12.4 Generische Klassen und Schnittstellen der Java API

- Das Paket java.util bietet unter anderem Algorithmen, Klassen und Schnittstellen zur (effizienten) Verwaltung von Mengen beliebiger Datenobjekte.
  - Listen
  - Suchbäume
  - Sortierverfahren
- Wichtige generische Schnittstellen
  - Comparable
    - Vergleich von zwei Objekten
  - Iterator
    - Durchlauf durch eine Datenstruktur unabhängig von der konkreten Implementierung

Interface List<E>

- Das Interface ist eine zentrale Schnittstelle in java.util.
  - Das Interface hat noch weitere Ober-Interfaces, wie z. B. Iterable<E>. Dieses Interface werden wir gleich betrachten.
  - Das Interface List wird von vielen Klassen implementiert, wie z. B.
    - ArrayList<E>
    - LinkedList<E>
    - SortedList<E>
- Wichtige Methoden im Interface
  - boolean add(E e)
  - E get(int index)
  - List<E> subList(int fromIndex, int toIndex)

Interface Comparable<T>

- Für geordnete Daten verwendet man das Interface Comparable<T>, das folgende Methode für Objekte o vom Typ T vorschreibt.

```
int compareTo(T o);
```

- Das Ergebnis von compareTo ist vom Typ int. Es gilt folgende Konvention:
  - Wenn a.compareTo(b) negativ ist, interpretiert man dies als a < b.
  - Wenn a.compareTo(b) 0 ist, interpretiert man dies als a == b.
  - Wenn a.compareTo(b) positiv ist, interpretiert man dies als a > b.
- Beispiel

```
class Point2D implements Comparable<Point2D> {
...
public int compareTo(Point2D o) {  // lexikografischer Vergleich
    int xc = Double.compare(x, o.x); // Verwenden
    if (xc < 0)
        return -1;
    else
        return (xc == 0)? Double.compare(y,o.y) : 1;
    }
}
```

Interface Iterator<E>

- Ein Behälter ist ein generischer Datentyp zur Verwaltung von Mengen beliebiger Objekte.
  - Listen sind nur ein Beispiel für einen Behälter.
- Ein Iterator liefert die Elemente eines Behälters (oder eines Teils) in einer spezifischen Reihenfolge („Aufzählung der gewünschten Elemente“).
  - Die Methode `hasNext()` liefert `true`, wenn bei der aktuellen Aufzählung der Elemente des Behälters noch weitere Elemente anstehen.
  - Die Methode `next()` produziert das nächste Element der Aufzählung.
  - Die Methode `remove()` entfernt das Element aus dem Behälter, das zuletzt mit next abgeliefert wurde. Diese Methode ist optional – d.h. kann auch weggelassen werden und führt dann ggf. zu einer Ausnahme.

```
public interface Iterator<E> {
  public boolean hasNext();
  public E next();
  public void remove();
}
```

Beispiel: Ein Iterator für Listen

```
import java.util.List;
/**
 *  Ein Iterator, der alle Elemente in einer Liste liefert.
 */
public class MyListIterator<E> implements Iterator<E> {
        List<E> list; // Zu durchlaufende Liste

        public MyListIterator(List<E> l) {
                list = l;
        }

        public boolean hasNext() {
                return !list.isEmpty();
        }

        public E next() {
                E tmp = list.get(0);
                list = list.subList(1,list.size()); // Liefert die Restliste
                return tmp;
         }

}
```

Die remove()-Methode ist als Default-Methode im Interface Iterator implementiert und wirft dort eine UnsupportedOperationException.

Interface Iterable<E>

- Analog zu einem Array kann auch eine Liste mit der **for-each**-Schleife durchlaufen werden.
- Voraussetzung hierfür ist, dass die Liste noch die generische Schnittstelle Iterable<E> implementiert.

```
public interface Iterable<E> {
  public Iterator<E> iterator();
}
```

  - Die Schnittstelle List erweitert die Schnittstelle Iterable

```
public interface List<E> extends Iterable<E>{ ... }
```

und die Klasse LinkedList implementiert die Methode.

```
public class LinkedList<E> implements List<E>{
    ...

    public Iterator<E> iterator() {
         return new MyListIterator<E>(this);
    }
}
```

Anwendung der for-each Schleife

- Da unsere Klasse LinkedList die Schnittstelle Iterable implementiert, kann der Durchlauf durch die Listen mit einer for-each-Schleife erfolgen.

```
public static void main(String[] args) {
    List<Point2D> list = new LinkedList<Point2D>();
    list.add(new Point2D (.2,.3));
    list.add(new Point2D (.3,.4));
    list.add(new Point2D (.4,.5));
    list.add(new Point2D (.5,.6));

    // Ausgabe aller Punkte mit der for-each Schleife
    for (Point2D p: list)
        System.out.println("Punkt: " + p);
}
```

- Die for-each-Schleife kann für alle Klassen, die Iterable implementieren, genauso wie bei Arrays benutzt werden.

12.5 Java Generics im Detail

- Bisher haben wir am Beispiel von Listen die wichtigsten Konzepte generischer Klassen erklärt.
- In diesem Abschnitt sollen weitere Details von Java Generics behandelt werden.

Syntax

- Java Generics unterstützt die Parametrisierung mit Typen bei
  - Klassen
  - Schnittstellen
  - Methoden
- Anzahl der Parameter
  - beliebig, aber i. A. ist die Anzahl kleiner 3.
- Syntax
  - Angabe der Parameterliste in eckigen Klammerpaar „< … >“
    - Bei Klassen und Schnittstellen nach dem entsprechenden Namen.
    - Parameter werden durch Kommata getrennt.

Beschränkungen

- **Keine** primitive Typen als Argument für einen Typparameter.

- **Keine** Aufrufe von Konstruktoren des Typparameters T

  `T x = new T();` funktioniert nicht!

- **Kein** Aufruf von statischen Methoden des Typparameters T

  `double avg = T.myStaticMethod();` funktioniert nicht!

- **Keine** Verwendung des Typparameters T in statischen Methoden/bei statischen Felddeklarationen

- **Keine** Allokation eines Arrays von Typparameter T

  `T[ ] arr = new T[12];` funktioniert nicht!

  `List<Point2D>[] larr = new List<Point2D>[12];` funktioniert nicht!

  `List<Point2D>[] larr = new List[12];` **funktioniert (ab Java 1.7)!**

Beschränkungen

- **Keine** primitive Typen als Argument für einen Typparameter.

```
List<int> li;
```

```
//das Folgende funktioniert aber:
List<Integer> = ...;
li.add(1);
// dabei wird der int 1 in ein
// automatisch in ein Integer-Objekt
// umgewandelt.
```

T

ktioniert nicht!

ers T

ktioniert nicht!

ktioniert nicht!

unktioniert nicht!

List<Point2D>[] larr = new List[12];

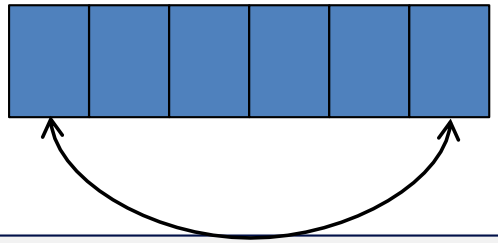
funktioniert (ab Java 1.7)!

Generische Methoden

- Objektmethoden dürfen Typparameter der Klasse verwenden
- Außerdem erlaubt Java, sowohl statischen Methoden als auch Objektmethoden eigene Typparameter zu deklarieren.
  - Die Liste der generischen Typen wird vor dem Rückgabetyp der Methode angegeben.

- Beispiel

```
// class Main
public static <T> void swap(T[] arr) {
    T tmp = arr[0];
    arr[0] = arr[arr.length-1];
    arr[arr.length-1] = tmp;
}
```



Automatische Typ-Bestimmung

- Oft kann der Java-Compiler den Typ für Typparameter selbst bestimmen

- Bei Erzeugung einer Instanz:
  - `//explizite Typangabe:`
    `List<Integer> li = new LinkedList<Integer>();`
  - `//Automatische Typbestimmung`
    `List<Integer> li = new LinkedList<>();`

- Beim Aufruf einer Methode: (gegeben: `Integer[] arr = null;`)
  - `//explizite Typangabe:`
    `Main.<Integer>swap(arr);`
  - `//Automatische Typbestimmung`
    `Main.swap(arr);`

Einschränkung des Typparameters

- Bei den bisherigen generischen Typen erlauben wir beliebige Typen bei der Instanziierung.

```
public static void main(String[] args) {
    // Liste mit Punkten
    List<Point2D> listp = new LinkedList<>();

    // Liste mit Konten
    List<Konto> listk = new LinkedList<>();

    // Liste mit Integern
    List<Integer> listi = new LinkedList<>();

    // Liste mit Objekten
    List<Object> listo = new LinkedList<>();
}
```

- Dies ist nicht immer erwünscht, da in bestimmten Fällen von Typen gewisse Eigenschaften gefordert werden.
  - Z. B. sollte eine geordnete Liste nur mit Typen instanziiert werden, die die Schnittstelle Comparable unterstützen.

extends-Klausel

- Bei einer generischen Klasse kann diese Eigenschaft durch Angabe des Schlüsselworts **extends** und einer Klasse oder Schnittstelle gefordert werden.
- Beispiel
  - Es soll eine generische Klasse erstellt werden, um beliebige Zahlen zu addieren.
    - Anmerkung:
      Die Klasse Number aus der Java API ist die Oberklasse von all diesen Klassen wie z. B. der Klasse Integer.
  - Lösung

```
class Accumulator<T extends Number> {
                ...
   // Über T kann jetzt auf die Methoden der Klasse
   // Number zugriffen werden.
}
```

```
Accumulator<Integer> ai = new Accumulator<>();     // funktioniert
```

```
Accumulator<String> ai = new Accumulator<>();     // funktioniert nicht.
```

extends-Klausel mit Schnittstellen

- Um eine Ordnung in der Liste sicherzustellen, sollte der generische Typ die Schnittstelle Comparable unterstützen.
  - Zusätzlich müssten wir in unserer Listenimplementierung die Methode

    `boolean add(T elem)`

    noch so ändern, dass die Objekte entsprechend der Ordnung in der Liste liegen.
- Entsprechend zu Klassen kann auch die extends-Klausel bei Schnittstellen verwendet werden.
  - Die Schnittstelle Comparable ist aber wiederum generisch.
- Der generische Typ T darf wieder als Typparameter der Schnittstelle/Klasse verwendet werden, die T implementieren/erweitern soll.

```
class SortedList<T extends Comparable<T>> { ... }
```

extends-Klausel mit mehreren Schnittstellen


```
public interface Cat {
    void miau();
}
```

```
public interface Dog {
    void wuff();
}
```

- Die Methode catDog erwartet ein Objekt einer Klasse, die sowohl das Interface Cat als auch das Interface Dog implementiert

```
public <T extends Dog & Cat> void catDog(T t) {
    t.miau();
    t.wuff();
}
```

extends-Klausel mit Klasse und Schnittstelle

- In manchen Situationen ist es also nützlich, dass der Typparameter mehrere Schnittstellen unterstützen soll. Es ist sogar Folgendes möglich:
  - Sei A eine Klasse oder Schnittstelle und seien BI, CI Schnittstellen, dann wird durch

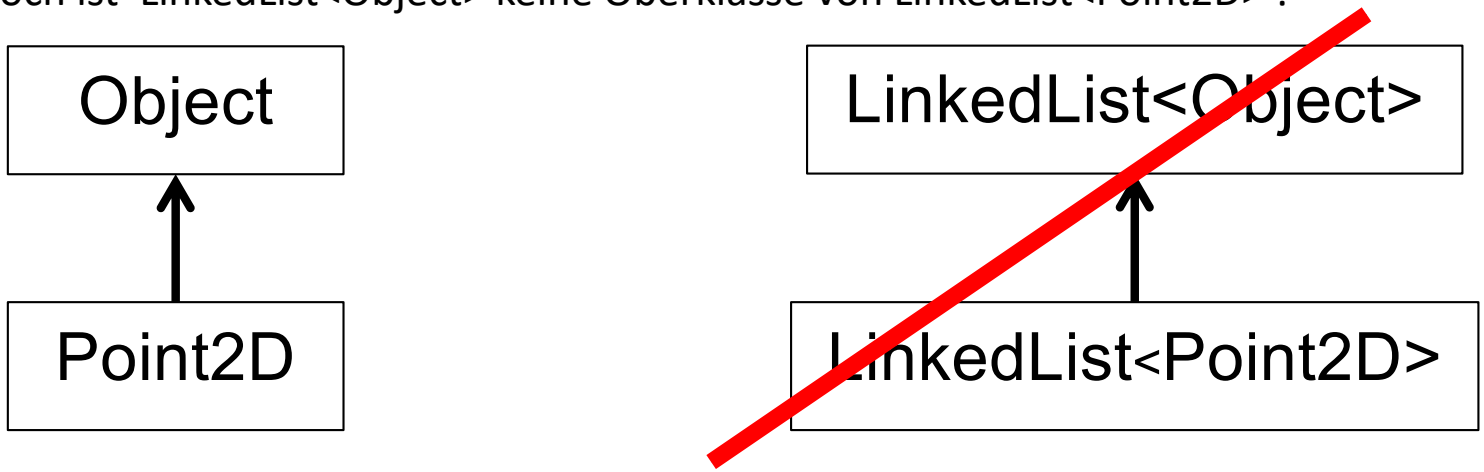
```
class MyGenericClass<T extends A & BI & CI> { ... }
```

    gefordert, dass bei der Instanziierung nur Klassen verwendet werden können, die alle Schnittstellen implementieren und (im Fall, dass A eine Klasse ist) A erweitern.

- Wichtig dabei ist, dass nur der erste Typparameter eine Klasse sein darf. Alle anderen Parameter müssen Schnittstellen sein.

12.6 Wildcards – Motivation (1)

- Bei den bisherigen Möglichkeiten in Generics gibt es noch ein paar Probleme.
  - Die Klasse Object ist Oberklasse von Point2D.
  - Jedoch ist LinkedList<Object> keine Oberklasse von LinkedList<Point2D> !



  - Damit ist dieser Programmschnipsel **nicht** korrekt.

```
LinkedList<Point2D> points = new LinkedList<Point2D>();// funktioniert

LinkedList<Object> objects = new LinkedList<Point2D>();// nicht erlaubt
```

  - Es gibt also keine Polymorphie zwischen den beiden Listen-Klassen.

Warum ist es nicht erlaubt?

- Betrachten wir folgende Situation
  - Eine Methode zum Einfügen eines neuen Konto-Objekts in eine Liste vom Typ LinkedList<Object>.

```
void addSomethingToList (LinkedList<Object> lif) {
    // Wir fügen jetzt etwas zu lif hinzu, wie z. B. ein Konto:
    lif.add(new Konto());
}
```

- Aufruf der Methode mit einem Parameter vom Typ LinkedList<Point2D>
  - Das sollte aber verhindert werden, da **Konto keine Unterklasse von Point2D** ist.

```
public static void main(String[] args) {
    // Liste mit Punkten
    LinkedList<Point2D> points = new LinkedList<>();
    points.add(new Point2D(0.2,0.3));

    addSomethingToList(points);      // Funktioniert nicht!

}
```

Warum sollte es erlaubt sein?

- Es soll eine Methode bereitgestellt werden, um eine beliebige Liste auszugeben.

```
void printList (LinkedList<Object> lif) {
    for (Object e: lif)
        System.out.println(e);
}
```

- Der Aufruf der Methode printList für die Liste points ist leider nicht erlaubt!

```
public static void main(String[] args) {
    // Liste mit Punkten
    LinkedList<Point2D> points = new LinkedList<>();
    points.add(new Point2D(.2,.3));

    printList(points);          // Funktioniert nicht!

}
```

Wildcard-Typ

- Durch Verwendung eines Wildcard-Typs wird dieses Problem behoben.

```
void printList (LinkedList<?> lif) {
    for (Object e: lif)
        System.out.println(e);
}
```

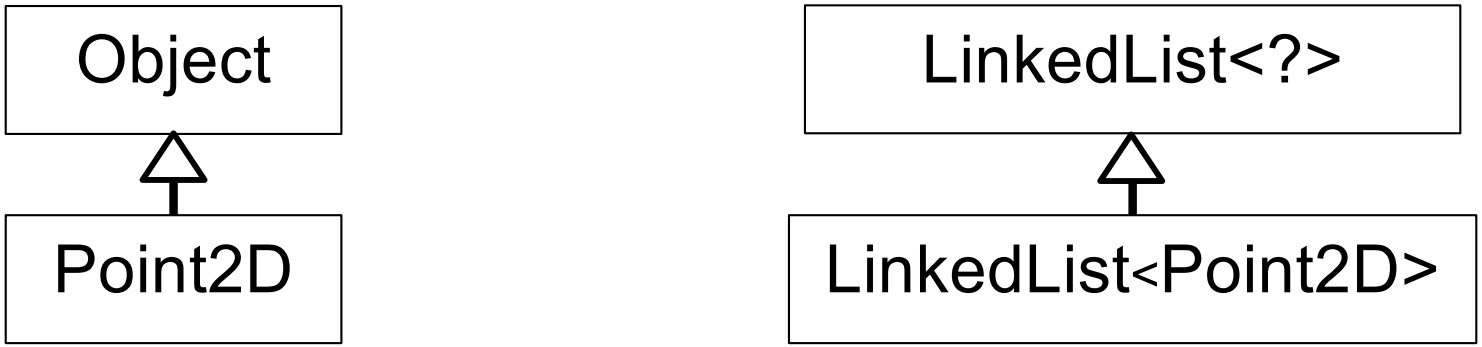
- Der Aufruf der Methode für die Liste points funktioniert jetzt!

```
public static void main(String[] args) {
    // Liste mit Punkten
    LinkedList<Point2D> points = new LinkedList<>();
    points.add(new Point2D(.2,.3));
    printlist(points);          // alles in Ordnung!
}
```

- Wichtige Beobachtungen
  - In der Methode printList dürfen wir nicht die Methode add aufrufen, da **diese Methode den Typparameter in der Parameterliste verwendet (boolean add(T t) {…})**
  - **Grund hierfür ist, dass der Typparameter der generischen LinkedList<?> unbekannt ist.**

Beziehung zwischen Klassen

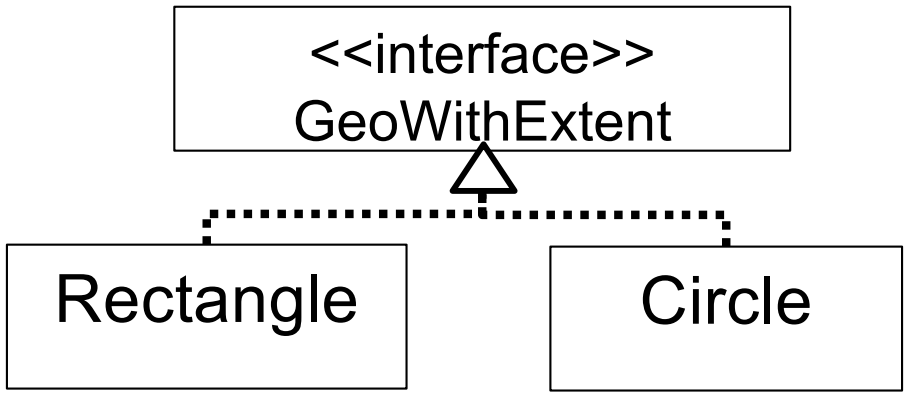
- Durch LinkedList<?> wird ein Obertyp für alle LinkedList-Klassen zur Verfügung gestellt.



- Jedoch gelten folgende zwei Einschränkungen:
  1. **Liefern Methoden den Typparameter der LinkedList<?> als Ergebnis, können wir nur davon ausgehen, dass das Ergebnis zur Klasse Object gehört.**
  2. **Es darf kein Aufruf einer Methode von LinkedList<?> erfolgen, in der die Parameterliste den Typparameter verwendet.**
- **Die erste kann durch nach oben beschränkte Wildcards gelockert werden.**
- **Die zweite durch nach unten beschränkte Wildcards aufgehoben werden.**

Motivation für oben beschränkte Wildcards

- Betrachten wir folgende Klassenhierarchie



  - Die Schnittstelle GeoWithExtent besitzt eine Methode area() zur Flächenberechnung.
  - Die Klassen Circle und Rectangle sind zwei Klassen, die die Schnittstelle implementieren.
- Wir betrachten im Folgenden drei verschiedene Listen.

```
LinkedList<GeoWithExt> geos = LinkedList <>();
LinkedList<Rectangle> rects = LinkedList <>();
LinkedList<Circle> circles = LinkedList <>();
```

- Können wir eine generische Methode bereitstellen, um für alle drei Listen die Summe der Flächeninhalte der Objekte zu berechnen?

Obere Schranke für Wildcards

- Benutzen wir den normalen Wildcard LinkedList<?> steht uns die Methode double area() nicht zur Verfügung.
- Deshalb gibt es nach **oben beschränkte Wildcards**, bei der wir nach **?** das Schlüsselwort **extends** und ein Typ als obere Schranke hinzufügen können.
- In unserem Beispiel:

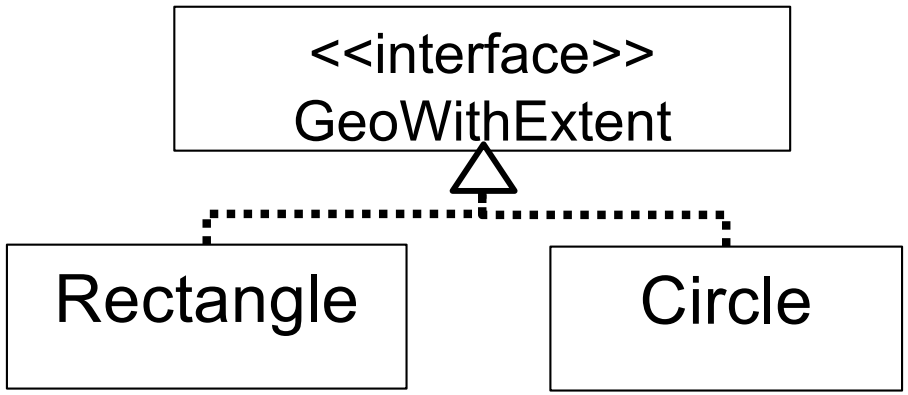
```
static double areaOverAll(LinkedList<? extends GeoWithExtent> list) {
    double res = 0.0;
    for (GeoWithExtent s: list) {
        res += s.area();
    }
    return res;
}
```

  - Diese Methode kann für alle Listen parametrisiert mit einer Unterklasse von GeoObjectsWithExtent verwendet werden.

```
double resg = areaOverAll(geos);          // Rechtecke und Kreise
double resr = areaOverAll(rects);         // Rechtecke
double resc = areaOverAll(circles);       // Kreise
```

Motivation für unten beschränkte Wildcards

- Wir betrachten wieder die Klassenhierarchie



- Die Klasse Circle soll die Methode double getRadius() besitzen.
- Ist es sinnvoll, ein Objekt der Klasse LinkedList<GeoWithExt> an eine Variable der Klasse LinkedList<Circle> zu übergeben?
  - Wenn ja, wie können wir dies in Java unterstützen?
- Wir betrachten im Folgenden zwei Beispiele.

Beispiel 1

```
static double getAllRadii(LinkedList<Circle> circles){
    double res = 0;
    for (Circle  c: circles) {
        res += c.getRadius();
    return res;
}


public static void main(String[] args){
    LinkedList<GeoWithExt> geos = new LinkedList<GeoWithExt>();
    geos.add(new Rectangle());
    double total = getAllRadii(geos); // // Funktioniert nicht!
}
```

- In diesem Fall (lesender Zugriff) darf die Übergabe des Objekts der Listenoberklasse an einer Variable einer Listenunterklasse **nicht** erfolgen.
  - **Grund: In einer Liste vom Typ LinkedList<GeoWIthExt> können Objekte einer Klasse sein, für die keine Methode getRadius() existiert.**

    ➔ Der Compiler zeigt zu Recht einen Fehler an!

Beispiel 2

```
static void addCircle(LinkedList<Circle> circles){
        circles.add(new Circle());
}


public static void main(String[] args){
    LinkedList<GeoWithExt> geos = new LinkedList<GeoWIthExt>();
    geos.add(new Rectangle());
    addCircle(geos);                    // Funktioniert leider auch nicht!
}
```

- In diesem Fall könnte man eigentlich den Aufruf der Methode addCircle erlauben, aber der Compiler ist zu restriktiv.

Untere Schranke

- Damit der Compiler solche Programme akzeptiert gibt es in Java nach unten beschränkte Wildcards.
  - Nach dem **?** folgt das **Schlüsselwort super** und eine Klasse L als untere Schranke der Klassen K, die beim Aufruf verwendet werden dürfen.
- Beispiel

```
static void addCircle(LinkedList<? super Circle> circles){
        circles.add(new Circle());   // Schreibender Zugriff erlaubt
}

public static void main(String[] args){
    LinkedList<GeoWithExt> geos = new LinkedList<GeoWIthExt>();
    geos.add(new Rectangle());
    addCircle(geos);                    // Funktioniert
}
```

  - **Der Aufruf von add(new Circle()) ist erlaubt, da nur Listen mit folgenden Klassen möglich sind: LinkedList<Circle>, LinkedList<GeoWithExtent>, …, LinkedList<Object>**
  - **In all diesen Listenklassen darf ein Objekt der Klasse Circle oder einer Unterklasse von Circle hinzugefügt werden.**

Zusammenfassung Wildcards

- Wir haben zwei unterschiedlich beschränkte Wildcards
  - nach oben beschränkte Wildcards: <? extends OT>
    - **nur Methodenaufrufe ohne Typparameter in der Parameterliste**
    - Verwendung der null-Referenz beim Aufruf immer möglich
  - nach unten beschränkte Wildcards: <? super UT>
    - **nur Objekte der Klasse UT (und Unterklassen) als Parameter bei Methodenaufruf mit Typparameter in der Parameterliste.**
    - Im Fall, dass die Methode als Ergebnistyp den Typparameter hat, kann man das Ergebnis nur an eine Variable vom Typ Object übergeben.
- Wildcards sind ziemlich fortgeschrittene, aber nützliche Konzepte.
  - Beispiel: Methode copy aus der Klasse java.util.Collections

```
/** Copies all elements from one list into another */
static <T> void copy(List<? super T> dest, List<? extends T> src){…}
```