar nuvarande vikt i gram. */
  def weight: Int = ???

  /** Ändrar vikten till w gram.
   *  w ska vara positiv, blir annars 0 */
  def weight_=(w: Int): Unit = ???
}
```

class JVegitable

```java
/** Skapar en grönsak. */
JVegitable(String name);

/** Returnerar namnet. */
String getName();

/** Returnerar nuvarande vikt i gram. */
int getWeight();

/** Ändrar vikten till weight gram.
 *  w ska vara positiv, blir annars 0 */
void setWeight(int weight);
```

Övning: Implementera dessa specifikationer

Specification **Vegitable**

```
/** Representerar en grönsak. */
class Vegitable(val name: String) {

  /** Returnerar nuvarande vikt i gram. */
  def weight: Int = ???

  /** Ändrar vikten till w gram.
   *  w ska vara positiv, blir annars 0 */
  def weight_=(w: Int): Unit = ???
}
```

class JVegitable

```
/** Skapar en grönsak. */
JVegitable(String name);

/** Returnerar namnet. */
String getName();

/** Returnerar nuvarande vikt i gram. */
int getWeight();

/** Ändrar vikten till weight gram.
 *  w ska vara positiv, blir annars 0 */
void setWeight(int weight);
```

Fördjupning:
Kasta undantaget `IllegalArgumentException` vid försök till negativ vikt.
Läs om undantag i Java här: docs.oracle.com/javase/tutorial/essential/exceptions/

Oföränderlig datatyp i Scala och Java

En oföränderlig datatyp implementeras i **Scala** helst som en

Oföränderlig datatyp i Scala och Java

En oföränderlig datatyp implementeras i **Scala** helst som en **case**-klass:

```
case class Person(name: String, age: Int){
  def isAdult = age >= Person.AdultAge
}

object Person {
  val AdultAge = 18
}
```

En oföränderlig datatyp i **Java** med **motsvarande** funktionalitet kräver egen implementation av dessa metoder:

- en getter för varje attribut
- `equals`
- `hashcode` (förklaras i forts.kurs)
- `apply`
  (men man kallar nog den `create` el. likn.; namnet måste ju skrivas)
- `toString`
- `copy`
  (men det finns ju inte namngivna parametrar och defaultargument så denna blir osmidig)
- `unapply`
  (men det finns ju inte mönstermatchning så denna struntar man nog i)

Variabeldeklarationer i Java

Syntax för variabeldeklaration i Scala och Java

Exempel på variabeldeklarationer i

Scala

```
var i1: Int = 0
var i2 = 0
var i3 = 0: Int
var p = new Point(0, 0)
var (x, y) = (0, 0)
val a = 0
final val Constant = 42
```

- i2 härledd typ; går ej i Java
- i3 typ i uttryck; går ej i Java
- (x, y) mönster i init; går ej i Java
- **val** ger ”engångsinit”; ingen exakt motsvarighet i Java men **final** kan ofta användas i stället

Java

```
int i1 = 0;
int i4;
Point p = new Point(0, 0);
final int CONSTANT = 42;
```

- i4 ej explicit init; går ej i Scala

Loopar i Java

For-sats i Scala och Java

Scala

```
val s = "Abbasillen"

//Loopa framlänges:

for (i <- 0 until s.length) prinlnt()

//Loopa baklänges:
```

Java

Huvudprogram i Java

Huvudprogram i Scala och Java

Array i Java

Syntax för Array i Scala och Java

Primitiva arrayer i Java

- Primitiva arrayer (med [] efter typ) i Java har **fördelar**:[1]
  - Det är den snabbaste indexerbara datastrukturen i JVM: att läsa och uppdatera ett element på en viss plats är mycket effektiv om man vet platsens index.
  - Fungerar lika bra med både primitiva värden och objektreferenser
- ... men också **nackdelar**:
  - Man måste bestämma sig för antalet element vid new.
  - Man kan ta i lite extra när man allokerar om man behöver plats för fler senare, men då måste man hålla reda på hur många platser man använder och veta var nästa lediga plats finns.
  - Det är krångligt att stoppa in (eng. *insert*) och ta bort (eng. *delete*) element.
  - Vill man ha fler platser måste man allokera en helt ny, större vektor och kopiera över alla befintliga element.

[1] stackoverflow.com/questions/2843928/benefits-of-arrays

Polygon med primitiv vektorer

```
1  public class Polygon {
2      private Point[] vertices; // vektor med hörnpunkter
3      private int n;            // antalet hörnpunkter
4
5      /** Skapar en polygon */
6      public Polygon() {
7          vertices = new Point[1];
8          n = 0;
9      }
10
11     ...
```

Polygon med primitiv vektorer: stoppa in sist och vid behov skapa mer plats

Metoden `addVertex` i klassen `Polygon`
med attributet: `private Point[] vertices`

```
1     private void extend(){
2         Point[] oldVertices = vertices;
3         vertices = new Point[2 * vertices.length]; // skapa dubbel plats
4         for (int i = 0; i < oldVertices.length; i++) {  // kopiera
5             vertices[i] = oldVertices[i];
6         }
7     }
8
9     /** Definierar en ny punkt med koordinaterna x,y */
10    public void addVertex(int x, int y) {
11        if (n == vertices.length) extend();
12        vertices[n] = new Point(x, y);
13        n++;
14    }
```

Polygon med primitiv vektorer: stoppa in mitt i på angiven plats

Metoden `insertVertex` i klassen `Polygon` med attributet: **`private`** `Point[] vertices`

```
1     public void insertVertex(int pos, int x, int y) {
2         if (n == vertices.length) extend();   // utöka vid behov
3         for (int i = n; i > pos; i--) {       // flytta element bakifrån
4             vertices[i] = vertices[i - 1];
5         }
6         vertices[pos] = new Point(x, y);
7         n++;
8     }
```

ArrayList

Förändringsbar samling i Scala och Java

Varför ArrayList?

En betydande nackdel med primitiva vektorer är att vi kan behöva ”uppfinna hjulet” upprepade gånger:

- För varje ny klass med vektor-attribut (vektor av Point, Person, Turtle, ...) som vi vill ska klara insert och append, blir det en hel del att implementera och testa...

Det vore smidigt med en datastruktur ...

- som inte kräver att vi känner antalet element från början,
- där vi enkelt kan lägga till och ta bort element,
- som kan hantera element av olika typ (likt vektorer).

Från och med version 5 av Java (2004) så introducerades **generics** vilket möjliggör skapandet av klasser som kan erbjuda generell behandling av olika typer av objekt. Generiska klasser känns igen med syntaxen `Klassnamn<Typ>`, till exempel `ArrayList<Point>`

Fördjupning: se javase tutorial, mer om detta i fördjupningskursen.

Vad är ArrayList?

`ArrayList` är en standardklass i paketet `java.util` med många fördelar:

- Lagrar sina element i en snabbindexerad primitiv vektor.
- Fungerar för alla typer av objekt.
- Utökar vektorns storlek av sig själv vid behov.

Det finns också vissa nackdelar:

- Fungerar **inte** med primitiva typer `int`, `double`, `char`, ... (men det finns sätt komma runt detta)
- Kräver visst onödigt minnesutrymme om vi vet antalet från början och inte behöver automatisk utökning.
- Likt primitiva vektorer tar det tid att göra insert och delete.

Polygon med ArrayList

Klassen `Polygon`, nu med ett attribut av typen `ArrayList<Point>` som håller reda på hörnpunkterna:

```
public class Polygon {
    private ArrayList<Point> vertices; // lista med hörnpunkter

    /** Skapar en polygon */
    public Polygon() {
        vertices = new ArrayList<Point>();
    }

    ...
```

Det behövs inget attribut n eftersom vi inte själva behöver hålla reda på antalet allokerade platser: allokering, insättning, och utökning av antalet platser sköts helt automatiskt av `ArrayList`-klassen vid behov.

Viktiga operationer på ArrayList (Urval)

ArrayList

```
/** Skapar en ny lista */
ArrayList<E>();

/** Tar reda på elementet på plats pos */
E get(int pos);

/** Lägger in objektet obj sist */
void add(E obj);

/** Lägger in obj på plats pos; efterföljande flyttas */
void add(int pos, E obj);

/** Tar bort elementet på plats pos och returnerar det */
E remove(int pos);

/** Tar reda på antalet element i listan */
int size();
```

Lär dig vad som finns om ArrayList i java snabbreferens!
Läs mer om ArrayList i javadoc.

Överkurs för den nyfikne: kolla implementation av ArrayList här.

ArrayList är en *generisk* klass

- `ArrayList` är en så kallad **generisk** klass. Se t.ex. wikipedia.
- Namnet **E** är en **typparameter** till klassen.
  (Mer om detta i Programmeringsteknik – fördjupningskurs.)
- Typparameterns namn kan användas i implementationen av en generisk klass och kompilatorn kommer att *ersätta* typparametern med den *egentliga* typen vid kompilering.
- I fallet `ArrayList`: **E** ersätts med typen på de objekt som egentligen lagras i listan.

Exempel:

```
ArrayList<String> words = new ArrayList<String>();
words.add("hej");
words.add("på");
words.add("dej");
```

Övning ArrayList: new och add

Skriv kod som skapar en lista med element av typen `Point` och lägger in tre punkter i listan med koordinaterna (50, 50), (50,10) och (30, 40).

Övning ArrayList: new och add

Skriv kod som skapar en lista med element av typen `Point` och lägger in tre punkter i listan med koordinaterna (50, 50), (50,10) och (30, 40).

Lösning:

```
ArrayList<Point> vertices = new ArrayList<Point>();
vertices.add(new Point(50, 50));
vertices.add(new Point(50, 10));
vertices.add(new Point(30, 40));
```

Polygon med ArrayList: metoderna blir enklare

```
public void addVertex(int x, int y) {
    vertices.add(new Point(x, y));
}

public void move(int dx, int dy) {
    for (int i = 0; i < vertices.size(); i++) {
     vertices.get(i).move(dx, dy);
    }
}

public void insertVertex(int pos, int x, int y) {
 vertices.add(pos, new Point(x, y));
}

public void removeVertex(int pos) {
 vertices.remove(pos);
}
```

Se hela lösningen här:
compendium/examples/scalajava/list/Polygon.java

Polygon med ArrayList: iterera över alla hörnpunkter i draw

```
public void draw(SimpleWindow w) {
    if (vertices.size() == 0) {
        return;
    }
    Point start = vertices.get(0);
    w.moveTo(start.getX(), start.getY());
    for (int i = 1; i < vertices.size(); i++) {
        w.lineTo(vertices.get(i).getX(),
                 vertices.get(i).getY());
    }
    w.lineTo(start.getX(), start.getY());
}
```

Se hela lösningen här:
compendium/examples/scalajava/list/Polygon.java

Övning ArrayList: implementera metoden hasVertex

Skriv kod som implementerar denna metod i klassen `Polygon`:

```
/** Undersöker om polygonen har någon hörnpunkt med koordinaterna x, y. */
public boolean hasVertex(int x, int y) {
    ???
}
```

Utökad for-sats, även kallad for-each-sats: Smidigt sätt att iterera över alla element i en lista

- Antag att vi vill gå igenom alla element i en lista.

```
ArrayList<String> words = new ArrayList<String>();
```

Det finns två olika typer av for-satser som kan göra detta:

- Vanlig for-sats:

```
for (int i = 0; i < words.size(); i++) {
    System.out.println(i + ": " + words.get(i));
}
```

- Utökad for-sats, även kallad **for-each-sats**:

```
for (String s: words) {
    System.out.println(s);
}
```

Syntax:

```
for (Elementtyp loopvariabel: samling) { ... }
```

Utökad for-sats med vektorer

Utökad for-sats fungerar även med primitiva vektorer:

```
String[] stringArray = {"hej", "på", "dej"};
for (String s: stringArray){
    System.out.println(s);
}
```

OBS! Vi får ingen indexvariabel i utökad for-sats.

Autoboxing

Generiska klasser (t.ex. ArrayList) med primitiva typer

- Elementen i `ArrayList` anger elementens typ.
- Men vad gör man om man vil ha element av primitiva typer, så som `int` och `double`? Detta går alltså **INTE**:
  ~~`ArrayList<int> list = new ArrayList<int>();`~~

- Javas lösning på problemet består av två delar:
  - Klasser som packar in primitiva typer, (eng. *wrapper classes*)
  - Speciella regler för implicita konverteringar, s.k. ”auto-boxing” (eng. *Boxing / Unboxing conversions*)

Detta kan bli ganska komplicerat och det finns fallgropar, se kapitel 12.8 i ankboken.

(Om du är nyfiken på alla intrikata detaljer, se java tutorial och javaspecifikationen.)

Wrapper-klassen `Integer`

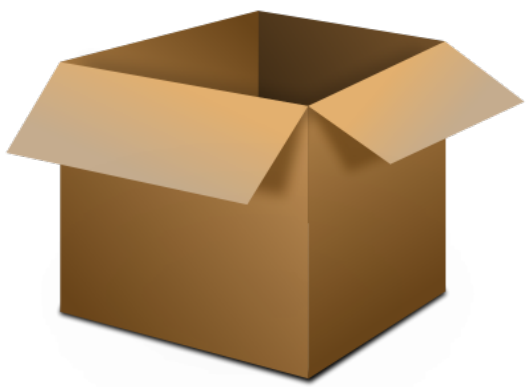
En skiss av klassen `Integer`
(ligger i paketet `java.lang` och importeras därmed implicit):

```
public class Integer {
    private int value;

    public static final MIN_VALUE = -2147483648;
    public static final MAX_VALUE = 2147483647;

    public Integer(int value) {
        this.value = value;
    }

    public int intValue() {
        return value;
    }
    ...
}
```



Javadoc för klasen `Integer` finns här:

`http://docs.oracle.com/javase/8/docs/api/java/lang/Integer.html`

Wrapper-klasser i `java.lang`

Primitiv typ	Inpackad typ
boolean	Boolean
byte	Byte
short	Short
char	Character
int	Integer
long	Long
float	Float
double	Double

OBS!

I ankboken kallas wrapper-klasserna för ”typklasser”, men termen ”type class” används ofta till något helt annat inom datalogin, vilket kan skapa förvirring.

Övning: primitiva versus inpackade typer

Med papper och penna:

- Deklarera en variabel med namnet `gurka` av den primitiva heltalstypen och initiera den till värdet 42.
- Deklarera en referensvariabel med namnet `tomat` av den inpackade (”wrappade”) heltalstypen och initiera den till värdet 43.
- Rita hur det ser ut i minnet.

Exempel: Lista med heltal

```
1  package week10.generics;
2
3  import java.util.ArrayList;
4  import java.util.Scanner;
5
6  public class TestIntegerList {
7      public static void main(String[] args) {
8          ArrayList<Integer> list = new ArrayList<Integer>();
9          Scanner scan = new Scanner(System.in);
10         System.out.println("Skriv heltal med blank emellan. Avsluta med <CTRL+D>:");
11         while (scan.hasNextInt()) {
12             int nbr = scan.nextInt();
13             Integer obj = new Integer(nbr);
14             list.add(obj);
15         }
16         System.out.println("Dina heltal i omvänd ordning:");
17         for (int i = list.size() - 1; i >= 0; i--) {
18             Integer obj = list.get(i);
19             int nbr = obj.intValue();
20             System.out.println(nbr);
21         }
22         scan.close();
23     }
24 }
```

Koden finns här: compendium/examples/scalajava/TestIntegerList.java

Specialregler för wrapper-klasser

- Om ett `int`-värde förekommer där det behövs ett `Integer`-objekt, så lägger kompilatorn automatiskt ut kod som skapare ett `Integer`-objekt som packar in värdet.
- Om ett `Integer`-objekt förekommer där det behövs ett `int`-värde, lägger kompilatorn automatiskt ut kod som anropar metoden `intValue()`.

Samma gäller mellan alla primitiva typer och dess wrapper-klasser:

`boolean`	$\Leftrightarrow$	`Boolean`
`byte`	$\Leftrightarrow$	`Byte`
`short`	$\Leftrightarrow$	`Short`
`char`	$\Leftrightarrow$	`Character`
`int`	$\Leftrightarrow$	`Integer`
`long`	$\Leftrightarrow$	`Long`
`float`	$\Leftrightarrow$	`Float`
`double`	$\Leftrightarrow$	`Double`

Exempel: Lista med heltal och autoboxing

```
1  package week10.generics;
2
3  import java.util.ArrayList;
4  import java.util.Scanner;
5
6  public class TestIntegerListAutoboxing {
7      public static void main(String[] args) {
8          ArrayList<Integer> list = new ArrayList<Integer>();
9          Scanner scan = new Scanner(System.in);
10         System.out.println("Skriv heltal med blank emellan. Avsluta med <CTRL+D>:");
11         while (scan.hasNextInt()) {
12             int nbr = scan.nextInt();
13             list.add(nbr);   // motsvarar: list.add(new Integer(nbr));
14         }
15         System.out.println("Dina heltal i omvänd ordning:");
16         for (int i = list.size() - 1; i >= 0; i--) {
17             int nbr = list.get(i);  // motsvarar: int nbr = list.get(i).intValue();
18             System.out.println(nbr);
19         }
20         scan.close();
21     }
22 }
```

Koden finns här: scalajava/generics/TestIntegerListAutoboxing.java

Fallgropar vid autoboxing

- Jämförelser med == och `!=`
- Kompilatorn hittar inte förväxlad parameterordning, t.ex. `add(pos, nbr)` i fel ordning: ~~`add(nbr, pos)`~~

Läs mer i kapitel 12.8 i ankboken.

Fallgrop med generiska samlingar: metoden contains kräver implementation av equals

Antag att vi vill implementerar `hasVertex()` i klassen `Polygon` genom att använda metoden `contains` på en lista. Hur gör vi då?

Fallgrop med generiska samlingar: metoden contains kräver implementation av equals

Antag att vi vill implementerar `hasVertex()` i klassen `Polygon` genom att använda metoden `contains` på en lista. Hur gör vi då?

```
public boolean hasVertex(int x, int y){
    return vertices.contains(new Point(x, y)); // FUNKAR INTE om ...
    // ... inte Point har en equals som kollar innehållslikhet
}
```

Vi behöver implementera metoden `equals(Object obj)` i klassen `Point` som kollar innehållslikhet och ersätter den `equals` som finns i `Object` som kollar referenslikhet, eftersom metoden `contains` i klassen `ArrayList` anropar `equals` när den letar igenom listan efter lika objekt.
Se exempel här: scalajava/generics/TestPitfall3.java
Överkurs: vissa samlingar kräver även att man implementerar hashcode

Iterera över samling i Scala och Java

Fördjupning

Undantag i Java

Fördjupning: Gränssnittet List i Java

Fördjupning: Skapa generisk Array av viss typ

Grumligt- och Nyfiken-på-lådan

Grumligt- och Nyfiken-på-lådan