Datenstrukturen und Abstrakte Datentypen

- Abstrakter Datentyp
- Idee der sequentiellen Struktur
- Einfach verkettete Liste
- Iteratorkonzept

Definition:

Ein **abstrakter Datentyp (ADT)** ist eine Menge von (unveränderlichen) Objekten, deren Eigenschaften vollständig durch eine endliche Menge von Operationen über diesen Objekte definiert sind. Die Realisierung der Objekte und der Operationen sind das "Geheimnis" des ADT.

Beispiele

- Klasse `java.lang.String` (hat der ein \0-Byte?)
- Klasse `java.math.BigDecimal` (wie ist das intern dargestellt?)

Bemerkungen

- Ein ADT kann unterschiedlich realisiert werden.
- Beschreibung des Verhaltens ist nicht ganz einfach.
- Der Begriff ADT stammt ursprünglich aus der funktionalen Programmierung..
- OOP übernimmt die Forderung nach abstrakter Schnittstelle

Ziel: Programmieren als Erweiterung des Typsystems.

Ein objektorientiertes System besteht aus Klassen, die reale Objekte darstellen und bei deren formaler Beschreibung vordefinierte und benutzerdefinierte Datentypen Verwendung finden.

Ideal verhalten sich neue Datentypen wie erwartet:

z.B. Scala-Klasse `Bruch`:

```
val a = Bruch(3, 4)
val b = Bruch(5, 6)
val c = a * b + a
println(b + Bruch(1,2))
```

Vorteile:

- Einfache Beschreibung des Verhaltens.
- Einfacheres Testen durch Zerlegung der Software in unabhängige Komponenten.
- Weiterentwicklung der Realisierung möglich, so lange die Schnittstelle unverändert bleibt.

Wie entwickelt man vernünftige Schnittstellen

- Keine darstellungsabhängigen Begriffe und Größen verwenden.
- Einfaches Verhalten bevorzugen.
- Nach Möglichkeit sollten immer alle Operationen anwendbar sein.
- Das Verhalten der Operationen sollte den Benutzererwartungen folgen.
- Wenn möglich, sollten bekannte syntaktische Formen benutzt werden (syntaktische Stetigkeit).
- Schnittstellen, die man nicht gut dokumentieren kann, sind meist schlecht.
- Das Konzept der Java-Interfaces ermöglicht eine syntaktische Formulierung von Schnittstellen (aber ist nicht dasselbe).

Klasseninvariante haben zunächst nichts mit Schnittstellen zu tun, ermöglichen aber eine korrekte und angemessene Implementierung (Anwendbarkeit von Operationen).

Sequenzielle Datenstrukturen (Listen)

Beispiel: a_0=1, a_1=2, a_2=4, a_3=8, ... $a_i = 2^i$

Eine Sequenz ist eine Ansammlung von Datenelementen, die in einer bestimmten (linearen) Reihenfolge stehen.

Mögliche Realisierungen:

1. Array: sehr **effizient**, platzsparend, vergänglich, starr.

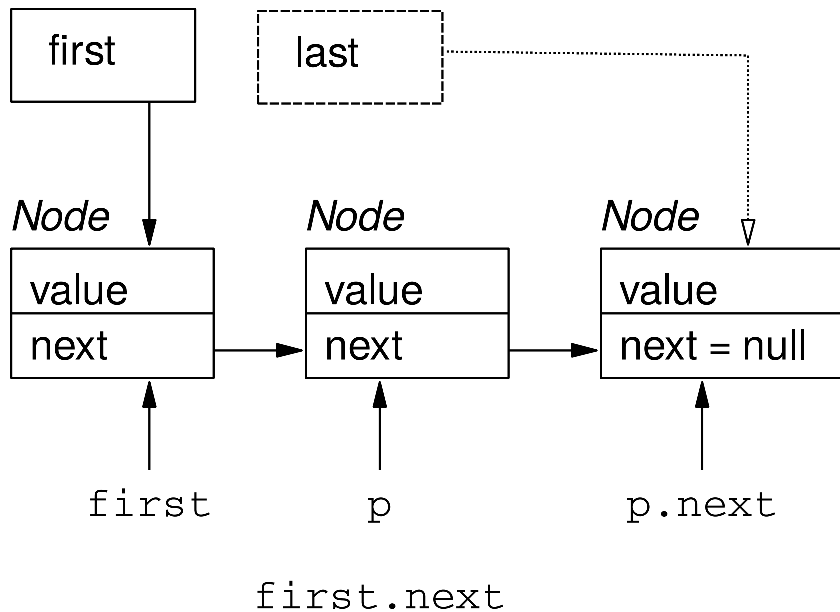
2. Datei: langsam, platzsparend, **unvergänglich**, relativ starr.

3. Verkettete Liste: relativ effizient, platzsparend, vergänglich, **flexibel.**

- Größe der Datenstruktur sollte dynamisch veränderbar sein.
- Änderungen in der Reihenfolge sollten einfach möglich sein.
- **Wenn wir alle Operationen von Sequenzen aufführen, haben wir einen abstrakten Datentyp** (vgl. java.util.List)
- hier geht es aber zunächst um eine besondere Implementierung

Idee der verketteten Liste

- **Array:** Reihenfolge wird implizit durch Anordnung im Speicher festgelegt.
- **Verkettete Liste:** Reihenfolge wird explizit festgelegt.
- **Minimales Konzept:** einfach verkettete Liste.
- (Das Ende kann ganz auf vielerlei Art markiert werden.)
- (*und die Variablen können natürlich auch anders heißen!*)




```
// Traversierung
// (Invariante ?)
Node p  =  first;
while (p != null) {
    ...
    p =  p.next;
}
// (Ziel ?)
```

Vergleich von Array und verketteter Liste

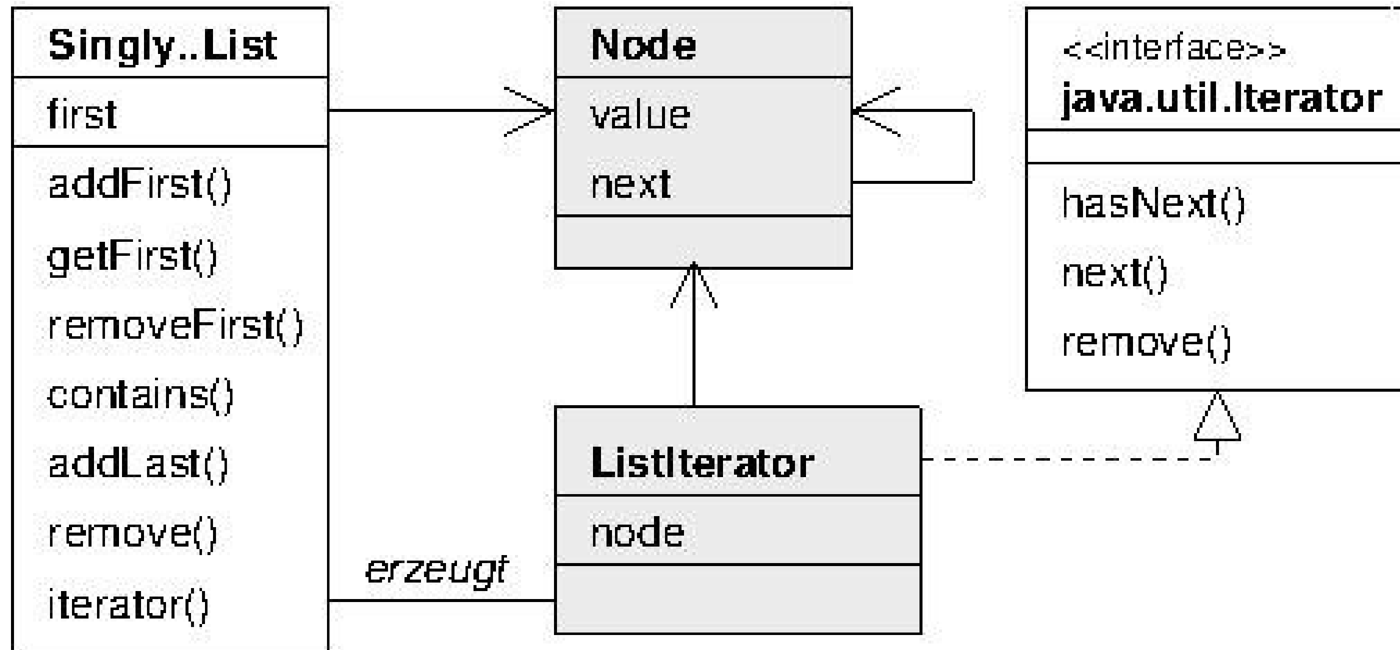
	Array	**verkettete Liste**
n-tes Element	**O(1)**	O(n)
Vergrößern	O(n)	**O(1)**
ein Element löschen	O(n)	**O(1)**
ein Element einfügen	O(n)	**O(1)**
Suchen	O(n)	O(n)
Sortieren	O(n log n)	O(n log n)

spezielle Array-/Listenstrukturen (z.B. binäre Suche)
können besseres Verhalten zeigen

Einfache Schnittstelle für die Listenklasse

```
public class SinglyLinkedList<T> {
    // 1. Konstruktor und leere Liste
    SinglyLinkedList()
    boolean isEmpty()

    // 2. Manipulationen an erster Stelle
    void addFirst(T obj)
    T getFirst()
    T removeFirst()

    // 3. Suchen und Ändern an unterschiedlicher Stelle
    boolean  contains(Object obj)
    void addLast(T obj)
    boolean remove(T obj)

    // 4. Das Iteratorkonzept
    T iterator()
}
```

Klassendiagramm für das Konzept der verketteten Liste

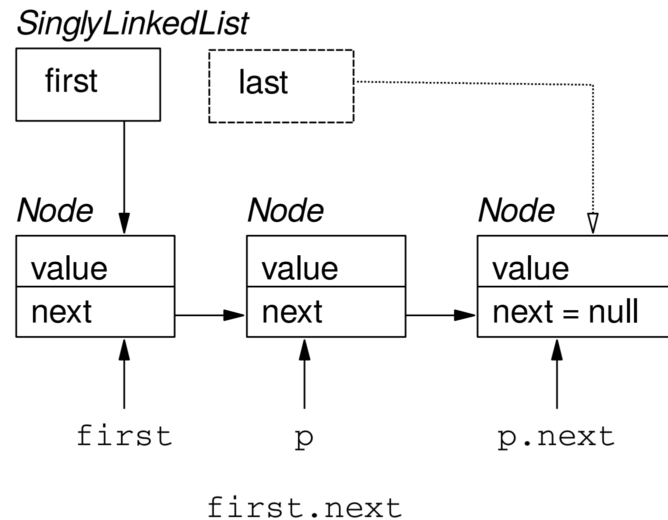


SinglyLinkedList

Interne Struktur der Listenklasse

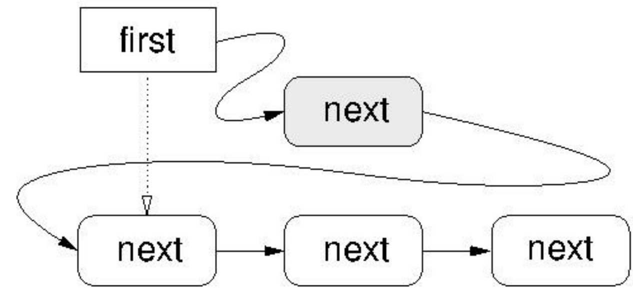
Klasseninvariante:
Es gilt `first == null` oder `first` enthält die Referenz des ersten Listenknotens. Dessen `next`-Feld enthält die Referenz des darauffolgenden Knotens. Der letzte Knoten ist dadurch gekennzeichnet, dass sein `next`-Feld gleich `null` ist.

```
class Node<T> {
   Node(T value, Node<T> next) {
        this.value = value;
        this.next = next;
    }
    T value;
    Node<T> next;
}

Node<T> first = null;

public SinglyLinkedList() {
}

public boolean isEmpty() {
    return first == null;
}
```



Operationen am Anfang der Liste

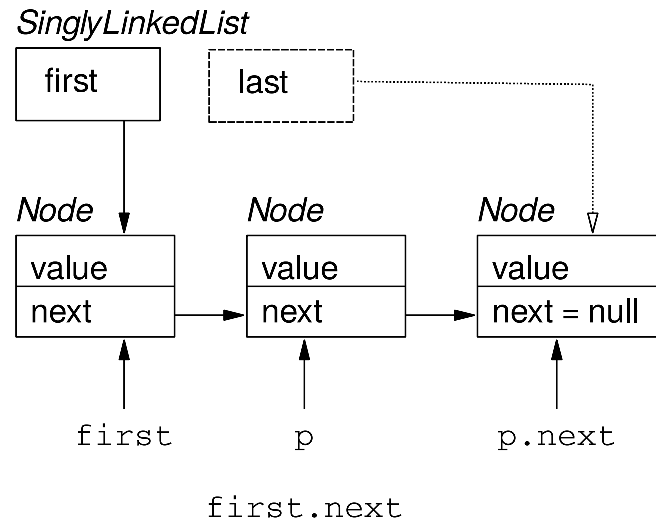
```
public void addFirst(T obj) {
    first = new Node<T>(obj, first);
}

public T getFirst() {
    if (first == null)
        throw new NoSuchElementException();
    return first.value;
}

public T removeFirst() {
    if (first == null)
        throw new NoSuchElementException();
    T result = first.value;
    first = first.next;
    return result;
}
```



Traversieren (= Durchlaufen) einer verketteten Liste

```
public boolean contains(Object obj) {
    Node<T> p = first;
    while (p != null && !p.value.equals(obj))
        p = p.next;
    return p!= null;
}
```

```
// Schema der
// Traversierung
Node<T> p = first;
while (p != null) {
    ...
    p = p.next;
}
```

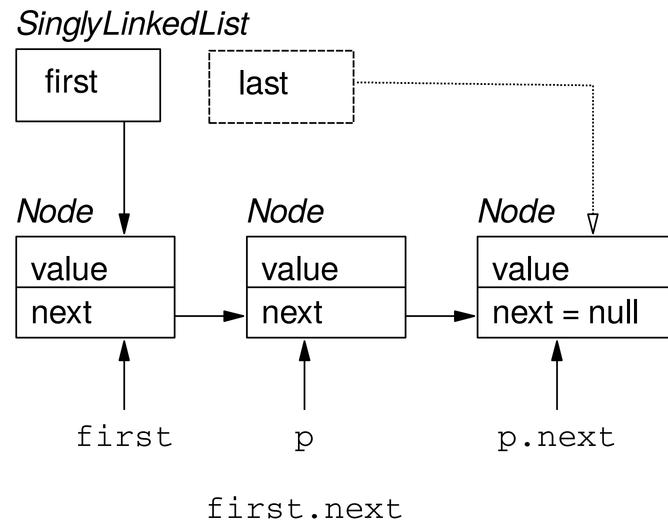


Anhängen am Ende einer verketteten Liste (1)

```
public void addLast(T obj) {
    // letzten Knoten suchen
    Node<T> p = first;
    while (p != null) p = p.next;
    // und jetzt ???
}
```

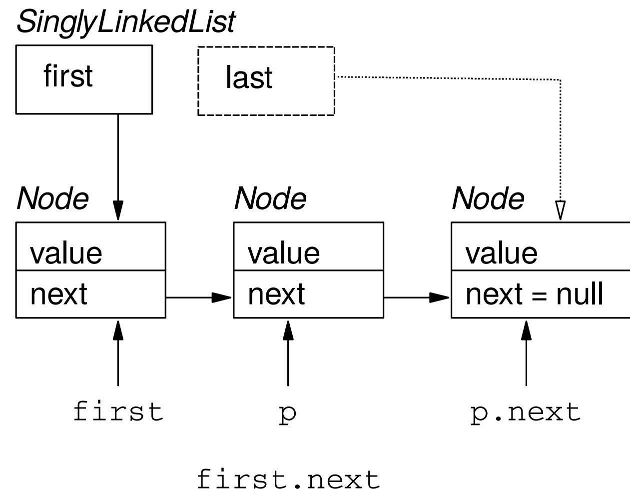
Idee:

```
Node<T> p = first; // Vorsicht: ist p != null?
while (p.next != null) p = p.next;
```



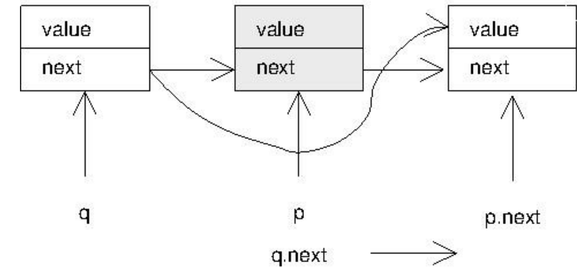
Anhängen am Ende einer verketteten Liste (2)

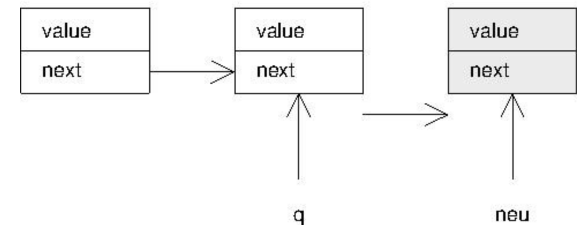
```
public void addLast(T obj) {
    // neuen Knoten erzeugen
    Node<T> newNode = new Node<T>(obj, null);
    if (first == null) {
        // noch kein Element vorhanden
        // somit wird dies der erste Eintrag
        first = newNode;
    }
    else {
        // letzten Knoten suchen
        Node<T> tempNode = first;
        while (tempNode.next != null)
            tempNode = tempNode.next;
        // ans Ende anhaengen
        tempNode.next = newNode;
    }
}
```



Anhängen am Ende einer verketteten Liste (3)

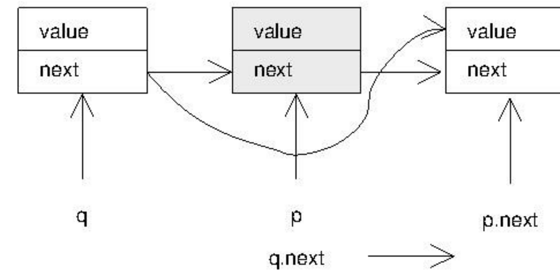
```
public void addLast(T obj) {
    // neuen Knoten erzeugen
    Node<T> newNode = new Node<T>(obj, null);
    if (first == null) {
        // noch kein Element vorhanden
        // somit wird dies der erste Eintrag
        first = newNode;
    }
    else {
        // letzten Knoten suchen
        Node<T> q = null;
        Node<T> p = first;
        while (p != null) {
            q = p;
            p = p.next;
        }
        // ans Ende anhaengen
        q.next = newNode;
    }
}
```

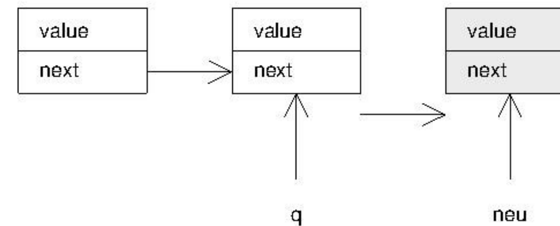




Löschen in einer verketteten Liste

```
public boolean remove(T obj) { // return true, wenn gefunden
    Node<T> p = first;
    Node<T> q = null;
    while (p != null && !p.value.equals(obj)) {
        q = p;
        p = p.next;
    }
    if (p == null)
        return false;
    else {
        if (p == first)
            first = first.next;
        else
            q.next = p.next;
        return true;
    }
}
```





Löschen des index-ten Elements

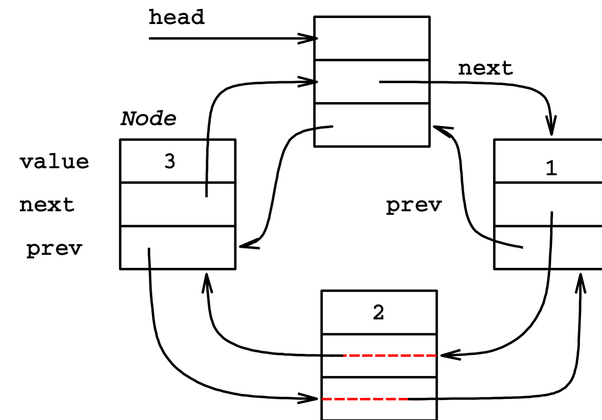
In Klasse List: `private int size = 0;`

```
public void remove(index) {
    if (index < 0 || index >= size)
        throw new IndexOutOfBoundsException();
    if (index == 0) {
        first = first.next;
    else {
        Node<T> p = first;
        for(int i = 1; i<index; i++)
            p = p.next;
        p.next = p.next.next;
        size--;
    }
}
```

Komfort = größerer Aufwand – keine Sonderfälle

- Zirkuläre Listen haben keine Sonderfälle
- Doppelt verkettete Listen machen Veränderungen einfacher
- Es entsteht mehr Speicheroverhead

```
public void remove(Object obj) {
    Node p = head.next;
    while (! obj.equals(p.value) p = p.next;
    if (p != head) {
        p.prev.next = p.next;
        p.next.prev = p.prev;
    }
}
```



Die funktionale Programmierung
verwendet auf der anderen Seite eine sehr effiziente
Form der einfach verketteten Liste (unveränderliche Liste)
(s. Praktikum 7: `util.FList`)

Beispiel (Einlesen von Zahlen und Mittelwert):

```
public static SinglyLinkedList<Integer> einlesen() {
    SinglyLinkedList<Integer> liste = new SinglyLinkedList<Integer>();
    while(true) {
        System.out.print("Bitte geben Sie eine Zahl ein (0 = Ende)");
        int n = scanner.nextInt();
        if (n == 0) return liste;
        liste.addLast(n);
    }
}
public static double mittelwert(SinglyLinkedList<Integer> liste) {
    int summe = 0, anzahl = 0;
    while (!liste.isEmpty()) {
        summe += liste.removeFirst();
        anzahl++;
    }
    return (double)summe / (double)anzahl;
}
public static void main(String[] args) {
    SinglyLinkedist<Integer> lst = einlesen();
    System.out.println("Mittelwert1: " + mittelwert(lst));
}
```

Besseres Verfahren

Der Nachteil des Beispiels ist, dass die Liste zerstört wird. Wir benötigen eine Methode zum zerstörungsfreien Bearbeiten der ganzen Liste.

1. Offenlegen der Datenstruktur `Node` und der Variablen `first`, `next` und `value` ermöglicht es, von außen ganz bequem alle gewünschten Operationen durchzuführen.

2. Wir bauen die Operation *Mittelwert* (und alle anderen benötigten Operationen) direkt in die Klasse `SinglyLinkedList` ein.

3. Wir kopieren die Liste in ein Array (Methode `toArray()`).

4. Wir überlegen uns ein Verfahren, das:
   a) das Geheimnisprinzip beibehält.
   b) keine ständigen Veränderungen der Klasse `SinglyLinkedList` erfordert.
   c) nacheinander den Zugriff auf die einzelnen Listenelemente ermöglicht.

Lösung: Iteratorkonzept
in Java:
`java.util.Enumeration` (veraltet, nicht zu verwechseln mit **`enum`**!)
`java.util.Iterator`
`java.util.ListIterator`

Schnittstelle Iterator

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

(Ohne Typparameter: T == Object ==> casts!)

Die Schnittstelle `Iterable` sollte von Container-Klassen implementiert werden, um die for-each Schleife zu ermöglichen:

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

Die Anwendung des Iterator-Konzepts

```
public static double mittelwert2(SinglyLinkedList<Integer> liste) {
    int summe = 0, anzahl = 0;
    for (Iterator<Integer> iter = liste.iterator(); iter.hasNext();) {
        summe+= = iter.next();
    }
    return (double)summe / (double)liste.size();
}
```

`Iterator` deklariert auch die Funktion `remove()`, erlaubt aber, dass diese nur eine `UnsupportedOperationException` wirft..

Wenn die Klasse `SinglyLinkedList` die Schnittstelle `Iterable` deklariert:

```
    SinglyLinkedList<Integer> liste;
    . . .
    for (int i : liste) summe += i;
```

Anmerkung

Der klassische Programmierstil sieht so aus:

- grundlegenden Programmiersprachen sind sehr technisch gehalten und erwarten, dass der Programmier sich auch um Details der Ausführung kümmmert.
- anwendungsbezogenene Sprachen (SQL, ABAP, PHP) sind auf einer problemorientierten Ebene angesiedelt, aber im Hinblick auf Effizienz und Anwendbarkeit eingeschränkt
- modernere Sprachen, wie Java, bieten gute Bibliotheken und mäßige, anwendungsfreundliche Erweiterungen (for-each, garbage collection, enum)

Der Trend geht dahin:

- dass der Compiler den Programmierer bei der Optimierung des Programms ablöst
- dass ein lesbares Programm einem technisch optimierten Programm vorgezogen wird
- dass Programmiersprachen wenige mächtige High-Level Konstrukte und eine bequeme Erweiterbarkeit bieten
- dass Sprachen die Implementierung von DSLs (anwendungsspezifische Sprachen) fördern

Historie:
Maschinencode → Assembler → C → Java → ???

Wie funktioniert die for-each-Schleife?

a) bei Arrays:

`for (T x : a)` ergibt:

```
for (int i = 0; i < a.length; i++) {
    T x = a[i];
    …
}
```

b) wenn `Iterable<T>` implementiert wurde:

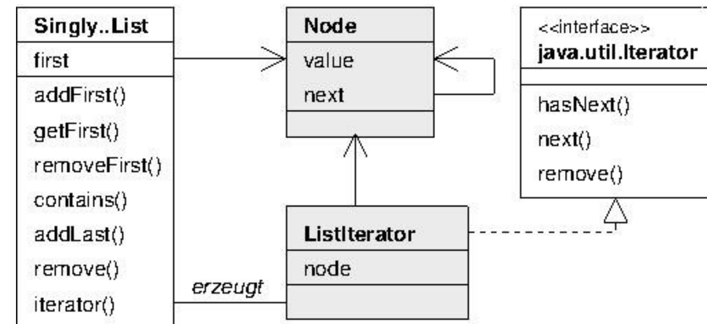
```
public interface Iterable<T> {
    public Iterator<T> iterator();
}

class Container<T> implements Iterable<T> { ...
    public Iterator<T> iterator() { ...
```

`for (T x : c)` ergibt:

```
for (Iterator<T> iter = c.iterator(); iter.hasNext();) {
    T x = iter.next();
    …
}
```

c) wenn weder Array noch Iterable => kein for-each !

```java
class ListIterator<T>  implements Iterator<T> {
    private Node<T> node;      // INV: zeigt auf den nächsten Knoten

    ListIterator(Node<T> firstNode)   {
        node   = firstNode;    // in innerer Klasse wäre first bekannt!
    }

    public T next() {
        if (node == null) throw new NoSuchElementException();
        T result = node.value;
        node = node.next;
        return result;
    }

    public boolean hasNext() {
        return node != null;
    }

    public void remove() {
        // damit's nicht zu kompliziert wird:
        throw new UnsupportedOperationException();
    }
}
```



Ausblick

Im Trend hin zu Funktionaler Programmierung werden anstelle von expliziter Wiederholung vermehrt Funktionen höherer Ordnung verwendet. Dies unterstützt auch die optimierte Ausführung in Multicorearchitekturen (s. Java 8)

Beispiel: *Mittelwert der Funktionswerte von f über einer Datenliste xs.*

Java:

```
double sf = 0.0;
for (double x : xs) sf += f(x);
double m = sf / xs.size();
```

Scala:

```
val m = xs.map(f).sum / xs.size
```

Es ist zu beachten, dass die konkrete Form der Datenstruktur in beiden Varianten keine Rolle spielt!