UNIVERSITÄT
LEIPZIG

Einführung in die Objekt-Orientierte Modellierung und Programmierung

Wintersemester 2025/2026

Dirk Zeckzer

Institut für Informatik

Teil XVIII

Algorithmen und Datenstrukturen – Java Collections

Klassen und Arrays

Klassen

- Heterogene Sammlung von Elementen **unterschiedlichen** Typs (Attribute)
- Methoden zum
  - Anlegen einer Instanz:

    ```
    Kreis kreis = new Kreis(5.0);
    ```
  - Gegebenenfalls
    - Ändern
    - Abfragen

    der Attribute

Arrays

- Homogene Sammlung von Elementen **gleichen** Typs
- Methoden zum
  - Anlegen eines Arrays:

    ```
    int[] a = new int[5];
    ```
  - Ändern der Werte:

    ```
    a[3] = 42;
    ```
  - Lesen der Werte:

    ```
    int i = a[2];
    ```
  - Bestimmen der Anzahl der Werte des Arrays:

    ```
    int numberOfElements = a.length;
    ```

Arrays und Klassen

Klassen

- Heterogene Sammlung von Elementen **unterschiedlichen** Typs (Attribute)
- Vorteil
  - Sehr mächtig zur Strukturierung
  - Kombination
    - Datenstrukturen (Attribute)
    - Algorithmen (Methoden)

Arrays

- Homogene Sammlung von Elementen **gleichen** Typs
- Vorteile
  - effizient
  - einfach
- Nachteil: unflexibel
  - Größe
    - muss beim Anlegen feststehen
    - kann nicht verändert werden
  - Einfügen und Löschen nicht möglich
    - am Anfang
    - zwischen zwei Elementen

Arrays und Klassen

Klassen

- Heterogene Sammlung von Elementen **unterschiedlichen** Typs (Attribute)
- Vorteil
  - Sehr mächtig zur Strukturierung
  - Kombination
    - Datenstrukturen (Attribute)
    - Algorithmen (Methoden)

Arrays

- Homogene Sammlung von Elementen **gleichen** Typs
- Vorteile
  - effizient
  - einfach
- Nachteil: unflexibel
- **Gesucht**:
  Alternativen zur Datenstruktur Array

Alternative Datenstrukturen

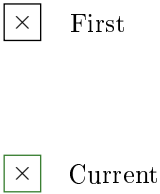
- Aufgabe
  - Einlesen homogener Elemente aus einer Datei
  - Beispiel: Jede Zeile enthält eine ganze Zahl
- Eigenschaften
  - Anzahl der Elemente nicht bekannt
- Lösungsansätze Array-basiert
  - Speichere Anzahl der Elemente in der ersten Zeile der Datei
  - Lies die Datei und ermittle die Anzahl der Zeilen der Datei
- Alternative:
  - Wähle andere Datenstruktur

Alternative Datenstrukturen

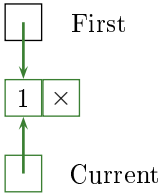
- Aufgabe
  - Einlesen homogener Elemente aus einer Datei
  - Beispiel: Jede Zeile enthält eine ganze Zahl
- Eigenschaften
  - Anzahl der Elemente nicht bekannt
- Anforderungen
  - Zu Beginn ist die Datenstruktur leer
  - Der Inhalt jeder Zeile wird an das Ende angehängt
- Allgemeine Anforderungen
  - Ist die Datenstruktur leer?
  - Anzahl der Elemente der Datenstruktur
  - Möglichkeit, die Elemente der Datenstruktur aufzuzählen
  - Datenstruktur leeren

Datenstruktur: konzeptionell

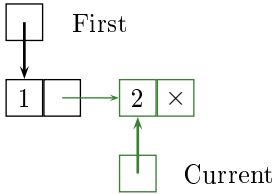
Leer



Einfügen des ersten Elements

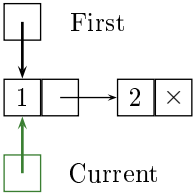


Einfügen eines weiteren Elements am Ende

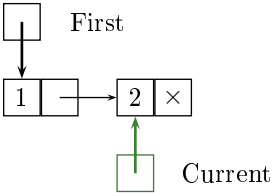


Datenstruktur: konzeptionell

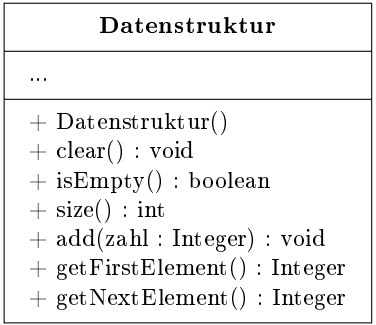
Lese erstes Element aus der Liste



Lese nächstes Element aus der Liste

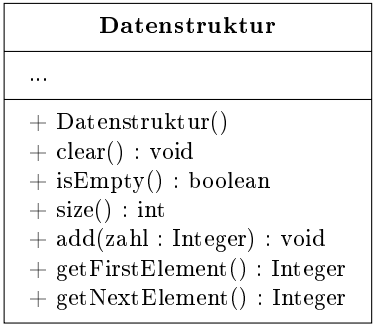


Datenstruktur



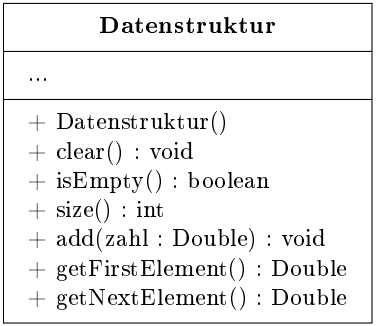

```java
package eoomp;

public class Datenstruktur {

  // Attribute

  /** Konstruktor
   */
  public Datenstruktur(
    // Parameter
  ) {
    // Initialisiere Datenstruktur
  }

  public void clear() {
    // Leeren der Datenstruktur
  }

  public boolean isEmpty() {
    // true <-> Datenstruktur ist leer
  }

  public int size() {
    // return: Anzahl der Elemente
  }

  ...
}
```

Datenstruktur



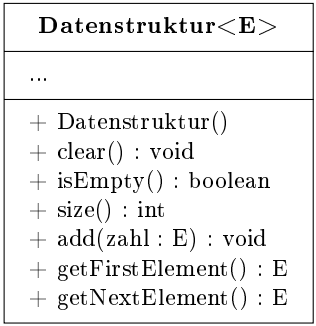

```java
package eoomp;

public class Datenstruktur {

  ...

  public void add(
    final Integer zahl
  ) {
    // Fuege Zahl am Ende der Datenstruktur hinzu
  }

  public Integer getFirstElement() {
    // Gehe zum Anfang der Datenstruktur
    // Gib erstes Element zurueck
  }

  public Integer getNextElement() {
    // Gehe zum naechsten Element der Datenstruktur
    // Gib naechstes Element zurueck
  }
}
```

Datenstruktur




```java
package eoomp;

public class Datenstruktur {

  ...

  public void add(
    final Double zahl
  ) {
    // Fuege Zahl am Ende der Datenstruktur hinzu
  }

  public Double getFirstElement() {
    // Gehe zum Anfang der Datenstruktur
    // Gib erstes Element zurueck
  }

  public Double getNextElement() {
    // Gehe zum naechsten Element der Datenstruktur
    // Gib naechstes Element zurueck
  }
}
```

Datenstruktur: Generics




```java
package eoomp;

public class Datenstruktur<E> {

  ...

  public void add(
    final E element
  ) {
    // Fuege Element am Ende der Datenstruktur hinzu
  }

  public E getFirstElement() {
    // Gehe zum Anfang der Datenstruktur
    // Gib erstes Element zurueck
  }

  public E getNextElement() {
    // Gehe zum naechsten Element der Datenstruktur
    // Gib naechstes Element zurueck
  }
}
```

Datenstruktur: Generics

Verwendung:

```
1     Datenstruktur<Kreis> kreise
2       = new Datenstruktur<>();
3     for (int i = 1;
4          i <= 10;
5          ++i) {
6       Kreis kreis
7         = new Kreis("Kreis " + i,
8                     i);
9       kreise.add(kreis);
10    }
```

```
1   package eoomp;
2
3   public class Datenstruktur<E> {
4
5     ...
6
7     public void add(
8       final E element
9     ) {
10      // Fuege Element am Ende der Datenstruktur hinzu
11    }
12
13    public E getFirstElement() {
14      // Gehe zum Anfang der Datenstruktur
15      // Gib erstes Element zurueck
16    }
17
18    public E getNextElement() {
19      // Gehe zum naechsten Element der Datenstruktur
20      // Gib naechstes Element zurueck
21    }
22  }
```

Collections

Collections

- Algorithmen und Datenstrukturen
  - zur effizienten und effektiven Verwaltung
  - von homogenen Elementen
  - gleichen Typs
- Basis: Klassen
  - Datenstrukturen $\rightarrow$ Attribute
  - Algorithmen $\rightarrow$ Methoden
- Package: `java.util`

In anderen Programmierparadigmen

- Datenstrukturen und Algorithmen können voneinander getrennt sein
- Kombination nur konzeptionell: keine explizite Kapselung des Zugriffs der Algorithmen auf die Datenstrukturen

Collections: Allgemeine Funktionalität

Allgemeine Funktionalität in Sammlungen von Elementen gleichen Typs T

`java.util.Collection<E>` (interface)

`void clear()`	Lösche alle Elemente aus der Collection
`boolean isEmpty()`	*true* gdw. Collection ist leer
`int size()`	Anzahl der Elemente der Collection
`boolean add(E o)`	*true* gdw. die Collection wurde verändert
`boolean remove(Object o)`	*true* gdw. ein Element wurde aus der Collection entfernt

Collections: Allgemeine Funktionalität

Allgemeine Funktionalität in Sammlungen von Elementen gleichen Typs T

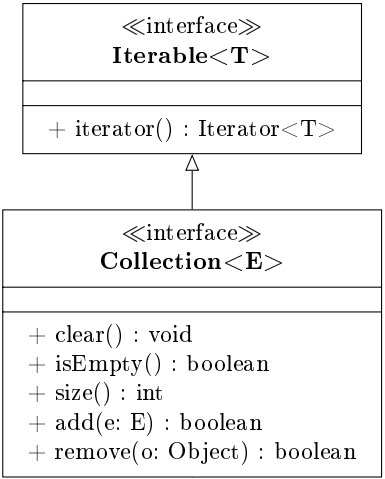
`java.lang.Iterable<T>` (interface)

`Iterator<T> iterator()` erzeugt Iterator

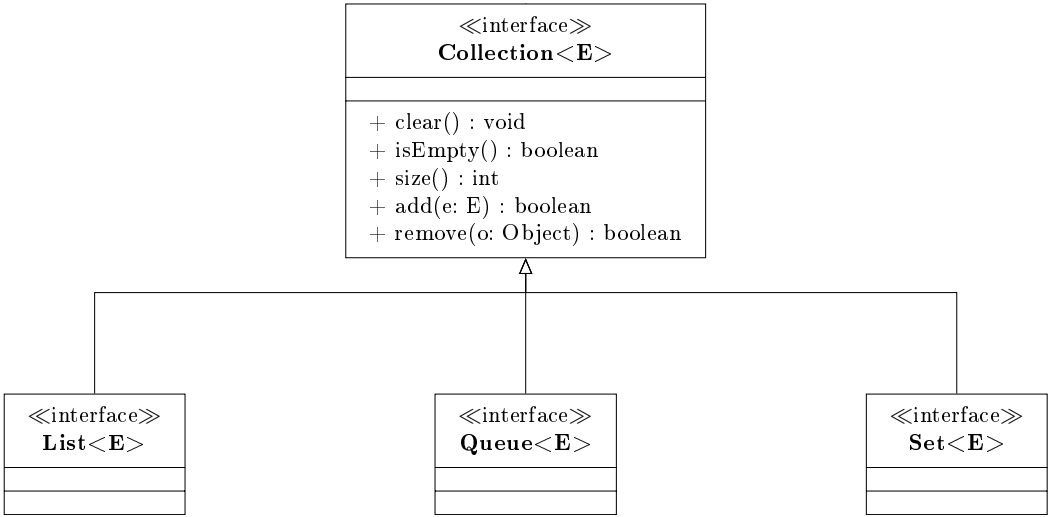
`java.util.Iterator<E>` (interface)

`boolean hasNext()`	gibt es ein weiteres Element?
`E next()`	gibt das nächste Element zurück, gibt `null` zurück, falls `hasNext` returns `false`

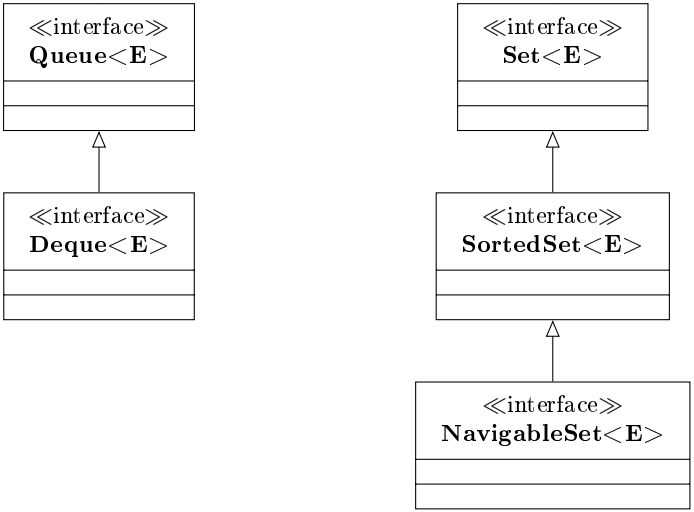
Collections: Allgemeine Funktionalität



Collections: Allgemeine Funktionalität



Collections: Allgemeine Funktionalität



Collections: Interface $\leftrightarrow$ Klasse

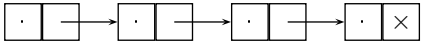
Interface wird als **Datentyp** verwendet

- Typ eines Attributes
- Typ einer Variablen
- Typ eines Parameters
- Typ des Rückgabewertes

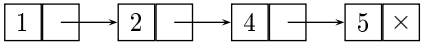
Klasse (Implementierung des Interfaces) wird zur **Instantiierung** verwendet

Listen: konzeptionell

Abstrakte Liste:

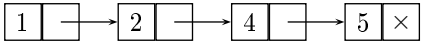


Liste mit Zahlen:

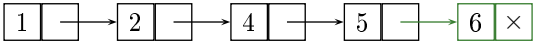


Listen: konzeptionell

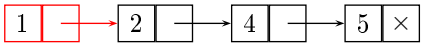
Liste mit Zahlen:



Anfügen am Ende:

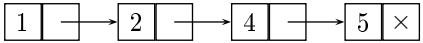


Erstes Element entfernen:

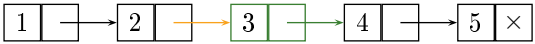


Listen: konzeptionell

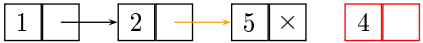
Liste mit Zahlen:



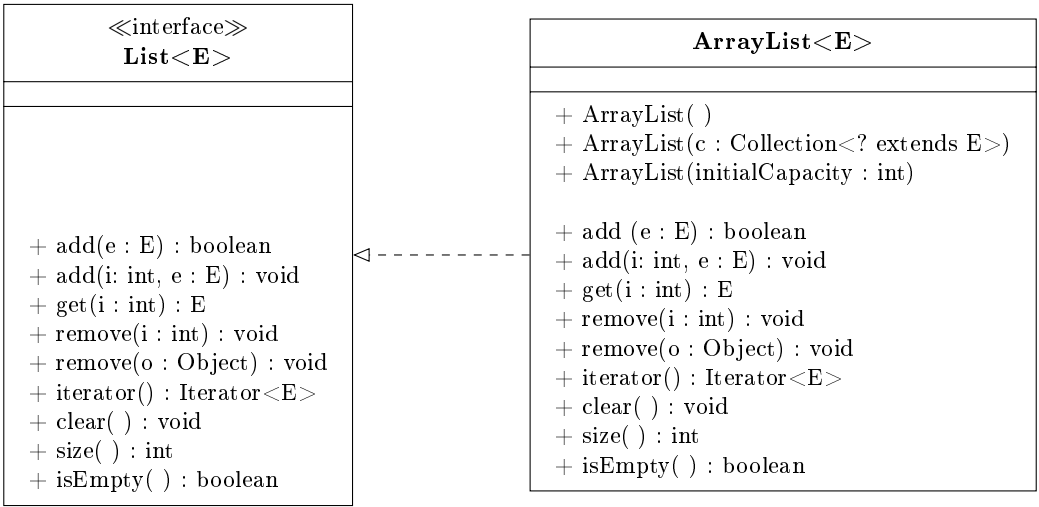
Einfügen in der Mitte (oder am Anfang):



Löschen in der Mitte (oder am Ende):



Listen: List – ArrayList



Listen Implementierung: ArrayList

java.util.ArrayList als Implementierung einer Liste

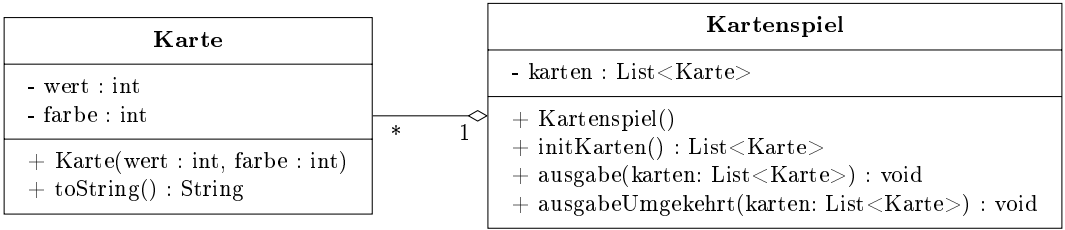
Eigenschaften:

- wahlfreier Zugriff (wie Array)
- wächst dynamisch (Liste)
- Verwendung: Erzeugung von Instanzen vom Typ `List`

Eigenschaften

- Größe
  - muss beim Anlegen **nicht feststehen**
  - kann **verändert** werden
- **Einfügen** und **Löschen** ist einfach
  - am **Ende**
  - **zwischen zwei Elementen**
  - am **Anfang**

Listen: Beispiel Kartenspiel



Listen: Beispiel Kartenspiel

```java
import java.util.List;
import java.util.ArrayList;

public class Kartenspiel {

  private final List<Karte> karten;

  public Kartenspiel() {
    // Initialisiere Liste von Karten
    karten = initKarten();

    // Ausgabe der Karten:
    // Reihenfolge wie in der Liste
    ausgabe(karten);

    // Ausgabe der Karten:
    // Reihenfolge umgekehrt wie in der
     Liste
    ausgabeUmgekehrt(karten);
  }

  ...
}
```

```java
public List<Karte> initKarten(
) {
  // Erzeuge Liste von Karten
  final List<Karte> karten =
    new ArrayList<Karte>();

  for (int farbe = 0;
       farbe <= 3;
       ++farbe) {
    for (int wert = 2;
         wert <= 14;
         ++wert) {
      karten.add(new Karte(wert, farbe));
    }
  }
  return karten;
}
```

Listen: Beispiel Kartenspiel

```java
import java.util.List;
import java.util.ArrayList;

public class Kartenspiel {

  private final List<Karte> karten;

  public Kartenspiel() {
    // Initialisiere Liste von Karten
    karten = initKarten();

    // Ausgabe der Karten:
    // Reihenfolge wie in der Liste
    ausgabe(karten);

    // Ausgabe der Karten:
    // Reihenfolge umgekehrt wie in der
     Liste
    ausgabeUmgekehrt(karten);
  }

  ...
}
```

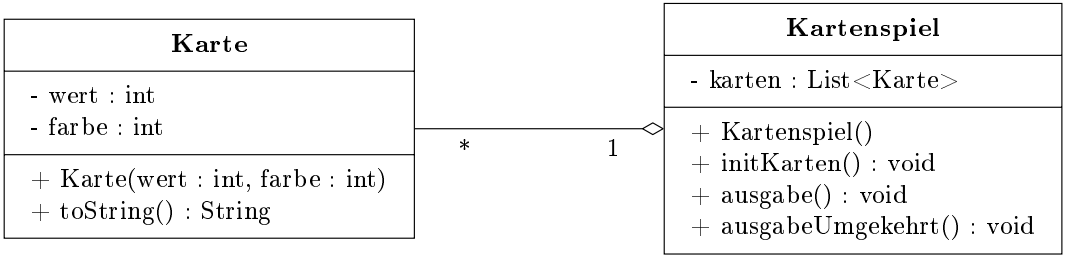
```java
public void ausgabe(
  final List<Karte> karten
) {
  if (karten == null) {
    return;
  }

  for (Karte karte : karten) {
    System.out.println(karte.toString());
  }
}
```

Listen: Beispiel Kartenspiel

```java
import java.util.List;
import java.util.ArrayList;

public class Kartenspiel {

  private final List<Karte> karten;

  public Kartenspiel() {
    // Initialisiere Liste von Karten
    karten = initKarten();

    // Ausgabe der Karten:
    // Reihenfolge wie in der Liste
    ausgabe(karten);

    // Ausgabe der Karten:
    // Reihenfolge umgekehrt wie in der
     Liste
    ausgabeUmgekehrt(karten);
  }

  ...
}
```

```java
public void ausgabeUmgekehrt(
  final List<Karte> karten
) {
  if (karten == null) {
    return;
  }

  for (int z = karten.size()-1;
       z >= 0;
       --z) {
    System.out.println(
      karten.get(z).toString()
    );
  }
}
```

Listen: Beispiel Kartenspiel



Listen: Beispiel Kartenspiel

```
1   public class Kartenspiel {
2
3     private List<Karte> karten;
4
5     public Kartenspiel() {
6       // Initialisiere Liste von Karten
7       initKarten();
8
9       // Ausgabe der Karten:
10      // Reihenfolge wie in der Liste
11      ausgabe();
12
13      // Ausgabe der Karten:
14      // Reihenfolge umgekehrt wie in der
          Liste
15      ausgabeUmgekehrt();
16    }
17
18    ...
19  }
```

```
1   public void initKarten() {
2     // Erzeuge Liste von Karten
3     karten = new ArrayList<Karte>();
4
5     for (int farbe = 0;
6          farbe <= 3;
7          ++farbe) {
8       for (int wert = 2;
9            wert <= 14;
10           ++wert) {
11        karten.add(new Karte(wert, farbe));
12      }
13    }
14  }
```

Listen: Beispiel Kartenspiel

```java
public class Kartenspiel {

  private List<Karte> karten;

  public Kartenspiel() {
    // Initialisiere Liste von Karten
    initKarten();

    // Ausgabe der Karten:
    // Reihenfolge wie in der Liste
    ausgabe();

    // Ausgabe der Karten:
    // Reihenfolge umgekehrt wie in der
     Liste
    ausgabeUmgekehrt();
  }

  ...
}
```

```java
public void ausgabe() {
  if (karten == null) {
    return;
  }

  for (Karte karte : karten) {
    System.out.println(karte.toString());
  }
}
```

```java
public void ausgabeIterator() {
  if (karten == null) {
    return;
  }

  for (Iterator<Karte> karteIterator
         = karten.iterator();
       karteIterator.hasNext();
      ) {
    Karte karte = karteIterator.next();
    System.out.println(karte.toString());
  }
}
```

Listen: Beispiel Kartenspiel

```java
public class Kartenspiel {

  private List<Karte> karten;

  public Kartenspiel() {
    // Initialisiere Liste von Karten
    initKarten();

    // Ausgabe der Karten:
    // Reihenfolge wie in der Liste
    ausgabe();

    // Ausgabe der Karten:
    // Reihenfolge umgekehrt wie in der
     Liste
    ausgabeUmgekehrt();
  }

  ...
}
```

```java
public void ausgabeUmgekehrt() {
  if (karten == null) {
    return;
  }

  for (int z = karten.size()-1;
       z >= 0;
       --z) {
    System.out.println(
      karten.get(z).toString()
    );
  }
}
```

Mengen: Gemeinsamkeiten

Datenstruktur **Menge** (`java.util.Set`):

Datenstruktur **sortierte Menge** (`java.util.SortedSet`):

- Sammlung von Elementen gleichen Typs
- Keine Duplikate
- `boolean contains(Object o)`
  `true` gdw. die Menge das Objekt enthält
- `boolean add(E e)`
  fügt `e` hinzu
  `false` gdw. die Menge hat das Objekt vor dem Hinzufügen enthalten

Mengen: Unterschiede

Datenstruktur **Menge** (`java.util.Set`):

- Keine Ordnung
- Keine Teilmengen

Datenstruktur **sortierte Menge** (`java.util.SortedSet`):

- Ordnung (Sortierung)
  - Kleinstes Element (`first()`)
  - Größtes Element (`last()`)
- Teilmenge (`subset(E from, E to)`); Beispiel:
  - Eingabe: $M = \{1,2,3,4,5\}$ (kein Java-Code)
  - Aufruf: `S = M.subset(2, 4);`
  - Ergebnis: $S = \{2,3\}$ (kein Java-Code)

Mengen Eigenschaften

Set

- Größe
  - muss beim Anlegen **nicht feststehen**
  - kann **verändert** werden
- Einfach
  - **Einfügen** von Elementen
  - **Test**, ob ein Element in der Menge enthalten ist

SortedSet

- Größe
  - muss beim Anlegen **nicht feststehen**
  - kann **verändert** werden
- Einfach
  - **Einfügen** von Elementen
  - **Test**, ob ein Element in der Menge enthalten ist
  - **Sortierung** (implizit)
  - **Erstes und letztes Element** bezüglich der Sortierung (implizit)
  - **Bildung von Teilmengen**

Mengen Implementierung: HashSet, TreeSet

java.util.HashSet
als Implementierung von

- Set

java.util.TreeSet
als Implementierung von

- Set
- SortedSet

Abbildungen: Gemeinsamkeiten

Datenstruktur **Abbildung** (`java.util.Map`):

Datenstruktur **sortierte Abbildung** (`java.util.SortedMap`):

- Schlüssel – Werte Paare:
  jedem Schlüssel wird eindeutig ein Wert zugeordnet
  `java.util.Map.Entry<K,V>`
  - `K getKey()`
  - `V getValue()`
- Keine Duplikate der Schlüssel
- Duplikate der Werte möglich

Abbildungen: Gemeinsamkeiten

Methode	Bedeutung
`V put(K key, V value)`	ein (key, value) Paar hinzufügen wenn schon ein Wert zum Schlüssel `key` in der Abbildung war, dann wird er zurückgegeben, und der neue Wert `value` hinzugefügt
`V get(K key)`	gibt den Wert `value` zum Schlüssel `key` zurück wenn es keinen Wert zum Schlüssel gibt, dann wird `null` zurückgegeben

Abbildungen: Gemeinsamkeiten

Methode	Bedeutung
`boolean containsKey(Object key)`	`return true`; gdw. die Abbildung enthält den angegebenen Schlüssel `key`
`boolean containsValue(Object value)`	`return true`; gdw. die Abbildung enthält den angegebenen Wert `value`

Abbildungen: Gemeinsamkeiten

Methode	Bedeutung
`Set<Map.Entry<K,V>> entrySet()`	Eine Menge mit den Einträgen
`Set<K> keySet()`	Eine Menge mit den Schlüsseln
`Collection<V> values()`	Eine `Collection<V>` mit den Werten

Abbildungen: Unterschiede

Datenstruktur **Abbildung** (`java.util.Map`):

- Keine Ordnung
- Keine Teilmengen der Abbildung

Datenstruktur **sortierte Abbildung** (`java.util.SortedMap`):

- Ordnung (Sortierung) auf den **Schlüsseln**
  - Kleinstes Element (`firstKey()`)
  - Größtes Element (`lastKey()`)
- Teilmenge (`subMap(K from, K to)`)

Abbildungen Eigenschaften

Map

- Größe
  - muss beim Anlegen **nicht feststehen**
  - kann **verändert** werden
- Einfach
  - **Einfügen** von Elementen
  - **Test**, ob ein Schlüssel in der Abbildung enthalten ist

SortedMap

- Größe
  - muss beim Anlegen **nicht feststehen**
  - kann **verändert** werden
- Einfach
  - **Einfügen** von Elementen
  - **Test**, ob ein Schlüssel in der Abbildung enthalten ist
  - **Sortierung nach Schlüsseln** (implizit)
  - **Erster und letzter Schlüssel** bezüglich der Sortierung (implizit)
  - **Bildung von Teilmengen**

Abbildungen Implementierung: HashMap, TreeMap

java.util.HashMap
als Implementierung von

- Map

java.util.TreeMap
als Implementierung von

- Map
- SortedMap

Abbildungen: Iteration über eine Abbildung

```java
public class ZiffernWorteMap {

  // Abbildung von Ziffern auf ihre Worte
  private final Map<Integer, String> ziffernWorte;

  public ZiffernWorteMap() {
    ziffernWorte = createMap();
  }

  private Map<Integer, String> createMap() {
    final Map<Integer, String> ziffernWorte = new HashMap<>();

    ziffernWorte.put(0, "Null");
    ziffernWorte.put(1, "Eins");
    ziffernWorte.put(2, "Zwei");
    ziffernWorte.put(3, "Drei");
    ziffernWorte.put(4, "Vier");
    ziffernWorte.put(5, "Fuenf");
    ziffernWorte.put(6, "Sechs");
    ziffernWorte.put(7, "Sieben");
    ziffernWorte.put(8, "Acht");
    ziffernWorte.put(9, "Neun");

    return ziffernWorte;
  }

  ...
}
```

Abbildungen: Iteration über eine Abbildung

```
1     public void gebeSchluesselAus() {
2       for (Integer schluessel : ziffernWorte.keySet()) {
3         System.out.println(schluessel);
4       }
5     }
```

```
1     public void gebeWertAus() {
2       for (String wert : ziffernWorte.values()) {
3         System.out.println(wert);
4       }
5     }
```

Abbildungen: Iteration über eine Abbildung

```
1     public void gebeSchluesselWertAus() {
2       for (Map.Entry<Integer, String> ziffernWort : ziffernWorte.entrySet()) {
3         System.out.println(ziffernWort.getKey()
4                            + " -> "
5                            + ziffernWort.getValue());
6       }
7     }
```

```
1     public void gebeSchluesselWertAus2() {
2       for (Integer schluessel : ziffernWorte.keySet()) {
3         System.out.println(schluessel
4                            + " -> "
5                            + ziffernWorte.get(schluessel));
6       }
7     }
```

HashSet, HashMap ↔ TreeSet, TreeMap

`HashSet`, `HashMap`

- Einfügen in konstanter Zeit
- Löschen in konstanter Zeit
- `contains`, `get` in konstanter Zeit
- Aufzählung aller Elemente in linearer Zeit
- Keine Sortierung
- Verschlechterung der Laufzeit zu linear, falls Hash-Funktion Kollisionen liefert

`TreeSet`, `TreeMap`

- Einfügen in logarithmischer Zeit
- Löschen in logarithmischer Zeit
- `contains`, `get` in logarithmischer Zeit
- Aufzählung aller Elemente in linearer Zeit
- Sortierung

Hash-Funktion und `HashSet<Character>`

Hashfunktion:

$$h: \begin{array}{ccc} V & \to & \mathbb{N} \\ v & \mapsto & h(v) \end{array}$$

- Beispiel:
  - $V = \{A, B, ..., Z\}$
  - $h_1(A) = 1,$
    $h_1(B) = 2,$
    $\ldots,$
    $h_1(Y) = 25,$
    $h_1(Z) = 26$
- h_1 ist perfekt:
  keine Kollisionen

- Beispiel:
  - $V = \{A, B, ..., Z\}$
  - $h_2(A) = 1, h_2(B) = 2, h_2(C) = 3, h_2(D) = 4$
    $h_2(E) = 1, h_2(F) = 2, h_2(G) = 3, h_2(H) = 4$
    $\ldots$
    $h_2(U) = 1, h_2(V) = 2, h_2(W) = 3, h_2(X) = 4$
    $h_2(Y) = 1, h_2(Z) = 2$
- h_2 ist nicht perfekt:
  viele Kollisionen

Hash-Funktion und HashSet<Character>

Beispiel:

- `HashSet<Character>`
- $V = \{A, B, ..., Z\}$
- $h_1(A) = 1,$
  $h_1(B) = 2,$
  $\ldots,$
  $h_1(Z) = 26$

1				14			
2				15			
3				16			
4				17			
5				18			
6				19			
7				20			
8				21			
9				22			
10				23			
11				24			
12				25			
13				26			

Hash-Funktion und `HashSet<Character>`

Füge 'G' ein:

Beispiel:

- `HashSet<Character>`
- $V = \{A, B, ..., Z\}$
- $h_1(A) = 1$,
  $h_1(B) = 2$,
  $\ldots$,
  $h_1(Z) = 26$

1			14		
2			15		
3			16		
4			17		
5			18		
6			19		
7	G		20		
8			21		
9			22		
10			23		
11			24		
12			25		
13			26		

Hash-Funktion und `HashSet<Character>`

Füge ‘G’, ‘Y’ ein:

Beispiel:

- `HashSet<Character>`
- $V = \{A, B, ..., Z\}$
- $h_1(A) = 1$,
  $h_1(B) = 2$,
  $\ldots$,
  $h_1(Z) = 26$

1	
2	
3	
4	
5	
6	
7	G
8	
9	
10	
11	
12	
13	

14	
15	
16	
17	
18	
19	
20	
21	
22	
23	
24	
25	Y
26	

Hash-Funktion und `HashSet<Character>`

Füge ‘G’, ‘Y’, ‘C’ ein:

Beispiel:

- `HashSet<Character>`
- $V = \{A, B, ..., Z\}$
- $h_1(A) = 1$,
  $h_1(B) = 2$,
  $\ldots$,
  $h_1(Z) = 26$

1			14		
2			15		
3	C		16		
4			17		
5			18		
6			19		
7	G		20		
8			21		
9			22		
10			23		
11			24		
12			25	Y	
13			26		

Hash-Funktion und `HashSet<Character>`

Füge ‘G’, ‘Y’, ‘C’, ‘W’ ein:

Beispiel:

- `HashSet<Character>`
- $V = \{A, B, ..., Z\}$
- $h_1(A) = 1$,
  $h_1(B) = 2$,
  $\ldots$,
  $h_1(Z) = 26$

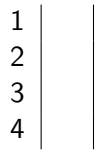
1	
2	
3	C
4	
5	
6	
7	G
8	
9	
10	
11	
12	
13	

14	
15	
16	
17	
18	
19	
20	
21	
22	
23	W
24	
25	Y
26	

Hash-Funktion und `HashSet<Character>`

Beispiel:

- `HashSet<Character>`
- $V = \{A, B, ..., Z\}$
- $h_2(A) = 1, h_2(B) = 2, h_2(C) = 3, h_2(D) = 4$
  $h_2(E) = 1, h_2(F) = 2, h_2(G) = 3, h_2(H) = 4$
  ...
  $h_2(U) = 1, h_2(V) = 2, h_2(W) = 3, h_2(X) = 4$
  $h_2(Y) = 1, h_2(Z) = 2$



Hash-Funktion und `HashSet<Character>`

Füge 'G' ein:

Beispiel:

- `HashSet<Character>`
- $V = \{A, B, ..., Z\}$
- $h_2(A) = 1, h_2(B) = 2, h_2(C) = 3, h_2(D) = 4$
  $h_2(E) = 1, h_2(F) = 2, h_2(G) = 3, h_2(H) = 4$
  ...
  $h_2(U) = 1, h_2(V) = 2, h_2(W) = 3, h_2(X) = 4$
  $h_2(Y) = 1, h_2(Z) = 2$

1	
2	
3	G
4	

Hash-Funktion und `HashSet<Character>`

Füge ‘G’, ‘Y’ ein:

Beispiel:

- `HashSet<Character>`
- $V = \{A, B, ..., Z\}$
- $h_2(A) = 1, h_2(B) = 2, h_2(C) = 3, h_2(D) = 4$
  $h_2(E) = 1, h_2(F) = 2, h_2(G) = 3, h_2(H) = 4$
  ...
  $h_2(U) = 1, h_2(V) = 2, h_2(W) = 3, h_2(X) = 4$
  $h_2(Y) = 1, h_2(Z) = 2$

1	Y
2	
3	G
4	

Hash-Funktion und `HashSet<Character>`

Füge ‘G’, ‘Y’, ‘C’ ein:

Beispiel:

- `HashSet<Character>`
- $V = \{A, B, ..., Z\}$
- $h_2(A) = 1, h_2(B) = 2, h_2(C) = 3, h_2(D) = 4$
  $h_2(E) = 1, h_2(F) = 2, h_2(G) = 3, h_2(H) = 4$
  ...
  $h_2(U) = 1, h_2(V) = 2, h_2(W) = 3, h_2(X) = 4$
  $h_2(Y) = 1, h_2(Z) = 2$

1	Y	
2		
3	G	l_3
4		

Ausweich-Liste
$l_3 = (C)$

Hash-Funktion und `HashSet<Character>`

Füge ‘G’, ‘Y’, ‘C’, ‘W’ ein:

Beispiel:

- `HashSet<Character>`
- $V = \{A, B, ..., Z\}$
- $h_2(A) = 1, h_2(B) = 2, h_2(C) = 3, h_2(D) = 4$
  $h_2(E) = 1, h_2(F) = 2, h_2(G) = 3, h_2(H) = 4$
  ...
  $h_2(U) = 1, h_2(V) = 2, h_2(W) = 3, h_2(X) = 4$
  $h_2(Y) = 1, h_2(Z) = 2$

1	Y	
2		
3	G	l_3
4		

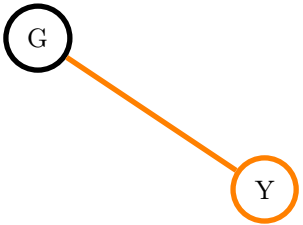
Ausweich-Liste
$l_3 = (C, W)$

`TreeSet<Character>`

Füge ‘G’ ein:

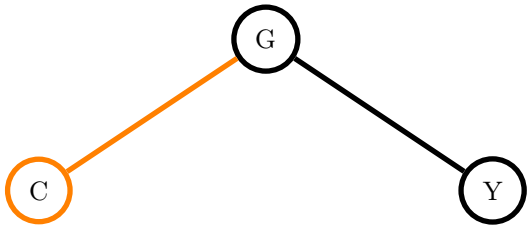
`TreeSet<Character>`

Füge ‘G’, ‘Y’ ein:



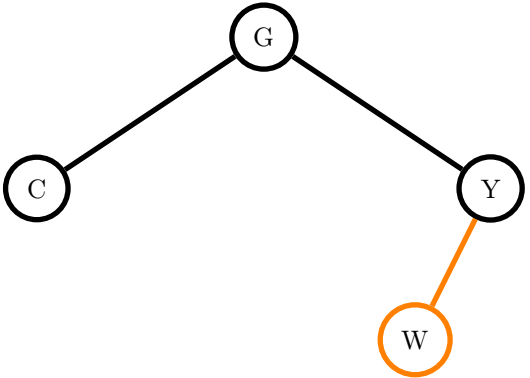
`TreeSet<Character>`

Füge ‘G’, ‘Y’, ‘C’ ein:



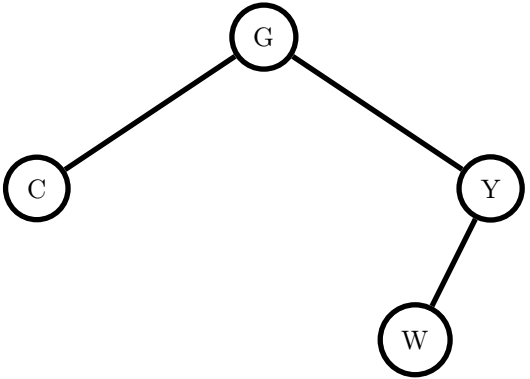
`TreeSet<Character>`

Füge ‘G’, ‘Y’, ‘C’, ‘W’ ein:



`TreeSet<Character>`

Suchbaum mit ‘G’, ‘Y’, ‘C’, ‘W’:

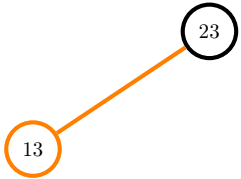


`TreeSet<Integer>`

Füge ‘23’ ein:

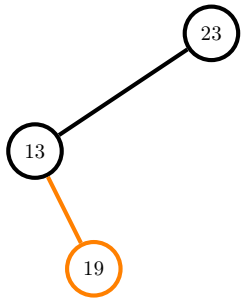
`TreeSet<Integer>`

Füge ‘23’, ‘13’ ein:



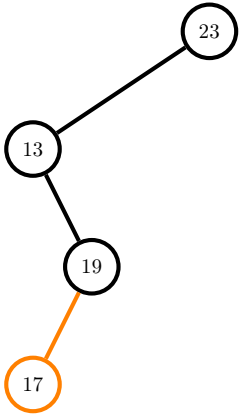
`TreeSet<Integer>`

Füge ‘23’, ‘13’, ‘19’ ein:



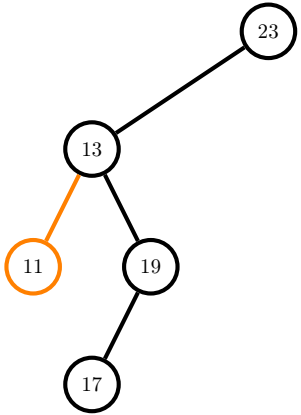
`TreeSet<Integer>`

Füge ‘23’, ‘13’, ‘19’, ‘17’ ein:



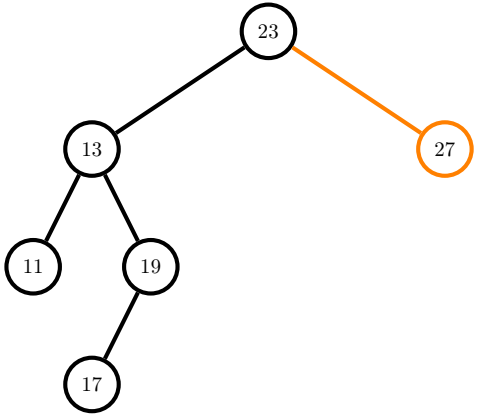
`TreeSet<Integer>`

Füge ‘23’, ‘13’, ‘19’, ‘17’, ‘11’ ein:



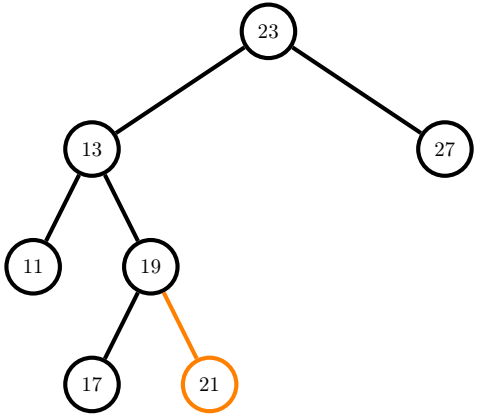
`TreeSet<Integer>`

Füge ‘23’, ‘13’, ‘19’, ‘17’, ‘11’, ‘27’ ein:



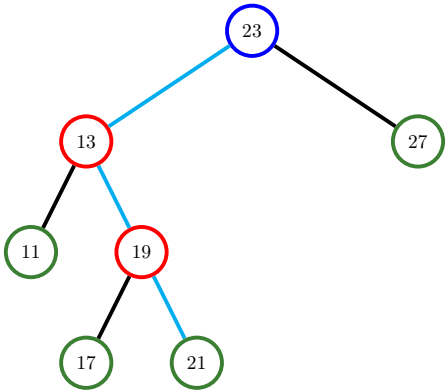
`TreeSet<Integer>`

Füge ‘23’, ‘13’, ‘19’, ‘17’, ‘11’, ‘27’, ‘21’ ein:



`TreeSet<Integer>`

Suchbaum mit ‘23’, ‘13’, ‘19’, ‘17’, ‘11’, ‘27’, ‘21’:



Farbkodierung

- Knoten
  - blau: Wurzel
  - rot: innere Knoten
  - grün: Blätter
- Kanten
  - cyan: Pfad von der Wurzel zum Blatt ‘21’