ETH zürich

Assignment 7

Felix Friedrich, Lars Widmer

TA lecture, Informatics II D-BAUG

April 2, 2014

“Präsenzstunden” Today

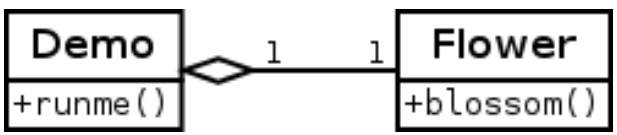
In the “other” room

- **HIL C 29**
- **15:00 - 18:00**
- **Timon Gehr** *(arriving 15:45)*

Outline

Aggregation in OOP

By now we used **aggregation** quite a lot. The class `Demo` **uses** the **abilities** of class `Flower` **by aggregation**. Therefore `Demo` contains *(has a)* an object of class `Flower`.




```
1 public class Demo
2 {    private Flower flower;
3      public runme()
4      {     flower.blossom();
5      }
6 }
```

Inheritance in OOP

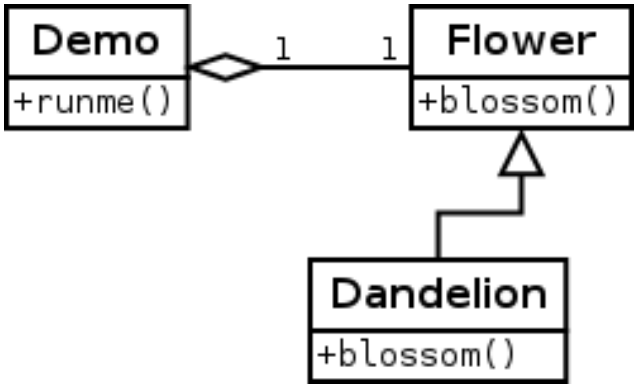
Comparison

Aggregation and **inheritance** are both ways to **reuse** an existing class in a new class. **Code reuse** in general allows to create new classes based on existing ones.

Inheritance

With **inheritance** we get the **is-a-relation** *(generalization)*. In addition to the aspect of **code reuse**, inheritance means **similarity** between the classes. Say: `Dandelion` **is a** `Flower`, thus class `Dandelion` **inherits** from class `Flower`.

`Dandelion` is a `Flower`




```
1 public class Dandelion
2     extends Flower {/* ... */}
```

Using Dandelion

`Dandelion` automatically inherits the `blossom`-Method from `Flower`. Nevertheless it can override it.

Since `Dandelion` **is a** `Flower`, it doesn't make any difference to use `Dandelion` **instead** of `Flower` in `Demo`.

Outline

Pets Example

Please order the following (german) words into a meaningful class diagram.

- Papagei
- Katze
- Säugetier
- Haustier
- Hund
- Vogel
- Rhodesian Ridgeback

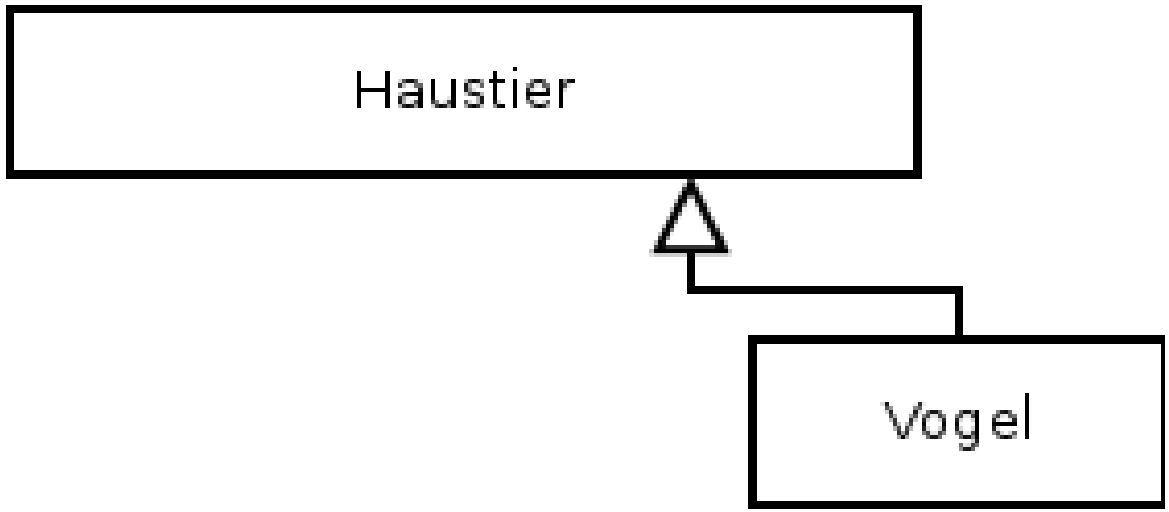
The image shows a Rhodesian Ridgeback. They are well known for lion hunting.

Pets Example

- Papagei
- Katze
- Säugetier
- Haustier
- Hund
- Vogel
- Rhodesian Ridgeback

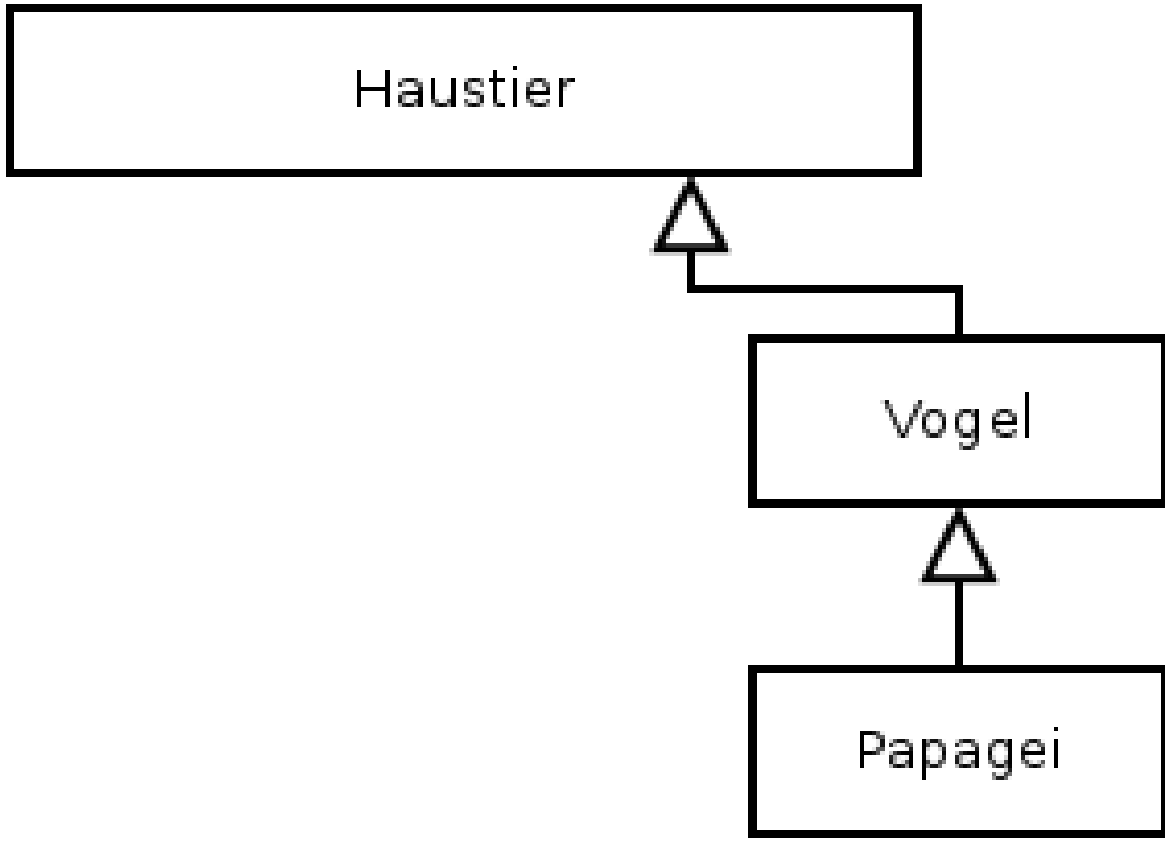
Pets Example

- Papagei
- Katze
- Säugetier
- Haustier
- Hund
- Vogel
- Rhodesian Ridgeback



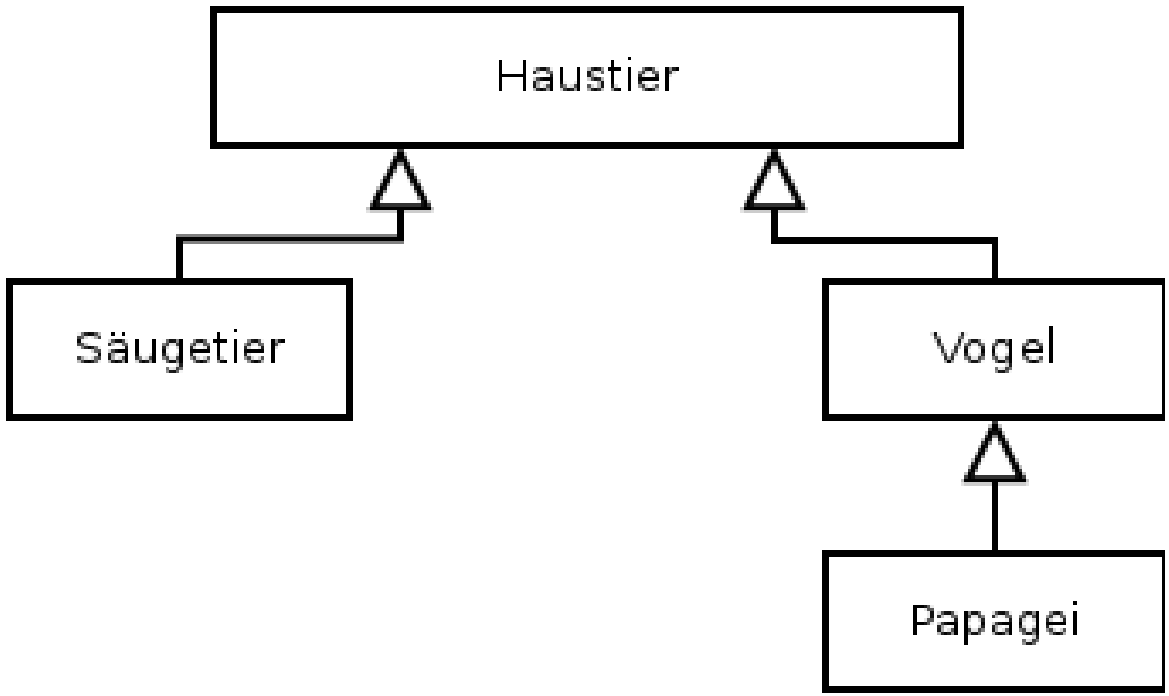
Pets Example

- Papagei
- Katze
- Säugetier
- Haustier
- Hund
- Vogel
- Rhodesian Ridgeback



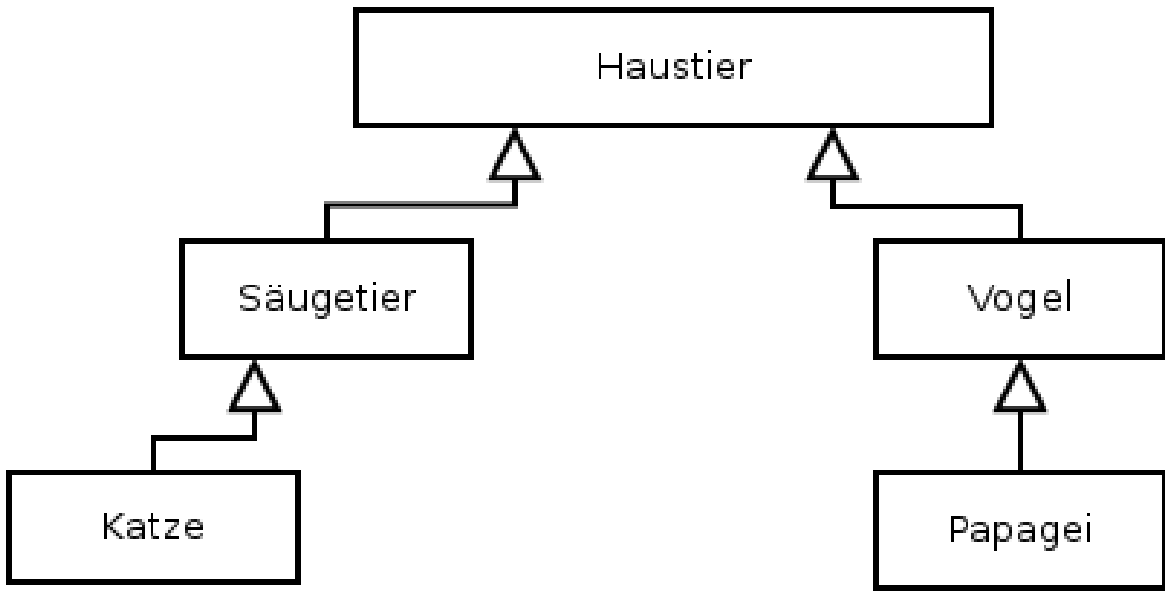
Pets Example

- Papagei
- Katze
- Säugetier
- Haustier
- Hund
- Vogel
- Rhodesian Ridgeback



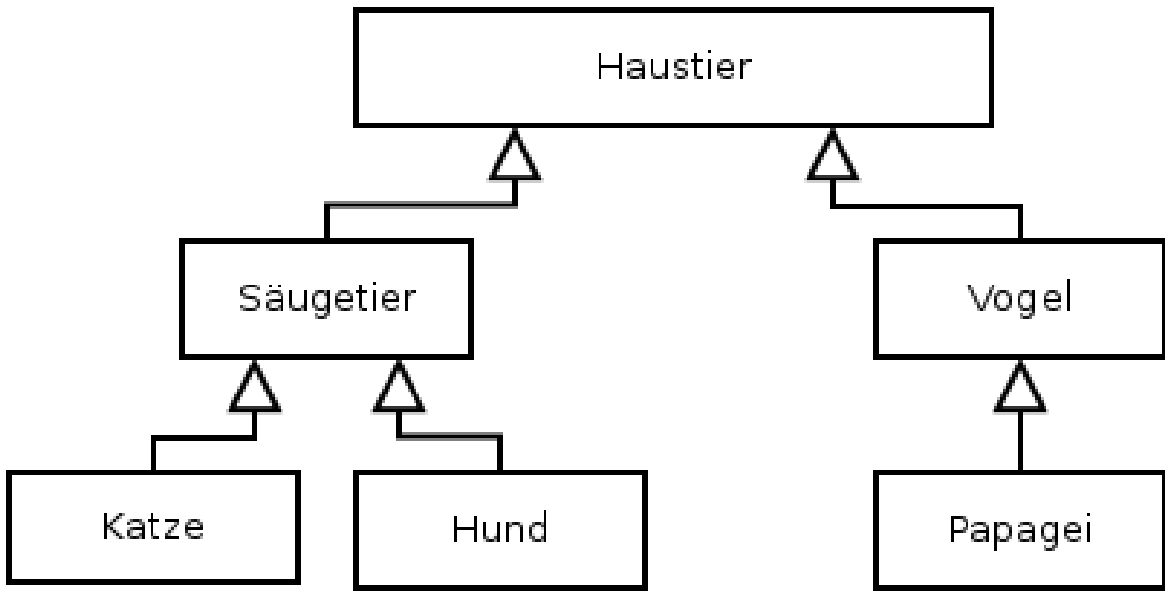
Pets Example

- Papagei
- Katze
- Säugetier
- Haustier
- Hund
- Vogel
- Rhodesian Ridgeback



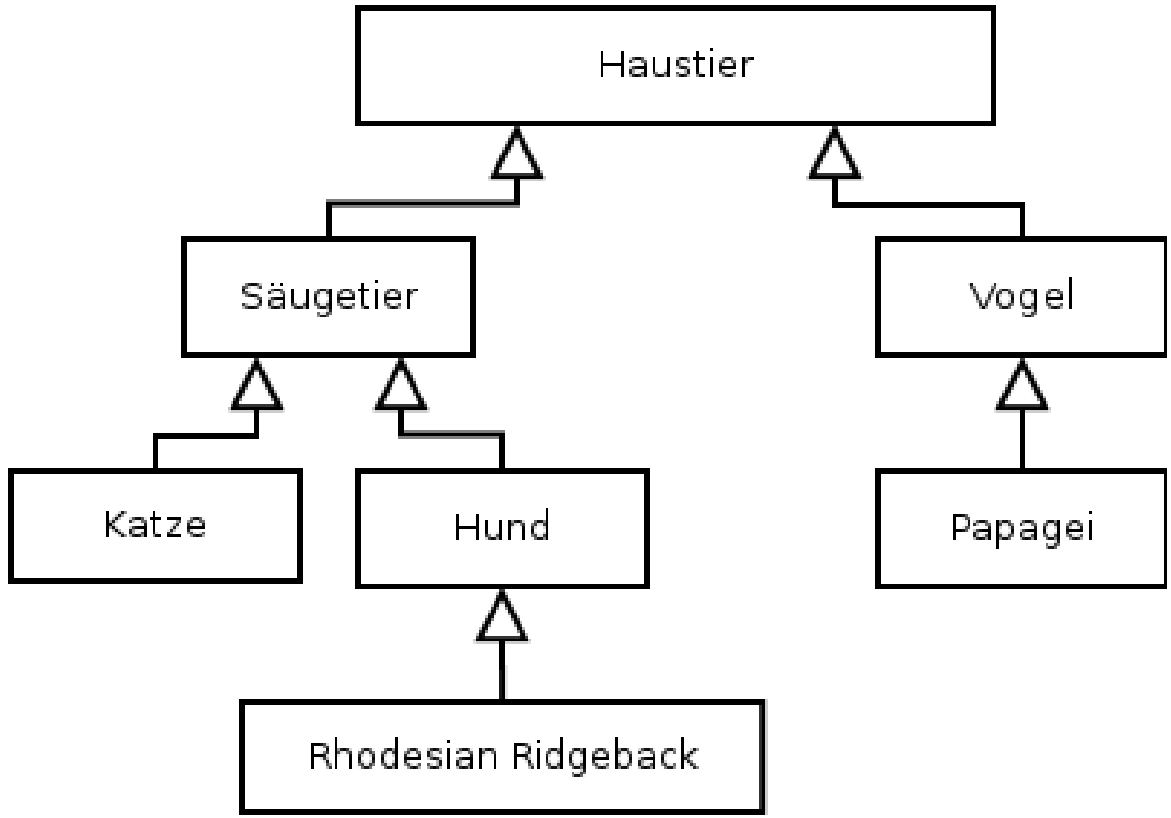
Pets Example

- Papagei
- Katze
- Säugetier
- Haustier
- Hund
- Vogel
- Rhodesian Ridgeback



Pets Example

- Papagei
- Katze
- Säugetier
- Haustier
- Hund
- Vogel
- Rhodesian Ridgeback



Outline

Interface

To allow **easy painting**, we wrote the class `EasyGraphics`.

It implements the following `interface`:

`EasyGraphics`

- `public void set(int x, int y, int rgb);`
- `public int get(int x, int y);`
- `public void repaint();`

Interface

`EasyGraphics`

- `public void set(int x, int y, int rgb);`
  Writes a pixel to the image. The image is like a two-dimensional array.

Interface

`EasyGraphics`

- `public void set(int x, int y, int rgb);`
  Writes a pixel to the image. The image is like a two-dimensional array.
- `public int get(int x, int y);`
  Reads a pixel from the image.

Interface

`EasyGraphics`

- `public void set(int x, int y, int rgb);`
  Writes a pixel to the image. The image is like a two-dimensional array.
- `public int get(int x, int y);`
  Reads a pixel from the image.
- `public void repaint();`
  Draws the image to the screen. We could draw the image everytime we set a pixel, but this would slow down the process.

EasyGraphics Usage for Painting

```
1 class Flower {
2     void blossom(EasyGraphics graph) {
3         graph.set(10, 10, Color.RED.getRGB());
4         graph.repaint();
5     }
6 }
```

EasyGraphics.set

The set-Method here paints a **single red pixel** at the coordinate (10/10). The RGB-value *(Red-Green-Blue)* of a color is just a single `int`-number. But the easiest way to define colors is to use the predefined class `Color` as in the example. Don't forget the according `import java.awt.Color;`

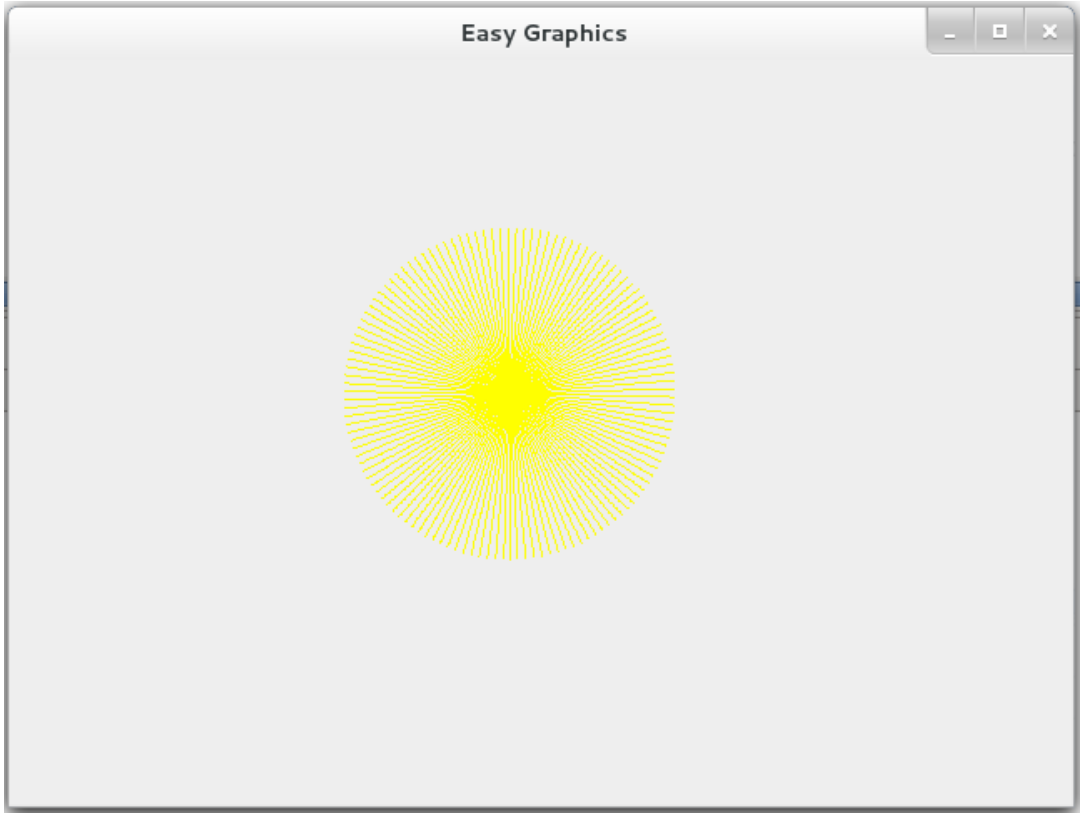
The `EasyGraphics` Window

By default the image is $640x480$ pixels in size. The single red pixel in the top left corner is quite hard to see. ... So let's create a subclass and override `draw` with something more beautiful.

Who can guess the Result?

```
1  class Dandelion extends Flower {
2      public void blossom(EasyGraphics graph) {
3          for (int i=0; i<628; i+=5) {
4              double s = Math.sin((double)i/100);
5              double c = Math.cos((double)i/100);
6              for (int ii=0; ii<100; ++ii) {
7                  graph.set((int)(300+ii*s),
8                            (int)(200+ii*c),
9                            Color.YELLOW.getRGB());
10             }
11         }
12         graph.repaint();
13     }
14 }
```

The `EasyGraphics` Window



Dandelion

Here's a "Dandelion" for you!

EasyGraphics usage in main

```
1 public class Demo {
2     public static void main(String[] args) {
3         EasyGraphics graph = new EasyGraphics();
4         Flower flower = new Dandelion();
5         flower.blossom(graph);
6     }
7 }
```

EasyGraphics

If you added the class `EasyGraphics` from the course website to your project, you can create an object of it similar to line 3. If there were many subclasses of `Flower` we could use an array of `Flower` and store objects of all different subclasses in it.

Outline

LinkedList Highscore Structure

```
1  public class Highscore {
2      private LinkedList<Player> highscore;
3
4      public Highscore(int s) {
5          highscore = new LinkedList<Player>();
6          for (int i=0; i<s; ++i) {
7              highscore.add(new Player("<empty>",0));
8          }
9      }
10     ...
11 }
```

The original `int`-array was substituted by a `LinkedList`.

Simplified Highscore

```
1 public void showHighscore() {
2     System.out.println("*** HIGHSCORE ***");
3     for (int i=0; i<highscore.size(); ++i) {
4         System.out.println(highscore.get(i));
5     }
6 }
```

showHighscore

Runs through the array in a for loop and prints the scores.

`for`-loop for lists

```
1 public void showHighscore() {
2     System.out.println("*** HIGHSCORE ***");
3     for (int i=0; i<highscore.size(); ++i) {
4         System.out.println(highscore.get(i));
5     }
6 }
```

Can be simplified to: *(... It's lovely, keep it in mind!)*

```
1 public void showHighscore() {
2     System.out.println("*** HIGHSCORE ***");
3     for (Player p : highscore) { // list-for
4         System.out.println(p);
5     }
6 }
```

Simplified Highscore

```
1  public void insert(Player newp) {
2      if (newp.score > highscore.getLast().score) {
3          int pos = 0;
4          for (Player p : highscore) { // list-for!
5              if (newp.score < p.score) {
6                  ++pos;
7              }
8          }
9          highscore.add(pos, newp); // add new entry
10         highscore.removeLast(); // maintain length
11     }
12 }
```

Thanks to the LinkedList we can easily insert at any place!
All we have to do is to **remove** the last element after insert.

Outline

Two different versions of `insert`

```
1 public void insert(Player newp) {
2     if (newp.score > highscore.getLast().score) {
```

```
1 int pos = 0;
2 for (Player p : highscore) { // list-for
3     if (newp.score < p.score) {
4         ++pos;
5     }
6 }
7 highscore.add(pos, newp);
8 highscore.removeLast();
```

```
1 highscore.removeLast();
2 highscore.add(newp);
3 highscore.sort();
```

```
1     }
2 }
```

The version on the right-hand-side looks much simpler.
But what's the **time complexity** of the two versions?

Sorted Insert vs. Insert & Sort

```
1 int pos = 0;
2 for (Player p : highscore) { // list-for!
3     if (newp.score < p.score) {
4         ++pos;
5     }
6 }
7 highscore.add(pos, newp);
8 highscore.removeLast();
```

Time Complexity: **Sorted Insert**

As it can be seen in the code, we only need one loop. On the average we need to go through half of the list.
Thus the runtime complexity is $\frac{n}{2} \Rightarrow O(n)$.

Looking at Sort

```
1 highscore.removeLast();
2 highscore.add(newp);
3 highscore.sort();
```

Listing 1: BubbleSort

```
1  boolean done = false; // not done
2  while (!done)
3  {   done = true; // assume done
4      ListElement le = first.next;
5      while (le.next != null)
6      {   if (le.data < le.next.data)
7          {   int tmp = le.data;
8              le.data = le.next.data;
9              le.next.data = tmp;
10             done = false;
11         }
12         le = le.next; // go to next
13     }
14 }
```

Insert & Sort

Bubble sort (code below) needs two nested loops. Each loop does up to n iterations.
Time Complexity: $\frac{n^2}{2}$
$\Rightarrow O(n^2)$

The difference is **huge**: for $n = 1000$, **sorted insert** is $1000x$ faster.

Outline

Class `ListElement<T>` Structure

```
1  public class ListElement<T>
2  {    public T data;
3       public ListElement<T> next;
4
5       public ListElement(T dat)
6       {    data = dat;
7            next = null;
8       }
9       public ListElement(T dat, ListElement<T> after)
10      {    data = dat;
11           next = after;
12      }
13 }
```

We kindly offer **two constructors**.

Method `ListElement<T>.toString()`

```
1 public class ListElement<T>
2 {    public String toString()
3      {    return "_"+data+"_";
4      }
5 }
```

`Object` is **the** Java-SUPER-class. **Every** class automatically inherits from `Object`. `Object` defines a method "`public String toString()`". Thus every class in Java offers this method. The method `toString` is automatically called, whenever an object is added to a String or printed. Thanks to this mechanism you can **print everything** in Java. **Overwriting** `toString` allows you to decide **which information** should be converted.

Class `LinkedList` Structure

```
1 public class LinkedList<T extends Comparable<T> >
2 {    private ListElement<T> first;
3      private ListElement<T> last;
4
5      public LinkedList()
6      {    first = new ListElement<T>(null, null);
7           last = first; // empty list
8      }
9 }
```

With `LinkedList<T extends Comparable<T> >` we enforce that the class T must implement the interface Comparable. This interface contains a method `compareTo(T)`. We wouldn't be able to sort without a compare-method.

Method `LinkedList.insert()`

```
1  public void insert(int pos, T dat)
2  {    ListElement<T> le = new ListElement<T>(dat);
3       ListElement<T> bef = locate(pos-1);
4       ListElement<T> aft = bef.next;
5       bef.next = le;
6       le.next  = aft;
7       if (aft == null)
8       {    last = le;
9       }
10 }
```

In the generic version we use `ListElement<T>` instead of `ListElement`. Other than that, there's not much to change.

Usage of generic `LinkedList<T>`

```
1  public class Test {
2      public static void main(String[] args) {
3          LinkedList<Integer> sll
4                  = new LinkedList<Integer>();
5          /* ... */
6          sll.insert(3,99);
7          sll.sort();
8          System.out.println(sll);
9          /* ... */
10     }
11 }
```

When instantiating the `LinkedList` we give the desired class in the arrow-brackets “<...>”. If we didn’t enforce `Comparable` we could use **every** class in the world.

Outline

Class `GrowingArray<T>` Structure

```
1 public class GrowingArray<T>
2 {
3     private Object[] data;
4
5     public GrowingArray()
6     {   data = new Object[10];
7     }
8
9 }
```

Since Java doesn't allow arrays of generic types, we have to use the superclass `Object`.

Method `GrowingArray<T>.set`

```
1 public void set(int ind, T val)
2 {    if (ind >= data.length)
3      {   resize(ind); // resize if too small
4      }
5      data[ind] = val;
6 }
```

Because of the **is-a**-relation between `Object` and every other class. Every class is also of type `Object`. So without conversion we can store every object in an array of type `Object`.

Method GrowingArray<T>.get

```
1 public T get(int ind)
2 {    if (ind >= data.length)
3      {   resize(ind); // resize if too small
4      }
5      return (T)data[ind];
6 }
```

To regain the original type, we use an unsafe type cast "`(T)`" on line 5. Eclipse therefore shows a warning.

Questions from your side?

Please

- Feedback?
- Additions?
- Remarks?
- Wishes?
- ...

All the Best!


use Geant4Runtime v2r51p1 IExternal
use ebfExt v2r301p3 IExternal
use xmlGeoDbs v1r15
use RootPolicy v2r1p2
use astro v0r6p1
use geometry v3r1
use facilities v2r7p2
use xml v4r3p1
use xmlUtil v2r10p2
use idents v2r10p1
use detModel v2r14p1
use Event v9r11
use GlastSvc v9r10p1
use mcRootData v2r11p5
use digiRootData v5r0p0
use reconRootData v4r3p3
use commonRootData v0r2p2
Linux
If you've ever built a TV set from scratch, you'll love Linux