# 12. Generische Klassen

- Problem bei der Erstellung von Software sind die sehr hohen Kosten.
  - Kosten für die Erstellung von 1 Zeile Programmcode: 10 – 50 Euro
  - Wartungskosten von existierendem Code: 50–75% der Gesamtkosten
- Eine Lösungsansatz für die Kostenreduktion ist der Einsatz von vorgefertigten Komponenten zur Entwicklung von Anwendungen, um die Anzahl der noch benötigten Zeilen eigenen Programmcodes zu minimieren.
- Beispiele
  - Statt I/O selbst zu programmieren greift man auf Standardsoftware zurück wie z. B. Datenbanksysteme
  - Java bietet mit dem JDK eine Vielzahl vorgefertigter Klassen, die man für die Entwicklung eigener Anwendungsprogramme nutzen kann.
- Die in Java unterstützen generischen Klassen erlauben es in einfacher Weise wiederverwendbare Komponenten selbst zu entwickeln.

# Aufbau des Kapitels

- Fallbeispiel: Listen mit 2-dimensionalen Punkten
  - Konzept einer Liste
- Generische Liste mit Object
  - Nachteile dieser Lösung
- Generische Listen als parametrisierte Klassen
- Generische Klassen aus der Java-Bibliothek
  - Schnittstellen Comparable, List und Iterable
- Details zu generischen Klassen
  - Schlüsselwort extends
  - Wildcards

# 12.1 Fallbeispiel: Listen

- Wichtige Aufgabenstellung in der Informatik
  - Dynamische Verwaltung einer Menge gleichartiger Datenobjekte, um diese wiederzufinden.
    - Gleichartig bedeutet, dass die Objekte zu einer Klasse gehören.
- Beispiele
  - Wir haben innerhalb einer betriebswirtschaftlichen Anwendung Klassen für Kunden, Produkte, Aufträge, Bestellungen, … erstellt.
    - In jeder dieser Klassen werden Datenfelder definiert, die den Zustand der Objekte aus diesen Klassen beschreiben.
  - In einem Unternehmen sollen jetzt Mengen von Objekten verwaltet werden, die aus einer Klasse stammen. Z. B. eine Menge aller Kunden eines Unternehmens.
    - Es soll möglich sein, neue Kunden in die Menge einzufügen und Kunden aus der Menge zu löschen.
    - Zudem soll es möglich sein, Kunden in der Menge unter Angabe von Suchprädikaten zu finden.
      - Suchprädikate hängen dabei von den Datenfelder des Objekts ab.
      - Suche nach dem Kunden mit KundenNr = 1234
      - Suche nach Kunden mit Wohnort = „Marburg“

# Mögliche Implementierungen

- Menge als Array
  - Ist die Maximalzahl möglicher Datenobjekte bekannt, so sollte am besten ein Array verwendet werden.

- Menge als einfach verkettete Liste
  - Ist die Anzahl der Datenobjekte nicht bekannt, kann man die Objekte dynamisch z.B. in einfach verketten Listen verwalten.
  - Im Gegensatz zu einem Array kann eine einfach verkettete Liste dynamisch (also zur Laufzeit) verlängert bzw. verkürzt werden.
    - Durch das Einfügen neuer Elemente wird eine Liste verlängert.
    - Wenn Elemente gelöscht werden, wird die Liste verkürzt.

# Was soll also eine Liste leisten?

- Die Minimalanforderung besteht aus folgenden Methoden
  - Hinzufügen eines Elements an das Ende der Liste

    *boolean add(Element e)*
  - Löschen eines Elements aus der Liste

    *boolean remove(Element e)*
  - Suchen nach einem Element in der Liste

    *boolean contains(Element e)*
  - Prüfen, ob die Liste leer ist.

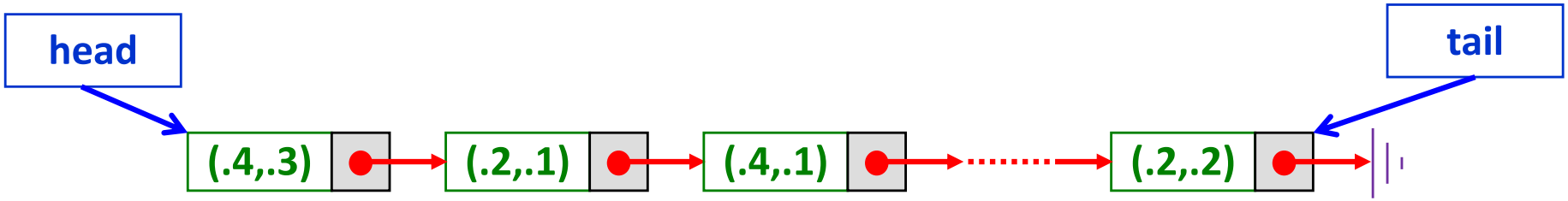
    *boolean isEmpty()*
  - Ausgabe des i-ten Elements aus der Liste

    *Element get(int i)*

- Was ist unter dem Datentyp *Element* zu verstehen?
  - Wir wollen Objekte mit beliebigem Typ speichern können! (dazu später mehr)

# Implementierung einer einfach verketteten Liste

- Zunächst machen wir folgenden Annahme
  - Der Datentyp Element entspricht der Klasse *Point2D*, die wir bereits aus früheren Kapiteln kennen. Wir wollen also eine Liste von Punkten verwalten.
- Eine Liste von Punkten besteht aus miteinander verketten Listenelementen. Jedes Listenelement hat
  - **ein Objekt der Klasse Point2D**
  - **eine Referenz auf das nächste Listenelement**




- Zusätzlich merken wir uns zwei Verweise **head** und **tail** auf das erste und letzte Listenelement.
  - Dadurch können schnell am Anfang und am Ende neue Punkte eingefügt werden.

# Umsetzung

- Zur Implementierung von Listen werden zwei neue Klassen implementiert.
  - Eine Klasse ListElement für die Listenelemente.
  - Eine Klasse LinkedList zur Verwaltung der Liste. Diese enthält:
    - Datenfelder *head* und *tail*
    - *Konstruktoren*
    - Methoden von Folie 567
    - Weitere Methoden wie z. B. equals und toString
- Für die Nutzung der Liste wird noch die Klasse Point2D benötigt.
  - Diese Klasse liegt bereits vollständig implementiert vor.

# Klasse ListElement

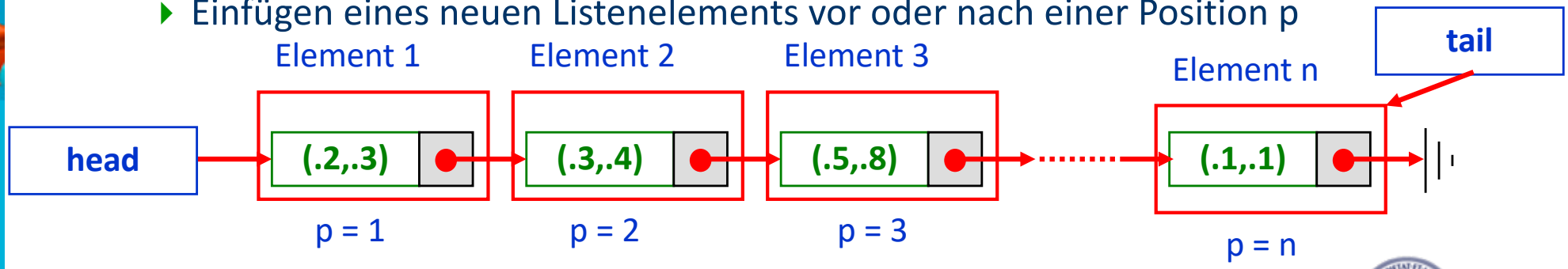
- Diese Klasse hat zwei Datenfelder:
  - Das Datenfeld *data* referenziert ein Objekt der Klasse Point2D.
  - Das Datenfeld *next* auf ein Objekt der Klasse ListElement
- Da innerhalb der Klasse wiederum auf Objekte der Klasse Bezug genommen wird, sprechen wir auch von einer rekursiven Klasse.
  - Viele der Algorithmen für Listen könnten deshalb auch rekursiv formuliert werden.
- Konstruktoren der Klasse
  - Einer der beiden Konstruktoren definiert ein neues Objekt vom Typ ListElement mit gegebenem data und null als next.
    - Erzeugung eines Listenelements ohne einen Nachfolger
  - Der andere Konstruktor definiert ein neues Objekt vom Typ ListElement mit gegebenem data und gegebenem Verweis next auf das nächste ListElement.
    - Erzeugung eines Listenelements mit Nachfolger

# Klasse ListElement

```
public class ListElement{
    protected Point2D data;
    protected ListElement next; // rekursive Klasse

        /**
         * Konstruktor der Klasse mit einem Parameter vom Typ Point2D
         * @param Objekt der Klasse Point2D, auf das Datenfeld data referenziert.
         */
         ListElement(Point2D in){
             this(in, null);
         }

        /**
         * Konstruktor der Klasse
         * @param in Objekt der Klasse Punkt, das in dem ListElement referenziert
         *           werden soll.
         * @param ref Ein gültiger Verweis auf ein ListElement.
         */
         ListElement(Point2D in, ListElement ref){
             next = ref;
             data = in;
         }
}
```

# Klasse LinkedList

- Diese Klasse enthält:
  - Die Verweise `head` und `tail`.
  - Konstruktoren
  - Methoden
    - Einfügen (am Ende), Löschen, Suchen, Prüfen auf leer, Liefern des ersten Elements
    - toString, um eine Liste einfach auszugeben.
  - Optional könnte man noch weitere Methoden implementieren
    - Ermitteln der Listenlänge
    - Einfügen eines neuen Listenelements vor oder nach einer Position p

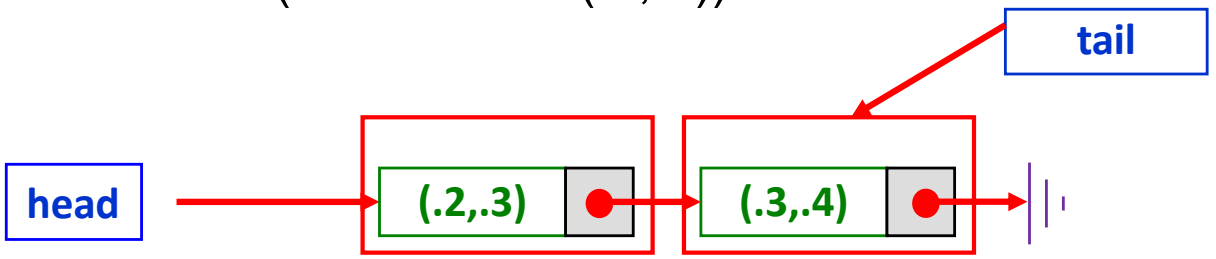


# Einfügen eines neuen Punkts

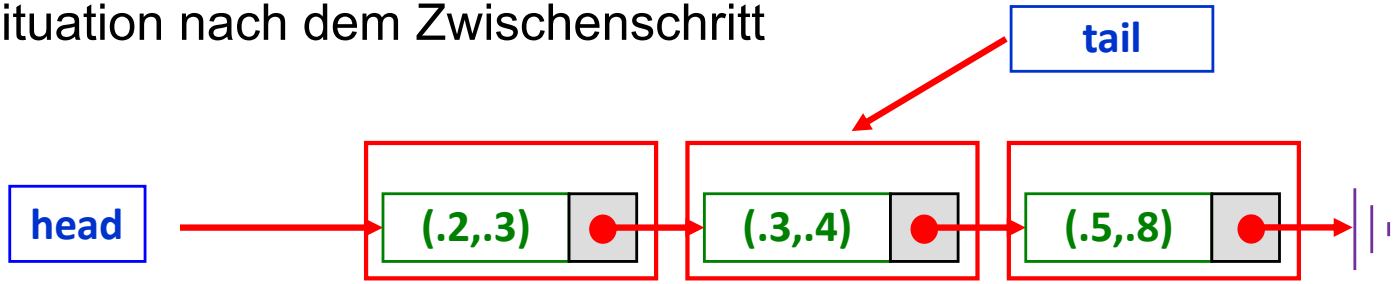
```
public class LinkedList {

        private ListElement head, tail;

        LinkedList() {
                head = tail = null;
        }

        /** Fügt einen neuen Punkt in die Liste ein.
         *  @param p der neue Punkt
         *  @return ob der Punkt erfolgreich eingefügt wurde.
         */
        public boolean add(Point2D p) {
                if (p == null) return false;                    // Solche Punkte nicht!
                if (head == null)                               // Liste ist leer
                        head = tail = new ListElement(p);
                else {                                          // Liste ist nicht leer
                        tail.next = new ListElement(p);         // Einfügen am Ende
                        tail = tail.next;
                }
                return true;
        }

        ...
}
```

# Beispiel

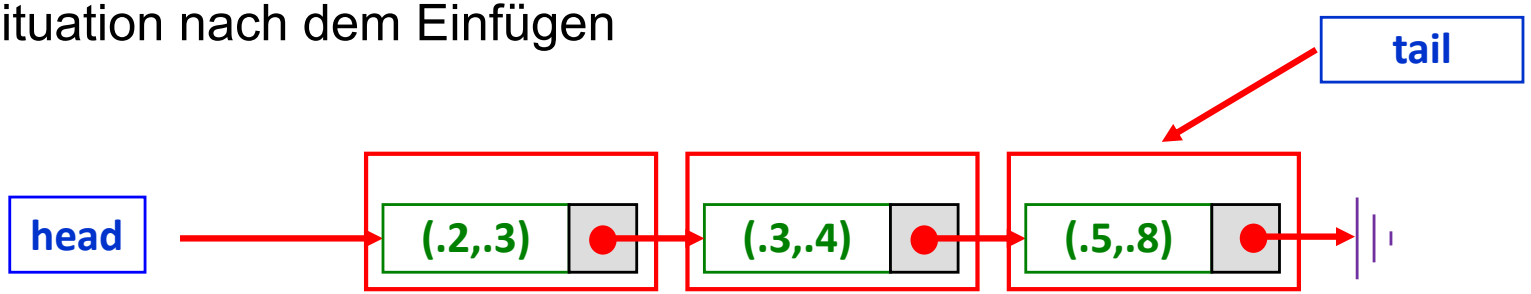
- Aufruf von add(new Point2D(.5,.8)) auf einer Liste mit 2 Elementen.




- Situation nach dem Zwischenschritt




- Situation nach dem Einfügen



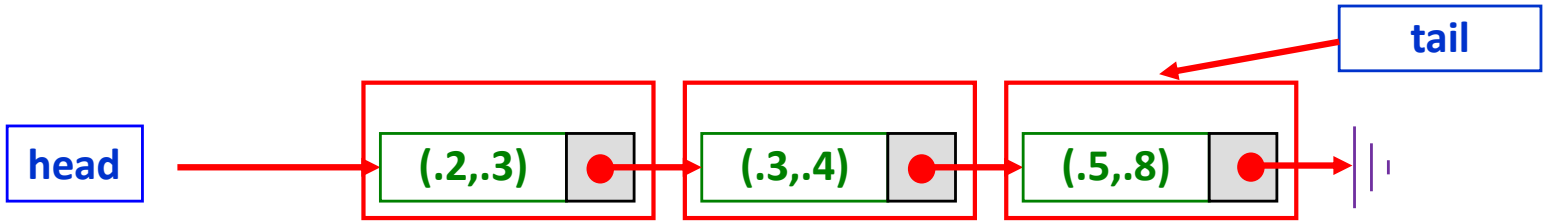
# Löschen eines Punkts

- Für einen gegebenen Punkt p wird nur das erste Listenelement mit einem Punkt q, der q.equals(p) erfüllt, gelöscht.
- Beim Durchlauf durch die Liste werden prev und cur verwendet.

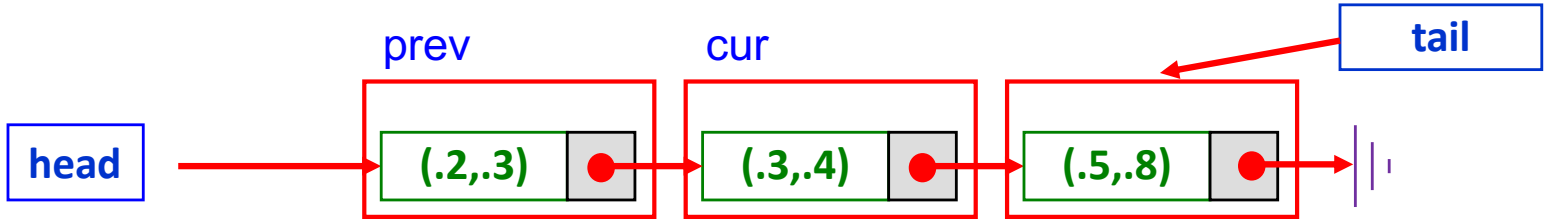
```
public boolean remove(Point2D p) {
        ListElement prev = null;
        for (ListElement cur = head; cur !=null; cur = cur.next) {
                if (p.equals(cur.data)) {                    // Überprüfe auf Gleichheit
                        if (prev == null)                    // Vorgänger existiert nicht
                                if (head == tail)            // Liste hat ein Element
                                        head = tail = null;
                                else                          // Löschen des 1. Elements
                                        head = head.next;
                        else
                                prev.next = cur.next;        // Freigeben des Speichers
                        return true;                         // Löschen erfolgreich
                }
                prev = cur;                                  // Element p nicht gefunden
        }                                                    // Anpassen von prev
        return false;                                        // Kein Element gelösc
}
```

# Beispiel

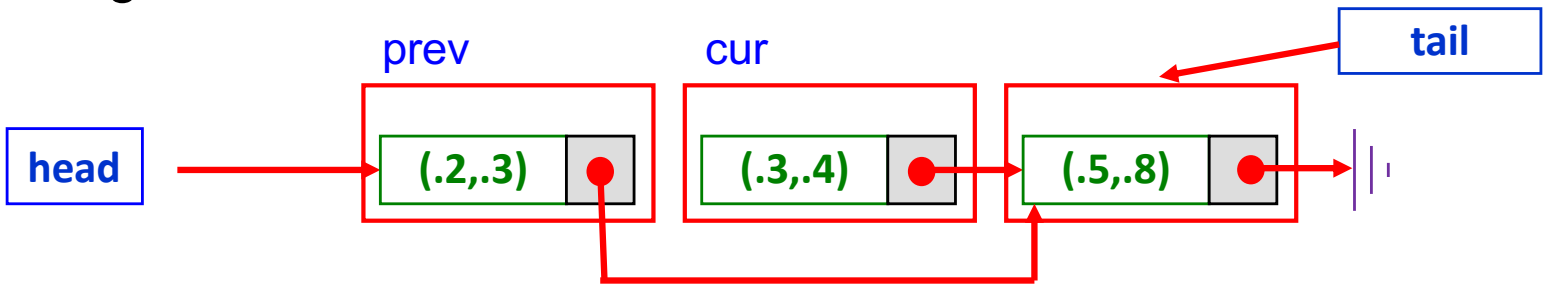
- Aufruf von remove (new Point2D(.3,.4)) auf folgender Liste.




- Situation nach dem Finden des zu löschenden Elements

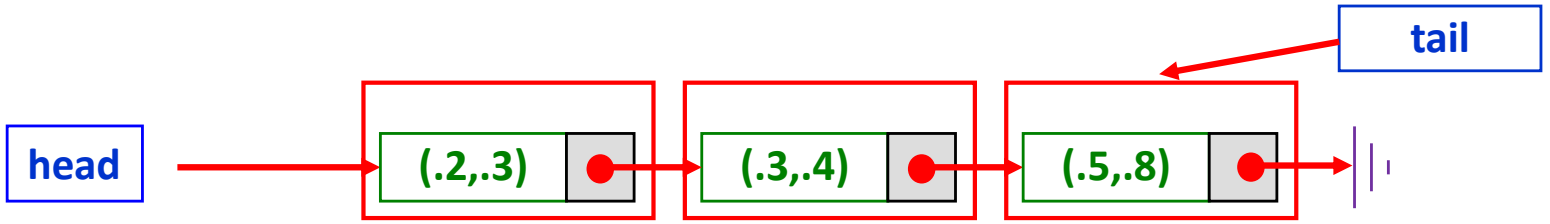



- Umhängen des Verweises

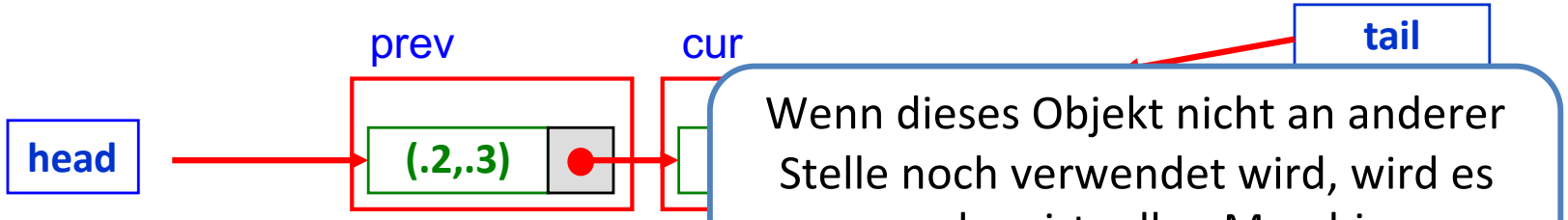


# Beispiel

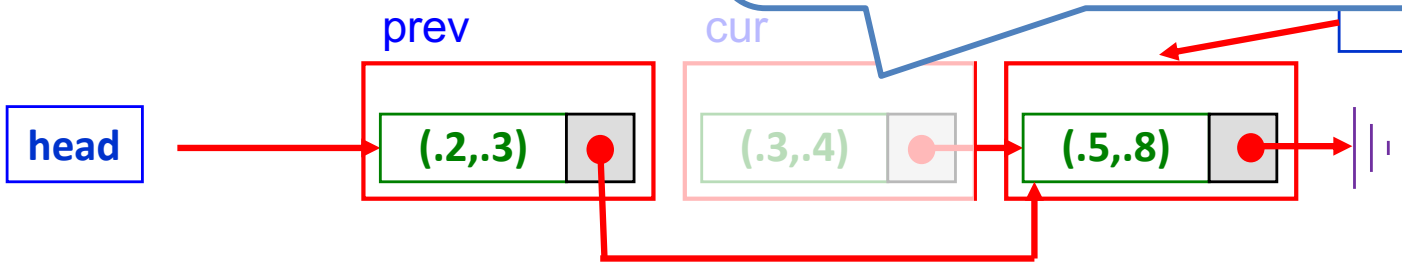
- Aufruf von remove (new Point2D(.3,.4)) auf folgender Liste.




- Situation nach dem Finden des zu löschenden Elements



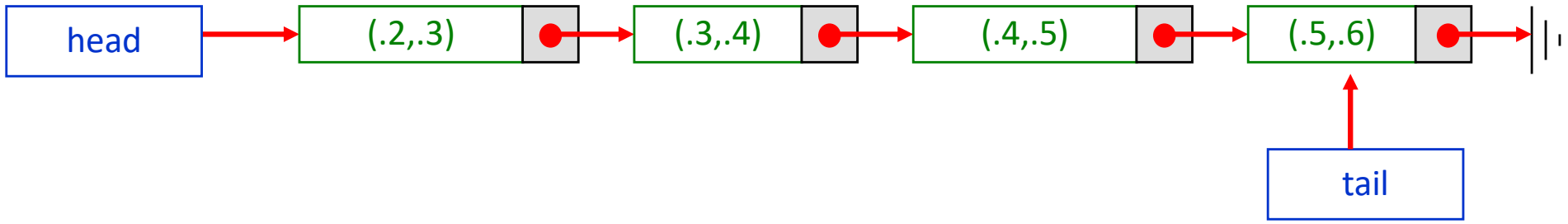

- Umhängen des Verweises



# Verwendung einer LinkedList

- Zunächst erzeugen wir eine Liste, dann fügen wir verschiedene Punkte ein:

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

    System.out.println("Liste " + list); // toString sollte überschrieben sein.
    System.out.println("Gefunden: " + list.contains(new Point2D (.3,.3)));
    System.out.println("x des 1. Punkts:" + list.get(0).getX());
}
```



# Diskussion

- Die Klasse LinkedList erfüllt nur die Anforderungen für eine Liste mit Objekten der Klasse Point2D.
- Die Implementierung ist nicht generisch genug.
  - Soll eine Liste für die Klasse Konto erstellt werden, müsste eine komplett neue Liste implementiert werden.
- Erster Lösungsvorschlag des Problems
  - Ersetze in der bisherigen Implementierung die Klasse Point2D durch die Klasse Object.
    - Damit kann die Implementierung für die Verwaltung von Objekten beliebiger Klassen verwendet werden.
  - Was wäre der Nachteil dieser Lösung?

# 12.2 Generische Listen mit Object

- Im Folgenden soll die zuvor genannte generische Lösung einer Liste vorgestellt werden.
  - *Verwendung der Klasse Object (statt Point2D) als Typ für das Datenfeld data.*
- Dadurch ergeben sich folgende Änderungen in der Klasse ListElement.

```
class ListElement{
    protected Object data;
    protected ListElement next;

        /** siehe oben */
         ListElement(Object in){
             this(in, null);
         }

        /** siehe oben */
         ListElement(Object in, ListElement ptr){
             next = ptr;
             data = in;
         }
}
```

# Einfügen eines Objekts in LinkedList

```
public class LinkedList implements DynamicSet {
        // Das Interface DynamicSet muss ebenfalls angepasst werden.
        private ListElement head, tail;

        // Hier müssen noch Konstruktoren eingefügt werden.

        /** siehe oben */
        public boolean add(Object p) {
                if (p == null) return false;
                if (head == null)
                        head = tail = new ListElement(p);
                else {
                        tail.next = new ListElement(p);
                        tail = tail.next;
                }
                return true;
        }
        /** Löschen eines Elements aus der Liste */
        public boolean remove(Object e) {…}

        /** Suchen nach einem Element in der Liste */
        public boolean contains(Object e) {…}

        ...
}
```

# Ausgabe eines Elements

```
public class LinkedList implements DynamicSet {
        private ListElement head, tail;

        // Hier müssen noch Konstruktoren eingefügt werden.
        /** siehe oben */
        public boolean add(Object p) { … }
        /** Löschen eines Elements aus der Liste */
        public boolean remove(Object e) {…}
        /** Suchen nach einem Element in der Liste */
        public boolean contains(Object e) {…}

        /** Prüft, ob eine Liste leer ist. */
        public boolean isEmpty() {
            return head == null;
        }

        /** Liefert das i-te Element aus der Liste */
        public Object get(int i) {
           if ((i < 0) || i >= size()) throw new IndexOutOfBoundsException();
           ListElement cur = head;
           for (int j = 0; j < i; j += 1)
               cur = cur.next;
           return cur.data;
        }
        ...
}
```

# LinkedList von Object

- Erster Lösungsvorschlag des Problems
  - Ersetze in der bisherigen Implementierung die Klasse Point2D durch die Klasse Object.

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

    System.out.println("Liste " + list); // toString s   te überschrieben sein.
    System.out.println("Gefunden: " + list.contains(new oint2D (.3,.3)));
    System.out.println("x des 1. Punkts:" + list.get(0).getX());
}
```

Das ist nicht erlaubt, da getX() nicht in Object deklariert ist.

# Nachteile der generischen Lösung

- Es kann nicht sichergestellt werden, dass eine Liste nur Objekte einer Klasse enthält.

  - Prinzipiell wäre es möglich, eine gemischte Liste mit Objekten der Klasse Konto und der Klasse Point2D zu erstellen.

- Die Methode get(int) liefert als Ergebnis nur Objekte der Klasse Object, auch wenn zuvor Objekte der Klasse Point2D eingefügt wurden.

  - Zusätzlich müssen diese Objekte mit einem *Down-Cast* noch in die entsprechende Klasse zurücktransformiert werden.
  - Dies funktioniert nur dann, wenn sich in der Liste nur Objekte einer Klasse befinden.

# 12.3 Generische Listen mit parametrisierten Typen

- Seit Java 5.0: parametrisierte Datentypen, meist generische Datentypen genannt.
  - Man spricht auch von parametrischer Polymorphie.
- Generische Datentypen können von einem oder mehreren Typ-Parametern abhängen.
- Es können sowohl Klassen als auch Schnittstellen auf diese Weise definiert werden.
  - Dadurch werden unsere Probleme bei der bisherigen generischen Implementierung einer Liste gelöst.

# Definition und Anwendung von generischen Datentypen

- In Java werden Typ-Parameter in spitzen Klammern nach dem Klassen- bzw. Schnittstellennamen angegeben.
  - Beispiele
    - interface List<E> { … }
    - class LinkedList<E> { … }
  - Ähnlich zu Parametervariablen in Methoden wird dadurch einer Klasse bzw. einer Schnittstelle ein Parameter übergeben. Der Unterschied ist, dass es sich dabei um einen Datentyp (Klasse) handelt.

# Definition und Anwendung von generischen Datentypen

- Innerhalb der Schnittstellen und Klassen kann der Typparameter (E) als (fast) normaler Datentyp verwendet werden, z.B.
  - Als Typ von Variablen und Parametern
  - Als Rückgabetyp
  - Als Typparameter
  - **ABER NICHT**: für Konstruktoraufrufe oder in der extends oder implements Klausel
- Erst bei der Deklaration von Variablen und der Erzeugung der Objekte einer generischen Klasse muss die konkrete Klasse für den Datentyp E festlegt werden.
  - Man spricht dann von einer Instanziierung des generischen Datentyps.

```
LinkedList<Konto> list = new LinkedList<Konto>();
```

  - Dadurch wird der Typ E in der Klasse LinkedList durch die Klasse Konto ersetzt.

# Modifizierte Klasse

Oberklasse kann auch generisch sein! Typ-Parameter kann in der Klassenerweiterung verwendet werden. Typparameter wird wieder als Typparameter verwendet.

```
class ListElement<E> implements List<E> {
    protected E data;
    protected ListElement<E> next;

        /**
         * Konstruktor der Klasse
         * @param in Objekt des Elementtyp E, das in dem ListElement
         *           referenziert werden soll.
         */
        ListElement(E in){
                this(in, null);
        }

        /**
         * Konstruktor der Klasse
         * @param in Objekt des Typs E, das in dem ListElement
         *           referenziert werden soll.
         * @param ptr Ein gültiger Verweis auf ein ListElement
         */
    ListElement(E in, ListElement<E> ptr){
         next = ptr;
         data = in;
    }
}
```

# Verwendung generischer Klassen

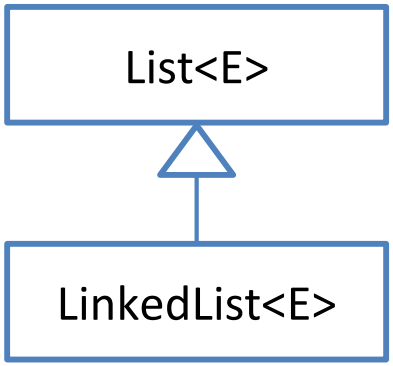
- Wie wir gesehen haben können in unserem Beispiel die Klasse und Schnittstelle relativ einfach generisch gemacht werden.
  - Zu beachten ist, dass bei der Klasse LinkedList<E> die Schnittstelle List<E> implementiert wird.
- Bevor die Details erläutert werden, wird zunächst die Verwendung einer generischen Klassen vorgestellt.
  - Erstellung einer Liste mit Objekten der Klasse Point2D

# Beispiel: Liste mit Punkten

## Vorteile

- In die Punktliste können nur Punkte eingefügt werden. Versucht man ein Objekt der Klasse Konto einzufügen, wird der **Compiler** ein Fehler melden.
- Beim Lesen eines Objekts der Klasse Point2D aus der Liste wird keine Cast-Operation mehr benötigt. Wir bekommen das Objekt mit der gewünschten Klasse zurück.
- Die Verwendung von generischen Klassen erfordert wenig Aufwand.

```
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));

        System.out.println("Liste " + list);
        System.out.println("Gefunden: " +
                list.contains(new Point2D (.3,.3)));
        System.out.println("x des 1. Punkts:" +
                list.get(0).getX());
}
```




Das ist OK: wir wissen, dass Element ein Point2D Sein muss.

# Beispiel: Liste mit Punkten

## Vorteile

- In die Punktliste können nur Punkte eingefügt werden. Versucht man ein Objekt der Klasse Konto einzufügen, wird der **Compiler** ein Fehler melden.
- Beim Lesen eines Objekts der Klasse Point2D aus der Liste wird keine Cast-Operation mehr benötigt. Wir bekommen das Objekt mit der gewünschten Klasse zurück.
- Die Verwendung von generischen Klassen erfordert wenig Aufwand.

```
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));
        list.add(new String("..."));
}
```

Gibt bereits beim Einfügen einen Fehler.