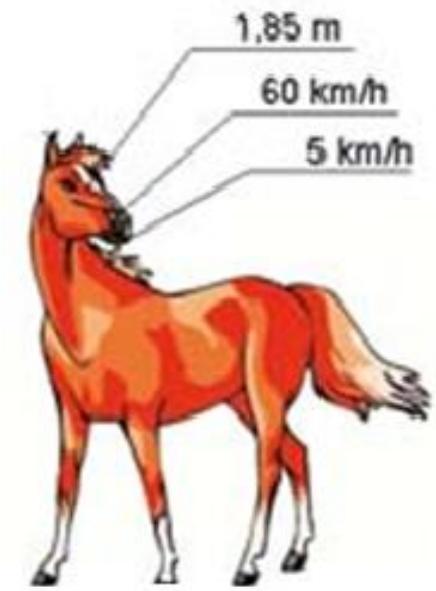
Kapitel 8: Arbeiten mit Objekten: Identität, Listen, Komparatoren, Kopien, Wrapper, Iterator

Inhalt - Überblick

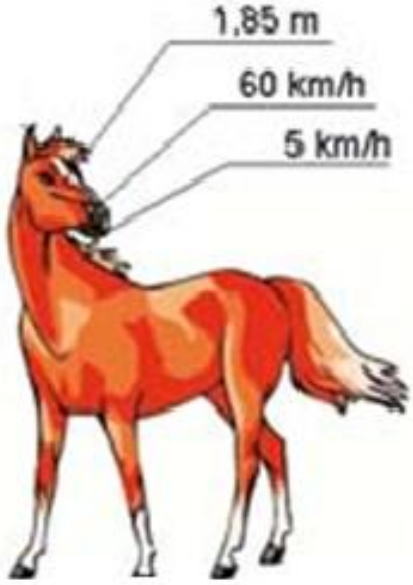
1. **Java Grundlagen: Entwicklungszyklus, Entwicklungsumgebung**
2. Datentypen, Kodierung, Binärzahlen, Variablen, Arrays
3. Ausdrücke, Operatoren, Schleifen und Verzweigungen
4. Blöcke, Sichtbarkeit und Methoden (Teil 1)
5. Grundkonzepte der Objektorientierung
6. Objektorientierung: Sichtbarkeit, Vererbung, Methoden (Teil 2), Konstruktor
7. Packages, lokale Klassen, abstrakte Klassen und Methoden, Interfaces, enum
8. Arbeiten mit Objekten: Identität, Listen, Komparatoren, Kopien, Wrapper, Iterator
9. Fehlerbehandlung: Exceptions und Logging
10. Utilities: Math, Date, Calendar, System, Random
11. Rekursion, Sortieralgorithmen und Collections
12. Nebenläufigkeit: Arbeiten mit Threads
13. Benutzeroberflächen mit Swing
14. Streams: Auf Dateien und auf das Netzwerk zugreifen

Identität / equals

Wann sind zwei Objekte “gleich”?



Pferd 1



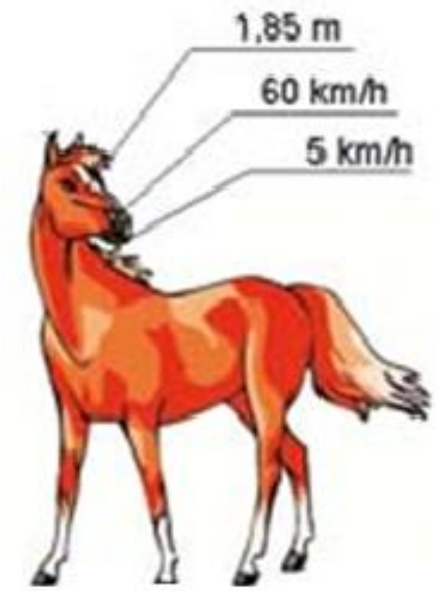
Pferd 2

```
2  public class Horse {
3
4      private long id;
5      private String name;
6      private int size;
7      private int maximumSpeed;
8      private float speed;
```

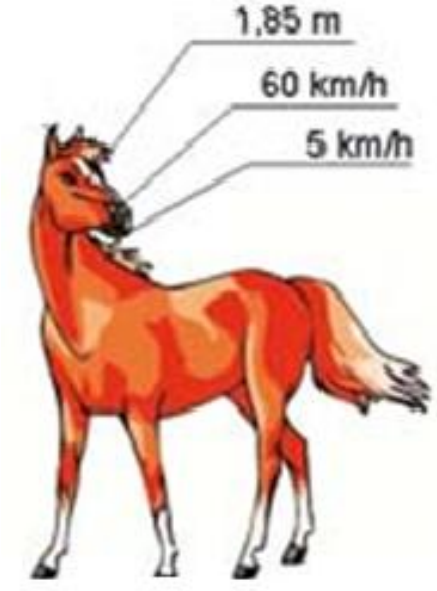
- Es geht um zwei Fragen:
  - Handelt es sich um die gleiche Objektinstanz (==)?
  - Handelt es sich um inhaltlich gleiche Objekte (equals)?

[Horse.java]

Wann sind zwei Objekte “gleich”?



Pferd 1



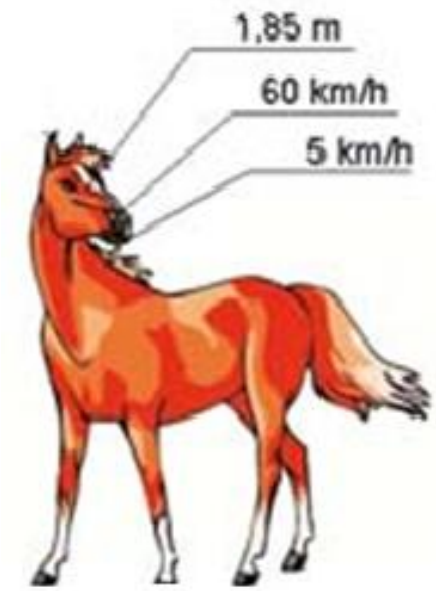
Pferd 2

```
Horse pferd1 = new Horse();
pferd1.setId(11);
pferd1.setName("Max");
pferd1.setSize(185);
pferd1.setSpeed(5);
pferd1.setMaximumSpeed(60);

Horse pferd2 = new Horse();
pferd2.setId(11);
pferd2.setName("Max");
pferd2.setSize(185);
pferd2.setSpeed(5);
pferd2.setMaximumSpeed(60);
```

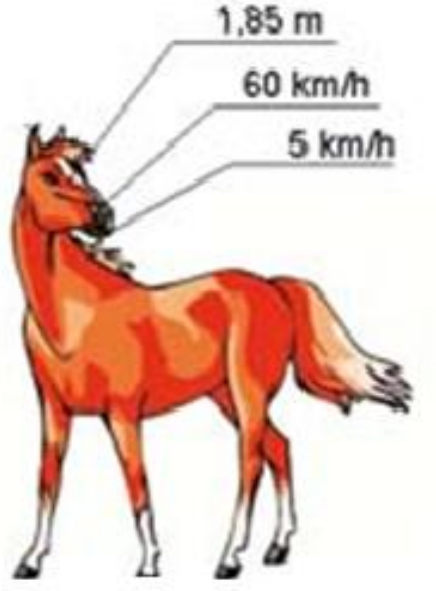
- equals testet, ob zwei Objekte den gleichen Inhalt haben
- equals muss hierfür implementiert werden, d. h. inhaltliche Identität muss genau spezifiziert werden

[Horse.java]

Wann sind zwei Objekte “gleich”?



Pferd 1



Pferd 2

```
Horse pferd1 = new Horse();
pferd1.setId(11);
pferd1.setName("Max");
pferd1.setSize(185);
pferd1.setSpeed(5);
pferd1.setMaximumSpeed(60);

Horse pferd2 = new Horse();
pferd2.setId(11);
pferd2.setName("Max");
pferd2.setSize(185);
pferd2.setSpeed(5);
pferd2.setMaximumSpeed(60);
```

- Test: Aufruf von equals, ohne equals zu überschreiben

```
if (pferd1.equals(pferd2)){
    System.out.println("Die Pferde haben den gleichen Inhalt");
} else {
    System.out.println("Die Pferde sind nicht gleich");
}
```

[Horse.java]

Wann sind zwei Objekte “gleich”?

- Test: Aufruf von equals, ohne equals zu überschreiben
  - In diesem Fall wird die Methode equals von Object genutzt
  - Diese prüft, ob es sich um identische Objektinstanzen handelt (==) und ignoriert den Inhalt

java.lang

Class Object

java.lang.Object

Method Summary

protected Object	**clone**() Creates and returns a copy of this object.
boolean	**equals**(Object obj) Indicates whether some other object is "equal to" this one.
protected void	**finalize**() Called by the garbage collector on an object when garbage

[Horse.java]

Equals überschreiben

- Damit legt man fest, in welchen Fällen zwei Objekte inhaltlich gleich sind
- Beispiel: „Zwei Pferde sind gleich, wenn sie die gleiche ID-Nummer haben“:

```
@Override
public boolean equals(Object obj) {
    if (this == obj)
        return true;
    if (obj == null)
        return false;
    if (getClass() != obj.getClass())
        return false;
    Horse other = (Horse) obj;
    if (id != other.id)
        return false;
    return true;
}
```

[Horse.java]

Equals übeschreiben

Beispiel: „Zwei Pferde sind gleich, wenn alle Attribute gleich sind“:

```
@Override
public boolean equals(Object obj) {
    if (this == obj)
        return true;
    if (obj == null)
        return false;
    if (getClass() != obj.getClass())
        return false;
    Horse other = (Horse) obj;
    if (id != other.id)
        return false;
    if (maximumSpeed != other.maximumSpeed)
        return false;
    if (name == null) {
        if (other.name != null)
            return false;
    } else if (!name.equals(other.name))
        return false;
    if (size != other.size)
        return false;
    if (Float.floatToIntBits(speed) != Float.floatToIntBits(other.speed))
        return false;
    return true;
}
```

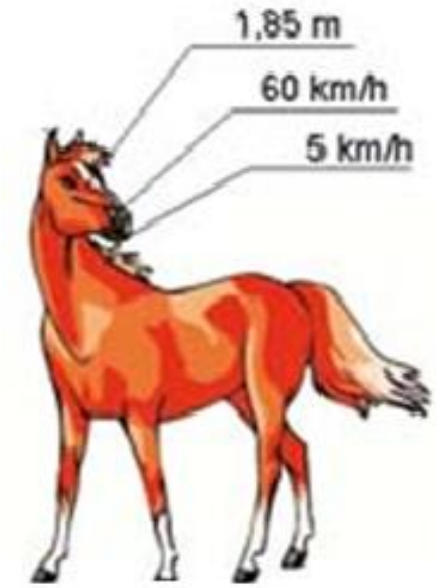
[Horse.java]

- Auch weitere API-Methoden nutzen equals:

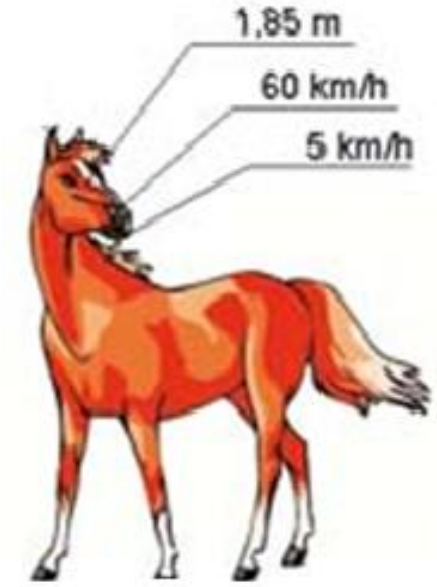
```
// Beispiel: equals und Listen
List<Horse> pferde = new ArrayList<Horse>();
pferde.add(pferd1);
pferde.add(pferd2);

Horse pferd3 = new Horse();
pferd3.setId(11);
pferd3.setName("Max");
pferd3.setSize(185);
pferd3.setSpeed(5);
pferd3.setMaximumSpeed(60);

if(pferde.contains(pferd3)){
    System.out.println("Das Pferd gibt es schon");
}
```

[Horse.java]

Wann sind zwei Objekte “gleich”?



Pferd 1



Pferd 2

```
Horse pferd1 = new Horse();
pferd1.setId(11);
pferd1.setName("Max");
pferd1.setSize(185);
pferd1.setSpeed(5);
pferd1.setMaximumSpeed(60);

Horse pferd2 = new Horse();
pferd2.setId(11);
pferd2.setName("Max");
pferd2.setSize(185);
pferd2.setSpeed(5);
pferd2.setMaximumSpeed(60);
```

- == testet, ob es sind um die gleichen Objektinstanzen handelt
- Vergleiche dazu die Anwendung von == auf Grunddatentypen

[Horse.java]

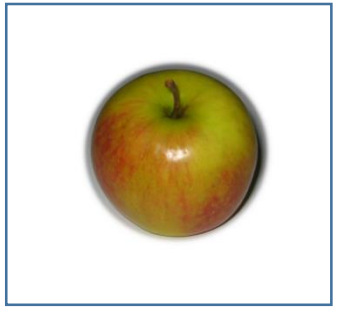
Listen

Alternative zu Arrays: Listen

- Listen sind teil des Collection-Konzepts von Java
- Sie befinden sie im Paket java.util
- Eine List ist eine beliebig große Liste von Elementen beliebigen Typs
- Der Zugriff auf eine Liste kann sequentiell oder über eine Indexzahl erfolgen (vgl. Arrays).
- Das Arbeiten mit Listen ist für den Programmierer meist komfortabler (bspw. kein Umkopieren bei einer Verlängerung der Liste)
- Aber: In bestimmten Situationen leidet die Performance

```
2  public class Apfel {
3
4      private int nummer;
5      private Farbe farbe;
6
7      public int getNummer() {
8          return nummer;
9      }
10     public void setNummer(int nummer) {
11         this.nummer = nummer;
12     }
13     public Farbe getFarbe() {
14         return farbe;
15     }
16     public void setFarbe(Farbe farbe) {
17         this.farbe = farbe;
18     }
19
20 }
```

0

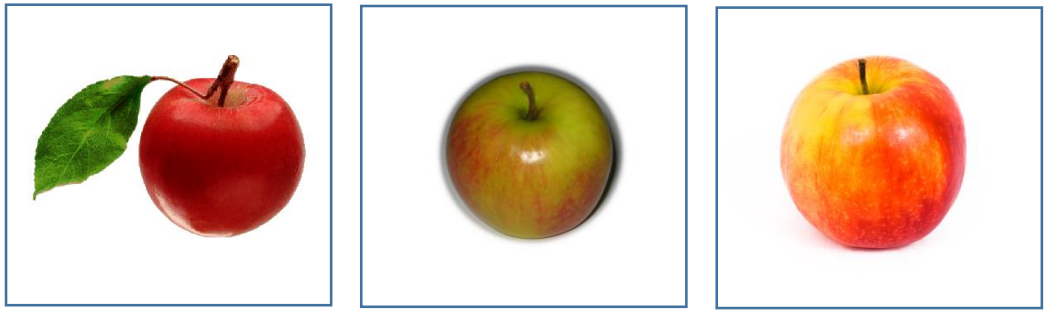


1

2

[Apfel.java]

Beispiel: Liste mit Äpfeln



0 1 2

```
1  import java.util.ArrayList;
2  import java.util.List;
3
4  public class Programm {
5
6      public static void main(String[] args) {
7          List<Apfel> liste = new ArrayList<Apfel>();
8      }
9
10 }
```

[Programm.java]

Wichtige Methoden 1/3

```
// Eine neue Liste anlegen (wir nutzen vorerst immer ArrayList)
List<Apfel> liste = new ArrayList<Apfel>();

// Drei Äpfel anlegen
Apfel apfel1 = new Apfel(Farbe.ROT, 101);
Apfel apfel2 = new Apfel(Farbe.GRUEN, 2099);
Apfel apfel3 = new Apfel(Farbe.GELB, 765);

// Einzelne Objekte in die Liste einfügen
liste.add(apfel1);
liste.add(apfel2);
liste.add(apfel3);

// Die Liste leeren
liste.clear();

// Einzelne Objekte mehrfach in die Liste einfügen
liste.add(apfel1);
liste.add(apfel1);
liste.add(apfel1);
```

[Programm.java]

Wichtige Methoden 2/3

```
// Ein Element "dazwischenschieben"
liste.add(2, apfel2);

// Ein Element über den Index referenzieren
Apfel einApfel = liste.get(1);

// Ein Element an eine Index-Stelle setzen (überschreibt)
liste.set(1, apfel3);

// Wie lange ist die Liste?
int length = liste.size();

// Ist die Liste leer?
boolean empty = liste.isEmpty();

// Ein bestimmtes Element aus der Liste entfernen
// Achtung: hier wird equals genutzt, um das zu entfernende Element zu finden
liste.remove(apfel1);

// Ein Element an einer bestimmten Stelle aus der Liste entfernen
liste.remove(1);
```

[Programm.java]

Wichtige Methoden 3/3

```
// Liste kommt mit einer vorgefertigten toString Methode
System.out.println(liste.toString());

// Eine zweite Liste mit den gleichen Äpfeln
List<Apfel> andereListe = new ArrayList<Apfel>();
andereListe.add(apfel1);
andereListe.add(apfel2);
andereListe.add(apfel3);

// Die zweite Liste in die erste Liste einfügen
liste.addAll(andereListe);

// Eine Liste in ein Array konvertieren
Apfel[] apfelArr = liste.toArray(new Apfel[liste.size()]);
```

[Programm.java]

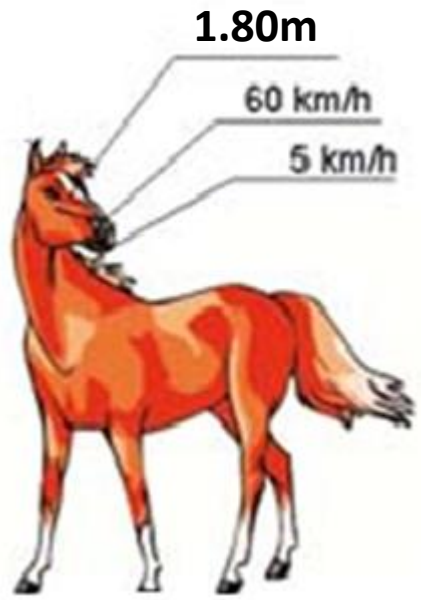
Liste durchlaufen (WH)

```
// Die Liste mit der Kurzform von for durchlaufen
for (Apfel apfel : liste){
    System.out.println(apfel);
}

// Das Array mit der Kurzform von for durchlaufen
for (Apfel apfel : apfelArr){
    System.out.println(apfel);
}
```

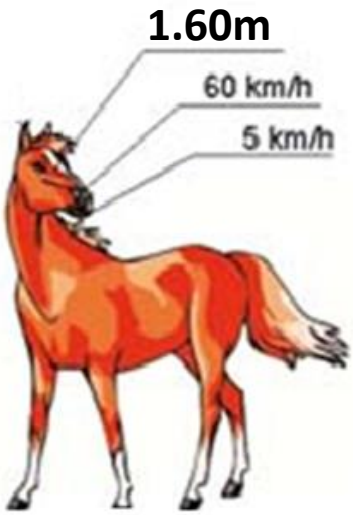
[Programm.java]

Comparable Interface

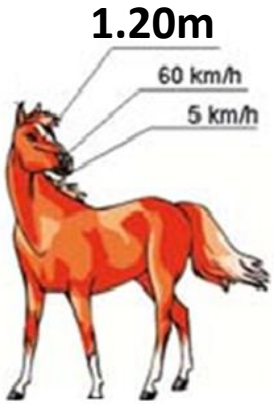
Wie lassen sich Objekte vergleichen?



Pferd 1



Pferd 2

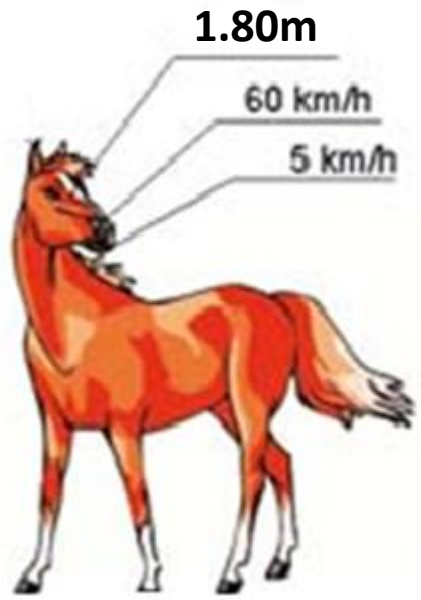


Pferd 3

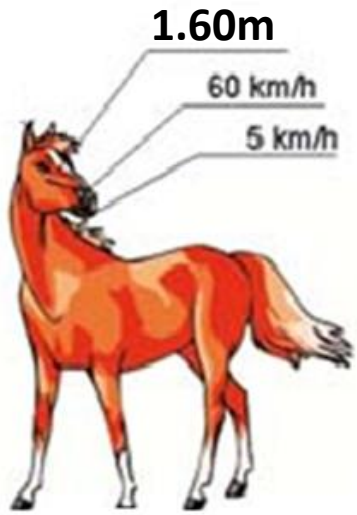
- Objekte lassen sich sortieren, d. h. in eine bestimmte Reihenfolge bringen

[Horse.java]

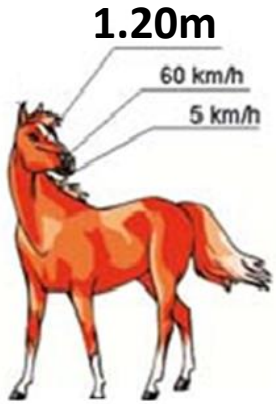
Wie lassen sich Objekte vergleichen?



Pferd 1



Pferd 2



Pferd 3

- Das Sortier-Kriterium kann der Programmierer selbst festlegen
- Beispielsweise: Die Größe des Pferdes

[Horse.java]

Comparable Interface

- Das Interface schreibt diese Methode vor:

```
public int compareTo(Object o)
```

- Diese Methode muss wie folgt implementiert werden:
  - Liefere 0 zurück, wenn das übergebene Objekt o an die gleiche Sortier-Stelle gehört, wie das vorliegende Objekt
  - Liefere einen Wert > 0 zurück, wenn das vorliegende Objekt hinter das übergebene Objekt o gehört
  - Liefere einen Wert < 0 zurück, wenn das vorliegende Objekt vor das übergebene Objekt o gehört

[Horse.java]

Beispiel

```
public class Horse implements Comparable{

@Override
public int compareTo(Object o) {
    if (o instanceof Horse){
        Horse other = (Horse) o;

        if (this.size < other.getSize()){
            return -1;
        }
        if (this.size > other.getSize()){
            return 1;
        }
        return 0;
    }
    // o ist kein Objekt der Klasse Horse
    return 0;
}
```

[Horse.java]

Eine Liste sortieren

- Sofern die Objekte eine Liste das Comparable Interface implementieren, kann diese wie folgt sortiert werden:

```
Collections.sort(liste);
```

- Die Reihenfolge der Sortierung wird durch die Implementierung der compareTo Methode festgelegt.

- Am Rande: Die Objekte einer Liste können wie folgt zufällig angeordnet werden:

```
Collections.shuffle(liste);
```

[Programm.java]

Beispiel

```
Horse pferd1 = new Horse("Anna", 120);
Horse pferd2 = new Horse("Max", 160);
Horse pferd3 = new Horse("Moritz", 180);

// Ist das pferd1 "kleiner als" pferd2?
if (pferd1.compareTo(pferd2) < 0){
    System.out.println("pferd1 ist kleiner als pferd2");
}
if (pferd1.compareTo(pferd2) > 0){
    System.out.println("pferd1 ist größer als pferd2");
}
if (pferd1.compareTo(pferd2) == 0){
    System.out.println("pferd1 ist genauso groß wie pferd2");
}
```

[Horse.java]

Beispiel

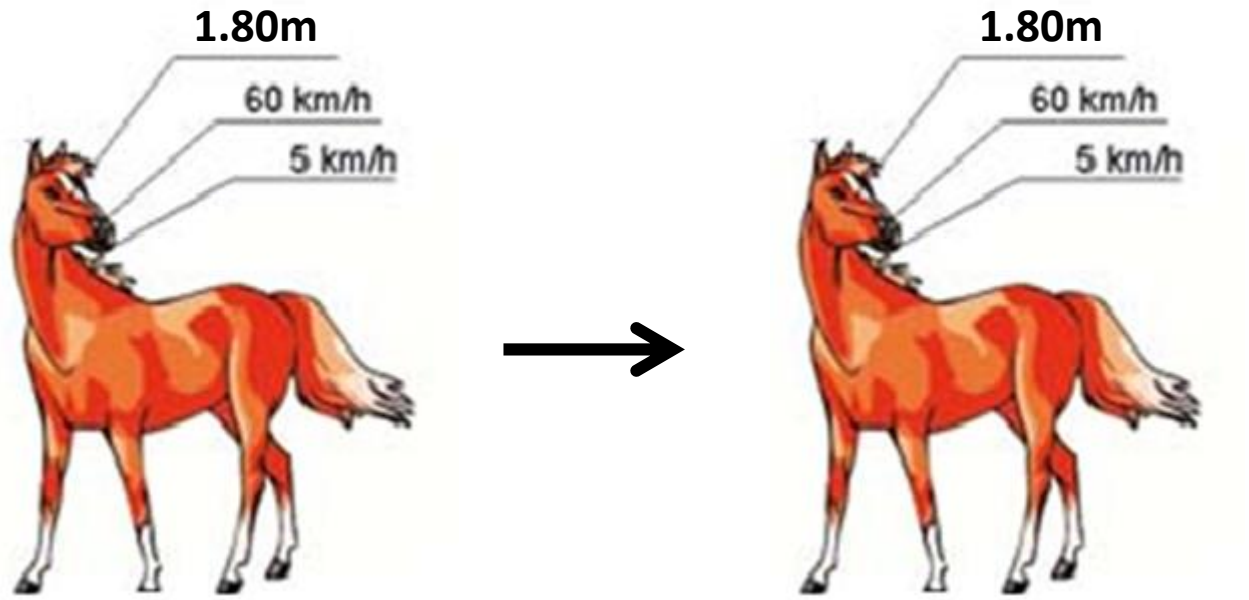
```
// Eine Liste mit Pferden sortieren
List<Horse> liste = new ArrayList<Horse>();
liste.add(pferd3);
liste.add(pferd1);
liste.add(pferd2);

// Die Liste anzeigen
// Beachten Sie die Reihenfolge der Pferde in der Liste
System.out.println(liste);

// Die Liste sortieren
Collections.sort(liste);

// ... und anzeigen
System.out.println(liste);

// Eine Liste mit Pferden durchmischen
Collections.shuffle(liste);
```

[Horse.java]

- Wie können mehrstufige Sortierkriterien implementiert werden?
- Beispiel:
  - Kriterium 1: Größe des Pferdes
  - Kriterium 2: Maximalgeschwindigkeit des Pferdes

Objekte kopieren

Wie lassen sich Objekte kopieren?



Pferd 1 Pferd 1

Was ist genau mit „Kopie“ eines Objekts gemeint?

- Ein „zweites“ Objekt mit der gleichen OID?
- Ein zweites Objekt der gleichen Klasse mit identischen Attributen?
  - Werden in diesem Fall alle Attribute (und deren Attribute) kopiert?

Kopieren per **clone()**

clone

```
protected Object clone()
                throws CloneNotSupportedException
```

Creates and returns a copy of this object. The precise meaning of "copy" may depend on the class of the object. The general intent is that, for any object x, the expression:

```
x.clone() != x
```

will be true, and that the expression:

```
x.clone().getClass() == x.getClass()
```

will be `true`, but these are not absolute requirements. While it is typically the case that:

```
x.clone().equals(x)
```

will be `true`, this is not an absolute requirement.

Kopieren per **clone()**

- Der clone-Kontrakt:
  - Kopie und Original sind verschiedene Objekte (unterschiedliche OIDs)
  - Kopie und Original sind vom selben Typ (gleiche Klasse)
  - Kopie und Original sollten gleich sein im Sinne von equals()

Cloneable

- Klassen, die eine **clone()** Methode anbieten, müssen das Interface **Cloneable** implementieren
- **Cloneable** ist ein reines Marker-Interface
- Es zeigt an, dass die implementierende Klasse mit **clone()** kopiert werden darf
- Wird **Cloneable** nicht implementiert, so liefert ein **clone()**-Aufruf eine **CloneNotSupportedException**

Cloneable

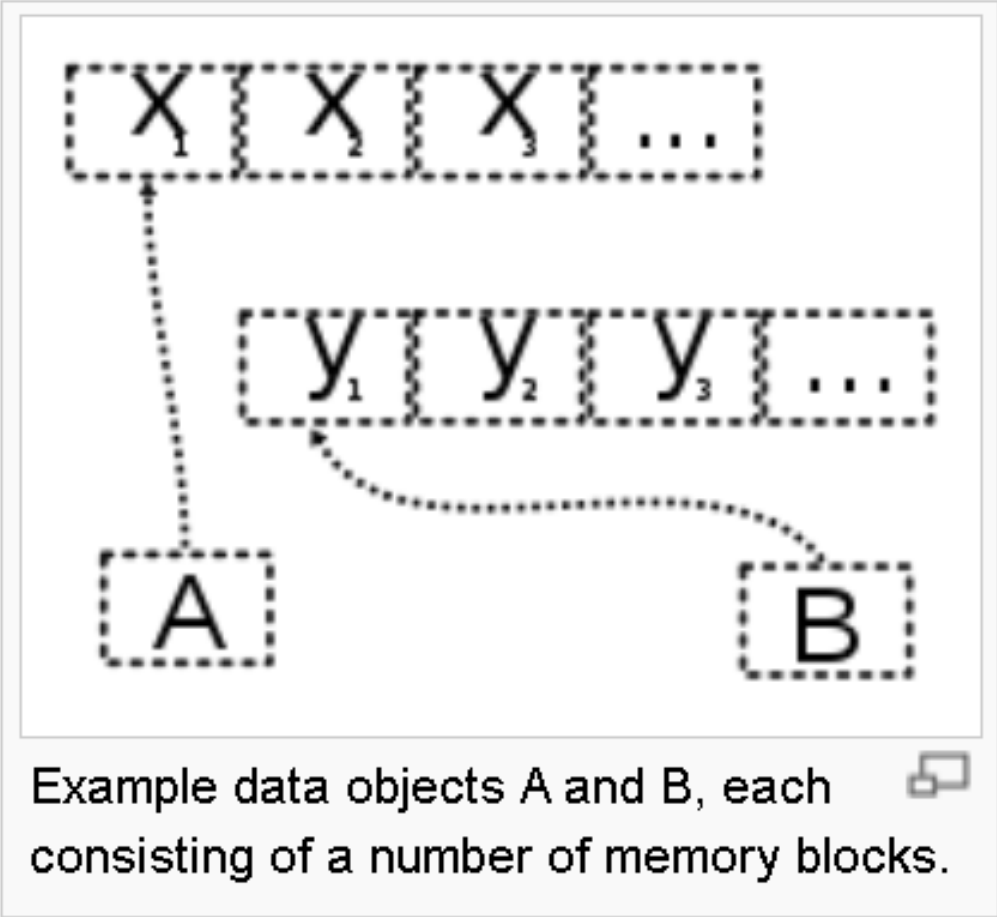
- Wenn man eine Klasse „Cloneable“ machen will, so muss man
  - das Interface **Cloneable** implementieren und
  - die Methode **clone()** public anbieten

```
38      public Object clone() {
39          try {
40              return super.clone();
41          } catch (CloneNotSupportedException e) {
42              // e.printStackTrace();
43          }
44          return null;
45      }
```

Cloneable

- Wenn man eine Klasse „Cloneable“ machen will, so muss man
  - das Interface **Cloneable** implementieren und
  - die Methode **clone()** public anbieten
  - … und das Objekt am besten gleich casten:

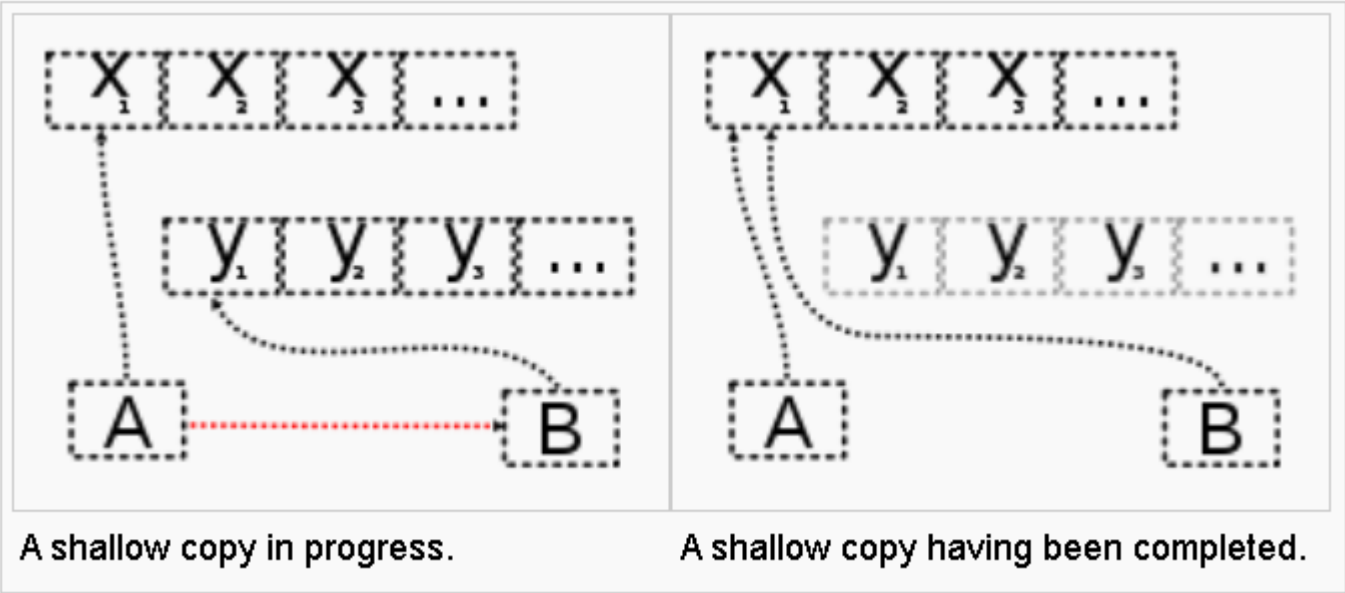
```
38      public Horse clone() {
39          try {
40              return (Horse) super.clone();
41          } catch (CloneNotSupportedException e) {
42              // e.printStackTrace();
43          }
44          return null;
45      }
```



Example data objects A and B, each consisting of a number of memory blocks.

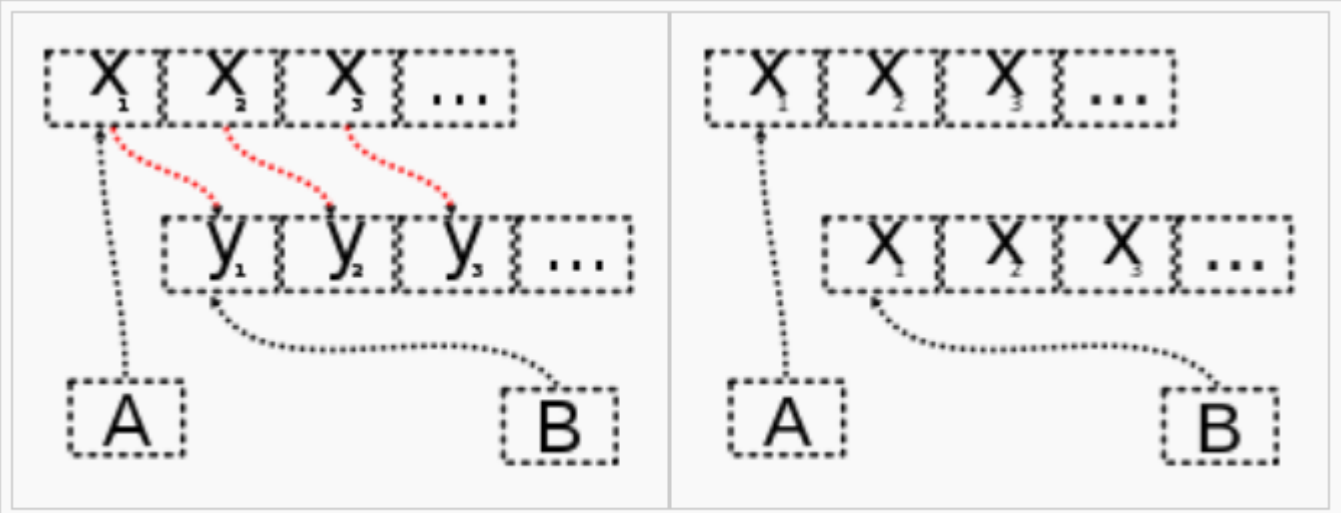
Quelle: http://en.wikipedia.org/wiki/Object_copy

- Flache Kopie (shallow copy):



Quelle: http://en.wikipedia.org/wiki/Object_copy

- Tiefe Kopie (deep copy):



A deep copy in progress. A deep copy having been completed.

Quelle: http://en.wikipedia.org/wiki/Object_copy

Flache vs. tiefe Kopie

- Ein Aufruf von clone() erzeugt ohne weitere Implementierungsschritte eine **flache Kopie**
- Eine **tiefe Kopie** muss implementiert werden, z. B.

```
54      public Horse deepClone() {
55          try {
56              Horse horseCopy = (Horse) super.clone();
57              horseCopy.sattel = (Sattel) horseCopy.sattel.clone();
58              return horseCopy;
59          } catch (CloneNotSupportedException e) {
60              // e.printStackTrace();
61          }
62          return null;
63      }
```

[Horse.java]

Arrays sind Cloneable

- Arrays enthalten eine clone() Methode:

```
29          Horse[] array = new Horse[3];
30          array[0] = new Horse("Max1", 180, "braun", 20);
31          array[1] = new Horse("Max2", 181, "blau", 21);
32          array[2] = new Horse("Max3", 182, "gruen", 22);
33
34          // Array kommt mit einem vorgefertigten clone()
35          Horse[] arrayCopy = array.clone();
```

- Diese erzeugt eine flache Kopie

Copy-Konstruktor

- Eine Alternative zu **clone**()
- Hiermit kann man tiefe und flache Kopien umsetzen:

```
14      // Copy Konstruktor
15      public Horse(Horse horse){
16          this.name = horse.name;
17          this.size = horse.size;
18          // Flache Kopie
19          this.sattel = horse.sattel;
20          // Alternative: Tiefe Kopie
21          // this.sattel = new Sattel(horse.sattel);
22      }
```

- **ArrayList** sieht beispielsweise einen Copy-Konstruktor vor (flache Kopie)

[Horse.java]

- Eine weitere (häufig genutzte) Alternative zu **clone**()
- Erstellt immer tiefe Kopien
- Besonders nützlich, wenn man externe Bibliotheken nutzt

```
public static Object deepCopy(Object o) throws Exception {
   ByteArrayOutputStream byteouts = new ByteArrayOutputStream();
   new ObjectOutputStream(byteouts).writeObject(o);
   ByteArrayInputStream byteAIns = new ByteArrayInputStream(byteouts.toByteArray());
   return new ObjectInputStream(byteAIns).readObject();
}
```

- … dazu später mehr (s. Abschnitt Serialisierung)

[Horse.java]

Wrapper / Autoboxing

Autoboxing und Unboxing

- In Java gibt es die elementaren Datentypen wie `int`, `float`, `char` und die Referenztypen.
- Die Referenztypen profitieren von den Vorteilen der Objektorientierung.
- Die elementaren Datentypen werden in Programmen mit intensiver Verarbeitung benötigt, um die Laufzeit knapp zu halten.
- Der Übergang von einem Typsystem zum anderen sieht wie folgt aus:

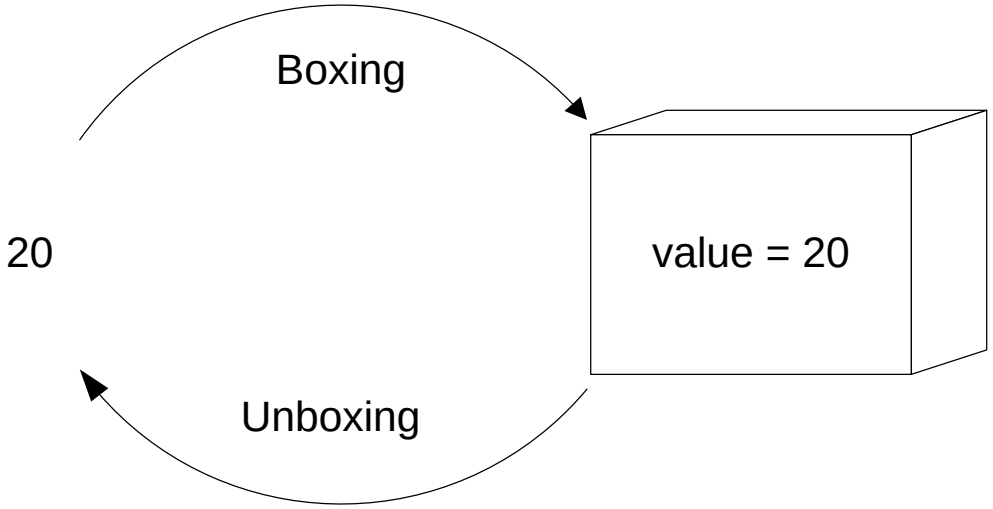
```
Integer i1 = new Integer (20);  // Sog. Boxing
int i      = i1. intValue();    // Sog. Unboxing
```

Autoboxing und Unboxing

Der Übergang von einem Typsystem zum anderen sieht wie folgt aus:

```
Integer i1 = new Integer (20); // Alt: Integer-Objekt
int i      = i1.intValue();    // Alt: Auslesen des Wertes
Integer i1 = 20;               // Neu: Sog. Autoboxing
  int i    = i1;               // Neu: Unboxing
```

Vergleichen Sie dies mit ein-/Auspacken.



Autoboxing und Unboxing

```
// Box-Typ anlegen
Integer i = new Integer(4);
// Unboxing
int j = i;

// Grunddatentyp anlegen
int x = 6;
// Boxing
Integer y = new Integer(x);
```

[Wrapper.java]

Zuordnung Referenztyp <-> Werttyp

Boolean	←→	**boolean**
Byte	←→	**byte**
Character	←→	**char**
Short	←→	**short**
Integer	←→	**int**
Long	←→	**long**
Float	←→	**float**
Double	←→	**double**

Weitere nützliche Methoden und Konstanten:

```
// Eine int Zahl zu einem String konvertieren
String string = Integer.toString(5);

// Eine Zahl aus einem String auslesen
int s = Integer.parseInt("7");

// Eine Zahl aus einem String auslesen
// Achtung: Wirft eine java.lang.NumberFormatException
// int t = Integer.parseInt("7Hallo");

// Für Optimierungsprobleme oftmals nützlich:
int max = Integer.MAX_VALUE;
int min = Integer.MIN_VALUE;
```

[Wrapper.java]

NaN: Not a Number

There are three kinds of operations that can return NaN:[4]

- Operations with a NaN as at least one operand.
- Indeterminate forms
  - The divisions 0/0 and ±∞/±∞
  - The multiplications 0×±∞ and ±∞×0
  - The additions ∞ + (−∞), (−∞) + ∞ and equivalent subtractions
  - The standard has alternative functions for powers:
    - The standard `pow` function and the integer exponent `pown` function define 0^0, 1^∞, and ∞^0 as 1.
    - The `powr` function defines all three indeterminate forms as invalid operations and so returns NaN.

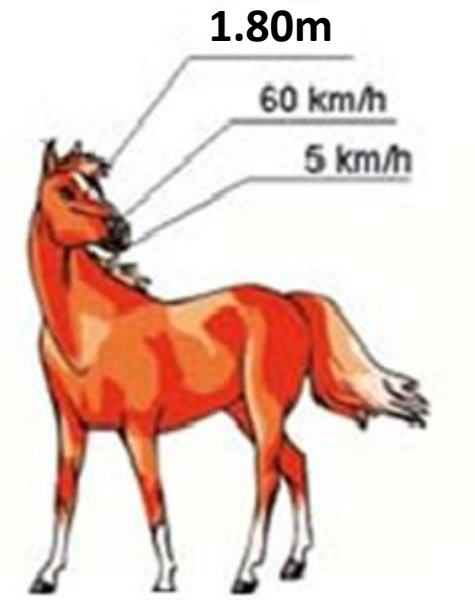
```
// Not a Number (NaN):
float f = Float.NaN;
// das liefert beispielsweise NaN:
float g = 0F / 0F;
float h = Float.NEGATIVE_INFINITY + Float.POSITIVE_INFINITY;
// Rechnen mit NaN:
float z = Float.NaN + 7.0F;
```

Werden bspw. genutzt, um fehlende Messwerte darzustellen

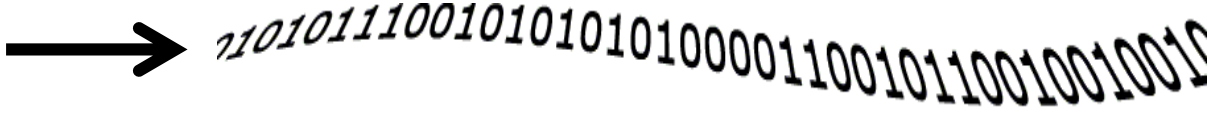
[Wrapper.java]

Serialisierung

Objekte speichern und übertragen



Pferd 1



Pferd 1

Wie kann ein Objekt, das im Arbeitsspeicher existiert, in eine Datei geschrieben werden oder über das Netzwerk übertragen werden?

Siehe Abschnitt 7 („Auf Dateien und auf das Netzwerk zugreifen“)

Iterator Interface

Iterator Interface

- Eine Liste von Elementen durchlaufen (… wie for, while)
- Nutzt einen **Cursor** beim Durchlaufen der Liste
- Ein Vorteil: Das Verändern der Liste (z. B. Objekte entfernen) während des Durchlaufens ist möglich

Iterator Interface

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

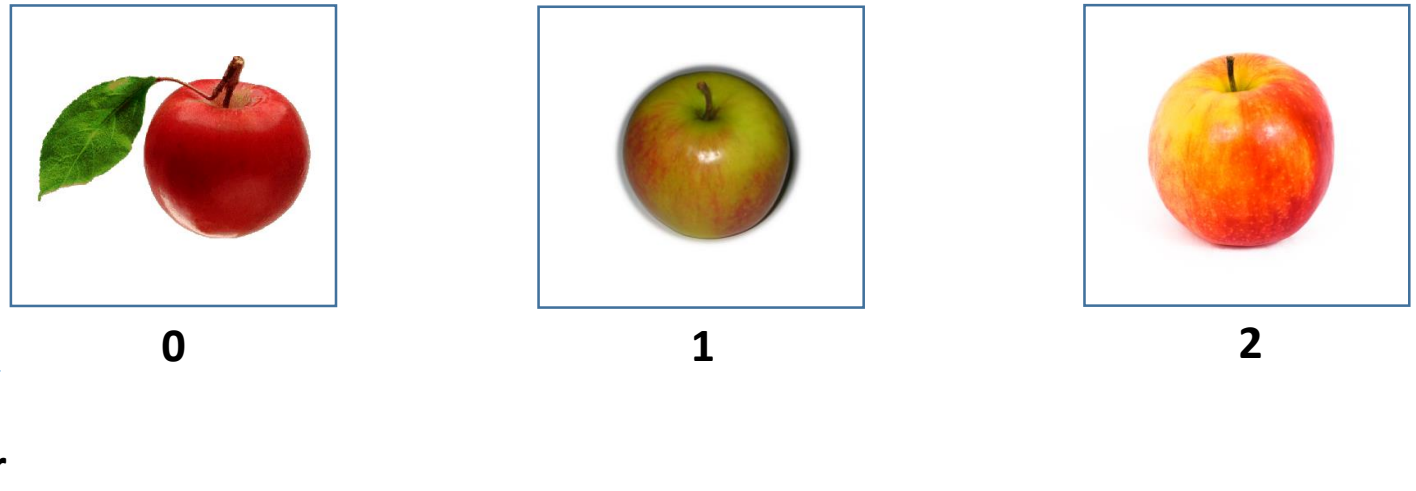
ListIterator Interface

Method Summary	
void	**add**(E o) Inserts the specified element into the list (optional operation).
boolean	**hasNext**() Returns `true` if this list iterator has more elements when traversing the list in the forward direction.
boolean	**hasPrevious**() Returns `true` if this list iterator has more elements when traversing the list in the reverse direction.
E	**next**() Returns the next element in the list.
int	**nextIndex**() Returns the index of the element that would be returned by a subsequent call to `next`.
E	**previous**() Returns the previous element in the list.
int	**previousIndex**() Returns the index of the element that would be returned by a subsequent call to `previous`.
void	**remove**() Removes from the list the last element that was returned by `next` or `previous` (optional operation).
void	**set**(E o) Replaces the last element returned by `next` or `previous` with the specified element (optional operation).

Cursor-Zugriff auf eine Liste

- Der **Cursor** zeigt nie direkt auf ein Element, immer davor bzw. dahinter (bzw. dazwischen)
- Der Cursor kann vor- und rückwärts bewegt werden
  - boolean **hasNext**() und **next**()
  - boolean **hasPrevious**() und **previous**()

liste =

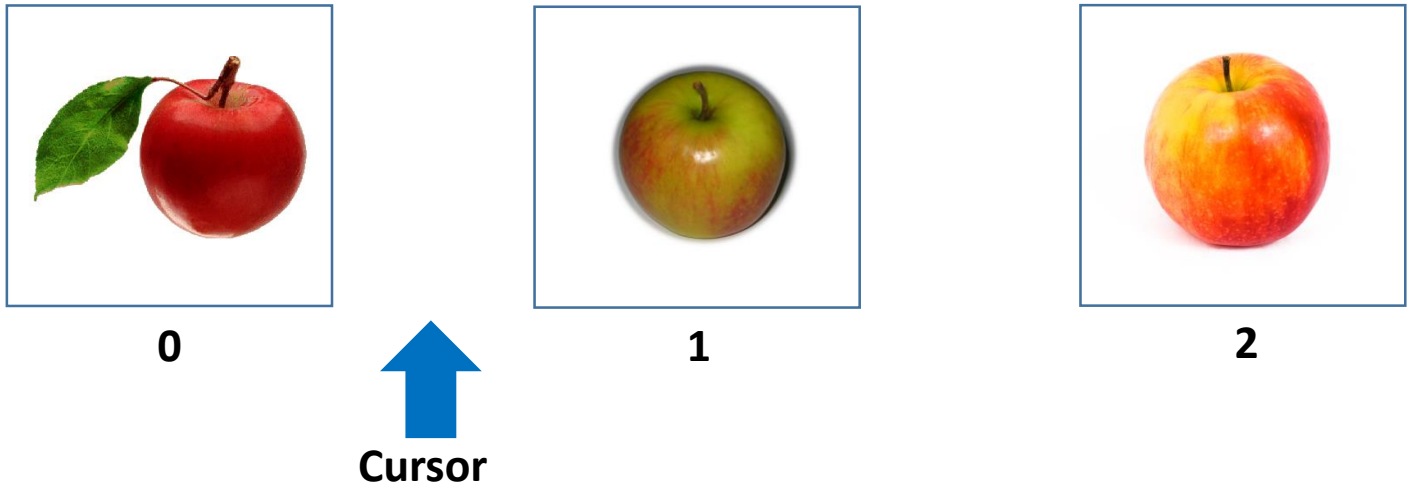


Cursor-Zugriff auf eine Liste

- Der **Cursor** zeigt nie direkt auf ein Element, immer davor bzw. dahinter (bzw. dazwischen)
- Der Cursor kann vor- und rückwärts bewegt werden
  - boolean **hasNext**() und **next**()
  - boolean **hasPrevious**() und **previous**()

Nach dem ersten Aufruf von next()

liste =

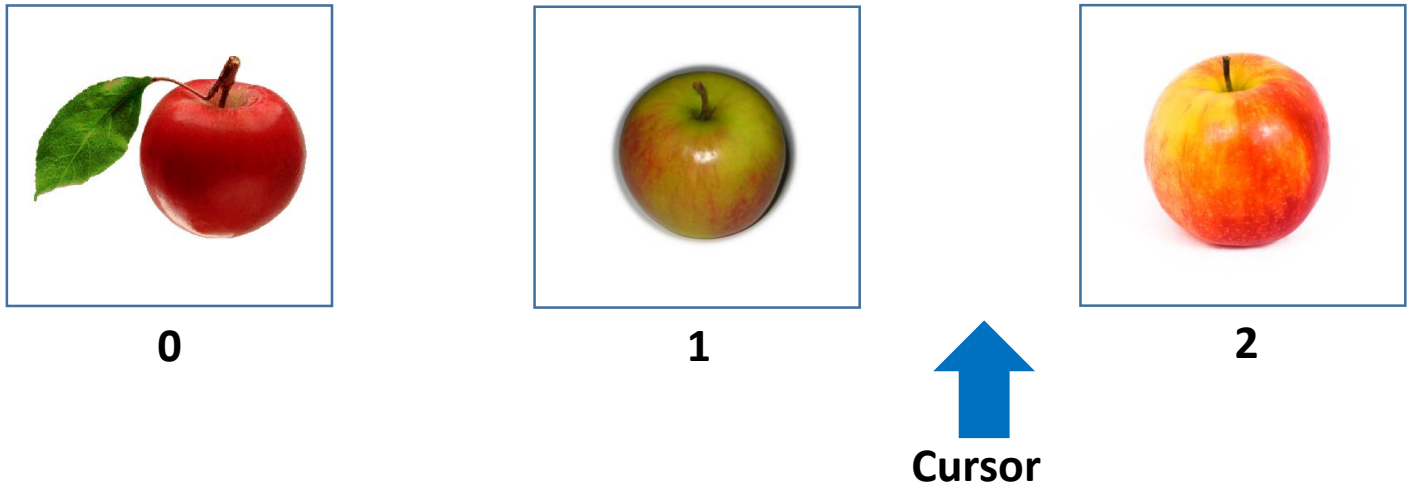


Cursor-Zugriff auf eine Liste

- Der **Cursor** zeigt nie direkt auf ein Element, immer davor bzw. dahinter (bzw. dazwischen)
- Der Cursor kann vor- und rückwärts bewegt werden
  - boolean **hasNext**() und **next**()
  - boolean **hasPrevious**() und **previous**()

Nach dem zweiten Aufruf von next() …

liste =



Beispiel

```
List<String> liste = new ArrayList<String>();

liste.add("String 1");
liste.add("String 2");
liste.add("String 3");
System.out.println(liste);

Iterator itr = liste.iterator();

while (itr.hasNext()){
    String currentString = (String) itr.next();
    System.out.println(currentString);
    itr.remove();
}

System.out.println(liste);
```

Fehlerhaftes Beispiel mit for

```
List<String> liste2 = new ArrayList<String>();

liste2.add("String 1");
liste2.add("String 2");
liste2.add("String 3");
System.out.println(liste2);

for(String str : liste2){
    liste2.remove(str);
}

System.out.println(liste2);
```

java.util.ConcurrentModificationException