Wichtige Datenstrukturen in Java

Boxed vs. Unboxed

```
public class Main {
  public static
    void main(String[] args) {
      int   a = 0;
      MyInt b = new MyInt();
      crazy(a,b);
      crazy(a,b);
      crazy(a,b);
  }
  public static
    void crazy(int c, MyInt d){
      c = c + 2;
      d.add(2);
      System.out.print(
        c + "/" +d.get() +"  ");
} }
public class MyInt {
  private int myint = 0;

  public int get() {
    return myint;
  }
  public void add(int x){
    this.myint = this.myint + x;
} }
```

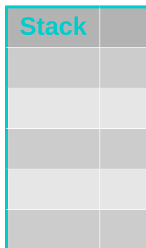
Ausgabe:

Boxed vs. Unboxed

```
public class Main {
  public static
    void main(String[] args) {
      int   a = 0;
      MyInt b = new MyInt();
      crazy(a,b);
      crazy(a,b);
      crazy(a,b);
  }
  public static
    void crazy(int c, MyInt d){
      c = c + 2;
      d.add(2);
      System.out.print(
        c + "/" +d.get() +"  ");
} }
public class MyInt {
  private int myint = 0;

  public int get() {
    return myint;
  }
  public void add(int x){
    this.myint = this.myint + x;
} }
```

Ausgabe: 2/2 2/4 2/6



Heap

Boxed vs. Unboxed

```
public class Main {
  public static
    void main(String[] args) {
      int   a = 0;
      MyInt b = new MyInt();
      crazy(a,b);
      crazy(a,b);
      crazy(a,b);
  }
  public static
    void crazy(int c, MyInt d){
      c = c + 2;
      d.add(2);
      System.out.print(
        c + "/" +d.get() +"  ");
} }
public class MyInt {
  private int myint = 0;

  public int get() {
    return myint;
  }
  public void add(int x){
    this.myint = this.myint + x;
} }
```

Ausgabe: 2/2 2/4 2/6

Stack	
a	0

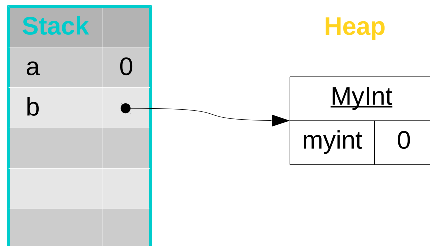
Heap

Boxed vs. Unboxed

```
public class Main {
  public static
    void main(String[] args) {
      int   a = 0;
      MyInt b = new MyInt();
      crazy(a,b);
      crazy(a,b);
      crazy(a,b);
  }
  public static
    void crazy(int c, MyInt d){
      c = c + 2;
      d.add(2);
      System.out.print(
        c + "/" +d.get() +"  ");
} }
public class MyInt {
  private int myint = 0;

  public int get() {
    return myint;
  }
  public void add(int x){
    this.myint = this.myint + x;
} }
```

Ausgabe: 2/2 2/4 2/6

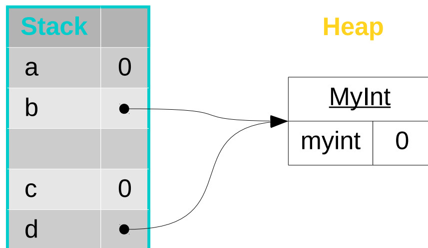


Boxed vs. Unboxed

```
public class Main {
  public static
    void main(String[] args) {
      int   a = 0;
      MyInt b = new MyInt();
      crazy(a,b);
      crazy(a,b);
      crazy(a,b);
  }
  public static
    void crazy(int c, MyInt d){
      c = c + 2;
      d.add(2);
      System.out.print(
        c + "/" +d.get() +"  ");
} }
public class MyInt {
  private int myint = 0;

  public int get() {
    return myint;
  }
  public void add(int x){
    this.myint = this.myint + x;
} }
```

Ausgabe: 2/2 2/4 2/6

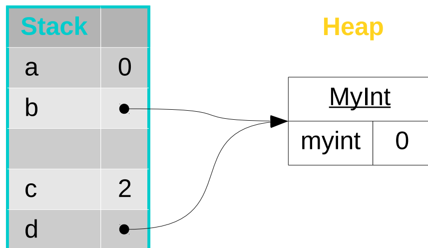


Boxed vs. Unboxed

```
public class Main {
  public static
    void main(String[] args) {
      int   a = 0;
      MyInt b = new MyInt();
      crazy(a,b);
      crazy(a,b);
      crazy(a,b);
  }
  public static
    void crazy(int c, MyInt d){
      c = c + 2;
      d.add(2);
      System.out.print(
        c + "/" +d.get() +"  ");
} }
public class MyInt {
  private int myint = 0;

  public int get() {
    return myint;
  }
  public void add(int x){
    this.myint = this.myint + x;
} }
```

Ausgabe: 2/2 2/4 2/6

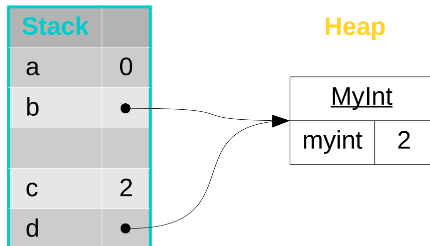


Boxed vs. Unboxed

```
public class Main {
  public static
    void main(String[] args) {
      int   a = 0;
      MyInt b = new MyInt();
      crazy(a,b);
      crazy(a,b);
      crazy(a,b);
  }
  public static
    void crazy(int c, MyInt d){
      c = c + 2;
      d.add(2);
      System.out.print(
        c + "/" +d.get() +"  ");
} }
public class MyInt {
  private int myint = 0;

  public int get() {
    return myint;
  }
  public void add(int x){
    this.myint = this.myint + x;
} }
```

Ausgabe: 2/2 2/4 2/6

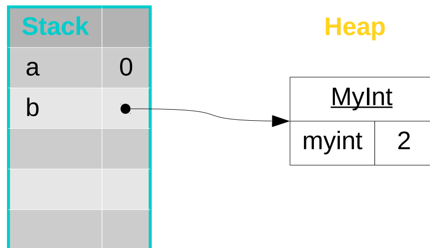


Boxed vs. Unboxed

```
public class Main {
  public static
    void main(String[] args) {
      int   a = 0;
      MyInt b = new MyInt();
      crazy(a,b);
      crazy(a,b);
      crazy(a,b);
  }
  public static
    void crazy(int c, MyInt d){
      c = c + 2;
      d.add(2);
      System.out.print(
        c + "/" +d.get() +"  ");
} }
public class MyInt {
  private int myint = 0;

  public int get() {
    return myint;
  }
  public void add(int x){
    this.myint = this.myint + x;
} }
```

Ausgabe: 2/2 2/4 2/6

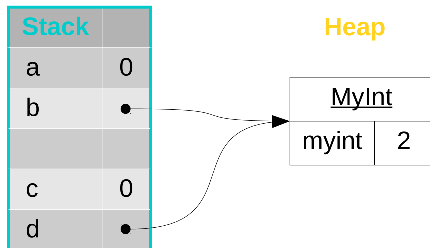


Boxed vs. Unboxed

```
public class Main {
  public static
    void main(String[] args) {
      int   a = 0;
      MyInt b = new MyInt();
      crazy(a,b);
      crazy(a,b);
      crazy(a,b);
  }
  public static
    void crazy(int c, MyInt d){
      c = c + 2;
      d.add(2);
      System.out.print(
        c + "/" +d.get() +"  ");
} }
public class MyInt {
  private int myint = 0;

  public int get() {
    return myint;
  }
  public void add(int x){
    this.myint = this.myint + x;
} }
```

Ausgabe: 2/2 2/4 2/6

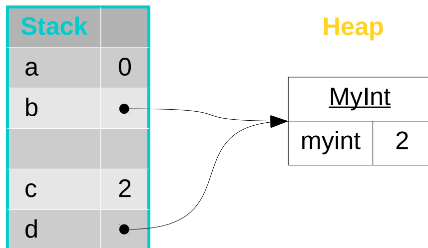


Boxed vs. Unboxed

```
public class Main {
  public static
    void main(String[] args) {
      int   a = 0;
      MyInt b = new MyInt();
      crazy(a,b);
      crazy(a,b);
      crazy(a,b);
  }
  public static
    void crazy(int c, MyInt d){
      c = c + 2;
      d.add(2);
      System.out.print(
        c + "/" +d.get() +"  ");
} }
public class MyInt {
  private int myint = 0;

  public int get() {
    return myint;
  }
  public void add(int x){
    this.myint = this.myint + x;
} }
```

Ausgabe: 2/2 2/4 2/6

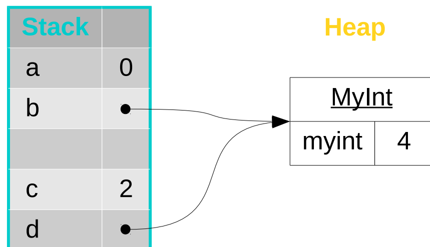


Boxed vs. Unboxed

```
public class Main {
  public static
    void main(String[] args) {
      int   a = 0;
      MyInt b = new MyInt();
      crazy(a,b);
      crazy(a,b);
      crazy(a,b);
  }
  public static
    void crazy(int c, MyInt d){
      c = c + 2;
      d.add(2);
      System.out.print(
        c + "/" +d.get() +"  ");
} }
public class MyInt {
  private int myint = 0;

  public int get() {
    return myint;
  }
  public void add(int x){
    this.myint = this.myint + x;
} }
```

Ausgabe: 2/2 2/4 2/6

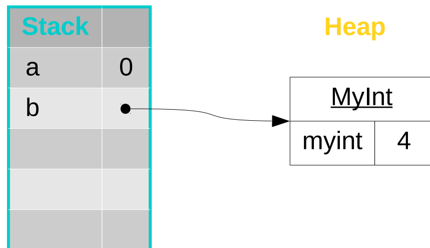


Boxed vs. Unboxed

```
public class Main {
  public static
    void main(String[] args) {
      int   a = 0;
      MyInt b = new MyInt();
      crazy(a,b);
      crazy(a,b);
      crazy(a,b);
  }
  public static
    void crazy(int c, MyInt d){
      c = c + 2;
      d.add(2);
      System.out.print(
        c + "/" +d.get() +"  ");
} }
public class MyInt {
  private int myint = 0;

  public int get() {
    return myint;
  }
  public void add(int x){
    this.myint = this.myint + x;
} }
```

Ausgabe: 2/2 2/4 2/6

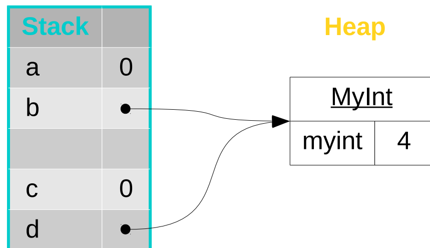


Boxed vs. Unboxed

```
public class Main {
  public static
    void main(String[] args) {
      int   a = 0;
      MyInt b = new MyInt();
      crazy(a,b);
      crazy(a,b);
      crazy(a,b);
  }
  public static
    void crazy(int c, MyInt d){
      c = c + 2;
      d.add(2);
      System.out.print(
        c + "/" +d.get() +"  ");
} }
public class MyInt {
  private int myint = 0;

  public int get() {
    return myint;
  }
  public void add(int x){
    this.myint = this.myint + x;
} }
```

Ausgabe: 2/2 2/4 2/6

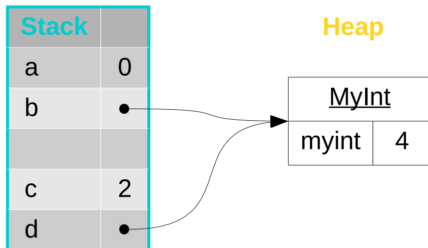


Boxed vs. Unboxed

```
public class Main {
  public static
    void main(String[] args) {
      int   a = 0;
      MyInt b = new MyInt();
      crazy(a,b);
      crazy(a,b);
      crazy(a,b);
  }
  public static
    void crazy(int c, MyInt d){
      c = c + 2;
      d.add(2);
      System.out.print(
        c + "/" +d.get() +"  ");
} }
public class MyInt {
  private int myint = 0;

  public int get() {
    return myint;
  }
  public void add(int x){
    this.myint = this.myint + x;
} }
```

Ausgabe: 2/2 2/4 2/6

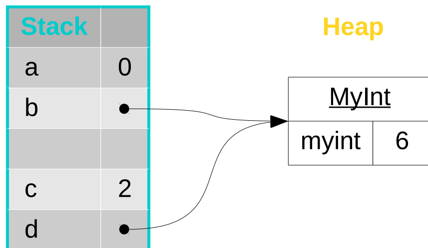


Boxed vs. Unboxed

```
public class Main {
  public static
    void main(String[] args) {
      int   a = 0;
      MyInt b = new MyInt();
      crazy(a,b);
      crazy(a,b);
      crazy(a,b);
  }
  public static
    void crazy(int c, MyInt d){
      c = c + 2;
      d.add(2);
      System.out.print(
        c + "/" +d.get() +"  ");
} }
public class MyInt {
  private int myint = 0;

  public int get() {
    return myint;
  }
  public void add(int x){
    this.myint = this.myint + x;
} }
```

Ausgabe: 2/2 2/4 2/6

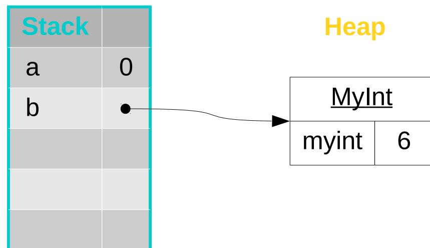


Boxed vs. Unboxed

```
public class Main {
  public static
    void main(String[] args) {
      int   a = 0;
      MyInt b = new MyInt();
      crazy(a,b);
      crazy(a,b);
      crazy(a,b);
  }
  public static
    void crazy(int c, MyInt d){
      c = c + 2;
      d.add(2);
      System.out.print(
        c + "/" +d.get() +"  ");
} }
public class MyInt {
  private int myint = 0;

  public int get() {
    return myint;
  }
  public void add(int x){
    this.myint = this.myint + x;
} }
```

Ausgabe: 2/2 2/4 2/6



Boxed vs. Unboxed

Primitive Datentypen: `int`, `double`, `boolean`
Klassen: `Integer`, `Double`, `Boolean`

Unterschiede

- Werte von `Integer`, `Double` oder `Boolean` sind Objekte
- Allerdings gibt es keine Setter-Methoden, d.h. Werte dieser Typen können nicht verändert werden; Seiteneffekte wie auf der vorangegangenen Folie können nicht auftreten!
- Java wandelt ggf. `int` in `Integer` um; manuell:

```
Integer x = Integer.valueOf(7);
int i = x.intValue();
```

- Primitive Datentypen nicht in Generics erlaubt

Generics

So wie es Arrays über `int[]` oder `double[]` gibt, erlaubt Java auch Klassen über andere Typen zu parametrisieren.
Diese **Typparameter** werden in spitze Klammern geschrieben und müssen Klassen sein.
Beispiele `ArrayList<Integer>` oder `ArrayList<Double>`

Arrays vs. ArrayList

```
double[] ary = { 1.2, 3.4 };                // Erstellen
                                            //  & Initialisieren
double d  = ary[0];                         // Zelle lesen
ary[1] = 6.9;                               // Zelle schreiben
System.out.println("Size:" + ary.length);   // "Size:2"
System.out.println(ary);                    // "[D@135fbaa4"
```

Klassische Arrays sollte man möglichst vermeiden und stattdessen `ArrayList` verwenden:

```
List<Double> ary= new ArrayList<Double>();// Erstellen
Collections.addAll(ary, 1.2, 3.4);        // Initialisieren
Double d = ary.get(0);                    // Zelle lesen
ary.set(1,6.9);                           // Zelle schreiben
System.out.println("Size:" + ary.size()); // "Size:2"
System.out.println(ary);                  // "[1.2, 6.9]"
```

Beachte:

1. Andere Initialisierung
2. `Double` statt `double`
3. `size()` statt `length`

Vorteile:

1. `toString()` geht nun
2. `add(index,wert)` möglich
3. `remove` möglich

Arrays vs ArrayList

Klassische Arrays sind in Java primär aus historischen Gründen enthalten. `ArrayList` hat zahlreiche Vorteile:

- Größenänderung einfacher ()
- Bessere Typsicherheit (späteres Plenum zu „Generics“)
- Bessere Wartbarkeit des Codes, da leichter auszutauschen gegen andere Datenstruktur

Für uns nennenswerte Nachteile sind:

- Aufwändigere Syntax durch reguläre Methodenaufrufe, z.B. `ary.set(7,8);` statt `ary[7]=8`
- Elemente müssen Objekte einer Klasse sein, d.h. `Double` statt `double`, `Integer` statt `int`, `Boolean` statt `bool`, etc.
- `ArrayList` ist immer 1-dimensional; möglich aber meist umständlich: `ArrayList<ArrayList<Double>>`

Weitere Unterschiede bei der Speicherverwaltung.

Schleifen funktionieren unverändert, mehr dazu später heute.

Datenstrukturen

Datenstrukturen legen fest, wie Daten im Speicher angeordnet werden und nach welchem Muster darauf zugegriffen wird.

Modulares Design von Datenstrukturen:

- **Interface** legt Funktionalität fest, spezifiziert Verhalten
- **Implementierung** legt Effizienz fest, Verbrauch von Zeit und Speicherplatz

Austausch der Implementierung darf Performanz beeinflussen, nicht jedoch semantische Korrektheit!

Datenstruktur Beispiele

Beispiele für beliebte Datenstrukturen in Java sind:

Struktur	Interface	Implementierung
Liste	`List<E>`	`ArrayList`, `LinkedList`, `Stack`, ...
Menge	`Set<E>`	`HashSet`, `TreeSet`, `EnumSet`, ...
Abbildung	`Map<K,V>`	`HashMap`, `TreeMap`, `EnumMap`, ...

`E` repräsentiert Typ der Elemente, ebenso `K` und `V`.

Tipp: Implementierung ist Subtyp des Interface!
Für Variablen und Attribute immer das Interface als Typ verwenden

```
List<E>      ary = new ArrayList<E>();
```

ist besser als

```
ArrayList<E> ary = new ArrayList<E>();
```

Ebenso bei Methodendeklarationen ist es besser

```
public void foo(     List<E> li){...};
```

zu schreiben als

```
public void foo(ArrayList<E> li){...};
```

Späterer Austausch der Datenstruktur dadurch leichter.

Interface Collection

Das Interface Collection<E> erfasst alle Arten von endlichen Sammlungen von Objekten eines Typs E.

Beispiel

Collection<Integer> bezeichnet Sammlungen von Zahlen; LinkedList<Integer> ist auch eine Sammlung von Zahlen und ist in der Tat auch ein Subtyp von Collection<Integer>.

Funktionalität

- Test, ob Element enthalten ist
- Hinzufügen / Entfernen eines Elementes
- Abfrage der Anzahl der enthaltenen Elemente
- Iteration über Elemente der Menge
- Streamen der Elemente

Interface `Collection<E>`

```
interface Collection<E> extends Iterable<E>
  int         size();
  boolean     isEmpty();
  boolean     add(E e);
  boolean     addAll(Collection<? extends E> c);
  boolean     contains(Object o);
  boolean     containsAll(Collection<?> c);
  boolean     remove(Object o);
  boolean     removeAll(Collection<?> c);
  boolean     retainAll(Collection<?> c);
  Stream<E>   stream();
  Stream<E>   parallelStream();
  Iterator<E> iterator();
```

Bemerkungen

- Vergleiche benutzen Methode `equals`
- Typ `Object` aus historischen Gründen,
  aber z.B. bei `remove`/`contains` ohnehin harmlos

Schleifen

Jedes Objekt vom Typ `Collection<E>` erlaubt Iteration über die enthaltenen Objekte:

```
Collection<E> collection;

for ( E e: collection ) {
  ...
}
```

Da `ArrayList<E>` auch das Interface `Collection<E>` implementiert, dürfen wir alle zuvor gezeigten Methoden verwenden und auch obige Schleifennotation:

```
List<Double> a = new ArrayList<>();
a.add(42);
a.add(77);
Double sum = 0;
for (Double d : a) { sum += a };
// sum == 119
```

Schleifen

Achtung: Innerhalb solcher Schleifen darf die Sammlung nicht verändert werden; also kein `add` oder `remove` in der Schleife!

```
List<Double> a = new ArrayList<>();
Collections.addAll(a, 42 77);
for (Double d : a) { ... };
```

Abhilfe 1: Iteration über Kopie der Sammlung

```
List<Double> a = new ArrayList<>();
Collections.addAll(a, 42 77);
List<Double> akopie = new ArrayList<>(a);
for (Double d : akopie) {
  ...
};
```

Schleifen

Achtung: Innerhalb solcher Schleifen darf die Sammlung nicht verändert werden; also kein `add` oder `remove` in der Schleife!

```
List<Double> a = new ArrayList<>();
Collections.addAll(a, 42 77);
for (Double d : a) { ... };
```

Abhilfe 2: Ergebnis in temporärer Variable

```
List<Double> a = new ArrayList<>();
Collections.addAll(a, 42 77);
List<Double> aresult = new ArrayList<>(); // leer
for (Double d : a) {
  ...
  aresult.add(d);
};
```

Nachteil: Ergebnis muss ggf. umkopiert werden,
`a = aresult;` funktioniert vermutlich nicht wie gewünscht!

Schleifen

Achtung: Innerhalb solcher Schleifen darf die Sammlung nicht verändert werden; also kein `add` oder `remove` in der Schleife!

```
List<Double> a = new ArrayList<>();
Collections.addAll(a, 42 77);
for (Double d : a) { ... };
```

Abhilfe 3: Gewöhnliche Schleife mit Zähler

```
List<Double> a = new ArrayList<>();
Collections.addAll(a, 42 77);
for (int i=0; i<a.size();i++) {
  Double d = a.get(i);
  ...
};
```

Nachteil: Zähler manipulierbar, schlechter zu lesen

Übersicht Datenstrukturen

Beispiele für beliebte Datenstrukturen in Java sind:

Struktur	Interface	Implementierung
Liste	`List<E>`	`ArrayList`, `LinkedList`, `Stack`, ...
Menge	`Set<E>`	`HashSet`, `TreeSet`, `EnumSet`, ...
Abbildung	`Map<K,V>`	`HashMap`, `TreeMap`, `EnumMap`, ...

Liste Eine *geordnete* Folge von Elementen, möglicherweise mit Duplikaten

Menge Eine *ungeordnete* Menge von Elementen, ohne Duplikate

Abbildung Zuordnung zwischen Objekten, z.B. ein Adressbuch ordnet jedem Namen Telefonnummern zu.

Listen

Geordnete Folge von Elementen gleichen Typs, möglicherweise mit Duplikaten.

```
Interface List<E> extends Collection<E>{
  // alt, aus Collection
  boolean isEmpty();                  // Liste leer?
  boolean add ( E e );                // Am Ende einfügen
  boolean remove(Object o);           // Erstes entfernen

  // neu, nur für Listen
  boolean add ( int index , E e ); // Bei index einfügen
  E get( int index );                 // Element bei index
  E remove( int index );     // Element bei index löschen
  ...
}
```

Implementierung: ArrayList

ArrayList<E> implementiert List<E> mit klassischen Arrays

a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]	a[9]
12	1	3	12	7	13	23			

Geeignet, falls

- Oft auf einzelne Element mit Index zugegriffen wird, also viele Zugriffe `a.get(index)` $O(1)$

Ungeeignet, falls

- Anzahl der Elemente sich oft ändert, also selten Aufrufe von `a.add(e)` oder `a.remove(e)` $O(n)$

Hinzufügen eines Elementes benötigt Umkopieren aller folgenden Elemente. Die Programmbibliothek macht dies zwar automatisch für uns, aber es kostet den Anwender Rechenzeit!

Implementierung: ArrayList

`ArrayList<E>` implementiert `List<E>` mit klassischen Arrays

a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]	a[9]
12	5	1	3	12	7	13	23		

Geeignet, falls

- Oft auf einzelne Element mit Index zugegriffen wird, also viele Zugriffe `a.get(index)` $O(1)$

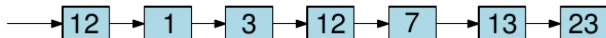
Ungeeignet, falls

- Anzahl der Elemente sich oft ändert, also selten Aufrufe von `a.add(e)` oder `a.remove(e)` $O(n)$

Hinzufügen eines Elementes benötigt Umkopieren aller folgenden Elemente. Die Programmbibliothek macht dies zwar automatisch für uns, aber es kostet den Anwender Rechenzeit!

Implementierung: LinkedList

`LinkedList<E>` implementiert `List<E>` als Kette von Objekten



Geeignet, falls

- Anzahl der Elemente sich oft ändert, $O(1)$
  also oft Aufrufe von `a.add(e)` oder `a.remove(e)`
- Wenn meistens sowieso *alle* Elemente der Reihe nach verwendet werden, z.B. mit Schleifen. $O(n)$

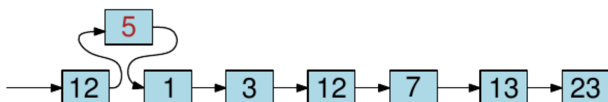
Ungeeignet, falls

- Oft auf einzelne Element mit Index zugegriffen wird, $O(n)$
  also viele Zugriffe `a.get(index)`

Verändern der Elementanzahl benötigt nur das Umsetzen von Zeigern, aber wenn man ein spezielles Element sucht, muss man sich durchhangeln.

Implementierung: LinkedList

`LinkedList<E>` implementiert `List<E>` als Kette von Objekten



Geeignet, falls

- Anzahl der Elemente sich oft ändert, also oft Aufrufe von `a.add(e)` oder `a.remove(e)` $O(1)$
- Wenn meistens sowieso *alle* Elemente der Reihe nach verwendet werden, z.B. mit Schleifen. $O(n)$

Ungeeignet, falls

- Oft auf einzelne Element mit Index zugegriffen wird, also viele Zugriffe `a.get(index)` $O(n)$

Verändern der Elementanzahl benötigt nur das Umsetzen von Zeigern, aber wenn man ein spezielles Element sucht, muss man sich durchhangeln.

Beispiel

```
List<Double> ary = new ArrayList<Double>(); // Erstellen
Collections.addAll(ary, 1.2, 3.4);          // Initialisieren
Double d = ary.get(0);                      // Zelle lesen
ary.set(1, 5.6);                            // Zelle schreiben
System.out.println("Size:" + ary.size());   // "Size:2"
System.out.println(ary);                    // "[1.2, 5.6]"
Double sum = 0;
for (Double f : ary) {
  sum += f
};                                          // sum=6.8
```

Geht ganz genauso, wenn man in der ersten Zeile `LinkedList` anstatt `ArrayList` schreibt, lediglich Laufzeit ändert sich bei Listen mit vielen Einträgen!

Mengen

Eine *ungeordnete* Menge von Elementen gleichen Typs,
ohne Duplikate

- Reihenfolge *unerheblich*
- Elemente maximal *einmal* enthalten sonst: Bag / MultiSet
- Gleichheit mit `equals` muss verfügbar sein, d.h. zwei Elemente `e1` und `e2` gelten als gleich, falls `e1.equals(e2)` den Wert `true` liefert!

Funktionalität

- Test, ob Element in Menge enthalten ist Primäre Op.
- Hinzufügen / Entfernen eines Elementes
- Bildung von Vereinigungs- / Schnittmenge
- Iteration über alle Elemente der Menge
  (in unvorhersehbarer Reihenfolge)

Interface `Set<E>`

```
interface Set<E> extends Collection<E>
  boolean contains(Object o);
  boolean add(E e);
  boolean addAll(Collection<? extends E> c);
  boolean remove(Object o);
  boolean removeAll(Collection<?> c);
  boolean retainAll(Collection<?> c);
  Iterator<E> iterator();
  ...
```

Nahezu gleich wie `Collection`, aber mit stärkeren Annahmen:

- jedes Element-Objekt darf nur einmal vorhanden sein
- Element-Objekte sollten möglichst immutable sein
- Vergleiche benutzen Methode `equals`
- Implementierung mittels **Hashing** oder **Suchbäumen**

Endliche Mengen

Die grundlegende Operationen auf einer Menge sollten möglichst schnell ablaufen:

- Element hinzufügen
- Element entfernen
- Test auf Elementschaft

Die Schnittstelle `Set<E>` wird z.B. durch folgende Klassen implementiert:

`TreeSet<E>` Operationen `add`, `remove`, `contains` haben immer logarithmischen Aufwand $O(\log n)$

`HashSet<E>` Operationen `add`, `remove`, `contains` haben höchstens linearen Aufwand, $O(n)$
in der Praxis jedoch meist konstant $O(1)$
Dies hängt vom Hashing ab gleich mehr dazu

Beipiel für Mengen

Wenn bei Minesweeper ein Feld ohne Minen in der Nachbarschaft aufgedeckt wurde, so musste gleiche der gesamte zusammenhängende Bereich ohne Minen aufgedeckt werden.
Mit Mengen kann man dies wie folgt lösen:

```
public Set<Feld> alleNachbarn(Feld startfeld) {
   Set<Feld> erledigt = new HashSet<>();
   Set<Feld> neue = new HashSet<>();
   neue.add(startfeld);
   while (!neue.isEmpty()) {
    Feld akt = neue.remove(neue.iterator().next());
    if (!erledigt.contains(akt)) {
      neue.addAll(getNachbarn(akt));
      erledigt.add(akt);
    }
   };
   return erledigt;
}
```

Identität und Gleichheit

Objekt-Identität `o1 == o2` ist wahr, falls `o1` und `o2` dasselbe Objekt sind (Vergleich der Speicheraddresse).

Objektgleichheit `o1.equals(o2)` ist wahr, falls beide Objekte gleiche Eigenschaften haben.

Beispiel:

```
Integer a = new Integer(1023);
Integer b = new Integer(1023);
(a==b)       // false
a.equals(b) // true
```

Die Methode `equals` wird von `Object` geerbt und sollte meistens überschrieben werden!

Methode equals

Die Methode `equals` wird von `Object` geerbt und sollte meistens überschrieben werden! Dabei müssen folgende Eigenschaften sichergestellt werden:

- Reflexiv: `x.equals(x)` ist immer wahr
- Symmetrisch: `x.equals(y) == y.equals(x)` gilt immer
- Transitiv: `x.equals(y)` und `y.equals(z)` impliziert `x.equals(z)`

Voraussetzung: `x`,`y`,`z` nicht `null`

Achtung: Die Implementierungen der Bibliotheken (also `ArrayList`, `TreeSet`, etc.) verlassen sich darauf!
Stimmt dies nicht, können beliebige Fehler zufällig auftauchen!

Der Code für `equals` kann von den IDEs oft automatisch erzeugt werden.

Beispiel: equals

```
public class MineCell {

  private final Position position;
  private boolean mine;
  private boolean flagged;

  @Override
  public boolean equals(Object o) {
      if (this == o) return true;
      if (o == null || getClass() != o.getClass()) return false;
      MineCell mineCell = (MineCell) o;
      if (mine    != mineCell.mine)    return false;
      if (flagged != mineCell.flagged) return false;
      return
        getPosition() != null
        ? getPosition().equals(mineCell.getPosition())
        : mineCell.getPosition() == null;
} }
```

⇒ Meistens alle Attribute mit `equals` vergleichen!

Hashing

Die Klasse `Object` enthält eine Methode

```
int hashCode();
```

Sie liefert zu jedem Objekt einen Integer, den **Hashcode** oder **Hashwert**, dies ist eine Art “Fingerabdruck” eines Objekts.

Idee: Wenn die Berechnung des “Fingerabdrucks” schnell geht, kann man damit Vergleiche abkürzen: Zwei Objekte mit verschiedenen “Fingerabdruck” braucht man nicht mehr mit `equals` vergleichen!

Aber in seltenen Fällen können verschiedene Objekte den gleichen Fingerabdruck haben.

Hashwerte

Die Spezifikation von `hashCode` besagt, dass zwei im Sinne von `equals` gleiche Objekte denselben Hashwert haben sollen.

Es muss also *immer* gelten:
Wenn `x.equals(y)` dann `x.hashCode() == y.hashCode()`

Es ist aber erlaubt, dass zwei verschiedene Objekte denselben Hashwert haben. Das ist kein Wunder, denn es gibt ja "nur" 2^{32} `int`-Werte.

Allerdings sorgt eine gute Implementierung von `hashCode` dafür, dass die Hashwerte möglichst breit gestreut (*to hash* = fein hacken) werden. Bei "zufälliger" Wahl eines Objekts einer festen Klasse sollen alle Hashwerte "gleich wahrscheinlich" sein.

Fallstrick Gleichheit / Hashing

Wenn man die Methode `equals` überschreibt, so *muss* man auch immer Methode `hashCode` überschreiben, so dass gilt:

Wenn `x.equals(y)` dann `x.hashCode() == y.hashCode()`

Achtung Gilt diese Eigenschaften nicht, so funktionieren `HashSet<E>`, `HashMap<K,V>`, etc. nicht mehr richtig! Es können beliebige Fehler zufällig auftauchen!

Tipp: `equals` und `hashCode` durch IDE generieren lassen! Der automatische erzeugte Code für `hashCode` ist üblicherweise gut genug für den effizienten Einsatz von `HashSet` oder `HashMap`.

Beispiel: hashCode

```
public class MineCell {

  private final Position position;
  private boolean mine;
  private boolean flagged;

  @Override
  public boolean equals(Object o) { ... };

  @Override
  public int hashCode() {
    int result =
      getPosition() != null ? getPosition().hashCode() : 0;
    result = 31 * result + (mine    ? 1 : 0);
    result = 31 * result + (flagged ? 1 : 0);
    return result;
} }
```

Hashfunktionen schreiben

- Gültige, aber auch besonders schlechte Hashfunktion:
  `return 1;`
- Eine brauchbare Hashfunktion bekommt man meistens, wenn man den Hashcode aller Attribute zusammenzählt, jeweils mit einer unterschiedlichen Primzahl als Vorfaktor
- Richtig gute Hashfunktion schreiben: Ziemlich kompliziert, für einfache Anwendungen meist nicht so wichtig

Implementierung von `Set` als Hashtabelle

Eine Möglichkeit, eine Menge zu implementieren, besteht darin, ein Array `s` einer bestimmten Größe `SIZE` vorzusehen und ein Element `x` im Fach `x.hashCode() % SIZE` abzulegen.

Das geht eine Weile gut, funktioniert aber nicht, wenn wir zwei Elemente ablegen möchten, deren Hashwerte gleich modulo `SIZE` sind. In diesem Falle liegt eine **Kollision** vor.

Um Kollisionen zu begegnen kann man in jedem Fach eine verkettete Liste von Objekten vorsehen.

Für `get` und `put` muss man zunächst das Fach bestimmen und dann die dort befindliche Liste linear durchsuchen.

Sind Kollisionen selten, so bleiben diese Listen recht kurz und der Aufwand hält sich in Grenzen.

Genaue stochastische Analyse erfolgt in “Effiziente Algorithmen”.

Endliche Abbildungen

Einer Menge von **Schlüssel**-Objekten der Klasse `K` (engl. **keys**) wird jeweils genau ein **Werte**-Objekt der Klasse `V` (engl. **values**) zugeordnet.

Eine **endliche Abbildung** (**finite map**) kann als zweispaltige Tabelle aufgefasst werden, wobei keine zwei Zeilen den gleichen Schlüssel haben. z.B. in Adressbuch

key	value
Martin	9341
Sigrid	9337
Steffen	9139

Funktionalität

- Abfragen ob Schlüssel vorhanden ist
- Abfragen des Wertes eines eingetragenen Schlüssels
- Eintragen/Entfernen von Schlüssel/Wert Zuordnungen

Schlüssel bilden wieder eine endliche Menge!

Interface `Map<K,V>`

```
interface Map<K,V>
  V          get(Object key);      // may return null
  V          remove(Object key);   // may return null
  V          put(K key, V value);  // may return null
  boolean         containsKey(Object key);
  boolean         containsValue(Object value);
  Set<K>          keySet();
  Collection<V>   values();
  ...
```

- Bildet **Schlüssel**-Objekte `K` auf **Werte**-Objekte `V` ab
- Schlüssel-Objekte sollten möglichst immutable sein
- Vergleiche benutzen `equals` und oft auch `hashCode`

Beispiel:

```
Map <String, String> dict = new HashMap <>();
dict.put("map", "abbilden");
dict.put("key", "Schluessel");
dict.put("hashmap", "Streuspeicher");
System.out.println(dict.get("key")); // "Schluessel"
```

Implementierungen von Abbildungen

- Implementierung `HashMap<K,V>` durch eine Hashtabelle; speichert Paare von Schlüsseln und Werten
  Nachteile:
  - Effizienz hängt stark von `hashCode` ab; $O(n)$
    meist darf man konstanten Aufwand erwarten $O(1)$
  - Einfügen kann rehashing erfordern

- Implementierung `TreeMap<K,V>` durch schnell durchlaufbare Suchbäume
  Nachteile:
  - Suchen, Einfügen und Löschen logarithmisch $O(\log n)$
  - Einfügen oder Löschen kann Restrukturierung erfordern

Anwendungsbeispiel

```
import java.util.*;
import javafx.scene.paint.Color;

public class MapTest {
    public static void main(String[] args) {
        Map<String,Color> favoriteColors =
                          new HashMap<String,Color>();
        favoriteColors.put("Juliet", Color.PINK);
        favoriteColors.put("Romeo" , Color.GREEN);
        favoriteColors.put("Adam"  , Color.BLUE);
        favoriteColors.put("Eve"   , Color.PINK);
        print(favoriteColors);
        favoriteColors.put("Adam"  , Color.YELLOW);
        favoriteColors.remove("Romeo");
        print(favoriteColors);
    }
```

```
    private static void print(Map<String,Color> m) {
        Set<String> keySet = m.keySet();
        Iterator<String> iter = keySet.iterator();
        while (iter.hasNext()) {
            String key = iter.next();
            Color value = m.get(key);
            System.out.println(key + "->" + value);
        }
    }
}
```

Funktionsprinzip

- Diese Zuordnung wird in einem Array fester Länge `n` gespeichert (tatsächlich ist die Länge veränderlich, der Einfachheit nicht genauer erläutert, wie das funktioniert)
- Die Speicherstelle für einen Datensatz (Schlüssel `k` und Wert `v`) hängt nur vom Schlüssel ab: `hashCode(k) % n`
- An jeder Speicherstelle wird eine Liste von Einträgen gespeichert
  $\Rightarrow$ An einer Stelle können viele Werte gespeichert werden; der korrekte Eintrag in der Liste wird mittels `K.equals` gesucht

27 0			39	40 22	5	15		17

Endliche Aufzählungen

Endliche Aufzählungen (engl. **Enumeration**) bieten sich immer dann an, wenn die Menge der Optionen vor dem Kompilieren bekannt ist.

Beispiele

- Booleans: `true`, `false` kein enum in Java
- Wochentage: Montag, Dienstag, . . . , Sonntag
- Noten: "Sehr gut",. . . ,"Ungenügend"
- Spielkarten: Kreuz, Pik, Herz, Karo
- Nachrichtentypen eines Protokolls: login, logout, chat, . . .
- Optionen, z.B. Kommandozeilenparameter

Aufzählungen dürfen sich mit der Programmversion ändern.

Probleme ohne `enum`

Aufzählungen werden von Anfängern oft mit finalen `int`/`String`-Konstanten realisiert, dies hat aber *Nachteile*:

- Keine Typsicherheit: `int`-Konstante `MONTAG` kann auch dort verwendet werden, wo eine Spielkartenfarbe erwartet wird.
- Keine Bereichsprüfung: Wert 42 kann übergeben werden, wo eine Wochentag `int` Konstante erwartet wird.
- Sprechende Namen nicht erzwungen:
  “Hacks” mit direkten Zahlen können auftauchen
- Geringe Effizienz: z.B. Vergleich von `String` Konstanten;
- Keine Modularität: Gesamt-Rekompilation bei Änderungen

Seit Java 1.5: `enum` für endliche Aufzählungen möglich!

`enum` Deklarationen

Beispiel

```
public enum Kartenfarbe { KREUZ, PIK, HERZ, KARO; };
```

Definiert Typ `Kartenfarbe` mit 4 Konstanten.

mit Komma getrennt, mit Semikolon abgeschlossen

Verwendung durch `Kartenfarbe.PIK`

`Kartenfarbe` ist dann eine reguläre Java-Klasse:

- Nur jeweils eine Objekt-Instanz pro statischer Konstante, d.h. es kann gefahrlos `==` verwendet werden
- Verschiedene Enums können gleiche Konstanten haben: Verwechslung wird durch Typsystem ausgeschlossen
- Erbe von `java.lang.Enum`, Methoden automatisch erstellt

java.lang.Enum

Folegende Methoden werden über `java.lang.Enum` automatisch für jedes `enum` bereitgestellt:

`boolean equals(Object other)`	Direkt verwendbar
`int hashCode()`	Direkt verwendbar
`int compareTo(E o)`	Vergleich gemäß Definitionsreihenfolge
`String toString()`	Umwandlung zur Anzeige
`static <T extends Enum<T>> valueOf(String)`	
`String name()`	NICHT verwenden!
`int ordinal()`	NICHT verwenden!

Erlaubt optimierte Versionen `EnumMap<K extends Enum<K>,V>` und `EnumSet<E extends Enum<E>>`

anstatt Bit-Felder immer `EnumSet` verwenden!

values()

Weiterhin wird für jedes enum eine statische Methode `values()` definiert, welche ein Array aller Konstanten liefert:

```
for (Kartenfarbe f : Kartenfarbe.values()) {
  System.out.println(f);
}
```

Reihenfolge der Konstanten wie in der Deklaration des enums!

Attribute

Konstanten können mit anderen Werten assoziiert werden, welche wie Attribute der enum-Klasse behandelt werden.

- Dazu Konstruktor und getter-Methoden definieren
- Konstruktoren müssen immer `private` sein
- Auch beliebige andere Methoden sind erlaubt

```
public enum Feld {
  FOREST("Wald",2), MEADOW("Wiese",2),  MOUNTAIN("Berg",1);
  private final String typ;
  private final int ertrag;

  private Feld(String typ, int ertrag) {
    this.typ = typ;
    this.ertrag = ertrag;
  }
  public int ertrag() { return ertrag; }
}
```

Zusammenfassung `Enum`

- Optionen können bei neuen Versionen leicht hinzugefügt werden
- `enum` Deklaration generiert Klasse mit statischen Instanzen (kein öffentlicher Konstruktor)
- Konstanten sind automatisch geordnet
- Nützliche Methoden automatisch generiert, z.B. `values()`
- `EnumSet` anstatt Bit-Felder verwenden
- Werte über `EnumMap` oder Attribute assoziieren

Streams

Streams sind Folgen von Werten. Ein Stream hat drei Phasen:

1. Erzeugung
2. Mehrere Transformationen der Elemente
3. Aggregation

Stream-Berechnung kann seriell oder auch parallel erfolgen.

Beispiel:

```
List<Integer> list;
/* ...Liste wird hier erstellt... */
list.stream()                               // Erzeugung
    .map(x -> x*x)                          // Transformation
    .filter(x -> x%2 == 0)                  // Transformation
    .forEach(x ->                           // Aggregation
      System.out.print(x+","));
```

Notation

Beispiel

```
List<Integer> oddnums = Stream.iterate(1, n->n+1)
                              .filter(\x -> x%2 > 0)
                              .limit(27)
                              .collect(Collectors.toList());
```

Das man die Transformation jeweils in eine eigene Zeile schreibt, ist eine nicht-bindende Konvention.

Hinweis:
Der Punkt `.` ist der gewöhnliche bekannte Methoden-Zugriff auf ein Objekt. Die Verkettung mehrerer solcher Zugriffe funktioniert also genau wie hier:

```
List<Integer> ilist;
String s = ilist.listIterator().next().toString();
```

Notation

Beispiel

```
Stream<Integer> istr = Stream.iterate(1, n->n+1);
istr = istr.filter(\x -> x%2 > 0).limit(27);
List<Integer> ilst = istr.collect(Collectors.toList());
```

Das man die Transformation jeweils in eine eigene Zeile schreibt, ist eine nicht-bindende Konvention.

Hinweis:
Der Punkt `.` ist der gewöhnliche bekannte Methoden-Zugriff auf ein Objekt. Die Verkettung mehrerer solcher Zugriffe funktioniert also genau wie hier:

```
List<Integer> ilist;
String s = ilist.listIterator()
               .next()
               .toString();
```

Beispiele: Stream Erzeugung

Aus Collections: Mit den `Collections<E>`-Methoden

```
Stream<E> stream();          und
Stream<E> parallelStream();
```

Mit Generatoren:

- Typ-spezifische Generatoren
  ```
  IntStream.rangeClosed(11,33)
  ```
- Unendliche Iteration
  ```
  Stream.iterate(0, n -> n + 10)
  ```
- Auflistung aller Werte
  ```
  Stream.of(2,4,42,69,111)
  ```

Aus Dateien:

```
Files.lines(Paths.get("file.txt),
            Charset.defaultCharset())
```

Beispiele: Stream Transformationen

- `Stream<T> filter(Predicate<? super T> predicate)`
  Filtert Elemente aus dem Strom heraus
- `Stream<T> skip(long n)`
  Entfernt feste Anzahl Elemente aus dem Strom
- `Stream<T> distinct()`
  Entfernt doppelte Elemente aus dem Strom
- `Stream<T> sorted(Comparator<? super T> comparator)`
  Sortiert alle Elemente des Stroms
- `Stream<R> map(Function<? super T,? extends R> mapper)`
  Anwendung einer Funktion auf einzelne Elemente
- `IntStream mapToInt(ToIntFunction<? super T> mapper)`
  Konvertierung der Elemente nach `Integer`

Transformationen betrachten jedes Element für sich; es sollten nicht über Seiteneffekte mehrere Elemente simultan beeinflusst werden!

Beispiele: Stream Aggregation

- `T reduce(T identity, BinaryOperator<T> accumulator)`
  Elemente werden durch binären Operator verknüpft
  z.B. aufaddieren, aufmultiplizieren, etc.
- `sum()`, `average()`, `max()`, `min()`, ...
  Vordefinierte Reduktionen
- `count()`
  Anzahl der verbleibenden Strom-Elemente zählen
- `void forEach(Consumer<? super T> action)`
  Einen Befehl für jedes Element ausführen
- `R collect(Collector<? super T,A,R> collector)`
  Wieder in einer `Collection` aufsammeln

Fazit Streams:

- erlauben kurze Beschreibung komplexer Operationen
- einfache Einführung paralleler Verarbeitung