EDA016 Programmeringsteknik för D

Läsvecka 10: Listor

Björn Regnell

Datavetenskap, LTH

Lp1-2, HT 2015

10 Listor

- Att göra denna vecka
- Repetition: Vektorer: utöka, stoppa in, ta bort, etc.
- ArrayList
- Exempel: Polygon med ArrayList
- Generisk klass
- Utökad for-sats: for-each
- ”Wrapper classes” och ”auto-boxing”
- Fallgropar vid autoboxing

Att göra i Vecka 10: Förstå ArrayList och skillnader/likheter jämfört med primitiva vektorer.

1. Läs följande kapitel i kursboken: 8.3-8.5, 12
   Begrepp:
   ArrayList, generisk datastruktur, utökad for-sats (for-each), automatisk in- och upp-packning (auto-boxing/-unboxing), typklasser.
2. Gör övning 9: ArrayList
3. Träffas i samarbetsgrupper och hjälp varandra
4. Gör Lab 8: vektorer, simulering av patiens

Repetition: Primitiva vektorer

- Primitiva vektorer (Array med []) i Java har **fördelar**:
  - 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.

Polygon med primitiv vektorer

```
1  package week10.vector;
2
3  public class Polygon {
4      private Point[] vertices; // vektor med hörnpunkter
5      private int n;            // antalet hörnpunkter
6
7      /** Skapar en polygon */
8      public Polygon() {
9          vertices = new Point[1];
10         n = 0;
11     }
12
13     ...
```

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     }
```

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: src/week10/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: src/week10/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.

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`

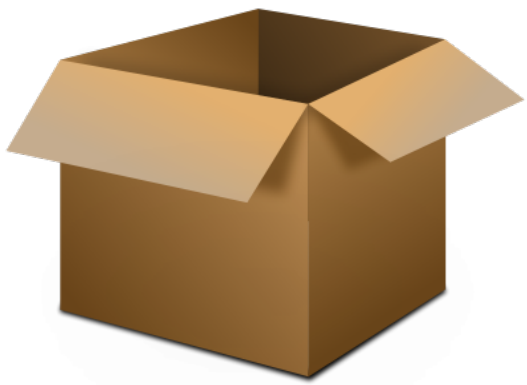
En skiss av klassen `Integer`
(ligger i paketet `java.lang` och importeras därmed implicit):

```
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: week10/generics/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`	⇔	`Boolean`
`byte`	⇔	`Byte`
`short`	⇔	`Short`
`char`	⇔	`Character`
`int`	⇔	`Integer`
`long`	⇔	`Long`
`float`	⇔	`Float`
`double`	⇔	`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: week10/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.