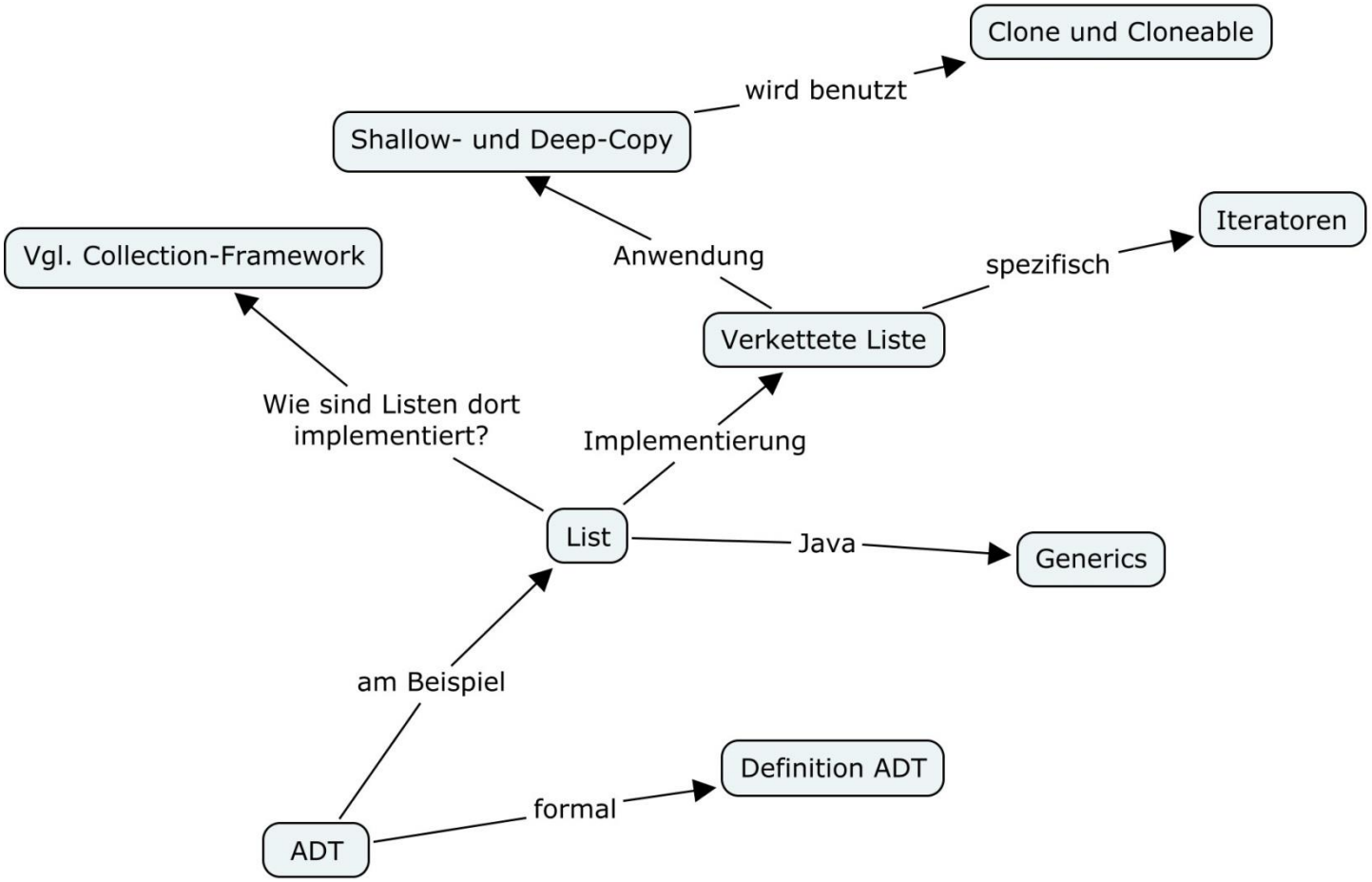
Abstract Datatypes

Abstract Datatypes



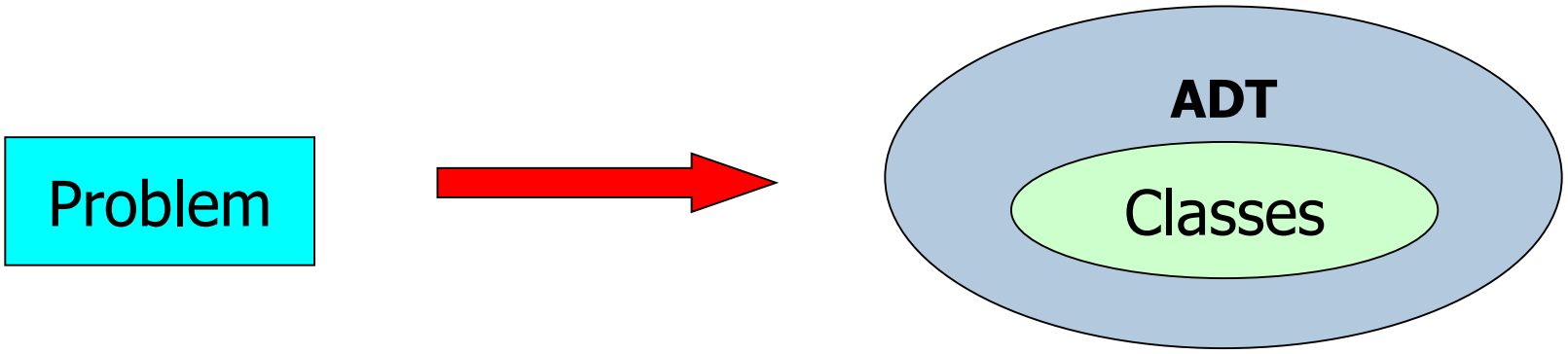
Educational Objective

- How to use Abstract Datatypes
- How to implement Abstract Datatypes
- How collections and iterators really work
- How to write your own ADT
- How to chose appropriate ADTs for a given problem

Abstract Datatype

An abstract datatype is a set of objects together with a set of operations.

Here we create ADTs by using Java classes.



Principles

- implementation independence
- precise specification
- simplicity
- encapsulation
- integrity
- modularity

Example: Complex Numbers

The set $C=R\times R=\{(x,y) \mid x,y\in R\}$ together with the following operations is called the set of complex numbers.

$+$	:	$C\times C$	$\rightarrow$	C
$*$	:	$C\times C$	$\rightarrow$	C
Re,Im	:	C	$\rightarrow$	R

General mathematical structures:
Group, ring, fields, vector spaces,…

Example: List

A list is an ordered set of elements.

For a list L=(A_0,...,A_{n-1}) A_{i+1} is called the successor of A_i and A_i the predecessor of A_{i+1}.

Index i is the position of A_i.

n is called length of the list. A list of length 0 is called empty list.

Lists can be implemented in many different ways. The most simple solution could be by using a simple Array (Disadvantages?).
Java provides Collection-Classes. Example: ArrayList.

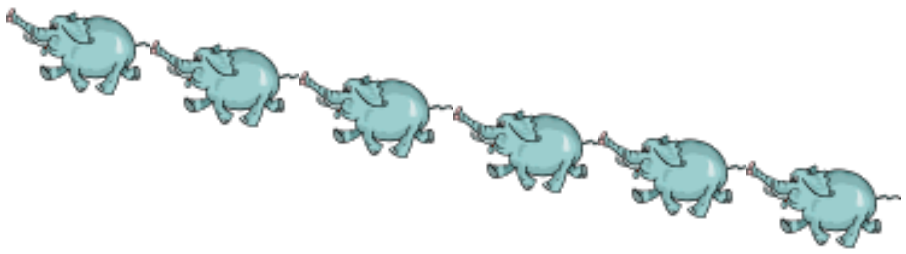
Operations on Lists

Typical list operations:

▪ append	Liste × Object →Liste
▪ insert	Liste × Object →Liste
▪ delete	Liste × Object →Liste
▪ seek	Liste × Object →Object
▪ …	

Compare Java List Interface

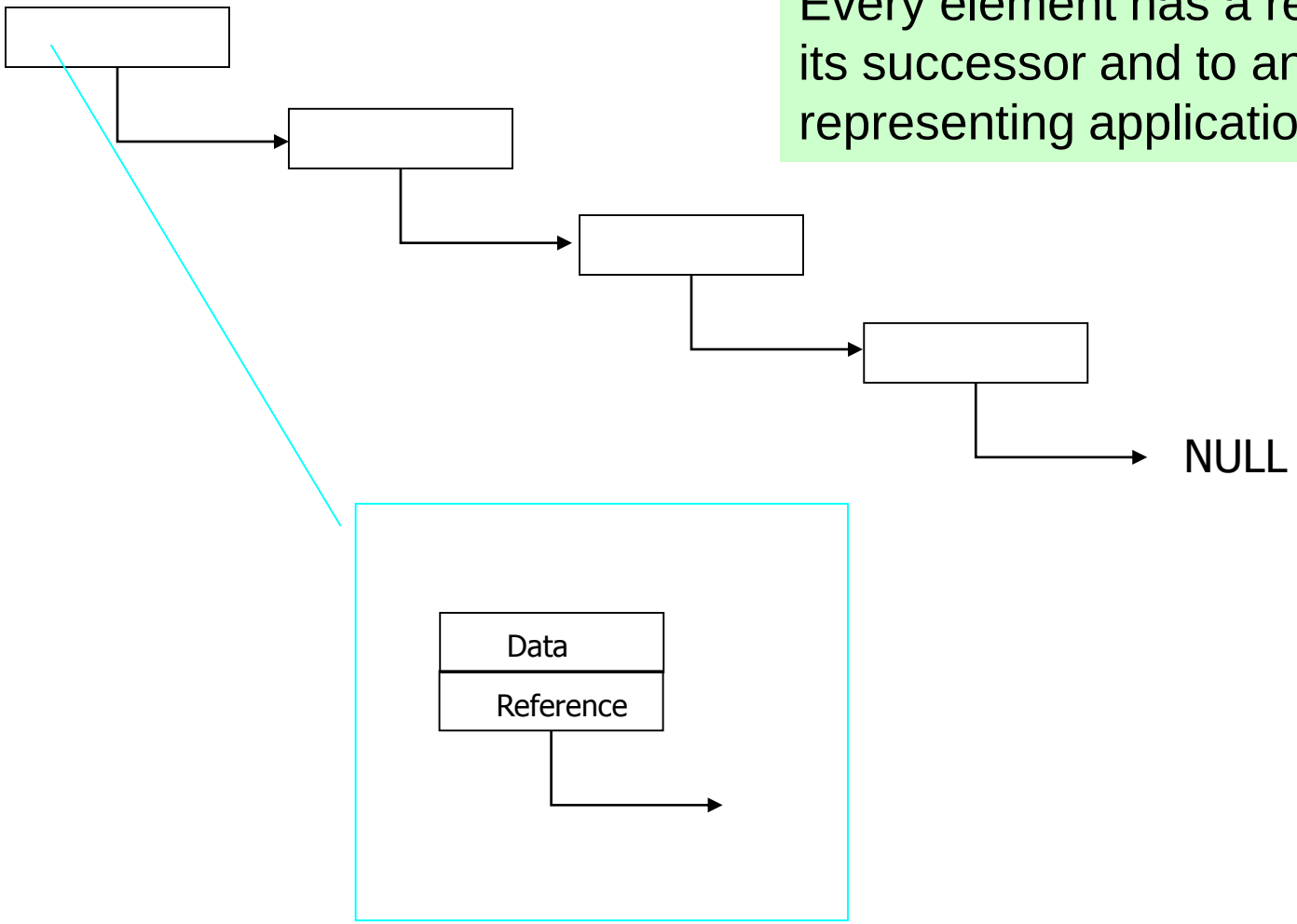
List



Each element has a reference to its next neighbour!

Singly-chained List

Every element has a reference to its successor and to an object representing application data.



Classes

Implementation using two classes:

Listenelement	list entries
VListe	the overall structure

The following code is slightly simplified and omits some usually advisable consistency checks.
We assume proper use.

```
public class Listenelement
{
    private Object data;
    private Listenelement successor;

    public Listenelement(){
    }
    public Object getData() {
        return data;
    }
    public void setData(Object obj) {
        data = obj;
    }
    public Listenelement getSuccessor() {
        return successor;
    }
    public void setSuccessor(Listenelement elt) {
        successor = elt;
    }
}
```

construktor

getter

setter

getter for successor

setter for successor

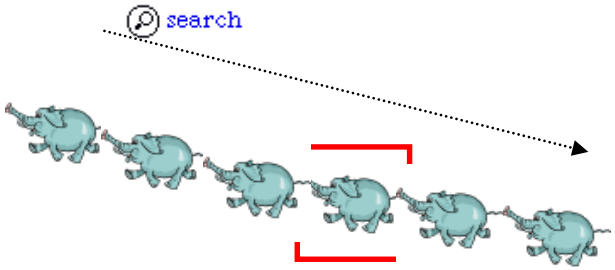
```
public class VListe
{
   private Listenelement start;

   public void setStart(Listenelement elt) {
        start = elt;
   }
   public Listenelement getStart() {
        return start;
   }
  public String toString() {
       String str = " ";
        Listenelement elt=start;
        while (elt!=null) {
            str = str +" "+ elt.toString();
            elt = elt.getSuccessor();
        }
        return str;
   }
  // … weitere Methoden auf den nächsten Seiten
}
```

setter

getter

override

Seek and Find

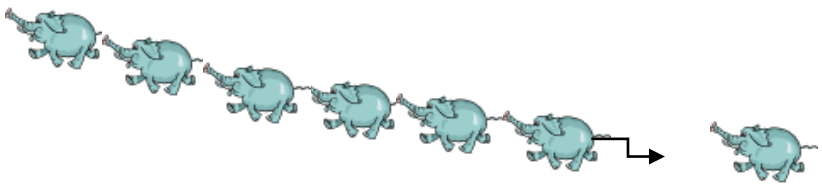



```
public Listenelement find(Listenelement find) {
     Listenelement elt = start;
     while (elt! = null) {
          if (elt == find)
               return elt;
          elt = elt.getSuccessor();
     }
     return null;
}
```

Complexity?

Is == OK?

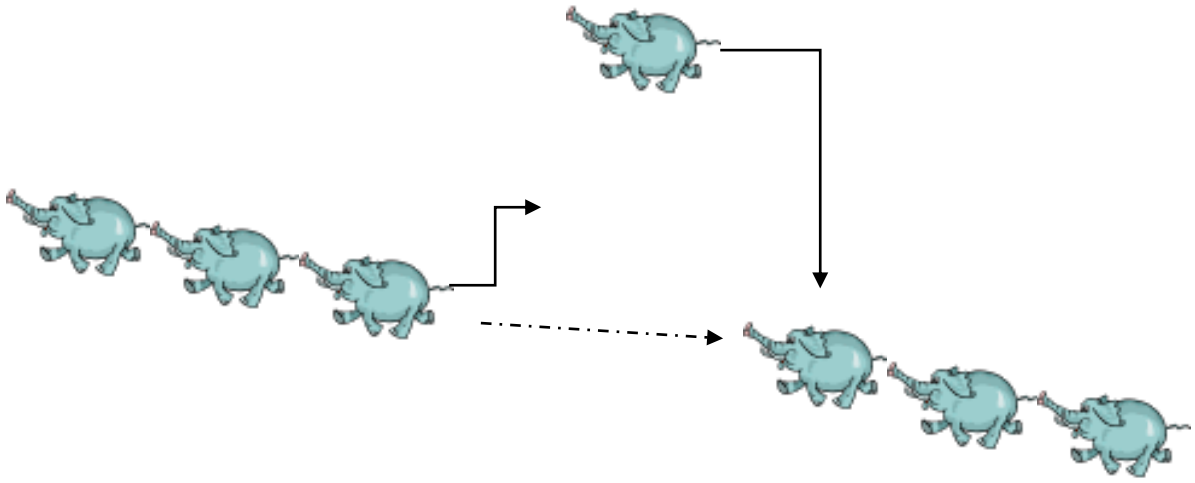
Append



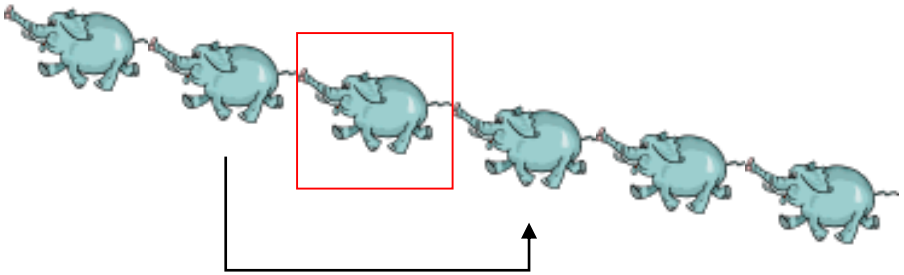
```
public void append(Listenelement neuelt) {
     if (start == null) {
         setStart(neuelt);
     } else {
         Listenelement elt = start;
         while (elt.getSuccessor()!= null)
            elt = elt.getSuccessor();
         elt.setSuccessor(neuelt);
     }
}
```

Other Operations

insert
(before or after
given element)

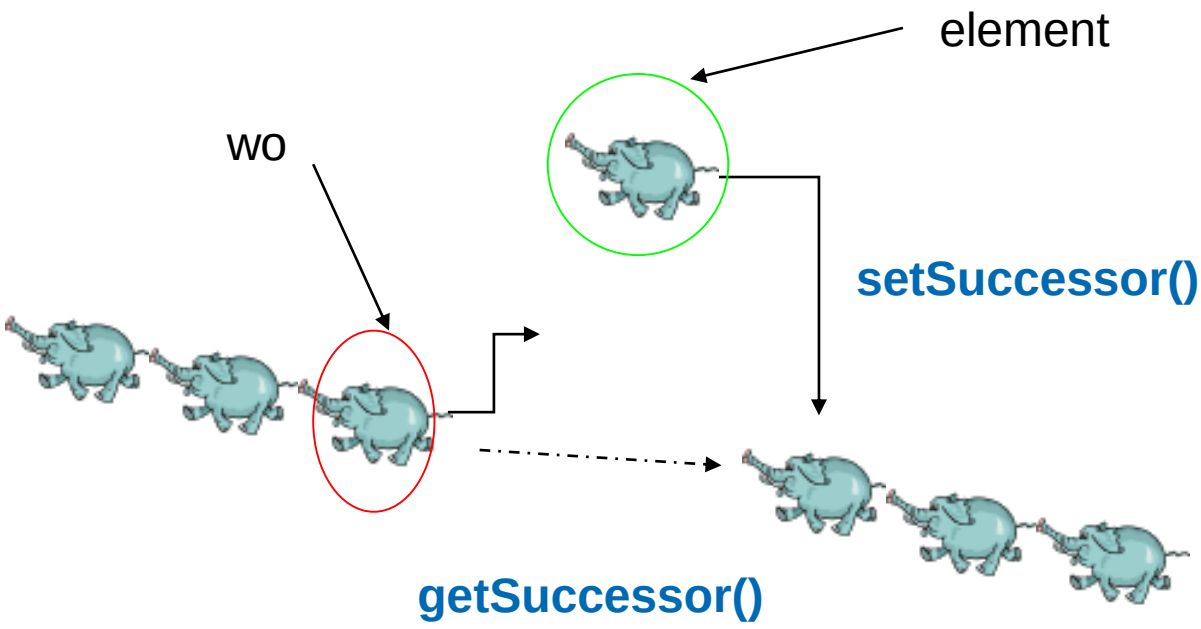


delete

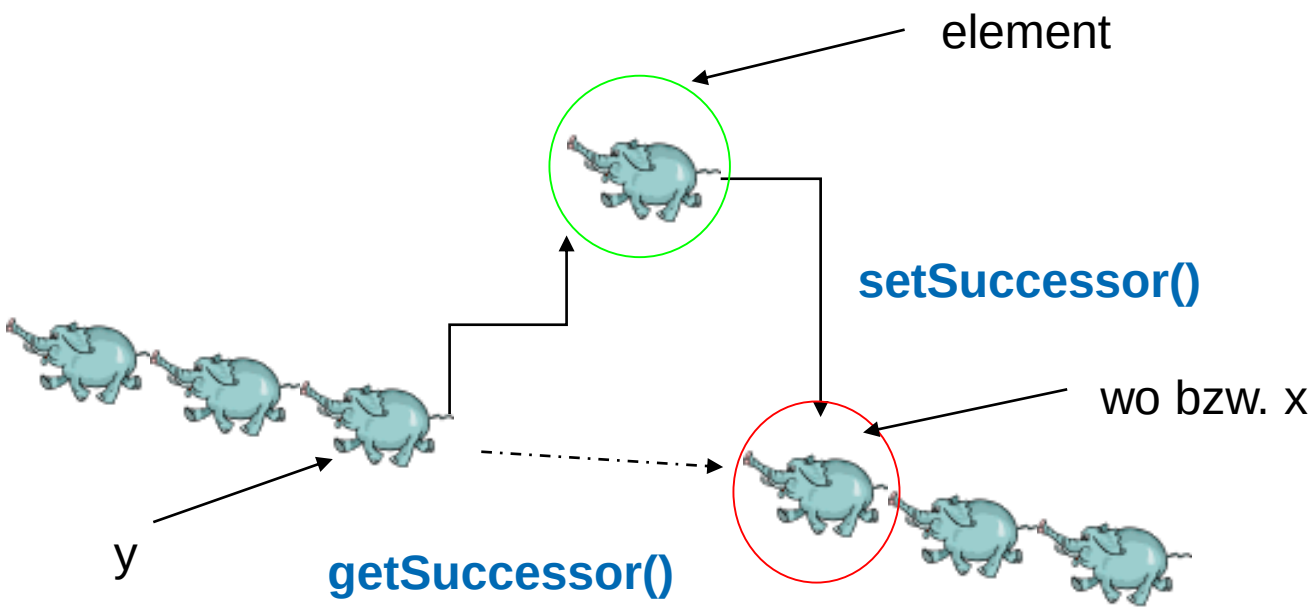


Insert after *wo*

```
public void insertAfter(Listenelement wo, Listenelement elt){
   elt.setSuccessor(wo.getSuccessor());
   wo.setSuccessor(elt);
}
```



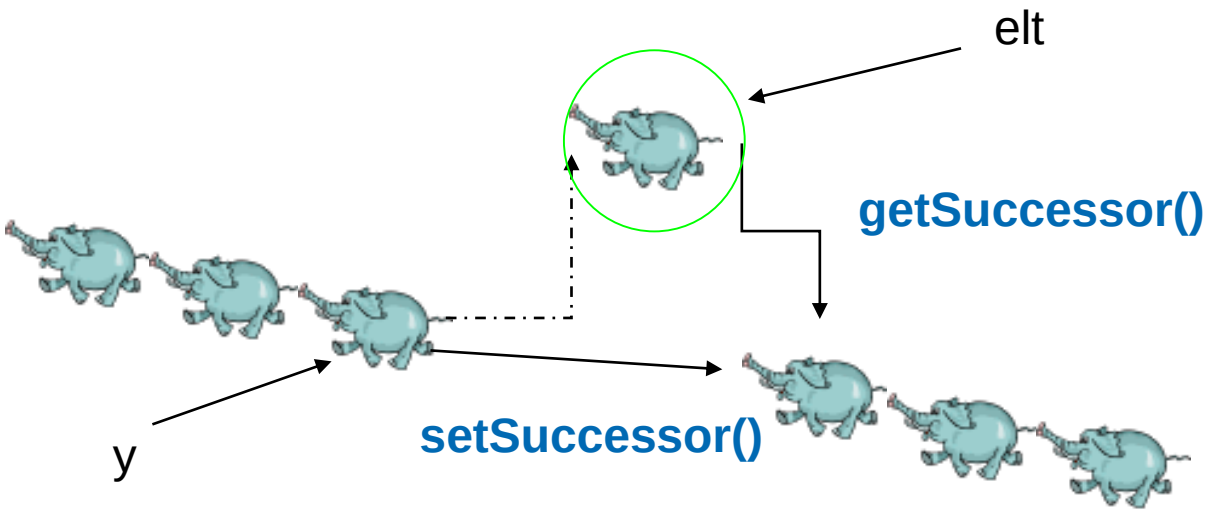
Insert Before *wo*



Insert Before *wo*

```
public void insertBefore(Listenelement wo, Listenelement elt){
  if (this.start==wo) {
     elt.setSuccessor(start);
     setStart(elt);
  } else {
     Listenelement x = start;
     Listenelement pred_wo;
     do {
        pred_wo = x;
        x = x.getSuccessor();
     } while ((x != null)&&(x != wo));
     if (x != null)
        insertAfter(pred_wo,elt);
 }
}
```

Delete



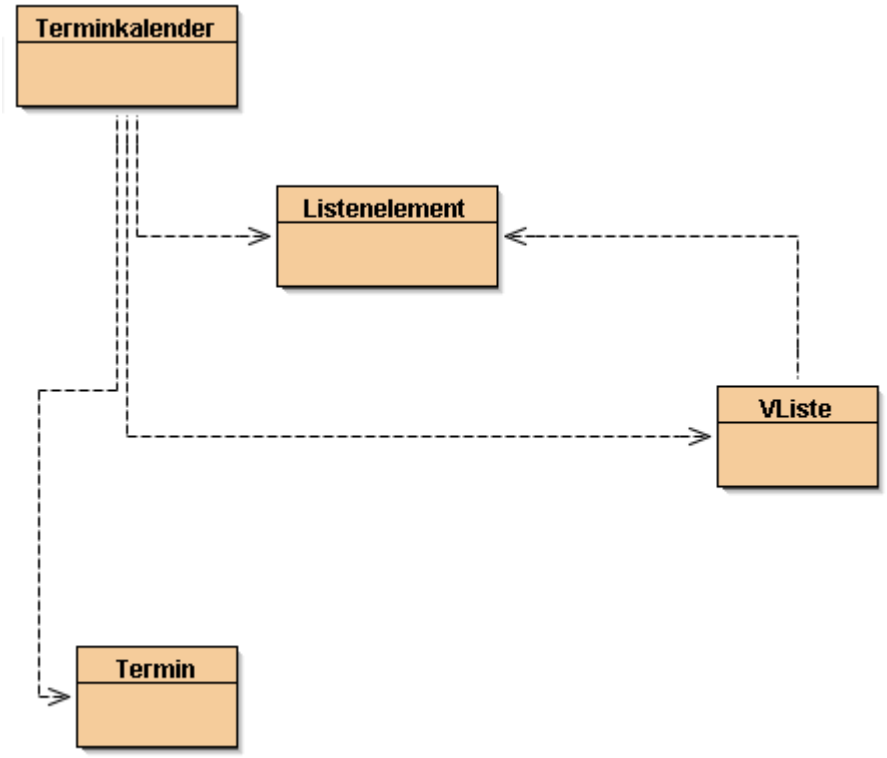
Note: *elt* still references its successor in the list!

Delete

```
public void delete(Listenelement elt) {
   if (start == null) return;
   if (this.start == elt)
       start = elt.getSuccessor();
   else {
       Listenelement x = start;
       Listenelement pred_elt;
       do {
          pred_elt = x;
          x = x.getSuccessor();
        } while ((x != null)&&(x != elt));
       if (x != null)
         pred_elt.setSuccessor(elt.getSuccessor());
}
```

Example

An apointment calendar contains appointments. Appointments are stored in a *VListe*.



Iterator

Iterators allow sequential access to elements of a container.
Iterators have to implement the Iterator-Interface, i.e. they have to implement the following methods:

Method Summary	
boolean	**hasNext**() Returns `true` if the iteration has more elements.
Object	**next**() Returns the next element in the iteration.
void	**remove**() Removes from the underlying collection the last element returned by the iterator (optional operation).

Class VListeIterator

```
import java.util.Iterator;

public class VListeIterator implements Iterator
{
    private VListe dieListe;
    private Listenelement current;
    private Listenelement lastCurrent;

    public VListeIterator(VListe dieListe)
    {
       this.dieListe = dieListe;
       current = dieListe.getStart();
    }

   … Methoden nächste Seite

}
```

the list

next current and current element

construktor

```
public boolean hasNext() {
    return (current != null);
}

public Object next() {
    lastCurrent = current;
    current = current.getSuccessor();
    return lastCurrent;
}

public void remove() {
    dieListe.delete(lastCurrent);
}
```

remove

`public void remove()`

Removes from the underlying collection the last element returned by the iterator (optional operation). This method can be called only once per call to `next`. The behavior of an iterator is unspecified if the underlying collection is modified while the iteration is in progress in any way other than by calling this method.

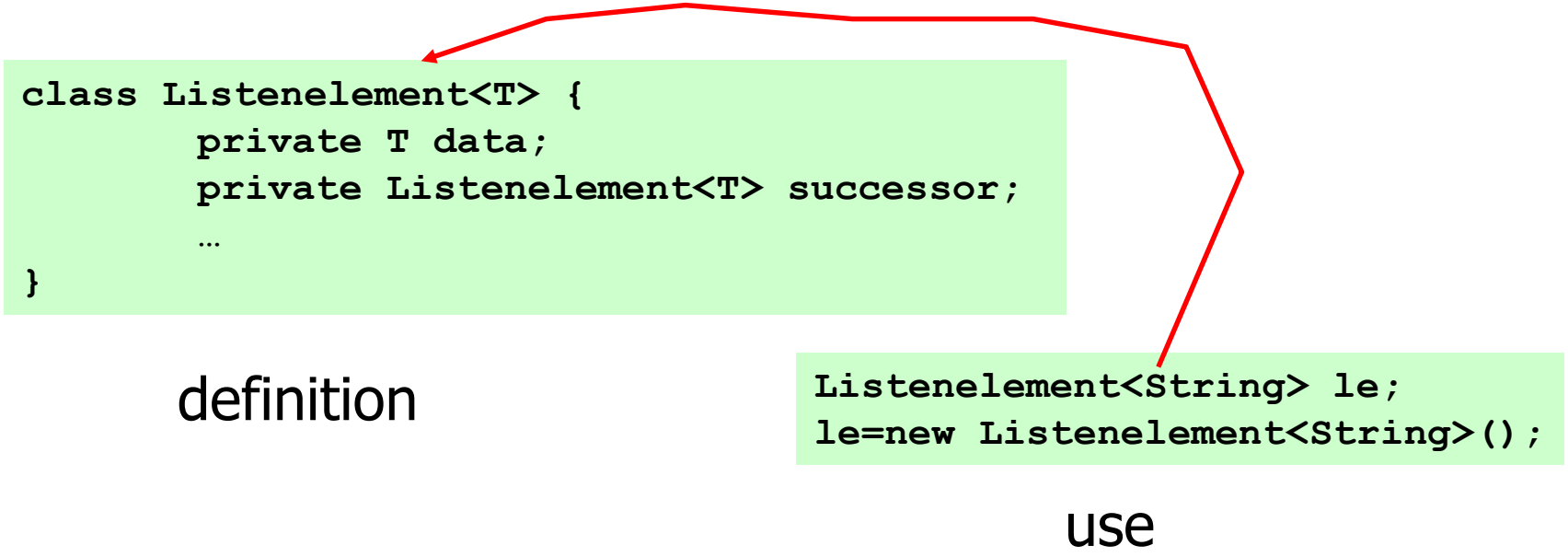
Improvements

Implementations so far are not type safe.

We have to use generics….

Generics

Generische Classes are available since Java version 1.5.
Generics use type variables that have to be put in <>-brackets.
„Real" types are substitutes for type variables in application code.



```
public class Listenelement<T>
{
    private T data;
    private Listenelement<T> successor;

    public Listenelement (){
    }
    public T getData() {
        return data;
    }
    public void setData(T obj) {
        data = obj;
    }
    public Listenelement<T> getSuccessor() {
        return successor;
    }
    public void setSuccessor(Listenelement<T> elt) {
        successor = elt;
    }
}
```

construktor

getter

setter

getter

setter

Exmaple

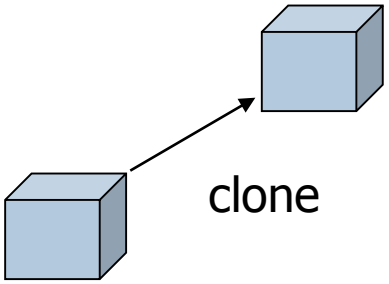
Make proper use of generics in VListe.

What changes are necessary for Listenelement and VListeIterator?

Copying Containers

Object provides method *clone* for copying objects.

An identical copy af an object is created.



Method Summary	
protected Object	**clone**() Creates and returns a copy of this object.

Attention: References are **copied** as well!

Constructor are not invoked!

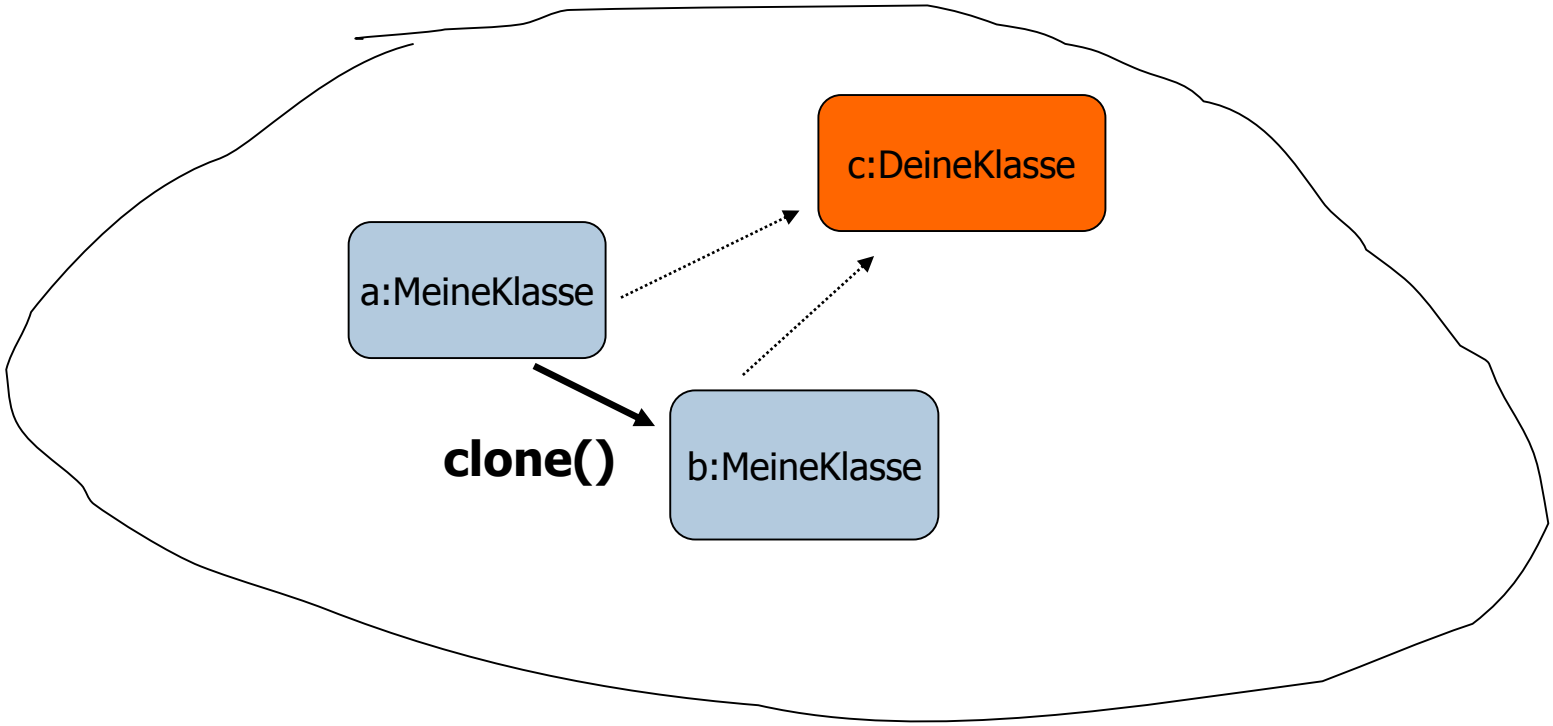
Remember: No entity without identity

Interface Cloneable

clone() ist a protected method of class Object (modifier: protected). Objects may only use clone(), if the class implements the Cloneable-Interface. No further code is required.
Call of clone() may throw CloneNotSuppoertedException!

```
public class MeineKlasse implements Cloneable {

  private DeineKlasse c;

  public MeineKlasse() {
   c=new DeineKlasse();
  }
  public MeineKlasse kopiere() throws CloneNotSupportedException {
   return (MeineKlasse) this.clone();
  }
}
```

Example

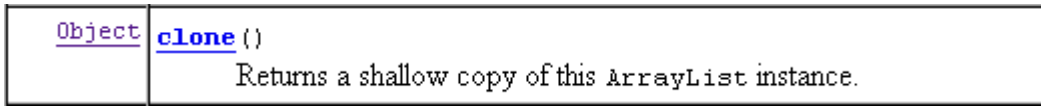


How could a copy of c be created?

Shallow Copy vs. Deep Copy

While copying containers and objects with references we have to distinguish between shallow copy (references are copied) and deep copy (refrenced objects are also cloned).
Typically we use shallow copy as arrays also do.

If arrays contain promitive data this is not an issue.



Object	**clone**() Returns a shallow copy of this `ArrayList` instance.

Example

How could a clone-methode of a list look like?

How could we implement shallow and how deep copy?

Educational Objective

- How to use Abstract Datatypes
- How to implement Abstract Datatypes
- How collections and iterators really work
- How to write your own ADT
- How to chose appropriate ADTs for a given problem