

# Vorlesung Programmieren

## Thema 13:

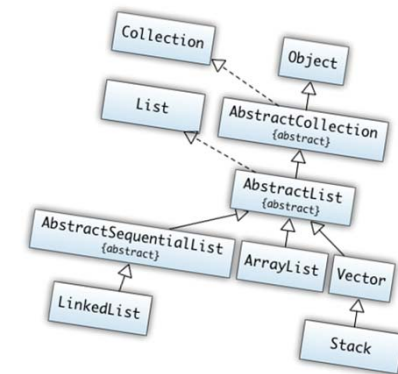
### Generische Programmierung und Collections

Olaf Herden

Fakultät Technik  
Studiengang Informatik

**HashSet**<String>

**ArrayList**<Integer>



Stand: 05/2024

- Einführung
- Mengen
- Listen
- Wildcards
- Selbst entwickelte generische Klassen
- Wrapper

- Ziel: Bestimmte Anzahl von Objekten eines bestimmten Typs verwalten
- Collection-/Container-Klasse: Nimmt Objekte gleichen Typs auf
- Beispiele:
  - Menge von Mitarbeitern
  - Multimenge von 2D-Objekten
  - Sortierte Menge von ganzen, positiven Zahlen
- Bisher: Felder
- Ermöglichen Verwaltung von Elementen gleichen Typs, z.B. `Mitarbeiter[ ]`  
`MitarbeiterFeld = new Mitarbeiter[20]`
- Probleme:
  - Feste Obergrenze (Feldvergrößerung nur durch Neuanlegen und Umkopieren)
  - Neue Implementierung aller Operationen (Einfügen, Entnehmen, Sortieren, ...) für jedes Feld

- Mehrfache Implementierung gleicher Funktionalität:
  - Unwirtschaftlich
  - Widerspricht Objektorientierung
- Wünschenswert:
  - Übernahme Aufgaben durch vorgefertigte Klassen
  - Collection-Klassen universell, d.h. für beliebigen Typ verwendbar
- Java stellt im Paket `java.util` (util = utility) solche Klassen zur Verfügung

- Bis Java 1.4:

- Alle Container enthalten Elemente vom Typ `Object`
- Insbesondere: Objekte verschiedenen Typs im gleichen Container möglich
- Beispiel:

```
(1) Vector myVector = new Vector();  
(2) myVector.add(new Arbeiter());  
(3) Arbeiter a = (Arbeiter) myVector.iterator().next();
```

- Probleme:

- Nicht Typ-sicher (Objekte verschiedener Typen können in gleichem Vektor gespeichert werden)
- Bei Lesezugriff Typumwandlung notwendig
- Laufzeitfehler möglich

- Idee:

- Anlegen einer Schablone
- Übergeben des konkreten Typs

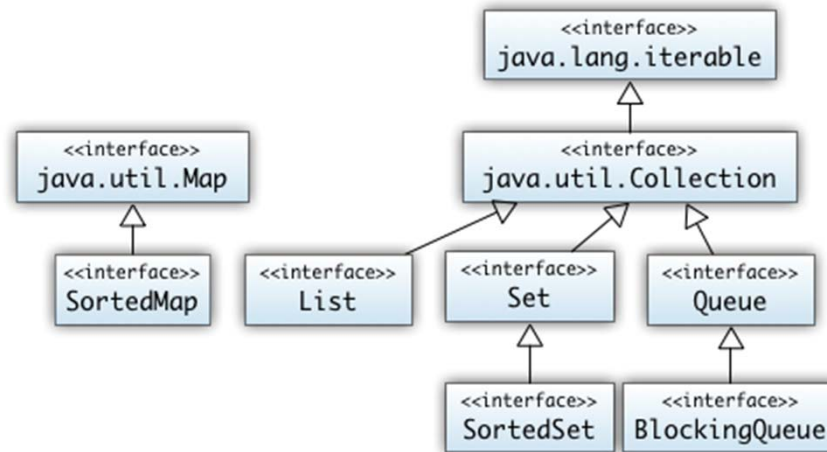
- Ab Java 5: Generische Datentypen (Generics)

- Beispiel:

```
(1) Vector<Arbeiter> myVector = new Vector<Arbeiter> ();  
(2) myVector.add(new Arbeiter());  
(3) Arbeiter a = myVector.iterator().next();
```

- Probleme behoben:
  - Typ-sicher (Compiler kann dies zur Übersetzungszeit überprüfen)
  - Damit keine Laufzeitfehler
  - Keine Typumwandlung notwendig

- Im Paket `java.util`
- Vier Gruppen:
  - Listen (abstrakte Klasse `List`): geordnete Datenstrukturen, Zugriff über Index, Unterschied zu Feldern: Einfügen an einer beliebigen Stelle
  - Mengen (abstrakte Klasse `Set`): mathematische Menge (Objekte können nur einmal enthalten sein), prüfen Objekt auf Enthaltensein, keine Reihenfolge oder Ordnung in Menge
  - Verzeichnisse (abstrakte Klasse `Map`): verallgemeinerte Felder, Wahl eines beliebigen Zugriffskriteriums, z.B. Personen nach Nachnamen verwalten und mit Hilfe eines gegebenen Nachnamens aufrufen
  - Warteschlangen (abstrakte Klasse `Queue`): Listen nach FIFO Prinzip (First In , First Out), kein wahlfreier Zugriff
  - Ergänzend: Klasse `Arrays` mit Basisfunktionen zum Suchen und Sortieren von Feldern



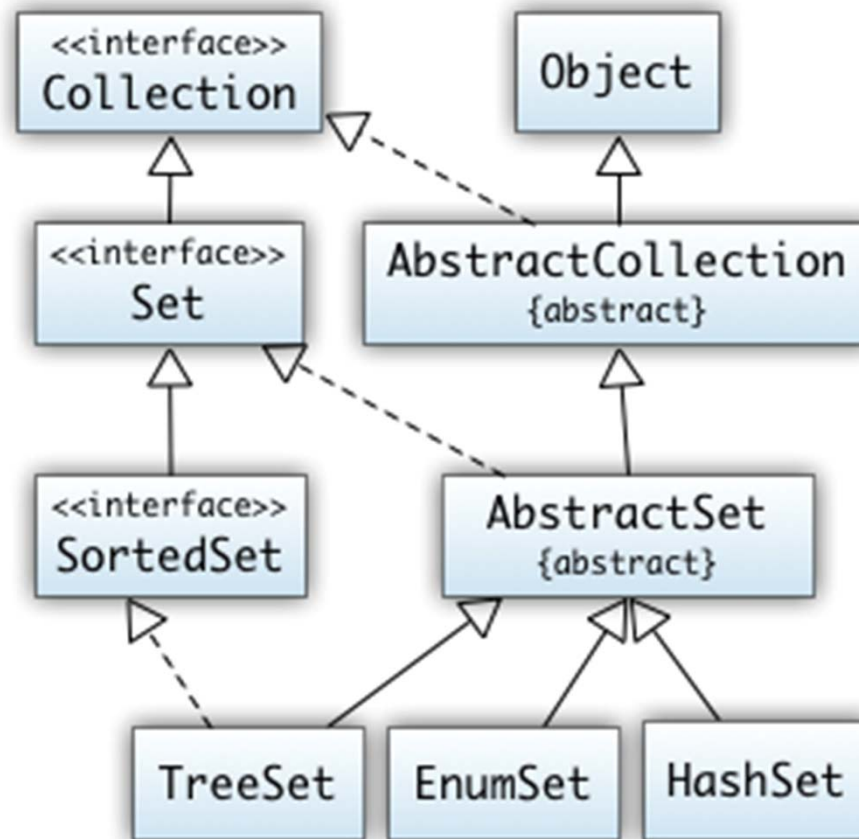
- Jede Gruppe hat abstrakte Klasse und Interface:

- Interfaces definieren welche Operationen für Gruppe erlaubt sind (funktionale Eigenschaft):
  - sortiert oder unsortiert
  - geordnet oder ungeordnet. Bleibt die Einfügereihenfolge erhalten?
  - sequenziell oder wahlfreier Zugriff
- Abstrakte Basisklasse implementiert Schnittstelle und bestimmt damit den Algorithmus, der zu bestimmter Effizienz führt



Klasse	Familie (implem. Schnittst.)	Zugriff	Duplikate erlaubt	Ordnung	Kommentar
<b>ArrayList</b>	Liste(List)	wahlfrei über Index	Ja	geordnet	effizientes Lesen über Index
<b>LinkedList</b>	Liste(List)	wahlfrei über Index	Ja	geordnet	effizientes Einfügen
<b>Vector</b>	Liste(List)	wahlfrei über Index	Ja	geordnet	synchronisiert(!)
<b>Stack</b>	Liste(List)	sequentiell (letztes Element)	Ja	geordnet	LIFO: Last In First Out
<b>HashSet</b>	Menge(Set)	ungeordnet; Test auf Existenz	Nein	ungeordnet	schnelles Lesen, Einfügen, Löschen
<b>TreeSet</b>	Menge(Set)	sortiert; Test auf Existenz	Nein	sortiert	
<b>LinkedList</b>	Warteschlange(Queue)	sequentiell, Nachfolger	Ja	geordnet	effizientes Einfügen (am Rand)
<b>PriorityQueue</b>	Warteschlange(Queue)	sequentiell, Nachfolger	Ja	sortiert	
<b>HashMap</b>	Verzeichnis(Map)	wahlfrei über Schlüssel	Keine Schlüsselduplikate, Werteduplikate erlaubt	ungeordnet	effizientes Lesen, Einfügen, Löschen
<b>TreeMap</b>	Verzeichnis(Map)	wahlfrei über Schlüssel	Keine Schlüsselduplikate, Werteduplikate erlaubt	sortiert	

- Einführung
- Mengen
- Listen
- Wildcards
- Selbst entwickelte generische Klassen
- Wrapper



- Wichtige Methoden:

Name	Bedeutung
<code>public boolean add(E e)</code>	Fügt Objekt e dem Set hinzu
<code>public void clear()</code>	Löscht Inhalt des Set
<code>public boolean contains(E e)</code>	Liefert true, falls Objekt e in Set, sonst false
<code>public boolean isEmpty()</code>	Liefert true, falls Set leer, sonst false
<code>public boolean remove(E e)</code>	Löscht Objekt e aus Set
<code>public int size()</code>	Liefert Anzahl Elemente des Set

- Weitere: <https://docs.oracle.com/javase/8/docs/api/java/util/Set.html>

- Abgeleitet von Set<E>, Elemente sortiert
- Wichtige Methoden:

Name	Bedeutung
<code>public SortedSet&lt;E&gt; subSet (E e, E f)</code>	Liefert Teilmenge von Element e bis Element f
<code>public SortedSet&lt;E&gt; headSet (E e)</code>	Liefert Teilmenge von Anfang bis Element e
<code>public SortedSet&lt;E&gt; tailSet (E e)</code>	Liefert Teilmenge von Element e bis Ende
<code>public E first()</code>	Liefert erstes Element
<code>public E last()</code>	Liefert letztes Element

- Weitere: <https://docs.oracle.com/javase/8/docs/api/java/util/SortedSet.html>

- Sortierte Speicherung:
  - Natürliche Ordnung oder
  - Wie in Interface Comparator definiert
- Kosten für Zugriffsmethoden `add()`, `remove()`, `contains()`:  $O(\log(n))$
- Nicht synchronisiert
- Anwendungsfälle:
  - Häufiges Iterieren über Menge mit bestimmtem Schlüsselkriterium
  - Erhöhte Kosten für Einfügen und Entfernen nicht relevant

- Intern als `HashMap` realisiert
- Keine Ordnung der Elemente  $\Rightarrow$  Beim Iterieren beliebige Reihenfolge
- Methoden `add()`, `remove()`, `contains()`, `size()`: konstanter Aufwand
- Operationen bei großen Datenmengen performanter als Methoden der Klasse `TreeSet`
- Anwendungsfälle:
  - Hohe Effizienz Einfüge-, Entfernungs- und Leseoperationen notwendig
  - Ordnung der Menge spielt keine Rolle

- Spezielle Implementierung für Enumerationen
- Interne Realisierung mit Bitvektoren
- Sehr effizient bei Mengenoperationen



```
( 1) // Anlegen eines Arbeiters, Facharbeiters und Managers
( 2) Arbeiter a = new Arbeiter(4711, "Meier", "Frank", 14.67);
( 3) Facharbeiter f = new Facharbeiter
( 4)           (4712, "Franken", "Peter", 18.33, "Chemie");
( 5) Manager m = new Manager(4713, "Kaiser", "Karl-Heinz",
( 6)           0.0, 0.0, "Mazda Cabrio", "S-ZZ 999");
( 7) // Anlegen HashSet-Objekt
( 8) HashSet<Mitarbeiter> h = new HashSet<Mitarbeiter>();
( 9) // Einfügen Objekte
(10) h.add(a);
(11) h.add(f);
(12) h.add(m);
(13) // Größe des HashSet ausgeben
(14) System.out.println("Größe des HashSet : " + h.size());
```

- Ermöglicht Aufzählen Container-Inhalt mittels Iteration
- Methoden:

Name	Bedeutung
<b>boolean</b> hasNext()	Liefert <code>true</code> , falls noch Elemente im Container, sonst <code>false</code>
<b>Object</b> next()	Liefert nächstes Objekt aus Container
<b>void</b> remove()	Entfernt zuletzt aufgelistetes Element aus Container

- Syntax für Iteratoren zum Auslesen:

```
for(<Objekttyp> <Variable> : <Container>){ ... }
```

- Navigation über alle Elemente des Containers
- Zugriff im Schleifenrumpf über Variablennamen
- Beispiel:

```
(1) // Ausgeben des Inhalts des HashSet  
(2) for(Mitarbeiter mit:h){  
(3)     mit.print();  
(4) }
```

führt zum Ausgeben aller Elemente des HashSet h

- Frühere Vorgehensweise (veraltet, kann aber in altem Code noch auftreten):
  - Anlegen des Iterators
  - Durchlaufen, solange noch Elemente vorhanden sind
  - Dabei Elemente bearbeiten

- Beispiel:

```
(1) for (Iterator i = h.iterator(); i.hasNext();){  
(2)     Object o = i.next();  
(3)     „Weitere Verarbeitung“  
(4) }
```

Anlegen des Iterators

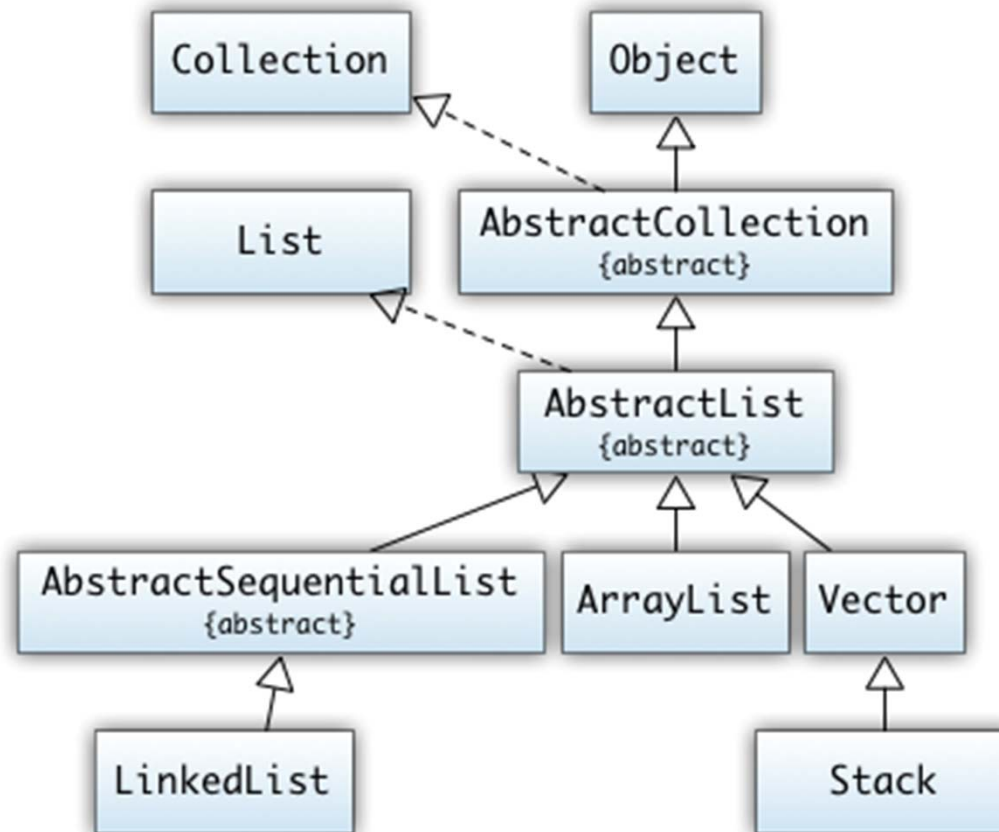
Durchlaufen, solange noch  
Elemente vorhanden

Objekt entnehmen und  
weiterverarbeiten

- Type-Cast im Schleifenrumpf, z.B.:

```
(1) Arbeiter o = (Arbeiter) h.get(i);  
(2) o.print();
```

- Einführung
- Mengen
- Listen
- Wildcards
- Selbst entwickelte generische Klassen
- Wrapper



- Wichtige Methoden:

Name	Bedeutung
<code>public void add(int i, E e)</code>	Fügt Element e an Position i ein, Nachfolger werden verschoben
<code>public E get(int i)</code>	Zugriff auf Objekt an Position i
<code>public void set(int i, E e)</code>	schreibender Zugriff auf ein Objekt an Position i
<code>public List&lt;E&gt; sublist(int von, int bis)</code>	Liefert Teilliste im vorgegebenen Intervall
<code>public void remove(int i)</code>	Entfernen Objekt an Index i
<code>public int size()</code>	Liefert Anzahl Elemente des HashSet

- Weitere: <https://docs.oracle.com/javase/8/docs/api/java/util/List.html>

- Dynamisch veränderbares Feld
- Eigenschaften:
  - Nicht synchronisiert
  - Methoden mit konstantem Aufwand: `size()`, `isEmpty()`, `get()`, `set()`, `iterator()`, `listIterator()`
  - Methode `add()` im Durchschnitt mit konstanter Ausführungszeit
  - Alle anderen Methoden linearer Aufwand (mit niedrigem konstanten Faktor im Vergleich zur Klasse `LinkedList`)



- Implementierung doppelt verzeigerter Liste
- Gut geeignet, wenn oft Elemente eingefügt oder entnommen werden müssen
- Beim indexierten, wahlfreien Zugriff auf Objekt muss Liste traversiert werden
- Nicht synchronisiert

- Wie `ArrayList`
- Aber synchronisiert
- Dadurch geringere Performance

- Einführung
- Mengen
- Listen
- Wildcards
- Selbst entwickelte generische Klassen
- Wrapper

- Bisher:
  - Anlegen von getypten Collections
  - Können auch Instanzen von Subklassen aufnehmen
  - z.B. kann `Vector<Mitarbeiter>` auch Arbeiter- oder Manager-Objekte aufnehmen
- Jedoch:
  - Bei Anlegen von `Vector<Mitarbeiter> v` und `Vector<Arbeiter> w` ist `w` nicht automatisch Unterobjekt von `v`
  - Manchmal jedoch hilfreich, um z.B. `w` an `v` zuzuweisen und Elemente von `w` als Mitarbeiter-Objekte zu nutzen
- Lösung: Wildcards ? `extends` und ? `super`

- Mit `Vector<? extends Mitarbeiter> vm = new Vector<Arbeiter>();` kann `vm` alle generischen `Vector`-Objekte referenzieren, die für Objekte einer Subklasse der Klasse `Mitarbeiter` angelegt wurden
- Damit: Elemente als `Mitarbeiter`-Objekte auslesen und nutzen
- Beispiel:

```
( 1) // Anlegen der Arbeiter-Objekte a,b,c wie vorher
( 2)     ...
( 3) // Anlegen eines Vector-Objektes
( 4) Vector<Arbeiter> va = new Vector<Arbeiter>();
( 5) // Einfügen der Objekte in den Vektor
( 6) va.add(a); va.add(b); va.add(c);
( 7) // Anlegen eines Vektors mit Wildcard
( 8) Vector<? extends Mitarbeiter> vm = va;
( 9) // Auslesen aller Objekte als Mitarbeiter-Objekte
(10) for (Mitarbeiter arb : vm){
(11)     arb.print();
(12) }
```

Hier wird Methode von  
Mitarbeiter aufgerufen

- Zugriff über Wildcard-Referenzen funktioniert nur lesend
- D.h. Hinzufügen eines `Arbeiter`-Objektes zu `vm` führt zur Fehlermeldung:

```
(1) Arbeiter d = new Arbeiter();  
(2) vm.add(d);
```

- Ist dies gefordert, so ist folgender Umweg notwendig:

```
(1) // Kopieren und Casten des Vektors in Vektor  
(2) // des konkreteren Typs  
(3) Vector<Arbeiter> va_hilf = (Vector<Arbeiter>) vm;  
(4) // Hinzufügen des neuen Elements zu diesem Vektor  
(5) va_hilf.add(d);  
(6) // Zurückkopieren des Vektors  
(7) vm = va_hilf;
```

- Analog: Einschränkung Wildcards mit ? super auch in andere Richtung der Vererbungshierarchie
- Beispiel:

```
Vector<? super Arbeiter> va = new Vector<Mitarbeiter>();
```

erlaubt für va jedes generische Container-Objekt mit Elementen des Typs Arbeiter oder einer Superklasse

- Einführung
- Mengen
- Listen
- Wildcards
- Selbst entwickelte generische Klassen
- Wrapper



- Erstelle generische Klasse mit folgenden Eigenschaften:
  - Multimenge
  - FIFO-Prinzip (First In – First Out), Schlange, Queue
  - Methoden:
    - Einfügen eines Elements
    - Entnehmen eines Elements
    - Lesen des ältesten Elements ohne zu entnehmen
    - Zählen der Elemente

```
( 1) public class MyQueue<T>{  
( 2)  
( 3)     private Object[] myArray;  
( 4)     private int numberOfElements;  
( 5)     private int maxSize;  
( 6)  
( 7)     public MyQueue(){  
( 8)         this.numberOfElements = 0;  
( 9)         this.maxSize = 100;  
(10)         myArray = new Object[this.maxSize];  
(11)     }  
(12)  
(13)     public void add(T t){  
(14)         this.myArray[this.numberOfElements] = t;  
(15)         this.numberOfElements++;  
(16)     }  
(17)     ...
```

```
...  
(17) public T get(){  
(18)     T retValue = (T) this.myArray[0];  
(19)     for (int i=0; i<this.numberOfElements; i++){  
(20)         this.myArray[i] = this.myArray[i+1];  
(21)     }  
(22)     this.numberOfElements--;  
(23)     return retValue;  
(24) }  
(25)  
(26) public int size(){  
(27)     return this.numberOfElements;  
(28) }  
(29) }
```

- Einführung
- Mengen
- Listen
- Wildcards
- Selbst entwickelte generische Klassen
- Wrapper

- Zwei Arten von Variablen in Java:
  - Variable primitiven Typs (z.B. `int`, `long`, ...)
  - Objekte
- Unterschiede:
  - Vergleichen (`==` vs. `equals()`)
  - Unterschiedliche Behandlung als Methodenparameter:
    - Variablen primitiven Typs als Wertparameter
    - Objekte als Referenzparameter
  - Aufnahme in Collection-Klassen nur von Objekten

- Wrapper-Klassen (Synonym: Ummantelungsklassen, Envelopes) := Verwandeln durch Kapselung Variablen primitiver Typen in Objekte
- Weitere Funktionalitäten:
  - Zugriff auf den Wert
  - Umwandlungsfunktionen
- In Java solche Klassen für alle primitiven Typen:
  - Haben jeweils Namen des Typs mit Großbuchstaben beginnend (z.B. Klasse `Boolean` für den Datentyp `boolean`)
  - Ausnahme: Klasse `Integer` für Datentyp `int`

- Beispiel:

```
(1) // Deklarieren und Initialisieren einer Integer-Variablen  
(2) int i = 5;  
(3) // Anlegen eines Integer-Objektes mit dieser Integer-Var.  
(4) Integer j = new Integer(i);  
(5) // Auslesen des Wertes aus dem Integer-Objekt in eine  
(6) // Integer-Variablen  
(7) int k = j.intValue();
```

- Anmerkungen:

- Wrapper-Objekte sind unveränderlich
- Direktes Zuweisen zwischen Objekten und Wrapper-Klassen und Variablen, d.h.

`Integer j = i` statt `Integer j = new Integer(i);`

und

`int k = j` statt `int k = j.intValue();`

wird als Boxing bzw. Unboxing bezeichnet

- Einführung:
  - Motivation für Generics
  - Collection-Klassen
- Mengen:
  - Interfaces `Set<E>` und `SortedSet<E>`
  - Klassen `HashSet`, `TreeSet` und `EnumSet`
  - Interface `Iterator`
- Listen:
  - Interface `List<E>`
  - Klassen `ArrayList`, `LinkedList` und `Vector`



- Wildcards:
  - Motivation
  - ? extends
  - ? super
  
- Wrapper:
  - Motivation
  - Beispiel

- **Aufgabe 1**

a) Realisieren einen `Integer`-Stapel. Dabei soll die Klasse `Stack` aber nicht verwendet werden, sondern eine dynamische Struktur wie in der Vorlesung „Algorithmen und Datenstrukturen“ vorgestellt selbst realisiert werden.

Schreiben Sie eine Klasse `IntegerStack`, die den Stapel mit den Methoden `public void push(int i)` und `public int pop()` zur Verfügung stellt.

Schreiben Sie dann eine Klasse `Anwendung`, in der die Funktionalität Ihres `Integer`-Stapels anhand der im Skript vorgestellten Reihenfolge von Operationen demonstriert wird.

b) Realisieren Sie einen generischen Stapel und testen Sie auch diesen in einer `Anwendung`.

c) Realisieren Sie jetzt die Klasse `Anwendung` unter Verwendung der Klasse `Stack` aus der Java-Klassenbibliothek.

## • Aufgabe 2

Ergänzen Sie das Beispiel der Schlange aus der Vorlesung, so dass sich die Feldgröße dynamisch anpasst! Dabei soll

- bei einem Belegungsgrad von 80% die allokierte Feldgröße verdoppelt werden
- bei einem Belegungsgrad von 20% die allokierte Feldgröße halbiert werden
- die Feldgröße aber nie den Wert 20 unterschreiten

## • Aufgabe 3

Es soll die Einfügeperformance der beiden Klassen LinkedList und ArrayList verglichen werden.

Entwickeln Sie dazu ein Programm, das 10.000, 20.000, 50.000, 100.000, 200.000, 500.000, 1.000.000, 2.000.000, 5.000.000, 10.000.000, 20.000.000 und 50.000.000 zufällig generierte Zeichenketten der Länge 10 bis 16 aufnimmt.

Führen Sie jede Messung 10-mal durch und dokumentieren Sie die Werte! Berechnen Sie jeweils den Mittelwert!

Hinweise:

- Eine Zufallszahl zwischen 0 und 1 erzeugen Sie mit `Math.random()`
- Verwenden Sie für die Zeitmessung folgenden Code:

Hinweise:

- Eine Zufallszahl zwischen 0 und 1 erzeugen Sie mit `Math.random( )`
- Verwenden Sie für die Zeitmessung folgenden Code:

```
long timeStart = System.currentTimeMillis();
```

```
<Code, dessen Ausführungszeit gemessen werden soll>
```

```
long timeEnd = System.currentTimeMillis();
```

```
long elapsedTime = timeEnd - timeStart;
```