EDAA20
Programmering och databaser

Föreläsning 12 – ArrayList, wrapper-klasser, sorterad följd, equals

2023-10-03, Niklas Fors

Klassen `ArrayList` används för ***listor***, som hanterar insättning och borttagning åt oss. Elementen lagras internt i en vektor.

```
// Skapa en tom lista där Point-objekt kan sättas in
ArrayList<Point> points = new ArrayList<Point>();

// Lägg in punkt sist i listan
points.add(new Point(50, 50));

// Iteration
for (int i = 0; i < points.size(); i++) {
  // Gör något med punkten points.get(i)
}
```

Viktiga metoder i `ArrayList`

```
/** Skapar en tom lista med element av typen E. */
ArrayList<E>();

/** Returnerar elementet på plats pos. */
E get(int pos);

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

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

/** Returnerar antalet element. */
int size();

...
```

Inga primitiva typer!

Typargumentet får ***inte*** vara en primitiv typ:

```
ArrayList<int> numbers = new ArrayList<int>(); // Kompileringsfel
```

Istället måste man använda en ***wrapper***-*klass*:

```
ArrayList<Integer> numbers = new ArrayList<Integer>();  // OK
```

Wrapper-klasser

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

Automatisk omvandling

Använd en `Integer`-lista som om den innehåller vanliga heltal:

```
ArrayList<Integer> numbers = new ArrayList<Integer>();
numbers.add(3);
numbers.add(5);

// Summera talen
int sum = 0;
for (int x: numbers) {    // for-each-sats
  sum += x;
}
```

Java omvandlar automatiskt mellan `Integer` och `int` när det behövs.

Läs in heltal och summera dem

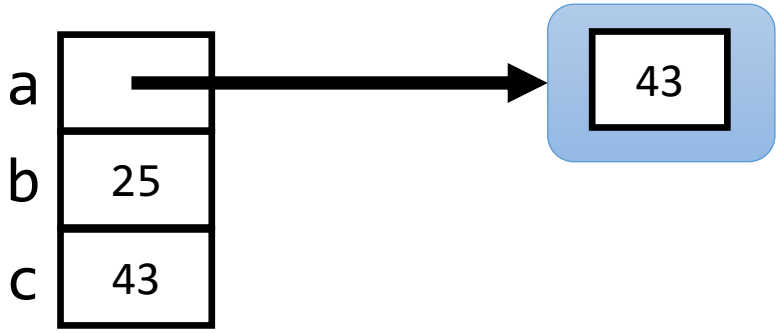
```
public class ArrayListTest {
  public static void main(String[] args) {
    ArrayList<Integer> numbers = new ArrayList<Integer>();

    // Läs in godtyckligt många heltal och lägg dem i listan
    Scanner scan = new Scanner(System.in);
    while (scan.hasNextInt()) {
      numbers.add(scan.nextInt());
    }

    // Gör något med talen i listan, exempelvis summera dem
    int sum = 0;
    for (int x: numbers) {
      sum += x;
    }
    System.out.println(sum);
  }
}
```

Wrapper-klass

Wrapper-klasser används när man behöver "packa in" ett värde av en primitiv datatyp i ett objekt.

Exempel:

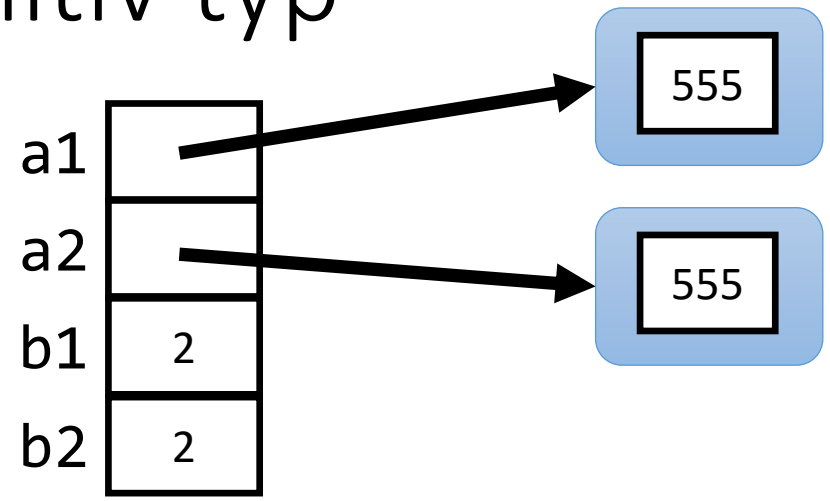
```
Integer a = Integer.valueOf(43);
int b = 25;
int c = a.intValue();
```



(`Double` har `doubleValue` istället för `intValue` osv)

Övning: wrapper-klass vs primitiv typ

```
Integer a1 = Integer.valueOf(555);
Integer a2 = Integer.valueOf(555);
int b1 = 2;
int b2 = 2;
```

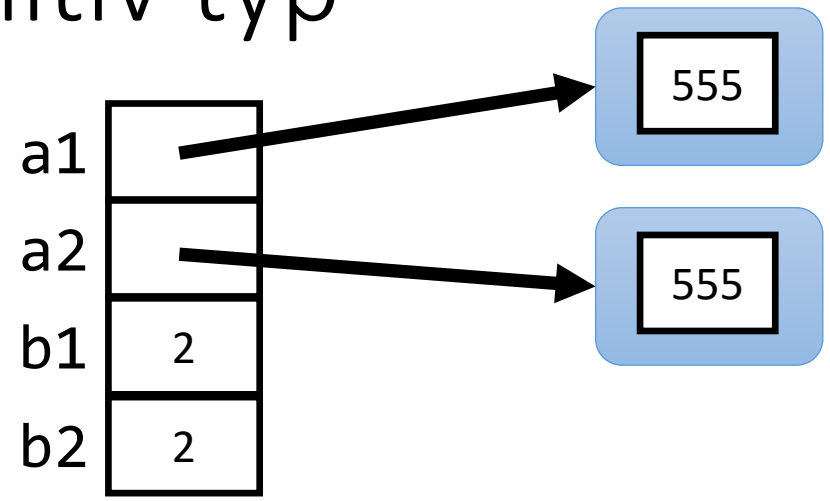



```
System.out.println(a1 == a2);
System.out.println(a1.intValue() == a2.intValue());
System.out.println(b1 == b2);
```

Vad skrivs ut?

Övning: wrapper-klass vs primitiv typ

```
Integer a1 = Integer.valueOf(555);
Integer a2 = Integer.valueOf(555);
int b1 = 2;
int b2 = 2;
```




```
System.out.println(a1 == a2);                          // False
System.out.println(a1.intValue() == a2.intValue());    // True
System.out.println(b1 == b2);                          // True
```

Vad skrivs ut?

(Ibland ger valueOf tillbaka en referens till samma objekt flera gånger pga prestandaskäl. Exempelvis för heltalen -128 till 127.)

Automatisk omvandling

Autoboxing, automatisk omvandling från `int` till `Integer` när det behövs

```
// Ekvivalenta satser:
Integer a = 42;          // autoboxing
Integer a = Integer.valueOf(42);
```

Unboxing, automatisk omvandling från `Integer` till `int` när det behövs

```
// Ekvivalenta satser:
int i = a;               // unboxing
int i = a.intValue();
```

Detta gäller för alla wrapper-klasser.

Med och utan autoboxing/unboxing

Med autoboxing/unboxing:

```
ArrayList<Integer> numbers = ...
numbers.add(3);  // autoboxing
numbers.add(5);  // autoboxing


int sum = 0;
for (int x: numbers) { // unboxing
  sum += x;
}
```

Utan autoboxing/unboxing:

```
ArrayList<Integer> numbers = ...
numbers.add(Integer.valueOf(3));
numbers.add(Integer.valueOf(5));


int sum = 0;
for (Integer x: numbers) {
  sum += x.intValue();
}
```

(Så här programmerar man dock inte)

Fler metoder

I wrapper-klasserna finns det en del metoder och konstanter.

```
// Omvandla sträng till int
int n = Integer.parseInt("42");

// Omvandla int till sträng
String s = Integer.toString(42);

// Största talet en int kan ha
int max = Integer.MAX_VALUE;
```

`ArrayList` vs vektor

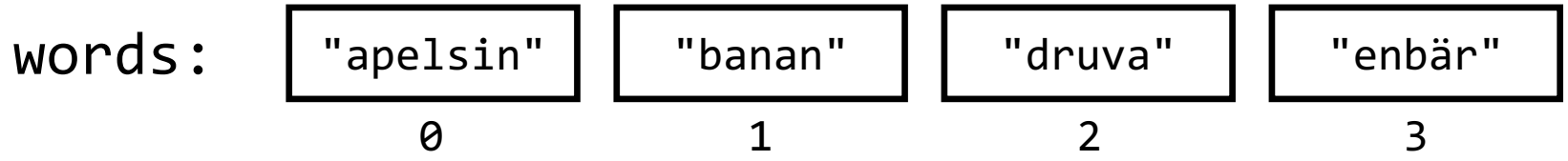
- `ArrayList` passar bra när antal element inte är bestämt. Klassen har färdiga metoder för att lägga till element, ta bort element, osv.
- En vektor passar bra för primitiva typer och när man på förhand vet antal element. Exempel:

```
int[] nbrs = new int[6];
```

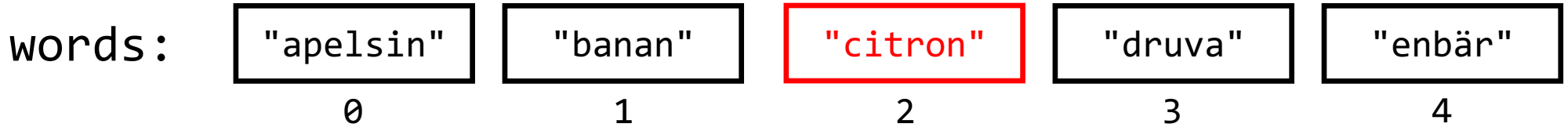
- Observera att listor skiljer sig från vektorer, exempelvis:
  - Ny lista är tom från början
  - Ny vektor har en given storlek

Insättning i sorterad följd

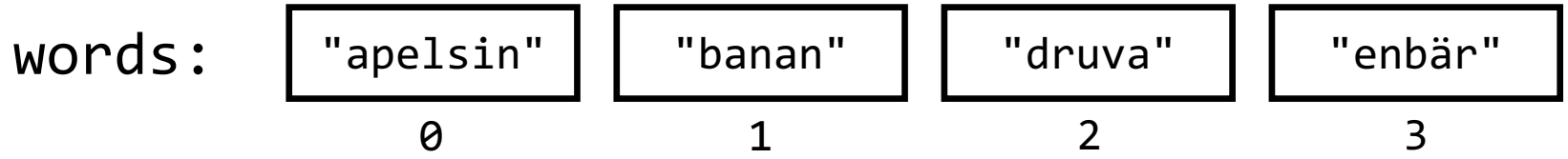
Givet en sorterad lista, exempelvis en lista av ord:



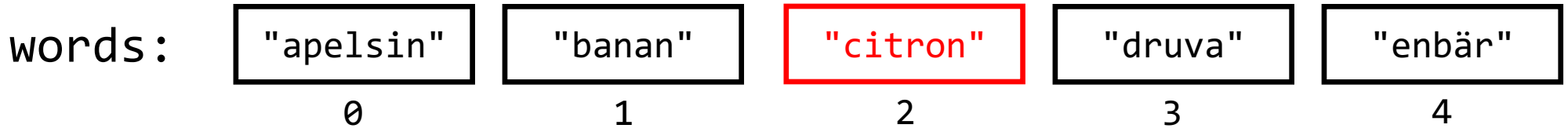
Så vill vi lägga in ett nytt element, exempelvis ordet *citron*, på rätt plats:



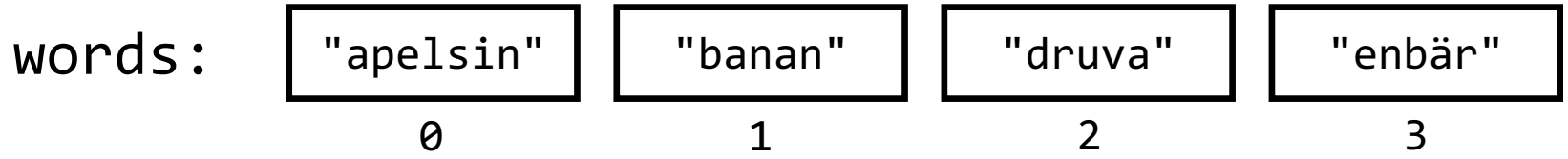
```java
// Sorterad lista av ord
ArrayList<String> words = new ArrayList<String>();
words.add("apelsin");
words.add("banan");
words.add("druva");
words.add("enbär");
```




```java
String word = "citron"; // Nytt ord som ska läggas in
words.add(2, word);     // Lägg in ordet på index 2
```

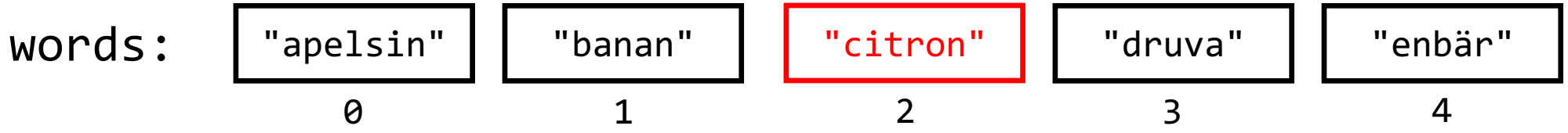


```java
// Sorterad lista av ord
ArrayList<String> words = new ArrayList<String>();
words.add("apelsin");
words.add("banan");
words.add("druva");
words.add("enbär");
```




```java
String word = "citron"; // Nytt ord som ska läggas in
words.add(2, word);     // Lägg in ordet på index 2
```

Att ordet *citron* ska läggas in på index 2 ska vi beräkna!

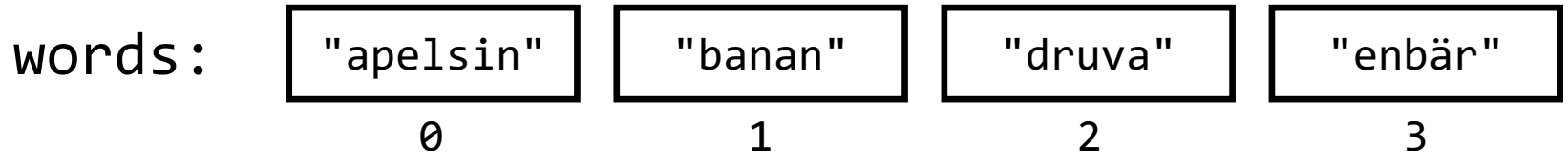


Algoritm: insättning i sorterad följd

Lösningsidé: gå igenom listan tills vi kommer till ett "för stort" ord

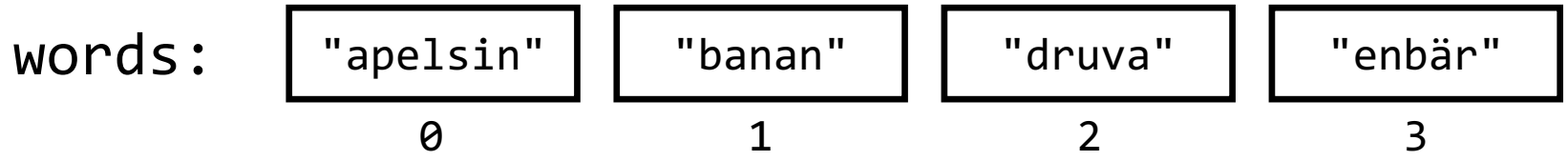
Algoritm:

```
pos = 0
så länge pos < antalet element
         och ordet på plats pos är före word alfabetiskt {
  öka pos med 1
}
sätt in word på plats pos
```



Java: insättning i sorterad följd

```
int pos = 0;
while (pos < words.size()
       && words.get(pos).compareTo(word) < 0) {
  pos++;
}
words.add(pos, word);
```

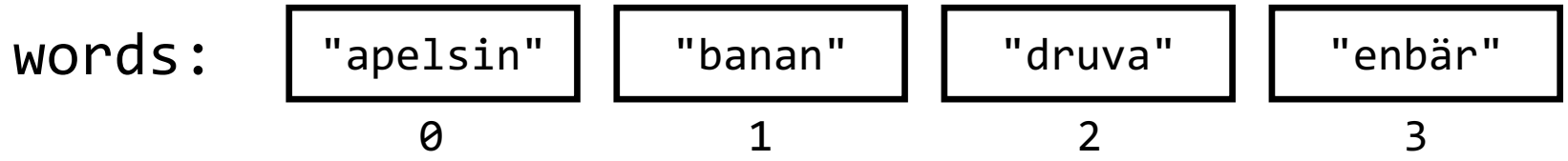


Java: insättning i sorterad följd

```
int pos = 0;
while (pos < words.size()
       && words.get(pos).compareTo(word) < 0) {
  pos++;
}
words.add(pos, word);
```

Hur elementen sorteras.

I labb 10 ska kort (Card-objekt) sorteras efter valör



Övning - sorterad lista av punkter

Specifikation av Point:

```
Point(int x, int y);
int getX();
int getY();
void move(int dx, int dy);
```

```
// Sorterad lista av punkter efter x-värde
ArrayList<Point> points = new ArrayList<Point>();
points.add(new Point(10, 50));
points.add(new Point(45, 100));
points.add(new Point(70, 20));

Point newPoint = new Point(50, 70);
int pos = 0;
while (pos < points.size()
       && punkten på plats pos har mindre x-värde än newPoint) {
  pos++;
}
points.add(pos, newPoint);
```

Övning: skriv färdigt koden så att den nya punkten sätts in på rätt plats i listan. Punkterna är sorterade efter x-värde.

Lösning - sorterad lista av punkter

Specifikation av Point:

```
Point(int x, int y);
int getX();
int getY();
void move(int dx, int dy);
```

```
// Sorterad lista av punkter efter x-värde
ArrayList<Point> points = new ArrayList<Point>();
points.add(new Point(10, 50));
points.add(new Point(45, 100));
points.add(new Point(70, 20));

Point newPoint = new Point(50, 70);
int pos = 0;
while (pos < points.size()
       && points.get(pos).getX() < newPoint.getX()) {
  pos++;
}
points.add(pos, newPoint);
```

Övning: skriv färdigt koden så att den nya punkten sätts in på rätt plats i listan. Punkterna är sorterade efter x-värde.

Logiska uttryck

För ett logiskt uttryck, där **a** och **b** är två godtyckliga deluttryck, som:

```
a && b
```

Då kommer Java enbart att evaluera **b** om **a** evalueras till **`true`**

*(Det är onödigt att evaluera **b** om vi redan vet att hela uttrycket blir falskt)*

Logiska uttryck

För ett logiskt uttryck, där **a** och **b** är två godtyckliga deluttryck, som:

a || b

Då kommer Java enbart att evaluera **b** om **a** evalueras till **`false`**

`ArrayList` har några metoder för sökning:

```
/** Söker upp ett element som matchar obj. Returnerar true om sådant
  * element finns, annars false. */
boolean contains(Object obj);

/** Söker upp ett element som matchar obj. Returnerar index för
  * första förekomsten av elementet, -1 om elementet inte finns. */
int indexOf(Object obj);

/** Tar bort första förekomsten av objektet obj, om det finns.
  * Returnerar true om ett element togs bort. */
boolean remove(Object obj);
```

OBS! Ovanstående metoder kräver att elementklassen implementerar metoden `equals(Object)`.

Klassen `String` och wrapper-klasserna (`Integer`, `Double`, …) gör detta.

Metoden `contains`

Metoden `contains` fungerar på en lista av strängar,
eftersom klassen `String` implementerar metoden `equals`:

```
ArrayList<String> words = new ArrayList<String>();
words.add("apelsin");
words.add("banan");
words.add("druva");
words.add("enbär");

System.out.println(words.contains("druva"));  // true
```

Jämförelse av referenser

När två referensvariabler jämförs är det referenserna som jämförs, ***inte*** innehållet! Exempel:

```
Point p1 = new Point(10, 30);
Point p2 = new Point(10, 30);
if (p1 == p2) {   // referenserna jämförs => false
  ...
}
```

Jämförelse av innehåll

Ska ***innehållet*** jämföras behöver man antingen anropa `getX()` och `getY()` på båda punkterna:

```
Point p1 = new Point(10, 30);
Point p2 = new Point(10, 30);
if (p1.getX() == p2.getX() && p1.getY() == p2.getY()) {
   ...
}
```

Eller så kan klassen `Point` implementera metoden `equals(Object)`. Då kan man skriva

```
if (p1.equals(p2)) {
   ...
}
```

Överkurs: Implementera metoden `equals` i `Point`

```
public class Point {
  private int x, y;

  ...

  public boolean equals(Object other) {  // Parametern måste ha typen Object
    // Kontrollera om other är en punkt
    if (other instanceof Point) {
      Point otherP = (Point) other; // typomvandla så vi kan hämta x,y
      return x == otherP.x && y == otherP.y;
    } else {
      return false;
    }
  }
}
```

Klassen Object betyder "alla sorters objekt"

Operatorn `instanceof` ger `true` om `other` refererar till ett objekt av typen `Point`

Tillbaka till klassen `Polygon`

```
public class Polygon {
  private ArrayList<Point> vertices;

  /** Skapar en polygon med 0 hörnpunkter. */
  public Polygon() {
    vertices = new ArrayList<Point>();
  }

  /** Lägger till en ny punkt med
      koordinaterna x, y. */
  public void addVertex(int x, int y) {
    vertices.add(new Point(x, y));
  }

  /** Returnerar true om någon av
      hörnpunkterna har koordinaterna x,y. */
  public boolean hasVertex(int x, int y) {
    ...

  }
}
```

```java
public class Polygon {
  private ArrayList<Point> vertices;
  ...
  public boolean hasVertex(int x, int y) {
    for (Point p: vertices) {
      if (p.getX() == x && p.getY() == y) {
        return true;
      }
    }
    return false;
  }
}
```

Linjärsökning

(iterera över listan och sök efter om en punkt finns eller inte)

Detta behöver man kunna.

```java
public class Polygon {
  private ArrayList<Point> vertices;
  ...
  public boolean hasVertex(int x, int y) {
    return vertices.contains(new Point(x, y));
  }
}
```

Nu när klassen `Point` implementerar metoden `equals` så kan vi anropa metoden `contains` på listan, som söker åt oss!

Detta är överkurs.

Checklista

- Förklara begreppet generisk klass
- Använda klassen `ArrayList`
  - Deklarera och skapa en lista
  - Sätta in element
  - Ta bort element
  - Gå igenom en lista och behandla alla element
  - Söka efter ett element i listan (med en `for`-sats)
  - Sätta in ett element i en sorterad följd