

EP2-Theorie 2.Test

Vorwort

Diese Stoffsammlung/Zusammenfassung enthält den Stoff, der in der EVC Vorlesung der TU Wien im Sommersemester 2025 vorgetragen wurde, der auch in den jeweiligen Skripten und Slides zu finden ist. Die Struktur dieser Zusammenfassung basiert demnach auch der des Skriptums.

Disclaimer

Vieles der Zusammenfassung wurde mit AI generiert, basiert allerdings nur auf Inhalte der Unterlagen. Die Stellen die mit AI generiert wurden, wurden von mir überprüft und mit den Unterlagen verglichen, aber auch ich kann Fehler machen.

Demnach, falls sich irgendwo Fehler befinden oder es Verbesserungsvorschläge gibt, bitte an [@xmozz](#) auf Discord wenden.

Inhaltsverzeichnis

7. Equals, hashCode, Hash-Tabelle
8. Abstraktionshierarchie, Iterator, Baumoperationen
9. Sortieren, Sichtweisen vs Kopien, Effizienz und Zuverlässigkeit
10. Garantie statt Zufall
11. IO über Streams, Validierung von Eingabedaten, DbC
12. Statisches Programm-Verstehen, Code-Review, Programmdokumentation
13. Testen Qualität Optimierung

7. Equals, hashCode, Hash-Tabelle

Gleichheit und equals()

- ep2-07_equals, hashCode, Hash-Tabelle, p.2|slides
- Skriptum, p.84

Die `equals(Object o)` Methode

- Sollte in den meisten Klassen überschrieben werden.
- Akzeptiert jedes Objekt oder `null` als Argument.
- Dient dem inhaltlichen Vergleich zweier Objekte (im Gegensatz zu `==` für Identitätsvergleich).
- Die Standardimplementierung in `Object` ist `{ return this == obj; }` (vergleicht Identität).
 - Implementierung in `Object`: `{ return this == obj; }`
- Meist unterscheidet sich inhaltliche Gleichheit von der Identität (`==`) → **Überschreiben ist notwendig**.
- Überschreiben notwendig, wenn inhaltliche Gleichheit von Identität abweichen soll.
- Die Definition von "inhaltlich gleich" ist nicht vorgegeben, meist:
 - Alle Objektvariablen haben gleiche Inhalte.
 - Organisatorische Variablen werden ignoriert.
 - Manchmal kommt es nur auf enthaltene Einträge an (z.B. bei Bäumen).

Beispielimplementierung (`equals` in `BoxedText` -Erweiterung)

- Vergleich auf Identität (`this == o`) zur Effizienzsteigerung.
- Prüfung auf `null` und den dynamischen Typ der anderen Objekts (`o.getClass() != BoxedText.class`).
- Typumwandlung (Cast) zu `BoxedText`.
- Vergleich relevanter Objektvariablen (`textHeight`, `textWidth`).
- Elementweiser Vergleich des `text`-Arrays.

Allgemeine Struktur beim Überschreiben von `equals`

- Parameter vom Typ `Object` kann `null` sein.
- Notwendig: `null`-Vergleich, Typvergleich, Typumwandlung.
- Vergleich mit `this` am Anfang zur Effizienz.
- Die ersten 6 Zeilen im Beispiel sind typisch.
- Der restliche Code hängt von den zu vergleichenden Attributen ab.

Eigenschaften von `equals` (Muss jede Implementierung erfüllen)

- Für `x != null` gilt `x.equals(null)` ist immer `false` (**keine Gleichheit mit null**).
 - **Daraus ableitbare Bedingung:** `x != null → !x.equals(null)`
- **Äquivalenzrelation:**
 - **Reflexiv:** Für `x != null` gilt `x.equals(x)` ist immer `true` (**Identität ⇒ Gleichheit**).
 - **Daraus ableitbare Bedingung:** `x != null → x.equals(x)`

- **Symmetrisch:** Für `x != null` und `y != null` liefert `x.equals(y)` dasselbe Ergebnis wie `y.equals(x)`.
 - **Daraus ableitbare Bedingung:** $x \neq \text{null} \& y \neq \text{null} \rightarrow x.equals(y) == y.equals(x)$
 - **Transitiv:** Wenn `x.equals(y)` und `y.equals(z)` beide `true` sind, dann ist auch `x.equals(z)` `true`.
 - **Daraus ableitbare Bedingung:** `x.equals(y) & y.equals(z) \rightarrow x.equals(z)`
 - **Konsistent:** Solange `x` und `y` nicht geändert werden, liefern wiederholte Aufrufe von `x.equals(y)` immer dasselbe Ergebnis (**keine Seiteneffekte**).
 - **Daraus ableitbare Bedingung:** `x` und `y` unverändert $\rightarrow x.equals(y) == x.equals(y)$
-

Die `equals(Object o)` Methode

- Sollte in den meisten Klassen überschrieben werden.
- Akzeptiert jedes Objekt oder `null` als Argument.
- Dient dem inhaltlichen Vergleich zweier Objekte (im Gegensatz zu `==` für Identitätsvergleich).
- Die Standardimplementierung in `Object` ist `{ return this == obj; }` (vergleicht Identität).
 - **Implementierung in Object:** `{ return this == obj; }`
- Meist unterscheidet sich inhaltliche Gleichheit von der Identität (`==`) \rightarrow **Überschreiben ist notwendig**.
- Überschreiben notwendig, wenn inhaltliche Gleichheit von Identität abweichen soll.

Typische Implementierung von `equals`:

```
public boolean equals(Object obj) {
    if (this == obj) {
        return true; // Optimierung, garantiert x.equals(x)
    }
    if (obj == null || getClass() != obj.getClass()) {
        return false; // dyn. Typen gleich, garantiert !x.equals(null)
    }
    NameOfThisClass that = (NameOfThisClass) obj; // Zugriff über 'that'
    ... // typspezifischer Vergleich von this und that
}
```

Was bedeutet „Gleichheit“? Entscheidende Fragen

- Unter welchen Bedingungen werden zwei Objekte als gleich angesehen?
- Welche Objektvariablen in den Vergleich einbeziehen?
- Die Antworten auf diese Fragen sind **nicht allgemeingültig** und hängen von der jeweiligen **Abstraktion** ab.
- Häufige Vorgehensweise:
 - Einbeziehung **aller Objektvariablen**, deren Werte (direkt oder indirekt) nach außen sichtbar werden könnten.

- Bei strukturierten Daten (z.B. Listen): Übereinstimmung der **Strukturen** (z.B. gleiche Reihenfolge).
 - **Eigene Ansätze notwendig, wenn Gleichheit keine gleiche Struktur voraussetzt:**
 - **Beispiel Suchbäume:** Zwei **Suchbäume** können als gleich angesehen werden, wenn sie die **gleichen Einträge** enthalten, auch wenn die Reihenfolge des Einfügens (und damit die Struktur des Baums) unterschiedlich war.
-

int hashCode()

- Gibt eine fast beliebige ganze Zahl zurück.
- Wird beispielsweise in Hash-Tabellen benötigt.
- **Implementierung in Object**: Leitet eine Zahl von der Speicheradresse des Objekts ab.
- **Bedingungen:**
 - Wenn `x.equals(y) true` ist, dann muss `x.hashCode() == y.hashCode()` gelten.
 - Solange das Objekt `x` unverändert bleibt, muss `x.hashCode()` immer denselben Wert zurückgeben (`x` unverändert \rightarrow `x.hashCode() == x.hashCode()`).
- **Wünschenswert (aber nicht garantierbar):** Wenn `!x.equals(y)` gilt, sollten die Hash-Werte möglichst verschieden sein (`!x.equals(y) \rightarrow x.hashCode() möglichst verschieden von y.hashCode()`), um Kollisionen in Hash-Tabellen zu minimieren.
- **Wichtige Regel:** Wenn `equals()` überschrieben wird, **muss auch hashCode() überschrieben werden**.
- **Implementierungshinweis:** Bei der Implementierung von `hashCode()` müssen die gleichen Bedingungen für die Gleichheit beachtet und **keine anderen Variablen** als in der `equals()`-Methode einbezogen werden.

Die `hashCode()` Methode

- Muss immer zusammen mit `equals()` überschrieben werden.
- Die in `Object` vordefinierte Methode ist `int hashCode()`.
- Gibt eine fast beliebige ganze Zahl zurück.
- Wird beispielsweise in Hash-Tabellen benötigt.
- **Implementierung in Object**: Leitet eine Zahl von der Speicheradresse des Objekts ab.
- Wird die Methode nicht überschrieben, gibt sie einen Ausschnitt aus der Objektadresse zurück.
- **Eigenschaften von hashCode :**
 - Hat `x.equals(y)` als Ergebnis `true`, dann gibt `x.hashCode()` dieselbe Zahl zurück wie `y.hashCode()`.
 - Solange das Objekt `x` unverändert bleibt, müssen wiederholte Aufrufe von `x.hashCode()` stets gleiche Ergebnisse liefern. Es gibt also einen Zusammenhang zwischen `hashCode()` und `equals()`.
- **Bedingungen:**
 - Wenn `x.equals(y) true` ist, dann muss `x.hashCode() == y.hashCode()` gelten.
 - Solange das Objekt `x` unverändert bleibt, muss `x.hashCode()` immer denselben Wert zurückgeben (`x` unverändert \rightarrow `x.hashCode() == x.hashCode()`).
- **Wünschenswert (aber nicht garantierbar):** Wenn `!x.equals(y)` gilt, sollten die Hash-Werte möglichst verschieden sein (`!x.equals(y) \rightarrow x.hashCode() möglichst verschieden von y.hashCode()`), um Kollisionen in Hash-Tabellen zu minimieren.
- **Wichtige Regel:** Wenn `equals()` überschrieben wird, **muss auch hashCode() überschrieben werden**.
- **Implementierungshinweis:** Beim Implementieren von `hashCode()` ist darauf zu achten, dass unterschiedliche Objekte zu möglichst unterschiedlichen Ergebnissen von `hashCode()` führen, obwohl gleiche Ergebnisse nicht auszuschließen sind. Es müssen die gleichen Bedingungen für die Gleichheit beachtet und **keine anderen Variablen** als in der `equals()`-Methode einbezogen werden.

Eine Implementierung, die nicht alle dieser Bedingungen erfüllt, muss als fehlerhaft betrachtet werden, auch wenn weder der Compiler noch das Laufzeitsystem einen Fehler meldet.

Richtige und falsche Annahmen bezüglich `hashCode()` und `equals()`

Folgende Bedingung ist immer erfüllt:

- `x.hashCode() != y.hashCode() → !x.equals(y)` (Wenn zwei Objekte unterschiedliche Hash-Werte haben, können sie nicht gleich sein gemäß `equals()`).

Folgende Bedingung ist manchmal erfüllt:

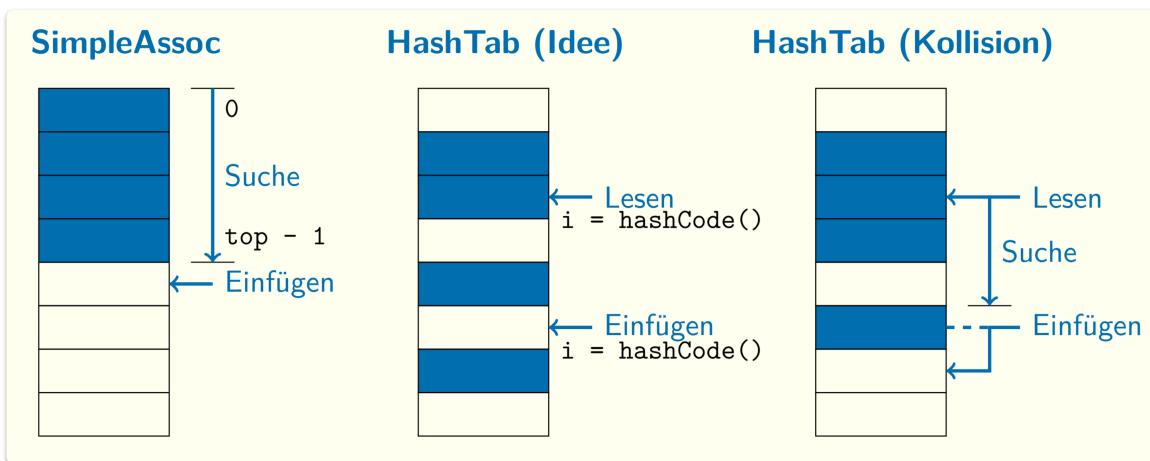
- `x.equals(y) → x.toString().equals(y.toString())` (Dies gilt nur, wenn die `toString()`-Methode ausschließlich Informationen liefert, die für die Gleichheit relevant sind).

Folgende Bedingungen dürfen NICHT angenommen werden:

- `x.hashCode() == y.hashCode() → x.equals(y)` (Objekte mit gleichem Hash-Wert können ungleich sein - Kollision).
- `x.toString().equals(y.toString()) → x.equals(y)` (Die String-Präsentation kann für ungleiche Objekte gleich sein).
- `x.hashCode()` ist die Speicheradresse von `x` (Die Standardimplementierung in `Object` leitet zwar einen Wert von der Speicheradresse ab, aber das ist keine garantierte oder die einzige mögliche Implementierung).

Hash-Tabellen

Funktionsweise von Hash-Tabellen



- Hash-Tabellen sind Datenstrukturen, die Einträge in Arrays ablegen.
- Die **Indizes** im Array entsprechen im Wesentlichen den aus den Einträgen **errechneten Hash-Werten**.
- Dies ermöglicht eine effiziente Überprüfung, ob ein bestimmtes Objekt im Array enthalten ist:
 1. Berechne den Hash-Wert des Objekts.
 2. Überprüfe den Eintrag an dem entsprechenden Index.
- **Problem: Hash-Werte sind nicht eindeutig.**
 - Mehrere Objekte können denselben Hash-Wert haben.
 - Dies führt zu **Kollisionen**, bei denen mehrere Objekte auf denselben Index im Array abgebildet werden.

- **Umgang mit Kollisionen:** Es gibt verschiedene Strategien.
 - **Lineare Suche (im Beispiel Listing 3.16):**
 - Der berechnete Hash-Wert dient als **Anfangsindex**.
 - Bei einer Kollision wird linear nach dem nächsten freien Platz gesucht.
 - Die Suche kann abgebrochen werden, sobald ein leerer Eintrag gefunden wird.
 - **Anpassung von Hash-Werten an Arraylängen:**
 - Hash-Werte sind in der Regel nicht direkt an die Größe des Arrays angepasst.
 - **Lösung:** Verwende nur einen passenden Ausschnitt aus dem Bitmuster des Hash-Werts.
 - **Beispiel:** Die niederwertigsten Bits werden verwendet.
 - Oft werden aber auch höherwertige Bits gewählt.
-

Vergleich

	lineare Liste	Suchbaum	Hash-Tabelle
Reihenfolge der Daten bleibt erhalten	+	-	-
Daten in sortierter Reihenfolge zugreifbar	-	+	-
Suche meist effizient	-	+	+
sicher vor Ausreißern (Laufzeit)	+	-	-
sicher vor Ausreißern bei üblichen Daten	+	-	+
wenig anfällig für Fehler	+	-	-

8. Abstraktionshierarchie, Iterator, Baumoperationen

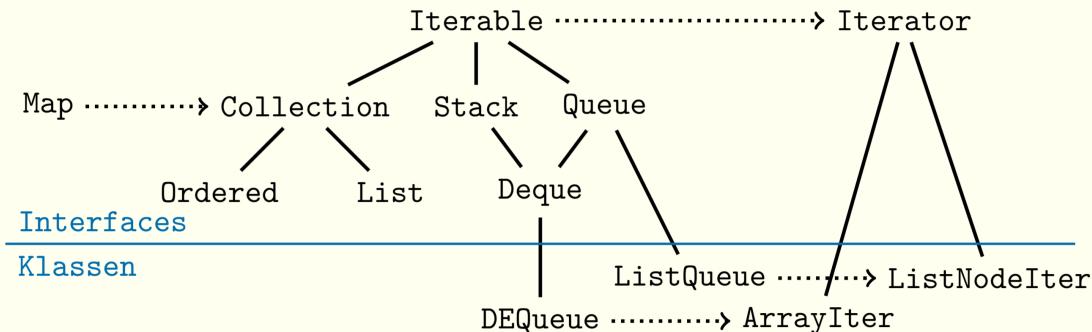
Abstraktionshierarchie

Skriptum, p.91

Beispiel:

oberstes Gebot: möglichst einheitliche Schnittstellen

meist mehrere Abstraktionshierarchien, die eng zusammenarbeiten



Wir wollen ein gemeinsames Konzept

Wir schauen uns heute an wie ein Collectionsframework implementiert sein kann

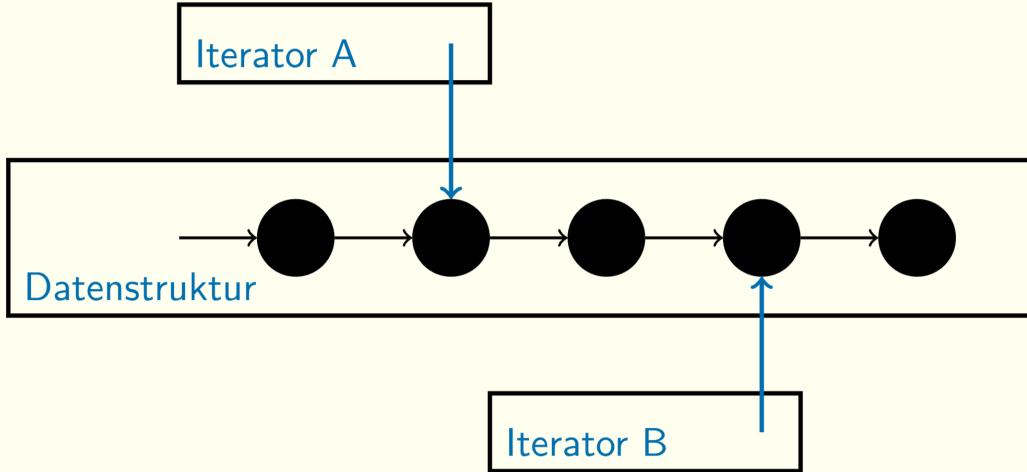
Beobachtung:

- Konzeptuelle Gemeinsamkeiten zwischen ADTs und Datenstrukturen (grundlegende Operationen: Einfügen, Suchen, Löschen etc.).
- Bedeutende Unterschiede im Detail (Verwendungsmuster, gezielte Nutzung von Eigenschaften, historische Entwicklung/unterschiedliche Benennung).

Ziel:

- Aufbau einer möglichst einheitlichen Abstraktionshierarchie.
- Berücksichtigung begründbarer Unterschiede.

Iterator auf einer Datenstruktur



Zentrale Idee: Anwendungsfälle definieren Interfaces mit typischen Operationen (Stack, Queue, Liste, Hashtabelle etc.).

Wichtiges Konzept: Das `Iterable`-Interface in Java ermöglicht die Verwendung von Iteratoren.

Interfaces:

- `Iterable` (Listing 3.17):

```
public interface Iterable extends java.lang.Iterable<String> {
    Iterator iterator();
}
```

- Ermöglicht die Erzeugung eines `Iterator`-Objekts.
- `Iterator` (Listing 3.18):

```
public interface Iterator extends java.util.Iterator<String> {
    String next();
    boolean hasNext();
}
```

- `next()`: Gibt den nächsten Eintrag zurück.
- `hasNext()`: Gibt `true` zurück, wenn es weitere Einträge gibt.
- **Wichtig:** `next()` verändert die Datenstruktur im Gegensatz zu `poll()` in einer Queue **nicht**.
- Mehrere Iteratoren können gleichzeitig über dieselbe Datenstruktur iterieren.

Verwendung von Iteratoren (Listing 3.19):

- Lange Variante (A):

```
private static void printAllA ( Iterable data ) {
    Iterator i = data . iterator ();
    while (i . hasNext ()) System . out . println ( i . next ());
}
```

- Kurze Variante (B) (ForEach-Schleife):

```
private static void printAllB ( Iterable data ) {
    for ( String s : data ) System . out . println ( s );
}
```

- Die ForEach-Schleife erzeugt automatisch einen `Iterator` für Objekte, die `Iterable` implementieren.

Ergänzung zu Stacks und Queues:

- Iteratoren erlauben das **mehrfache Durchlaufen** von Datenstrukturen.
- Die **Reihenfolge der Iteration ist beliebig wählbar** (nicht zwingend LIFO oder FIFO).

Weitere Interfaces: (Datenstrukturen + Datenstrukturen 2)

- **Stack** (Listing 3.20):

```
public interface Stack extends Iterable {
    void push ( String element );
    String pop ();
    String peek ();
    int size ();
}
```

- Erweitert `Iterable` und definiert typische Stack-Operationen.

- **Queue** (Listing 3.21):

```
public interface Queue extends Iterable {
    void add ( String element );
    String poll ();
    String peek ();
    int size ();
}
```

- Erweitert `Iterable` und definiert typische Queue-Operationen.

- **Deque** (Double-Ended-Queue) (Listing 3.22):

```
public interface Deque extends Queue , Stack {
    void addFirst ( String element );
    void addLast ( String element );
    String pollFirst ();
    String pollLast ();
    String peekFirst ();
    String peekLast ();
}
```

- Verallgemeinert `Stack` und `Queue`.
- Muss Methoden der Obertypen enthalten (z.B. `pop()` und `poll()`).
- Manche Methoden sind semantisch gleich (`add` und `addLast`).

- **Map (Assoziative Datenstruktur) (Listing 3.23):**

```
public interface Map {
    String put ( String key , String value );
    String get ( String key );
    String remove ( String key );
    boolean containsKey ( String key );
    boolean containsValue ( String value );
    Collection keys ();
    Collection values ();
    int size ();
}
```

- Kein Untertyp von `Iterable`, da die Iteration über Schlüssel oder Werte erfolgen kann.
- Bietet `keys()` und `values()`, die `Collection`-Objekte zurückgeben, über die iteriert werden kann.
- **Collection (Sammlung von Datenelementen) (Listing 3.24):**

```
public interface Collection extends Iterable {
    void add ( String element );
    int contains ( String element );
    int removeAll ( String element );
    void clear ();
    int size ();
    String [] toArray ();
}
```

- Erweitert `Iterable`.
- Schreibt **keine bestimmte Reihenfolge** der Einträge vor.
- `clear()`: Löscht alle Einträge.
- `toArray()`: Ermöglicht Zugriff auf alle Einträge als Array.
- **Ordered (Sortierte Sammlung) (Listing 3.25):**

```
public interface Ordered extends Collection { }
```

- Erweitert `Collection` und garantiert, dass Einträge in `toArray()` und bei Iteration **sortiert** vorliegen.
- Führt keine neuen Methoden ein.
- **List (Über Index zugreifbare Sammlung) (Listing 3.26):**

```
public interface List extends Collection {
    int indexOf ( String element );
    String get ( int index );
    boolean add ( int index , String element );
    String set ( int index , String element );
    String remove ( int index );
    void sort ();
    Iterator sortedIterator ();
}
```

- Erweitert Collection.
- Ermöglicht Zugriff auf Elemente über einen Index.
- sort() : Sortiert die Liste (die Sortierung kann durch add(index, ...) oder set(index, ...) wieder aufgehoben werden).
- sortedIterator() : Liefert einen Iterator, der die Elemente in sortierter Reihenfolge durchläuft.

Anmerkung zu extends java.lang.Iterable<String> und extends java.util.Iterator<String> :

- Diese Klauseln stellen Verbindungen zu vordefinierten Java-Interfaces her, um die Verwendung von ForEach-Schleifen zu ermöglichen.
- Bei Verzicht auf ForEach-Schleifen sind diese Klauseln nicht notwendig.

Implementierung von Interfaces

Mögliche Implementierungen:

- Stack, Queue, Deque, Ordered, List: Arrays und Listen (wegen linearer Ordnung).
- Map, Collection: Bäume und Hash-Tabellen (Hash-Tabellen besonders gut geeignet).
- Ordered: Bäume.

Ungeeignete Implementierungen:

- Stack, Queue, Deque, List (abgesehen von Ordered): Übliche Hash-Tabellen und wenig geeignet: Bäume (wegen nötiger linearer Ordnung).

Eigenschaften von Implementierungen:

- Unterschiedliche Effizienz bei Operationen (Suchen, Einfügen, Löschen etc.).
- Ideal: Verfügbarkeit verschiedener Implementierungen zur Auswahl der optimalen Variante für spezifische Anwendungsfälle (je nach Häufigkeit der Operationen).

Kombination von Implementierungstechniken:

- Beispiel: Hash-Tabelle mit Baum zur Kollisionsbehandlung (Anmerkung: Mit guten Hash-Funktionen ist dies oft nicht sehr sinnvoll).

Iterator-Implementierung

Beispiel: Iterator für eine Queue basierend auf einer Liste (ListQueue)

- Die Queue -Implementierung (ListQueue) benötigt eine Methode zur Erzeugung eines spezifischen Iterators.
- Ausschnitt aus ListQueue (Listing 3.27):

```
public class ListQueue implements Queue {
    // Implementierung von ListQueue erweitert Listing 2.22, Seite 64
    public Iterator iterator () {
        return new ListNodeIter ( head );
    }
}
```

- Die `iterator()`-Methode erzeugt eine Instanz von `ListNodeIter` und übergibt den `head` der Liste.
- Implementierung des Iterators `ListNodeIter` (Listing 3.28):

```
public class ListNodeIter implements Iterator {
    private ListNode n; // Listing 2.18, Seite 61
    public ListNodeIter ( ListNode n ) { this . n = n; }
    public boolean hasNext () { return n != null ; }
    public String next () {
        if ( n == null ) { return null ; }
        String result = n. value ();
        n = n. next ();
        return result ;
    }
}
```

- `n`: Aktueller Knoten der Liste.
- Konstruktor: Initialisiert `n` mit dem Startknoten.
- `hasNext()`: Gibt `true` zurück, solange der aktuelle Knoten nicht `null` ist.
- `next()`: Gibt den Wert des aktuellen Knotens zurück und bewegt `n` zum nächsten Knoten.

Beispiel: Iterator mit direktem Zugriff auf ein Array (DEQueue)

- Iteratoren benötigen oft Zugriff auf interne Implementierungsdetails der Datenstruktur (z.B. das Array).
- Iteratoren werden daher meist gemeinsam mit den Datenstrukturen implementiert.
- Ausschnitt aus `DEQueue` (Listing 3.29):

```
public class DEQueue implements Deque {
    // Implementierung von DEQueue erweitert Listing 2.16, Seite 53
    public Iterator iterator () {
        return new ArrayIter ( es , head , count );
    }
}
```

- Die `iterator()`-Methode erzeugt eine Instanz von `ArrayIter` und übergibt das Array (`es`), den Startindex (`head`) und die Anzahl der Elemente (`count`).
- Implementierung des Iterators `ArrayIter` (Listing 3.30):

```
public class ArrayIter implements Iterator {
    private String [] es ;
    private int i , count ;
    public ArrayIter ( String [] es , int i , int count ) {
        this . es = es ; this . i = i; this . count = count ;
    }
    public boolean hasNext () { return count > 0; }
    public String next () {
        if ( count <= 0) { return null ; }
        count - -;
        return es [ i ++ & ( es . length - 1)];
    }
}
```

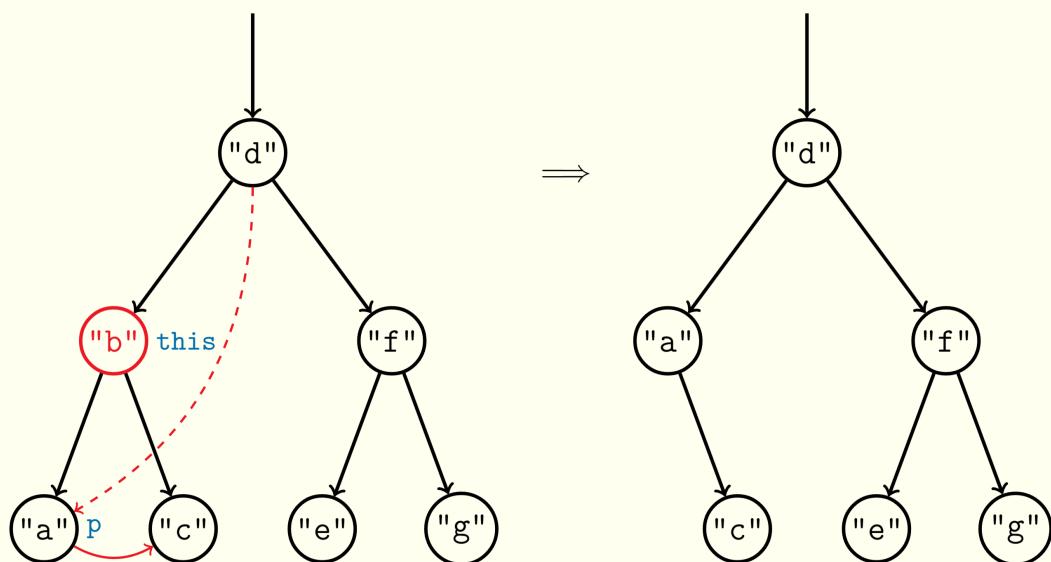
```

    }
}
}
```

- `es` : Das Array, das die Elemente speichert.
- `i` : Aktueller Index.
- `count` : Anzahl der verbleibenden Elemente.
- Konstruktor: Initialisiert die Attribute mit den übergebenen Werten.
- `hasNext()` : Gibt `true` zurück, solange noch Elemente vorhanden sind (`count > 0`).
- `next()` : Gibt das Element am aktuellen Index zurück, dekrementiert `count` und inkrementiert `i` (unter Berücksichtigung des Array-Umlaufs mittels Bitwise-AND-Operation).

Baumoperationen

Entfernen eines Knotens aus binärem Suchbaum



Implementierung von `Ordered` als binärer Suchbaum (Listing 3.31, 3.32, 3.33):

- Nutzt die inhärente Sortiereigenschaft eines binären Suchbaums.

`OrderedTree` (Listing 3.31):

- Implementiert das `Ordered`-Interface.
- `root` : Wurzelknoten des binären Suchbaums (`OTNode`).
- `cnt` : Array der Größe 1 zur Speicherung der Anzahl der Elemente. Verwendung eines Arrays ermöglicht die Übergabe als Referenz und somit die Änderung der Größe innerhalb von Methoden.
- **Konstruktoren:**
 - Standardkonstruktor: Erzeugt einen leeren Baum.
 - Konuktoren mit `String[]` oder `Iterable`: Fügen die Elemente in den Baum ein (gleicher Rumpf, unterschiedliche Parameter).
- `add(String e)` : Fügt ein Element in den Suchbaum ein (hält die Suchbaum-Eigenschaft aufrecht).
- `contains(String e)` : Gibt die Anzahl der Vorkommen eines Elements zurück.
- `removeAll(String e)` : Entfernt alle Vorkommen eines Elements und gibt die Anzahl der entfernten Elemente zurück.
- `clear()` : Leert den Baum.

- `size()` : Gibt die Anzahl der Elemente zurück.
- `toArray()` : Erzeugt ein sortiertes Array der Elemente.
- `toArray(String[] array)` : Füllt ein gegebenes Array mit den sortierten Elementen und gibt die Anzahl der eingefügten Elemente zurück.
- `iterator()` : Erzeugt einen Iterator (`OTIter`), der die Elemente in sortierter Reihenfolge durchläuft.
- `toString()` : Gibt eine String-Repräsentation der sortierten Elemente zurück.
- `equals(Object o)` : Vergleicht zwei `OrderedTree`-Objekte, indem die aus ihnen erzeugten Arrays verglichen werden.
- `hashCode()` : Berechnet den Hash-Code basierend auf dem sortierten Array der Elemente.

OTNode (Listing 3.32):

- Repräsentiert einen Knoten im binären Suchbaum.
- `elem` : Das im Knoten gespeicherte Element.
- `left, right` : Referenzen auf den linken und rechten Kindknoten.
- `fill(String[] array, int i)` : Füllt ein Array rekursiv mit den Elementen des Baums in sortierter Reihenfolge (linker Teilbaum, aktueller Knoten, rechter Teilbaum). `i` ist der aktuelle Index im Array. Gibt den nächsten freien Index zurück.
- `compare(String e)` : Vergleicht das gegebene Element `e` mit dem Element des aktuellen Knotens. Berücksichtigt auch `null`-Werte.
- `add(String e)` : Fügt ein Element rekursiv in den passenden Teilbaum ein (unter Beachtung der Suchbaum-Eigenschaft).
- `contains(String e)` : Sucht rekursiv nach dem Element `e` im Baum und zählt seine Vorkommen. Die Schleife wird weiter ausgeführt, um alle Vorkommen des Elements zu zählen, da Duplikate erlaubt sein könnten (obwohl ein typischer binärer Suchbaum keine Duplikate direkt unterstützt; hier wird die Zählung implementiert).
- `rmv(String e, int[] cnt)` : Entfernt rekursiv ein Element `e`.
 - Zeilen 49-56 (Fall `cmp == 0`): Wenn das Element gefunden wurde, wird die Anzahl dekrementiert.
 - Wenn der linke Teilbaum leer ist, wird der rechte Teilbaum zurückgegeben.
 - Andernfalls wird der rechteste Knoten des linken Teilbaums gesucht, dessen rechter Zeiger auf den rechten Teilbaum des zu löschenen Knotens gesetzt, und der linke Teilbaum wird zurückgegeben (Ersetzungsknoten-Strategie).
 - `root, left, right` werden neu gesetzt, um die Struktur des Baums nach dem Löschen zu aktualisieren.
- `iter(OTIter iter, boolean next)` : Hilfsmethode für den Iterator. Fügt Knoten in den Iterator-Stack ein (in umgekehrter Reihenfolge für die In-Order-Traversierung). `next` steuert, ob der rechte Kindknoten (für den nächsten Eintrag) oder der aktuelle Knoten behandelt wird. Gibt das Element des aktuellen Knotens zurück.

OTIter (Listing 3.33):

- Implementiert das `Iterator`-Interface für `OrderedTree`.
- `node` : Der als nächstes zu besuchende Knoten.
- `parent` : Der "übergeordnete" `OTIter`, der den Pfad zurück zur Wurzel im simulierten Stack repräsentiert.
- **Konstruktoren:**

- Standardkonstruktor: Erzeugt einen leeren Iterator (für den Fall eines leeren Baums).
- Konstruktor mit `OTNode n` und `OTIter p`: Erzeugt einen neuen `OTIter`, wobei `n` der aktuelle Knoten und `p` der vorherige `OTIter` (Elternteil im simulierten Stack) ist. Die Referenzen werden so manipuliert, dass eine Art verkettete Liste der zu besuchenden Knoten entsteht (simulierter Stack).
- `hasNext()`: Gibt `true` zurück, solange es noch einen zu besuchenden Knoten gibt (`node != null`).
- `next()`: Gibt das Element des aktuellen Knotens zurück und bewegt den Iterator zum nächsten Knoten (simuliert das Entnehmen vom Stack und das Besuchen des linken Kindes).
 - `todo`: Speichert den aktuellen Knoten.
 - `node` und `parent` werden aktualisiert, um den nächsten Knoten im simulierten Stack zu erreichen.
 - `todo.iter(this, true)`: Ruft die `iter`-Methode des aktuellen Knotens auf, um den rechten Teilbaum für die nächste Iteration vorzubereiten (fügt die Knoten des rechten Teilbaums in umgekehrter Reihenfolge auf den simulierten Stack).

Wieso sind `toArray` und `toString` sortiert?

- Die Sortierung ergibt sich aus der **In-Order-Traversierung** des binären Suchbaums: Zuerst der linke Teilbaum, dann der aktuelle Knoten, dann der rechte Teilbaum.
- `fill` implementiert diese Traversierung rekursiv.
- `iterator` und die Hilfsmethode `iter` implementieren diese Traversierung iterativ mithilfe des simulierten Stacks.

Umkehrung der Sortierreihenfolge (ohne `add` oder `compare` zu ändern):

- `fill`: Ändern Sie die Reihenfolge der rekursiven Aufrufe in `OTNode.fill` zu:

```

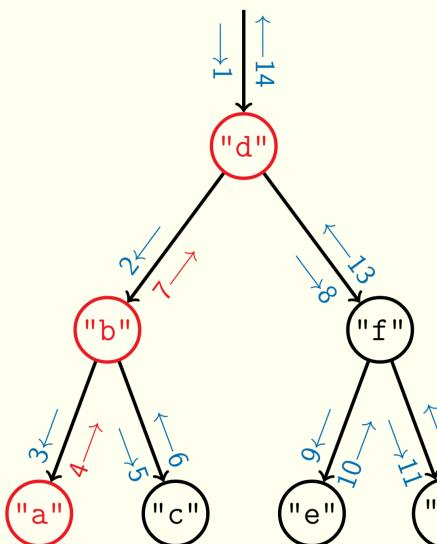
if ( right != null ) { i = right . fill ( array , i ); }
if ( i < array . length ) {
    array [ i ++] = elem ;
    if ( left != null ) i = left . fill ( array ,i );
}
  
```

- `OTNode.iter`: Ändern Sie den initialen Wert von `n` und die Reihenfolge der Zuweisungen in `OTNode.iter` zu:

```

OTNode n = next ? left : this ;
while (n != null ) {
    new OTIter (n , iter );
    n = n. right ;
}
return elem ;
  
```

Iteration über binären Suchbaum



Aufrufe von `next` erfolgen iterativ, daher keine rekursive Lösung bei einem Iterator (anders bei Verwendung mehrerer Iteratoren)

Lösungsansätze:

jeder Knoten referenziert Elter-Knoten

Simulation des Systemstacks
(Merken des aktuellen Pfads)

oder je ein Iterator pro Knoten

Funktionsweise des Iterators:

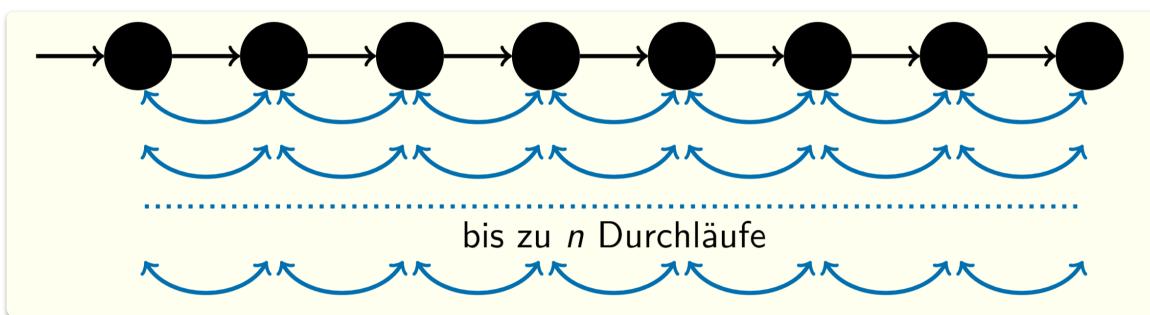
- `iterator() in OrderedTree`: Erzeugt ein neues `OTIter`-Objekt und startet die In-Order-Traversierung, indem es (falls die Wurzel nicht null ist) die `iter`-Methode der Wurzel aufruft, um die linke Baumhälfte auf den simulierten Stack zu legen.
- `iter() in OTNode`: Nimmt einen `OTIter` und ein `boolean next` entgegen. Es simuliert das "Besuchen" eines Knotens und das Vorbereiten des nächsten Knotens für die Iteration. Es erstellt neue `OTIter`-Objekte, um die Knoten des linken Teilbaums des aktuellen Knotens auf den simulierten Stack (`parent`-Kette von `OTIter`-Objekten) zu legen.
- `OTIter`: Verwaltet den Zustand der Iteration. Der `parent`-Zeiger in `OTIter` bildet intern eine **verkettete Liste (simulierter Stack)** von `OTIter`-Objekten. Jeder `OTIter` in dieser Kette repräsentiert einen Knoten im Baum, der noch besucht werden muss.
 - `hasNext()`: Prüft, ob noch Knoten im simulierten Stack vorhanden sind (`node != null`).
 - `next()`: Nimmt den obersten Knoten vom simulierten Stack (`node`), aktualisiert den Stack (`node = parent.node, parent = parent.parent`), und ruft dann `iter()` auf dem gerade besuchten Knoten auf, um den **rechten Teilbaum** für die zukünftige Iteration auf den simulierten Stack zu legen.

Vergleich in `equals` und Verwendung von `toArray` in `hashCode`:

- **`equals` vergleicht Arrays:** Da es sich um eine **sortierte** Datensammlung handelt, sind zwei `OrderedTree`-Objekte genau dann gleich, wenn sie die gleichen Elemente in der gleichen (sortierten) Reihenfolge enthalten. Der Vergleich der Arrays stellt sicher, dass sowohl die Elemente als auch ihre Reihenfolge übereinstimmen.
- **Vergleich von `toString`-Ergebnissen wäre falsch:** `toString` erzeugt eine durch Kommas getrennte String-Repräsentation. Die Struktur des Baums (z.B. die Anordnung der Knoten) würde dabei verloren gehen, und zwei Bäume mit der gleichen Menge an Elementen könnten unterschiedliche `toString`-Ausgaben haben, aber trotzdem die gleiche sortierte Menge repräsentieren.
- **`toArray` in `hashCode`:** Der `hashCode` muss für gleiche Objekte gleich sein. Da die Gleichheit über den Inhalt (als sortiertes Array) definiert ist, muss auch der Hash-Code auf diesem Inhalt basieren. Die Verwendung des Hash-Codes des Arrays stellt sicher, dass Objekte mit dem gleichen Inhalt den gleichen Hash-Code haben.

9. Sortieren, Sichtweisen vs Kopien, Effizienz und Zuverlässigkeit

Sortierverfahren für Listen



Das Sortieren ist eine fundamentale Operation auf Datenstrukturen. Während einfache Sortierverfahren für Arrays bekannt sind, erfordert die Anwendung auf rekursive Datenstrukturen wie Listen besondere Beachtung hinsichtlich der Zugriffseffizienz.

Bubblesort für einfach verkettete Listen

- Bubblesort ist ein einfaches, auch für lineare Listen geeignetes Sortierverfahren.
- Es traversiert die Liste wiederholt.
- In jeder Traversierung werden benachbarte Listenknoten in ihrer Reihenfolge verglichen.
- Bei Bedarf werden die Werte (Inhalte) benachbarter Knoten vertauscht.
- Dieser Prozess wird solange wiederholt, bis eine komplette Traversierung ohne Vertauschungen erfolgt ist, was bedeutet, dass die Liste sortiert ist.
- Die Implementierung (Listing 3.34) behandelt `null`-Werte und vertauscht die Werte innerhalb der Knoten, anstatt die Knoten selbst zu bewegen, was die Implementierung vereinfacht.
- Die `LinearList`-Klasse bietet eine `sort()`-Methode zur direkten Sortierung der Liste und eine `sortedIterator()`-Methode, die einen Iterator zurückgibt, der die Liste in sortierter Reihenfolge durchläuft, ohne die ursprüngliche Sortierung der Liste zu verändern.

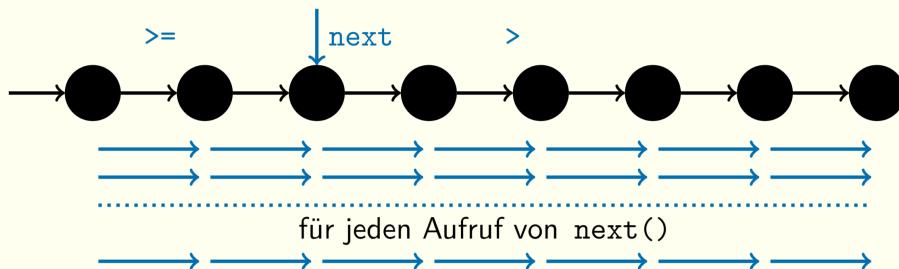
```

1  public class LinearList implements List {
2      private ListNode head;
3      public void sort() {
4          if (head != null) {
5              boolean modified;
6              do {
7                  modified = false;
8                  ListNode m, n = head;
9                  while ((n = (m = n).next()) != null) {
10                      String x=m.value(), y=n.value();
11                      if (x != null && (y == null
12                          || x.compareTo(y) < 0)) {
13                          m.setValue(y);
14                          n.setValue(x);
15                          modified = true;
16                      }
17                  }
18              } while (modified);
19          }
20      }
21      public Iterator sortedIterator() {
22          return new SortedListNodeIter(head);
23      }
24      // zahlreiche weitere Methoden zu implementieren ...
25  }

```

Sortieren durch Iterator (`SortedListNodeIter`)

Such für jeden Aufruf von `next()` den letzten Knoten mit dem kleinsten Wert größer (oder gleich) dem zurückgegebenen Wert (`next`)

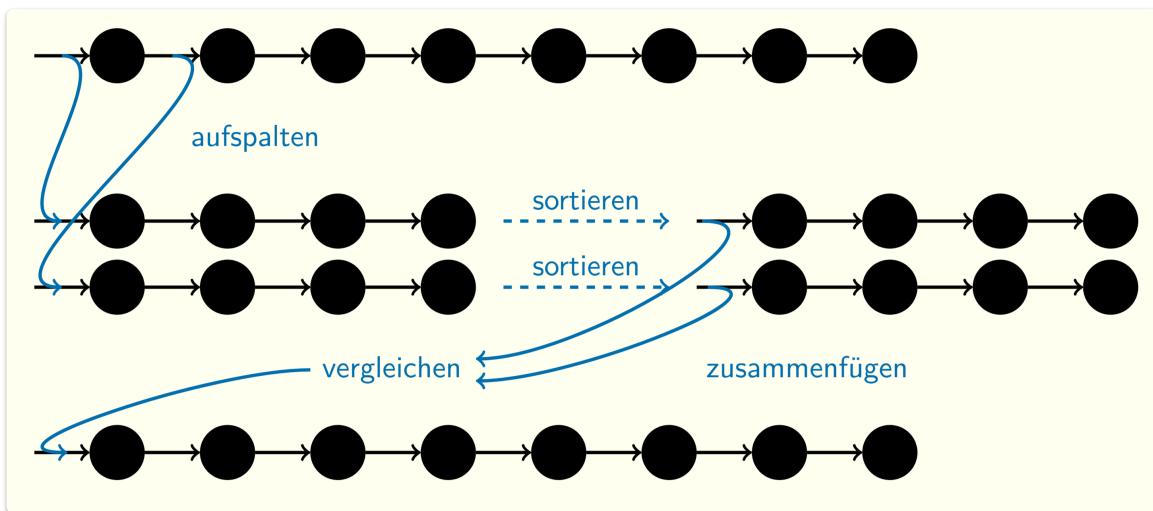


- Die `SortedListNodeIter`-Klasse (Listing 3.35) implementiert einen Iterator, der die Liste in sortierter Reihenfolge liefert.
- Vor jedem Aufruf von `next()` durchsucht der Iterator die gesamte Liste nach dem kleinsten Eintrag, der seit dem letzten Aufruf noch nicht zurückgegeben wurde.
- Die `prec()`-Methode (`precedes`) ist für den Vergleich zuständig und berücksichtigt dabei auch `null`-Werte.
- Der Iterator merkt sich den zuletzt zurückgegebenen Knoten (`next`) und berücksichtigt beim nächsten Durchlauf nur Einträge, die größer oder gleich dem Wert von `next` sind (bzw. echt größer, um Duplikate korrekt zu behandeln).
- Dieses iterative Sortieren ist ineffizient, da für jedes Element die restliche Liste durchsucht werden muss.

```

1  public class SortedListNodeIter implements Iterator {
2      private ListNode head, next;
3      public SortedListNodeIter(ListNode head) {
4          this.head = head;
5          setNext();
6      }
7      private static boolean prec(ListNode a,
8          ListNode b, int c) {
9          String x = a.value(), y = b.value();
10         if (y == null) { return x == null && c == 0; }
11         return x == null || y.compareTo(x) >= c;
12     }
13     private void setNext() {
14         int c = 0;
15         ListNode n = head, min = null;
16         while (n != null) {
17             if (n == next) { c = 1; }
18             else if ((next==null || prec(next,n,c))
19                     && (min==null || prec(n,min,0)))
20                 min = n;
21             n = n.next();
22         }
23         next = min;
24     }
25     public boolean hasNext() { return next != null; }
26     public String next() {
27         if (next == null) { return null; }
28         String result = next.value();
29         setNext();
30         return result;
31     }
32 }
```

Mergesort für Listenknoten



- Mergesort ist ein effizienter, rekursiver Sortieralgorithmus, der sich besonders gut für Listen eignet.
- Die Implementierung in `ListNode` (Listing 3.36) demonstriert eine rekursive Sortierung direkt auf den Listenknoten.
- Der Algorithmus teilt die Liste rekursiv in zwei annähernd gleich große Teillisten auf.

- Die Teillisten werden durch rekursive Aufrufe von `mergesort()` sortiert.
- Anschließend werden die sortierten Teillisten wieder zusammengeführt (gemergt), wobei nur die ersten Elemente der Teillisten verglichen werden müssen, um die Reihenfolge beim Zusammenführen zu bestimmen.
- Da das effiziente Einfügen am Ende der Liste ohne eine zusätzliche Referenz auf das letzte Element aufwendig wäre (Mergesort erfordert ein FIFO-Verhalten beim Zusammenführen), wird hier ein Trick angewendet: Mit jeder Rekursionsebene wird die Sortierreihenfolge umgekehrt. Das LIFO-Verhalten der rekursiven Aufrufe in Kombination mit der umgekehrten Sortierung in jeder Ebene resultiert in der gewünschten aufsteigenden Sortierung.
- Im Gegensatz zu Bubblesort (Listing 3.34) werden hier die Listenknoten selbst neu verkettet und nicht nur deren Werte vertauscht.

```

1  public class ListNode { // erweitert Listing 2.18, Seite 61
2      private String value;
3      private ListNode next;
4      private static boolean cmp(ListNode a,ListNode b) {
5          return a.value == null || (b.value != null
6              && a.value.compareTo(b.value) < 0);
7      }
8      public ListNode mergesort(boolean reverse) {
9          if (next == null) { return this; }
10         ListNode[] xs = new ListNode[2];
11         ListNode h, n = this;
12         for (int i = 0; n != null; i = (i + 1) & 1) {
13             h = n;
14             n = h.next;
15             h.next = xs[i];
16             xs[i] = h;
17         }
18         xs[0] = xs[0].mergesort(!reverse);
19         xs[1] = xs[1].mergesort(!reverse);
20         while (xs[0] != null) {
21             int i = (xs[1] == null
22                 || cmp(xs[0], xs[1]) == reverse) ? 0 : 1;
23             h = xs[i];
24             xs[i] = h.next;
25             h.next = n;
26             n = h;
27         }
28         while (xs[1] != null) {
29             h = xs[1];
30             xs[1] = h.next;
31             h.next = n;
32             n = h;
33         }
34         return n;
35     }
36 }
```

Vorteile des Vertauschens von Listenknoten

- Der Aufwand für das Vertauschen von Listenknoten ist unabhängig von der Größe und Art der Daten, die in den Knoten gespeichert sind.

- Es bietet eine bessere Robustheit im Umgang mit externen Referenzen auf einzelne Knoten. Wenn nur Verwaltungsdaten (wie `next`-Pointer) geändert werden, ist die Wahrscheinlichkeit negativer Auswirkungen auf die Korrektheit bei unerwarteten Änderungen der Knoteninhalte durch andere Programmteile geringer, als wenn die eigentlichen Daten in referenzierten Knoten verändert würden.

Die `LinearList`-Klasse (Listing 3.37) zeigt, wie die `mergesort()`-Methode der `ListNode`-Klasse genutzt werden kann, um die Liste effizient zu sortieren, indem der `head` der Liste nach dem Sortieren auf den neuen sortierten Anfang der Liste gesetzt wird.

```

1 public class LinearList implements List {
2     private ListNode head;
3     // Alternative Implementierung von sort – siehe Listing 3.34
4     public void sort() {
5         if (head != null) {
6             head = head.mergesort(false);
7         }
8     }
9 }
```

Sichtweisen und Kopien im Kontext von Hash-Tabellen

Dieser Abschnitt (3.2.4) erläutert das Konzept von Sichtweisen (Views) und Kopien im Zusammenhang mit der Datenstruktur Hash-Tabelle und dem Interface `Map`. Die Listings 3.38 bis 3.41 demonstrieren, wie unterschiedliche Perspektiven auf die gleichen Daten realisiert werden können und wie sich dies von unabhängigen Kopien der Daten unterscheidet.

HashTab implementiert Map (Listing 3.38)

- Die Klasse `HashTab` wird erweitert, um das Interface `Map` zu implementieren.
- Die interne Datenhaltung erfolgt in zwei String-Arrays: `ks` für Schlüssel (keys) und `vs` für Werte (values). `count` speichert die Anzahl der Einträge.
- Die `remove(String k)`-Methode behandelt Kollisionen, indem nach dem Entfernen eines Eintrags alle nachfolgenden Einträge bis zum nächsten leeren Slot ebenfalls entfernt und wieder neu in die Tabelle eingefügt werden, um die Suchbarkeit dieser Einträge zu gewährleisten.
- Die Methoden `keys()` und `values()` geben Objekte vom Typ `Collection` zurück, die eine Sicht auf die Schlüssel bzw. Werte der Hash-Tabelle darstellen. Die Implementierung von `keys()` wird detaillierter betrachtet und erzeugt ein Objekt der Klasse `KeySet`.
- `keyArray(String[] keys)` kopiert die Schlüssel der Hash-Tabelle in das übergebene Array.
- `keyIterator()` gibt einen Iterator zurück, der direkt über das Schlüssel-Array der `HashTab` iteriert.
- Die Methode `clear()` initialisiert die Schlüssel- und Werte-Arrays sowie den Zähler.
- `size()` gibt die aktuelle Anzahl der Einträge zurück.

```

1  public class HashTab implements Assoc, Map {
2      private String[] ks, vs;
3      private int count;
4      // übernimmt Inhalte von Listing 3.16 sofern sie hier nicht stehen
5      public HashTab() { clear(); }
6      public void clear() {
7          ks = new String[65];
8          vs = new String[65];
9          count = 0;
10     }
11     public String remove(String k) {
12         int i = find(k);
13         String result = vs[i];
14         if (ks[i] != null) {
15             ks[i] = vs[i] = null;
16             count--;
17             for (i = (i + 1) & (ks.length - 2);
18                  ks[i] != null;
19                  i = (i + 1) & (ks.length - 2)) {
20                 String ki = ks[i], vi = vs[i];
21                 ks[i] = vs[i] = null;
22                 count--;
23                 put(ki, vi);
24             }
25         }
26         return result;
27     }
28     public int size() { return count; }
29     public Collection keys() { return new KeySet(this); }
30     public int keyArray(String[] keys) {
31         int i = 0, j = -1;
32         while (i < keys.length && ++j < ks.length-1) {
33             if (ks[j] != null) { keys[i++] = ks[j]; }
34         }
35         return i;
36     }
37     public Iterator keyIterator() {
38         return new HashTabKeyIter(ks);
39     }
40     public Collection values() { ... }

```

Implementierung einer bestimmten Sichtweise auf HashTab (Listing 3.39: KeySet)

- Die Klasse `KeySet` implementiert das Interface `Collection` und stellt eine *Sichtweise* auf die Schlüssel der `HashTable`-Instanz dar, auf die sie referenziert (`tab`).
- `KeySet` enthält keine eigenen Daten, sondern nur eine Referenz auf die zugrundeliegende `HashTable`.
- Die Methoden von `KeySet` delegieren Operationen an die entsprechenden Methoden der `HashTable`-Instanz:
 - `add(String element)` fügt den übergebenen Schlüssel (ohne zugehörigen Wert, daher `null`) in die `HashTable` ein, falls er noch nicht existiert.
 - `contains(String e)` prüft, ob der Schlüssel `e` in der `HashTable` vorhanden ist.
 - `removeAll(String e)` entfernt den Schlüssel `e` und seinen Wert aus der `HashTable`, falls vorhanden.

- `clear()` ruft die `clear()`-Methode der `HashTable` auf.
- `size()` ruft die `size()`-Methode der `HashTable` auf.
- `toArray()` erstellt eine *Kopie* der Schlüssel aus der `HashTable` in einem neuen String-Array.
- `iterator()` gibt einen `HashTableKeyIter` zurück, der über die Schlüssel der `HashTable` iteriert.
- `toString()`, `equals(Object o)`, `hashCode()` sind weitere Methoden zur Repräsentation und zum Vergleich des `KeySet`-Objekts.

```

1  public class KeySet implements Collection {
2      private HashTab tab;
3      public KeySet(HashTable t) { tab = t; }
4      public void add(String element) {
5          if (!tab.containsKey(element))
6              tab.put(element, null);
7      }
8      public int contains(String e) {
9          return tab.containsKey(e) ? 1 : 0;
10     }
11     public int removeAll(String e) {
12         if (tab.containsKey(e)) {
13             tab.remove(e);
14             return 1;
15         }
16         return 0;
17     }
18     public void clear() { tab.clear(); }
19     public int size() { return tab.size(); }
20     public String[] toArray() {
21         String[] keys = new String[size()];
22         tab.keySet(keys);
23         return keys;
24     }
25     public Iterator iterator() {
26         return tab.keySetIterator();
27     }
28     public String toString() {
29         String s = "";
30         for (String e : this)
31             s += s.equals("") ? e : ", " + e;
32         return "{" + s + "}";
33     }
34     public boolean equals(Object o) { ... }
35     public int hashCode() { ... }
36 }
```

Iterator über die Schlüssel von HashTab (Listing 3.40: HashTabKeyIter)

- Die Klasse `HashTableKeyIter` implementiert das `Iterator`-Interface, um die Schlüssel der `HashTable` zu durchlaufen.
- Der Iterator greift direkt auf das interne Schlüssel-Array (`ks`) der `HashTable` zu.
- `hasNext()` prüft, ob es noch nicht-null-Schlüssel im Array ab der aktuellen Position (`i`) gibt.

- `next()` gibt den nächsten nicht-null-Schlüssel im Array zurück und inkrementiert den Index `i`.

```

1  public class HashTabKeyIter implements Iterator {
2      private String[] ks;
3      private int i;
4      public HashTabKeyIter(String[] a) { ks = a; }
5      public boolean hasNext() {
6          for (int j = i; j < ks.length - 1; j++) {
7              if (ks[j] != null) { return true; }
8          }
9          return false;
10     }
11     public String next() {
12         while (i < ks.length - 1 && ks[i] == null)
13             i++;
14         return ks[i++];
15     }
16 }
```

Collection-Sichtweise versus kopierte Schlüssel einer Hash-Tabelle (Listing 3.41: `useHashTable`-Methode)

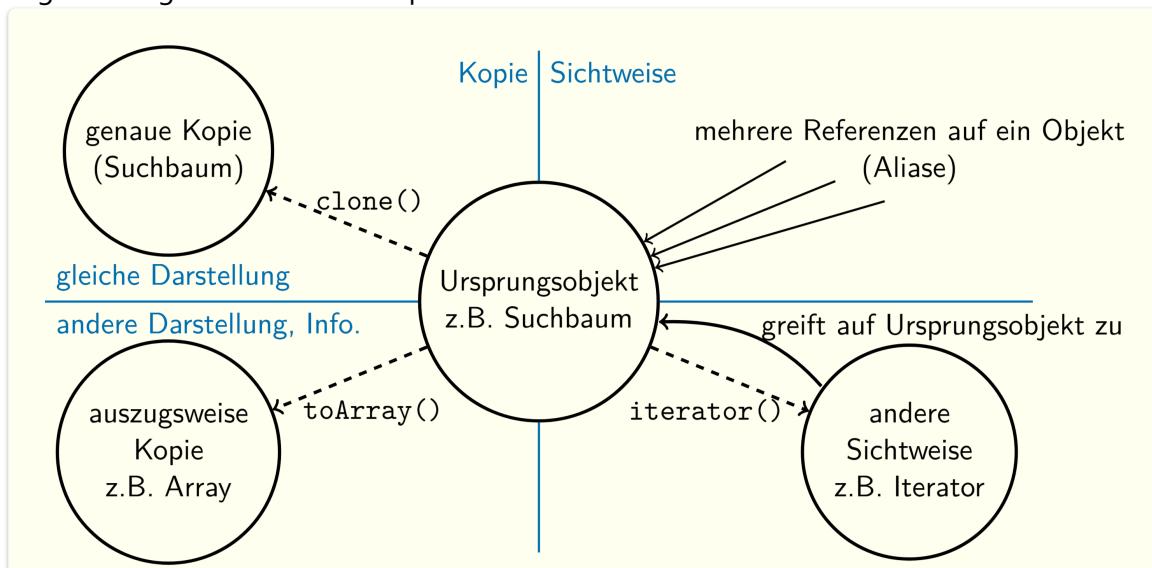
- Die Methode `useHashTable` demonstriert den Unterschied zwischen Sichtweisen und Kopien.
- Eine `HashTable`-Instanz (`map`) wird erstellt und mit Schlüssel-Wert-Paaren gefüllt.
- Eine `Collection`-Sicht der Schlüssel (`coll`) wird über `map.keys()` erzeugt.
- Eine *Kopie* der Schlüssel (`array`) wird über `coll.toArray()` erstellt.
- Die ersten drei `println`-Anweisungen (Zeilen 8-10) zeigen, dass `map`, `coll` und `array` zwar die gleichen Schlüsselmengen repräsentieren, aber unterschiedliche dynamische Typen haben.
- In den Zeilen 11-13 werden die Daten, auf die `map`, `coll` und `array` verweisen, modifiziert:
 - Ein neuer Eintrag wird in `map` eingefügt.
 - Ein neuer Schlüssel wird über die `coll`-Sicht hinzugefügt (was intern zu einem `put` in der `map` führt).
 - Das erste Element des `array` wird geändert.
- Die nachfolgenden `println`-Anweisungen (Zeilen 14-16) zeigen die Auswirkungen der Änderungen:
 - Änderungen über `map` sind auch in der `coll`-Sicht sichtbar und umgekehrt, da sie auf die gleichen zugrundeliegenden Daten in der `HashTable` zugreifen.
 - Änderungen am `array` haben keine Auswirkungen auf `map` und `coll`, und umgekehrt, da `array` eine unabhängige Kopie der Schlüssel darstellt.

```

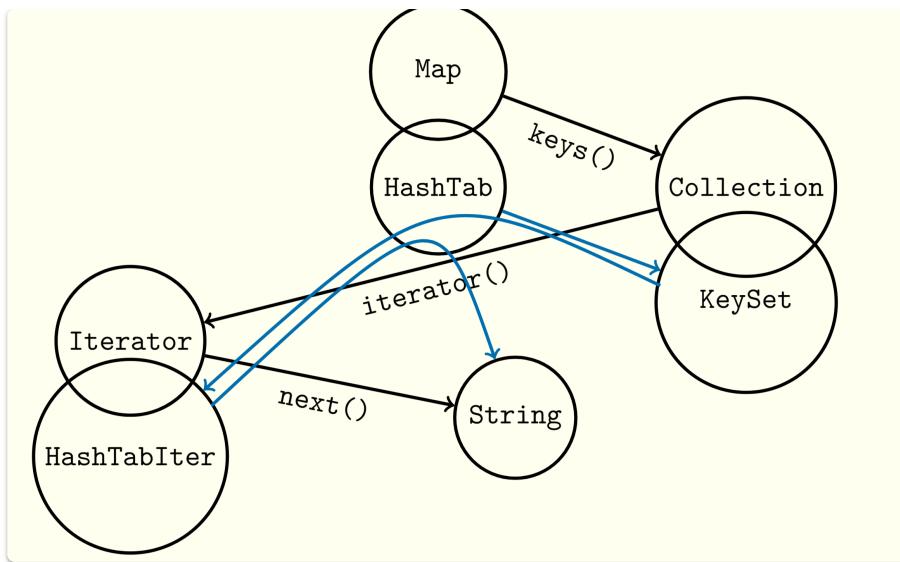
1  private static void useHashTable {
2      Map map = new HashTab();
3      for (char c = 'A'; c <= 'Z'; c++) {
4          map.put("key" + c, "value" + c);
5      }
6      Collection coll = map.keys();
7      String[] array = coll.toArray();
8      System.out.println(map);
9      System.out.println(coll);
10     System.out.println(Arrays.toString(array));
11     map.put("newKey", "newValue");
12     coll.add("collKey");
13     array[0] = "newA";
14     System.out.println(map);
15     System.out.println(coll);
16     System.out.println(Arrays.toString(array));
17 }
```

Zusammenfassung der Konzepte

- **Kopieren von Daten** erzeugt unabhängige Datenmengen. Änderungen in einer Kopie haben keine Auswirkungen auf das Original oder andere Kopien.
- **Unterschiedliche Sichtweisen** können auf gemeinsame Daten zugreifen. Änderungen der Daten über eine Sichtweise sind auch durch andere Sichtweisen sichtbar, da sie alle die gleichen zugrundeliegenden Daten manipulieren.



- Unterschiedliche Sichtweisen können die gemeinsamen Daten unterschiedlich nach außen darstellen.
- Das Beispiel von `KeySet` zeigt, dass manchmal zusätzlicher Implementierungsaufwand betrieben wird, um eine einheitliche Schnittstelle (`Collection`) für den Zugriff auf eine Datenstruktur bereitzustellen, auch wenn dies zu semantischen Inkonsistenzen führen kann (wie `add` in `KeySet`). Der Vorteil liegt in der Einheitlichkeit der API und der Vermeidung von Sonderlösungen für spezifische Datenstrukturen.



Wir laufen immer egal was wir machen, über die Zentrale Stelle wo die Daten liegen. Das ist so die Zentrale Idee von Sichtweisen. Wir arbeiten zwar mit Daten, haben aber nicht an jeder Stelle die Daten sondern verwenden die Daten von dem Ort, wo sie gespeichert sind.

Individuelle Eigenschaften und Effizienz

- Programmieraufgaben können auf vielfältige Weise gelöst werden.
- Der Einsatz vorhandener abstrakter Datentypen (ADTs) zielt idealerweise auf die Reduzierung des Programmieraufwands ab.
- Entscheidende Frage: Welche ADTs sollen wie kombiniert werden, um ein möglichst gutes Programm zu erhalten?

Wichtige Kriterien für die Beurteilung der "Güte" eines Programms:

- **Laufzeiteffizienz:** Wie schnell wird das Programm ausgeführt?
- **Spechereffizienz:** Wie viel Speicherplatz benötigt das Programm?
- **Energieeffizienz:** Wie viel Energie verbraucht das Programm (besonders relevant für mobile Geräte)?

Zielsetzung:

- Meist geht es nicht darum, die **bestmögliche** Effizienz in allen Bereichen zu erreichen.
- Vielmehr sollen aus Benutzersicht **inakzeptabel lange Wartezeiten** oder eine **zu kurze Akku-Laufzeit** vermieden werden.

Einfluss der Datenkenntnis:

- Je genauer die verarbeiteten Daten bekannt sind (z.B. Umfang, Struktur, Verteilung), desto leichter lassen sich die genannten Effizienzziele erreichen.

Rolle des Zufalls:

- Die Effizienz kann häufig vom Zufall abhängen (z.B. bei Hash-Tabellen, randomisierten Algorithmen).
- Dies ist nicht zwingend negativ, da das Verhalten einer großen Zahl zufälliger Ereignisse in der Gesamtheit oft zuverlässig abschätzbar ist.

- Voraussetzung dafür ist, dass die tatsächlichen Daten den erwarteten Umfang und die angenommene Zufallsverteilung aufweisen.

Umgang mit Unsicherheiten:

- Wenn die Daten nicht den Erwartungen entsprechen, müssen Vorkehrungen getroffen werden, um dennoch eine akzeptable Effizienz zu gewährleisten.
- Diese Vorkehrungen können jedoch sowohl den Programmieraufwand erhöhen als auch die Effizienz in bestimmten Fällen verringern.

Effizienz und Zuverlässigkeit typischer Datenstrukturen

Zusammenfassung der typischen Eigenschaften hinsichtlich Laufzeit, Speicher- und Energieverbrauch sowie Programmieraufwand:

• Arrays:

- **Effizienz:** Sehr effizient (konstant) für übliche Zugriffe (Index), solange keine häufige lineare Suche in großen Arrays oder häufiges Umkopieren erforderlich ist. Binäre Suche in sortierten Arrays ist recht effizient (logarithmisch). Effiziente Sortierverfahren (z.B. Quicksort) existieren.
- **Programmieraufwand:** Meist akzeptabel.

• Einfache lineare Listen: 4. Lineare Liste und co

- **Effizienz:** Zugriffe am Anfang (inkl. Einfügen/Löschen) und ggf. am Ende (abhängig von der Implementierung) sind sehr effizient (konstant). Suchen und die meisten anderen Zugriffsarten sind ineffizient (linear). Mergesort ist ein effizientes Sortierverfahren für Listen.
- **Speicher- und Energieverbrauch:** Etwas höher als bei Arrays aufgrund des Umgangs mit Referenzen, auch bei effizienten Operationen.
- **Programmieraufwand:** Sehr gering.

• Binäre Suchbäume: 6. Suchbäume

- **Effizienz:** Halten Einträge ständig sortiert und ermöglichen in der Regel eine einigermaßen effiziente Suche (durchschnittlich logarithmisch, im schlechtesten Fall linear). Sortieren ist nicht notwendig.
- **Programmieraufwand:** Höher als bei Listen.
- **Effizienz (Einfügen/Löschen):** Weniger effizient als bei Listen (logarithmisch bzw. linear).
- **Speicherverbrauch:** Höher als bei Listen.

• Hash-Tabellen: (7. Hashing)

- **Effizienz:** Ermöglichen eine recht effiziente Suche, Einfügen und Löschen (konstant), aber nur bei wenigen Kollisionen. Häufige Kollisionen führen zu Ineffizienz (linear).
- **Ordnung/Sortierung:** Keine inhärente Ordnung der Einträge, Sortieren ist nicht sinnvoll.
- **Energieverbrauch:** Recht niedrig bei wenigen Kollisionen.
- **Speicherverbrauch:** Größer als bei Listen, da Hash-Tabellen zur Vermeidung von Kollisionen nicht vollständig gefüllt sein dürfen.

- Der Speicher- und Energieverbrauch der bisher betrachteten Datenstrukturen (Arrays, Listen, Bäume, Hash-Tabellen) ist im Vergleich zu Datenstrukturen für parallele Programmierung relativ gering (im Wesentlichen linear). (Hinweis: Parallele Programmierung wird in EP2 nicht behandelt.)

Empfehlungen zur Auswahl von Datenstrukturen basierend auf typischen Nutzungsszenarien:

bevorzugt in dieser Reihenfolge:

1. Array (wenn nur selten Änderungen der Indexierung nötig)
2. lineare Liste (wenn Zugriffe nur an Enden erfolgen oder Liste kurz bleibt)
3. Hash-Tabelle (wenn keine Sortierung nötig)
4. Baum (sonst, diverse Arten)

- **Arrays:**

- Verwenden, wenn nur einfache Arrayzugriffe (über Index) benötigt werden.
- Auch geeignet für binäre Suche, wenn nur selten Änderungen an der Datenmenge erfolgen (da bei Änderungen eine erneute Sortierung notwendig sein kann).

- **Lineare Listen:**

- Verwenden, wenn Zugriffe hauptsächlich oder ausschließlich am Anfang oder Ende der Liste stattfinden.
- Auch geeignet, wenn die Länge der Liste voraussichtlich sehr kurz bleibt.

- **Hash-Tabellen:**

- Verwenden, wenn keine Sortierung der Daten erforderlich ist.

- **Diverse Arten von Bäumen (z.B. binäre Suchbäume):**

- Verwenden in Fällen, die nicht optimal zu Arrays, Listen oder Hash-Tabellen passen, insbesondere wenn eine gewisse Ordnung der Daten nützlich ist und ein Kompromiss zwischen Such-, Einfüge- und Löschoperationen gesucht wird.

Zufall

Effizienz von Hash-Tabellen, Bäumen, ... stark zufallsabhängig

solange Daten zufällig verteilt → **Zufall sehr zuverlässig**

Gefahr: Zufall unterbunden, Daten **nicht zufällig** verteilt

Reihe sortierter, gleicher oder beieinander liegender Werte \neq zufällig

z.B. Verteilung der Daten unbekannt, Zufall ungerechtfertigt angenommen

z.B. schlechte Datenverteilung absichtlich herbeigeführt (Denial-of-Service-Attack)

Gegenmaßnahmen verschlechtern Effizienz, Fehlersicherheit und Wartungsfreundlichkeit, nur notwendig, wenn Gefahr besteht

Zufall und Hash-Tabellen:

- Die Effizienz von Hash-Tabellen ist stark vom Zufall abhängig.
- Wesentliche Operationen (Suchen, Einfügen, Entfernen) sind effizient (konstant), wenn Kollisionsbehandlungen selten sind.
- Seltene Kollisionen treten auf, wenn sich die Hash-Werte der meisten Einträge in den für die Indexberechnung relevanten Bereichen voneinander unterscheiden.
- Die Wahrscheinlichkeit für gleiche Hash-Werte ist geringer in Hash-Tabellen mit geringer Füllung als in fast vollen Tabellen.

Maßnahmen zur Unterstützung des "Zufalls" und zur Verbesserung der Hash-Tabellen-Effizienz:

1. **Gute Hash-Funktionen:** Verwenden von Hash-Funktionen, die die Hash-Werte der Eingabedaten gut streuen und die Wahrscheinlichkeit von Kollisionen minimieren.
2. **Angemessener maximaler Füllgrad:** Beschränken des maximalen Füllgrads der Hash-Tabelle (z.B. auf ca. 80%), um die Wahrscheinlichkeit von Kollisionen gering zu halten.

Durch die Kombination dieser beiden Maßnahmen können Hash-Tabellen oft recht effiziente Operationen ermöglichen. Es ist jedoch wichtig zu beachten, dass *Ausreißer* möglich sind, d.h., gelegentlich können Zugriffe deutlich mehr Zeit als im Durchschnitt benötigen (wenn es doch zu Kollisionen kommt).

- Binäre Suchbäume zeigen ebenfalls eine starke Abhängigkeit von den eingefügten Daten bezüglich ihrer Effizienz.

Einfluss der Einfügereihenfolge auf die Baumstruktur:

- Die resultierende Struktur eines binären Suchbaums wird maßgeblich durch die Reihenfolge bestimmt, in der die Einträge in den Baum eingefügt werden.
- **Schlechter Fall (Entarteter Baum):**
 - Werden Elemente in streng aufsteigender oder absteigender Reihenfolge eingefügt, wird tendenziell nur einer der beiden Kind-Zeiger (`left` oder `right`) jedes Knotens verwendet, während der andere `null` bleibt.
 - Ein solcher Baum degeneriert zu einer linearen Liste.
 - In diesem entarteten Fall wird die Suche und das Einfügen ineffizient, da im Wesentlichen die gesamte "Liste" traversiert werden muss (lineare Komplexität).
- **Bester Fall (Balancierter Baum durch zufällige Einfügereihenfolge):**
 - Erfolgen die Einfügungen in einer zufälligen Reihenfolge, ist die Wahrscheinlichkeit sehr hoch, dass der Baum eine gut ausbalancierte Struktur aufweist.
 - In einem gut ausbalancierten Baum sind sowohl die Suche als auch das Einfügen von Elementen effizient (logarithmische Komplexität im Durchschnitt).
- **Auftreten entarteter Bäume:**
 - Entartete Bäume entstehen primär dann, wenn die Einfügereihenfolge der Daten nicht zufällig ist, sondern einer bestimmten Struktur folgt (diese muss nicht zwingend eine vollständig sortierte Reihenfolge sein).
- **Umgang mit strukturierten Eingabedaten:**
 - Wenn die Daten, die in einen binären Suchbaum eingefügt werden sollen, eine inhärente Struktur aufweisen, ist es notwendig, die Einfügereihenfolge so zu manipulieren (abhängig von der Struktur der Daten), dass ein gut geformter, sogenannter *balancierter Baum* entsteht.
 - Balancierte Bäume (wie z.B. AVL-Bäume, Rot-Schwarz-Bäume) garantieren auch im schlechtesten Fall eine logarithmische Komplexität für wichtige Operationen. Das Konzept der balancierten Bäume wird in Abschnitt 3.3.2 näher erläutert.
- Solange gelegentliche Effizienzeinbrüche tolerierbar sind, kann es sinnvoll sein, sich auf die Wahrscheinlichkeit einer guten Datenverteilung zu verlassen. Dies führt tendenziell zu einfacheren und effizienteren Programmen. Bei inakzeptabel häufigen oder langen Wartezeiten in der Praxis kann man später immer noch Mechanismen zur Vermeidung von Ausreißern implementieren.

- Bei typischen, nahezu zufälligen Datenverteilungen in der Praxis ist die wahrgenommene Effizienz meist zufriedenstellend. Die Wahrscheinlichkeit einer zufällig auftretenden, extrem schlechten Datenverteilung ist gering.

Bewusst schlechte Datenverteilung:

- In Ausnahmesituationen oder durch gezielte Manipulation (z.B. Denial-of-Service-Angriffe) kann eine sehr ungünstige Datenverteilung systematisch entstehen oder absichtlich herbeigeführt werden. Beispiele hierfür sind plötzliche, massive Anfragen für dasselbe Element in einem System oder gezielte Angriffe, die Schwächen in der Implementierung aufgrund schlechter Datenverteilung ausnutzen.
- Für Programme, bei denen solche Risiken bestehen, ist es entscheidend, sie gegen erhebliche Effizienzeinbrüche bei ungünstigen Datenverteilungen abzusichern. Die Vermeidung von Ausreißern wird in diesen Fällen zu einer Sicherheitsmaßnahme.
- Nicht nur Datenstrukturen, sondern auch viele Algorithmen sind von der Verteilung der Eingabedaten betroffen.

Sortieren und Zufall

- Beispiel: Quicksort:**
 - Ein häufig für Arrays verwendetes Sortierverfahren.
 - Im Durchschnitt und in den meisten Fällen sehr effizient ($O(n \cdot \log n)$) mit einem kleinen konstanten Faktor).
 - Hat jedoch einen Worst-Case von $O(n^2)$, vergleichbar mit dem schlechtesten Fall von Bubblesort.
 - Um den Worst-Case zu vermeiden, wird die Pivot-Elementauswahl meist nicht vom Anfang des Arrays vorgenommen, sondern aus der Mitte oder durch eine zufällige Wahl.
 - Quicksort wird selten für Listen verwendet, da der ineffiziente Zugriff auf Elemente in der Mitte die Wahrscheinlichkeit des Worst-Case kaum reduziert.
- Beispiel: Mergesort:**
 - Auch im schlechtesten Fall effizient ($O(n \cdot \log n)$), jedoch mit einem tendenziell größeren konstanten Faktor als Quicksort).
 - Im durchschnittlichen und besten Fall nicht besser als im schlechtesten Fall, da Listen unabhängig vom Inhalt immer gleich geteilt werden.
- Beispiel: Bubblesort:**
 - Im Durchschnitt und im schlechtesten Fall ineffizient ($O(n^2)$).
 - Kann im besten Fall (bereits sortierte Daten) effizienter sein als Quicksort und Mergesort ($O(n)$).
- Auswahl des Sortierverfahrens:**
 - Bei unbekannten Daten verwendet man für Arrays eher Quicksort und für Listen eher Mergesort.
 - Bei hoher Gefahr von Effizienzausreißern kann auch Mergesort für Arrays in Betracht gezogen werden.
 - Je nach spezifischen Eigenschaften der Daten und der Anwendungsfälle können viele andere Sortierverfahren sinnvoll sein.
- Beispiel: Sortierter Iterator (Listing 3.35):**
 - Generell sehr ineffizient, da bei jeder Suche nach dem nächsten Element die gesamte Liste durchlaufen wird.

- Kann in Spezialfällen effizient sein, z.B. wenn nur auf wenige Elemente am Anfang einer bereits sortierten Liste zugegriffen wird, da in diesem Fall eine vollständige Sortierung unnötig ist.

10. Garantie statt Zufall

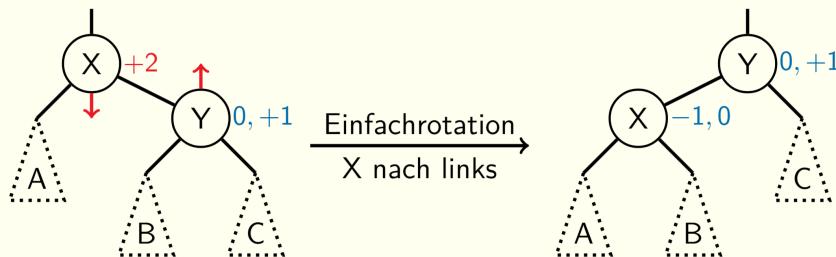
AVL-Baum

Für eine bessere Erklärung von AVL-Bäumen siehe [AlgoDat](#)

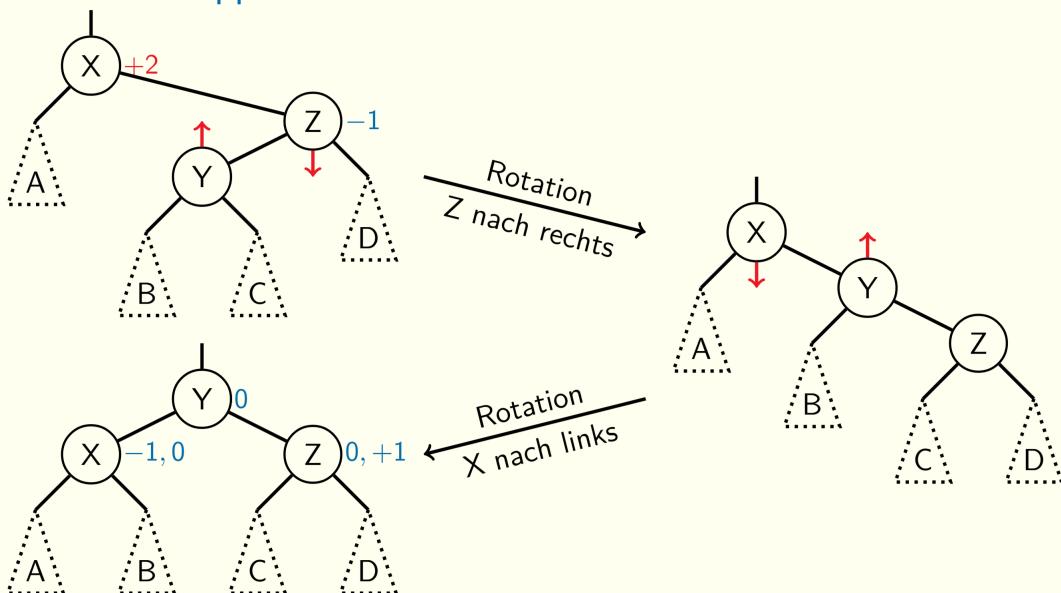
Kriterium für jeden Knoten in AVL-Baum (binärer Suchbaum):

Tiefe des linken Teilbaums unterscheidet sich von Tiefe des rechten Teilbaums um max. 1.

Rotieren wenn Kriterium verletzt (einfach oder doppelt, links oder rechts)



AVL-Baum – Doppelrotation



Balancierte Bäume: Garantierte Effizienz

Um das Problem **entarteter Bäume** (die wie eine Liste werden und die Effizienz stark reduzieren) zu vermeiden, werden **balancierte Bäume** verwendet.

- **Grundprinzip:** Bei einem Ungleichgewicht zwischen linkem und rechtem Teilbaum werden Knoten verschoben (Rotationen), um die Balance wiederherzustellen.
- **Vorteile:**
 - **Vermeidung von Effizienzproblemen:** Das Entarten des Baumes wird verhindert, was eine garantierte Laufzeit für Operationen (Suchen, Einfügen, Löschen) sicherstellt.
- **Nachteile/Mehraufwand:**
 - **Zusätzlicher Aufwand** beim Einfügen und Löschen (durch Rotationen).
 - **Höherer Speicherverbrauch** für Balance-Informationen pro Knoten.

- Erhöhtes Komplexität in Entwicklung und Wartung aufgrund vieler Fallunterscheidungen für Rotationen.

Beispiel: AVL-Baum

- **Eigenschaft:** Bei einem AVL-Baum darf sich die Höhe der beiden Teilbäume unter jedem Knoten um höchstens 1 unterscheiden.
- **Balance-Variable:** Jeder Knoten enthält eine `balance`-Variable, die den Höhenunterschied speichert. Erlaubte Werte sind 0, 1, -1.
- **Rotieren:** Tritt nach Einfügen/Löschen ein größerer Unterschied auf, wird der Baum durch **Rotieren** umstrukturiert.
 - **Rotation:** Zwei oder drei verkettete Knoten (mit ihren Teilbäumen) werden so vertauscht, dass die Höhenunterschiede ausgeglichen sind, während die Sortierung erhalten bleibt.
 - **Laufzeit:** Der zusätzliche Aufwand für Rotationen ist in Bezug auf die Laufzeit effizient.

Listing 3.42 Einfügen in AVL-Baum als Variante von Listing 3.32 (Ausschnitt)

```

1  public class OTNode {
2      private String elem;
3      private OTNode left, right;
4      private int balance = 0;
5
6      public OTNode add(String e) {
7          if (compare(e) > 0) {
8              if (right == null) {
9                  right = new OTNode(e);
10                 ++balance;
11             } else {
12                 OTNode n = right.add(e);
13                 if (n != null) {
14                     right = n;
15                     return this;
16                 }
17                 if (++balance > 1) {
18                     if (right.balance > 0) {
19                         n = right;
20                         balance = 0;
21                     } else {
22                         n = right.left;
23                         balance = right.balance = 0;
24                         if (n.balance > 0) {
25                             balance = -1;
26                         } else if (n.balance < 0) {
27                             right.balance = 1;
28                         }
29                         right.left = n.right;
30                         n.right = right;
31                     }
32                     n.balance = 0;
33                     right = n.left;
34                     n.left = this;
35                     return n;
36                 }
37             }
38         } else { // Symmetrischer Zweig für linken Teilbaum
39         }
40         return balance != 0 ? null : this;
41     }
42 }
```

- Java-spezifische Implementierungsherausforderungen (aus Listing 3.42 abgeleitet):

- Java hat keine Zeiger auf Objektvariablen wie C, was das Ändern des Elternknotens erschwert.
- Methoden können nur einen Ergebniswert zurückgeben, was problematisch ist, wenn sowohl ein ersetzender Knoten als auch Höheninformationen zurückgegeben werden müssen.
- add -Methode im Beispiel gibt `null` zurück, wenn der Teilbaum gewachsen ist (und `this` unverändert bleibt), oder den ersetzenden Knoten (oder `this` selbst), wenn die Höhe unverändert ist. Diese Fälle schließen sich gegenseitig aus.

Listing 3.43 Variante von Listing 3.31 als AVL-Baum

```

1  public class OrderedTree implements Ordered {
2      ...
3      public void add(String e) {
4          cnt[0]++;
5          if (root != null) {
6              OTNode n = root.add(e);
7              if (n != null) { root = n; }
8          } else { root = new OTNode(e); }
9      }
10     public int removeAll(String e) { ... };
11 }
```

- Aufrüfer der `add`-Methode müssen den Wurzelknoten (oder den jeweiligen Kindknoten) aktualisieren, falls die Methode einen neuen Knoten zurückgibt.

Kombination von Hash-Tabellen mit Balancierten Bäumen

Hash-Tabellen sind sehr effizient wenn Hash-Funktionen gut streuen.

aber:

gute Hash-Funktionen erfordern Expertenwissen
bekannte Hash-Funktionen können bewusst ausgehebelt werden

daher häufig kombinierte Strategie, z.B.

lineare Liste zur Kollisionsbehandlung und
zu lange Liste wird in balancierten Baum umgewandelt
(Sortierkriterium z. B. Speicheradresse damit jedes Objekt sortierbar)

Die Qualität von Hash-Funktionen ist entscheidend für die Effizienz von Hash-Tabellen. Um Ausreißer bei schlechten Hash-Funktionen oder hohem Füllgrad zu vermeiden, können Hash-Tabellen mit balancierten Bäumen kombiniert werden.

- **Konzept:** Das Array der Hash-Tabelle speichert nicht direkt Werte, sondern **Wurzeln von Bäumen**.
 - Alle Einträge, die denselben Array-Index (Hash-Kollision) ergeben, werden in einem dieser Bäume gespeichert.
- **Vorteile:**
 - **Robustheit:** Eine schlecht streuende Hash-Funktion reduziert die Effizienz maximal auf die eines Baumes (oder knapp darunter), verhindert aber keinen "Absturz" der Laufzeit auf $O(n)$.
 - **Gute Hash-Funktionen:** Bei gut streuenden Hash-Funktionen liegt die Effizienz nahe an der einfacher Hash-Tabellen ($O(1)$ im Durchschnitt).
 - **Füllgrad-Unabhängigkeit:** Die Effizienz ist weniger stark vom Füllgrad der Tabelle abhängig. Die Anzahl der Einträge kann sogar deutlich größer sein als die Array-Größe.

- **Flexibilität:** Die Größe des Arrays kann gewählt werden, um die Effizienz zwischen der einer Hash-Tabelle und der eines Baumes zu steuern.
- **Nachteile:**
 - **Hoher Implementierungsaufwand** und **Speicherverbrauch**. Es werden die Nachteile beider Datenstrukturen (Hash-Tabellen und Bäume) kombiniert.

Alternative: Listen statt Bäume (häufig in Java)

- Oft enthalten Arrays in Hash-Tabellen **lineare Listen von Einträgen** zur Kollisionsbehandlung.
- **Vorteil:** Listen sind **einfacher zu handhaben** als Bäume.
- **Nachteile:**
 - Die Suche in langen Listen ist **sehr aufwendig** ($O(n)$).
 - Erfordert **gut streuende Hash-Funktionen** und einen **nicht zu großen Füllgrad**.
 - **Kein Schutz** gegen schlechte Datenverteilung (ein Index kann extrem lang werden).

Hybrid-Ansatz: Liste und Baum Kombination

- **Mittelweg:** Für wenige Einträge auf einem Index wird eine lineare Liste verwendet. Bei einer bestimmten **Längeschwelle** wird die Liste in einen balancierten Baum umgewandelt.
- **Vorteil:** Zugriffe auf sehr kurze Listen sind oft effizienter als auf kleine Bäume (weniger Overhead).
- **Implementierung:** Der Typ der Array-Einträge kann ein Interface sein, das sowohl von Listen als auch von Bäumen implementiert wird. Dynamisches Binden entscheidet, welche Struktur verwendet wird.

Sicherheit durch mehrere Hash-Funktionen

(Hashcodes + 7. Hashing)

- Um die Robustheit gegen schlechtes Datenelement zu erhöhen, kann bei einer Kollision auf eine **andere Hash-Funktion** (oder einen anderen Bit-Muster-Bereich) mit anderer Streuung umgeschaltet werden.

Generizität (kurzgefasst aus Anwendersicht)

Klasse, Interface, Methode hat **Typparameter** (Parameter steht für Typ)

z.B. sind K und V Typparameter in Map<K, V>

bei Verwendung müssen Typparameter **statisch durch Referenztypen** ersetzt werden

z.B. Map<String, Integer> verwendet String für K und Integer für V,

z.B. Map<> wenn Compiler Typen aus Kontext errechnen kann,

aber Map ohne Parameter (Raw-Type) ist falsch obwohl vom Compiler akzeptiert

gebundene Typparameter sind nur durch Untertypen der **Schranke** ersetzbar

z.B. SortedSet<E extends Comparable<E>> hat Schranke Comparable<E> auf E

zu jedem elementaren Typ existiert entsprechender Referenztyp

z.B. Integer ist Referenztyp zu int, Char ist Referenztyp zu char;

durch **Auto-Boxing/Unboxing** fast wie elementarer Typ verwendbar

Angebot von Java: Collections Framework

--> Skriptum, p.118

Interfaces:

Collection<E>			Map<K, V>
Set<E>	List<E>	Queue<E>	SortedMap<K, V>
SortedSet<E>		Deque<E>	

Klassen:

	Set<E>	Map<K, V>
Hash-Tabelle	HashSet<E>	HashMap<K, V>
Baum	TreeSet<E>	TreeMap<K, V>
Hash-Tab.+Liste	LinkedHashSet<E>	LinkedHashMap<K, V>

	List<E>	Deque<E>
Array	ArrayList<E>	ArrayDeque<E>
Liste	LinkedList<E>	LinkedList<E>

Das **Java Collections Framework** (`java.util`) bietet eine Sammlung vordefinierter abstrakter Datentypen (Interfaces) und deren Implementierungen.

Wichtige Interfaces im Java Collections Framework:

- **Collection<E>** :
 - Basis-Interface für Datensammlungen, wobei `E` der Typ der Einträge ist.
 - Wird für allgemeine, iterierbare Datensammlungen verwendet.
 - Enthält zahlreiche, aber nicht spezifische Methoden.
 - **Vorsicht:** Einige Methoden wie `add`, `remove`, `clear` sind als "optional" gekennzeichnet. Implementierungen können hier `UnsupportedOperationException` werfen.
 - Ähnelt einer generischen Version von `Ordered` (aus Listing 3.24), mit `boolean`-Rückgabewerten für `add`, `contains`, `remove`.
- **Set<E>** :
 - Erweitert `Collection<E>`.
 - Stellt eine **Menge im mathematischen Sinne** dar: Gleiche Einträge dürfen **nicht mehrfach** vorkommen.
- **SortedSet<E>** :
 - Erweitert `Set<E>`.
 - Alle Einträge sind **vollständig sortiert**.
 - Typ `E` muss das Interface `Comparable<E>` implementieren, um Sortierbarkeit zu gewährleisten (verwendet `compareTo`).
 - Einträge sollten nicht verändert werden, um die Sortierung nicht zu zerstören.
 - Vergleichbar mit dem `Ordered`-Interface (Listing 3.25).
- **List<E>** :
 - Erweitert `Collection<E>`.
 - Erlaubt **mehrfaches Eintragen** desselben Objekts.
 - Unterstützt **Zugriffe über einen Index** (wie in Listing 3.26).
 - Wird aufgrund oft ineffizienter Indexzugriffe seltener verwendet als `ArrayList`.
- **Queue<E>** :

- Erweitert `Collection<E>`.
- Bietet Methoden einer **Warteschlange (Queue)** in zwei Varianten:
 - `offer`, `poll`, `peek`: Geben spezielle Werte (z.B. `false`, `null`) zurück, wenn Operationen nicht ausgeführt werden können.
 - `add`, `remove`, `element`: Werfen eine `Exception` (Laufzeitfehler) in solchen Fällen.
- **Deque<E>**:
 - Erweitert `Queue<E>`.
 - Bietet Methoden einer **doppelseitigen Warteschlange (Double-Ended Queue)** und eines **Stacks**.
 - Methoden ebenfalls in zwei Varianten (mit Rückgabewert oder Exception bei Fehlern).
- **Map<K,V>**:
 - Beschreibt **assoziative Datenstrukturen** (Schlüssel-Wert-Paare).
 - `K` ist der Typ der Schlüssel, `V` der Typ der assoziierten Werte.
 - Schlüssel dürfen nicht verändert werden, da Einträge sonst nicht mehr auffindbar wären.
 - Methoden ähneln denen aus Listing 3.23, sind aber zahlreicher.
- **SortedMap<K,V>**:
 - Erweitert `Map<K,V>`.
 - Alle Einträge sind **vollständig nach Schlüsseln sortiert** (mittels `compareTo`).
- Es gibt weitere Interfaces für parallele Programmierung und spezielle Aufgaben.

Implementierungen im Java Collections Framework:

Die meisten Implementierungen sind robust gegenüber schlechten Hash-Funktionen oder Datenverteilungen und streben gleichzeitig eine gute durchschnittliche Effizienz an. Implementierungsdetails sind oft nicht offen gelegt, aber der Quellcode ist zugänglich.

- **HashSet<E>**:
 - Implementiert `Set<E>` als **Hash-Tabelle**.
 - Konstruktor erlaubt die Angabe von Anfangsgröße und Füllgrad. Standardwerte sind meist gut.
 - **Reihenfolge der Iteratoren ist unvorhersehbar**.
 - `LinkedHashSet<E>`: Variante, die Einträge zusätzlich in einer linearen Liste verkettet, um die **Reihenfolge des Eintragens** bei Iteratoren zu bewahren.
- **HashMap<K,V>**:
 - Implementiert `Map<K,V>` als **Hash-Tabelle**.
 - Gleichtes gilt wie für `HashSet<E>`, inklusive der Variante `LinkedHashMap<K,V>`.
- **ArrayList<E>**:
 - Implementiert `List<E>` als **Array**.
 - Größe passt sich automatisch an.
 - **Effiziente Array-Zugriffe**, andere Operationen (Einfügen/Löschen in der Mitte) sind eher ineffizient.
 - Oft als Ersatz für normale Arrays verwendet, wenn die Größe im Voraus unbekannt ist; selten als typische "Liste" eingesetzt.
- **ArrayDeque<E>**:
 - Effiziente Implementierung von `Deque<E>` als Array mit automatischer Größenanpassung.
 - Fokus auf Effizienz schränkt Anwendbarkeit ein (z.B. `null` als Eintrag verboten).
- **LinkedList<E>**:

- Implementiert `List<E>` und `Deque<E>` als **doppelt verkettete Liste**.
- `null` als Eintrag ist erlaubt.
- `TreeSet<E>` :
 - Implementiert `SortedSet<E>` als **balancierter binärer Suchbaum** (kein AVL-Baum, oft Red-Black Tree).
 - Sortierungsmethode kann im Konstruktor festgelegt werden.
- `TreeMap<K,V>` :
 - Implementiert `Map<K,V>` als **balancierter Baum**, basierend auf derselben Struktur wie `TreeSet<E>`.
- **Arrays (Klasse, kein Interface):**
 - Enthält häufig verwendete **statische Methoden** auf Arrays (z.B. Sortieren, Suchen).
 - Nicht Teil des Java Collections Frameworks, aber eine wichtige Alternative für Datensammlungen, wenn Arrays passend sind.

Auswahl von Interface und Implementierung:

1. Beginne immer mit der Auswahl eines **geeigneten Interfaces** basierend auf der benötigten Funktionalität.
2. Wenn mehrere Interfaces in Frage kommen, wähle das **einfachere (Obertyp)**.
3. Entscheide dich danach für eine **Implementierung**, unter Berücksichtigung von Effizienzkriterien (siehe Abschnitt 3.3.1).
4. Verwende die konkrete Klasse (Implementierung) üblicherweise nur bei der **Objekterzeugung**.
5. An allen anderen Stellen (insbesondere Variablen Deklarationen) sollte das **Interface als Typ** verwendet werden.
 - **Vorteil:** Ermöglicht den einfachen Austausch der Implementierung an einer zentralen Stelle, ohne den restlichen Code anpassen zu müssen.
6. Methoden sind über die Interfaces hinweg **weitgehend vereinheitlicht und sprechend benannt**, was die Einarbeitung erleichtert.

Qualität in der Programmierung



Qualität in der Programmierung

Umgang mit und Vermeidung von Fehlern zur Laufzeit

- Ausnahmebehandlung

- Ein- und Ausgabe als Fallbeispiel

- Überprüfung von Eingabedaten

Fehlervermeidung und Effizienz durch statisches Programmverstehen

- Design-by-Contract (Schnittstellen)

- Hoare-Logik (interner Programmablauf)

- Code-Review

Kontrollieren und Verbessern der Qualität

- Testen

- Qualitätsbewusstsein

- Optimieren

Das Konzept der Softwarequalität ist vielschichtig. Dieses Kapitel behandelt verschiedene Aspekte, die zur Verbesserung der Programmqualität beitragen.

Umgang mit unerwarteten Zuständen (Laufzeit)

Werfen und Propagieren von Ausnahmen

Laufzeitfehler → **Ausnahme ausgelöst („geworfen“)**

Ausnahme ist Objekt eines von Throwable abgeleiteten Typs

explizit Ausnahme werfen: z.B. `throw new IllegalArgumentException();`

Ausnahme an Aufrufer der aktuellen Methode **propagiert** (weitergereicht)

Programmabbruch wenn über alle Methoden der Aufrufhierarchie propagiert, dabei **Stack-Trace** ausgegeben

```
Exception in thread "main" java.lang.IllegalArgumentException
  at Infinity.main(Infinity.java:4)
  at Infinity.main(Infinity.java:6)
```

Dieser Abschnitt behandelt die Fehlerbehandlung während der Programmausführung in Java, insbesondere durch Ausnahmebehandlung (Exception Handling) und die Validierung von Eingabedaten.

Ausnahmebehandlung in Java (Exception Handling)

Ausnahmebehandlung in Java

```

try {
    // normaler Block, außer dass catch und/oder finally folgt,
    // ausgeführt bis Ausnahme ausgelöst (oder bis Ende ohne Ausnahme)
} catch(A e) {
    // nur ausgeführt wenn im try-Block Ausnahme e vom Typ A ausgelöst,
    // danach ist e abgefangen, e wird nicht weiter propagiert
} catch(B e) {
    // Auffangen von Ausnahme vom Typ B,
    // beliebig viele catch-Blöcke
} finally { // optional
    // nach try-Block ausgeführt, egal ob Ausnahme ausgelöst/abgefangen
    // dient zum Aufräumen
}

```

Grundprinzip der Ausnahmebehandlung

In Java werden alle Laufzeitfehler einheitlich behandelt:

- **Ausnahme auslösen/werfen:** Wenn ein Programm in eine nicht normal fortsetzbare Situation gerät, wird der Programmfluss unterbrochen und eine **Ausnahme (Exception)** ausgelöst (`thrown`).
- **Propagation (Weitergabe):** Die ausgelöste Ausnahme wird schrittweise von der aktuellen Methode an ihren Aufrufer weitergegeben (`propagated`). Dabei werden die sogenannten Stack-Frames (Informationen der aufgerufenen Methoden auf dem Call Stack) abgebaut.
- **Programmbeendigung:** Wenn die Ausnahme bis zum initialen Aufruf von `main` propagiert wird und dort nicht behandelt wurde, wird das Programm beendet.
 - Dabei wird eine Fehlermeldung ausgegeben, die den Typ der Ausnahme und einen **Stack-Trace** enthält.
 - Der Stack-Trace listet alle Methoden und Programmstellen auf, die zum Zeitpunkt des Auslösens der Ausnahme aktiv waren, und hilft bei der Fehlersuche.
- **Ausnahmetypen:** Abhängig vom Grund des Fehlers gibt es viele verschiedene Ausnahmetypen, die jeweils durch eine Java-Klasse repräsentiert werden (z.B. `NullPointerException`, `StackOverflowError`).

Abfangen von Ausnahmen (Exception Handling)

Java ermöglicht es, den Propagationsprozess von Ausnahmen zu unterbrechen und sie zu behandeln.

Dies geschieht in einem **try-catch -Block**.

Listing 4.1 Auffangen einer Ausnahme – schlechter Programmierstil

```

1  public class HandledException {
2      private static void printLen(String s) {
3          System.out.print(s.length());
4      }
5      public static void main(String[] args) {
6          try {
7              printLen(null);
8              System.out.print("nicht ausgeführt");
9          }
10         catch(NullPointerException e) {
11             System.out.print("abgefangen ");
12         }
13         System.out.print("weiter");
14     }
15 }
```

• **Ablauf im `try-catch`-Block:**

1. Der Code im `try -Block` wird ausgeführt.
2. Tritt währenddessen eine Ausnahme auf, wird der `try -Block` sofort abgebrochen.
3. Das Programm sucht anschließend nach einem passenden `catch -Block`.
4. Ein `catch -Block` ist passend, wenn sein deklarierter Ausnahmetyp **ein Oberotyp des tatsächlich aufgetretenen Ausnahmetyps** ist.
5. Der passende `catch -Block` wird ausgeführt und ersetzt den abgebrochenen `try -Block`.
6. Nach der Ausführung des `catch -Blocks` wird die Programmausführung **nach dem gesamten `try-catch -Block fortgesetzt`**.
7. Hat kein `catch -Block` einen passenden Typ, wird die Ausnahme normal weiterpropagiert.

• **Beispiel (Listing 4.1):**

- Der Aufruf von `printLen(null)` in `main` (innerhalb des `try -Blocks`) führt zu einer `NullPointerException`.
- Diese wird von dem `catch (NullPointerException e)` -Block abgefangen.
- Die Ausgabe ist " abgefangen weiter ", da der `System.out.print(" nicht ausgeführt ");` im `try -Block` übersprungen wird und nach dem `catch -Block` fortgefahren wird.
- **Syntax `catch`:** Der Typ im `catch -Block` wird wie ein formaler Parameter deklariert (z.B. `NullPointerException e`), und dieser Parameter kann innerhalb des `catch -Blocks` verwendet werden, um auf das ausgelöste Ausnahmeobjekt zuzugreifen.

Wichtiger Hinweis: Vermeidung von Ausnahmen bei vorhersehbaren Fällen

Wenn der Programmablauf wie in Listing 4.1 **vorhersehbar** ist (z.B. wenn bekannt ist, dass ein Parameter `null` sein könnte), sollte man **Ausnahmen vermeiden**. Stattdessen sollten übliche Kontrollstrukturen wie **bedingte Anweisungen** (`if -Statements`) eingesetzt werden, um solche Fälle proaktiv zu behandeln. Ausnahmebehandlung sollte für **unerwartete** und **unvorhersehbare** Fehlerzustände reserviert sein.

Einsatzbereiche der Ausnahmebehandlung

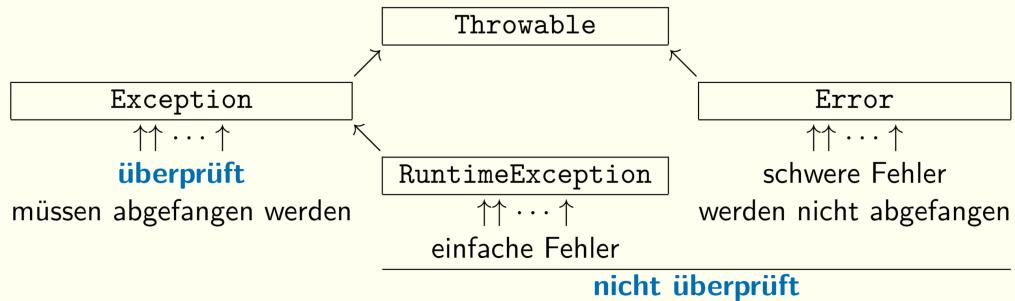
Ausnahmebehandlungen eignen sich hervorragend, um auf **außergewöhnliche Ereignisse** zu reagieren.

Listing 4.2 Auffangen von Ausnahmen zwecks freundlicherer Fehlermeldung

```

1 public static void main(String[] args) {
2     try { ... }
3     catch(Exception e) {
4         // store data and status information
5         System.out.println("Entschuldigung");
6     }
7 }
```

- Zweck von Listing 4.2:** Ein allgemeiner `catch (Exception e)`-Block kann alle Ausnahmen auffangen, die einen sauberen Programmabschluss zulassen. Dies ermöglicht es, eine benutzerfreundliche Fehlermeldung auszugeben oder andere Aufräumarbeiten durchzuführen, anstatt das Programm abzustürzen zu lassen.

Hierarchie der Ausnahmetypen in Java**Arten von Ausnahmen****überprüfte Ausnahmen** müssen

lokal in passendem `catch`-Block abgefangen werden
oder in Methode vorkommen, die `throws`-Klausel dafür enthält

```

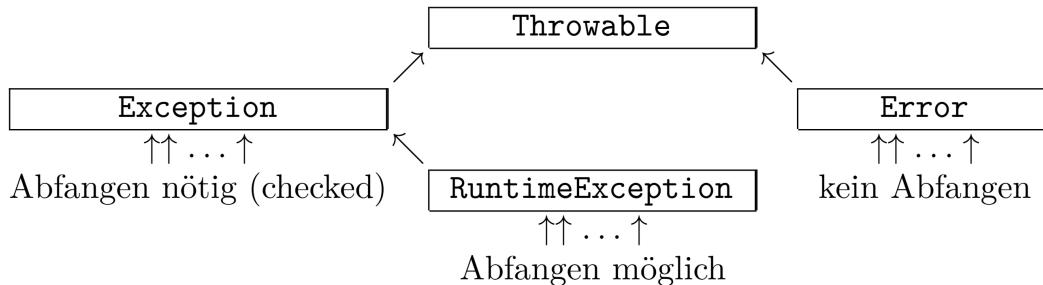
public class MyException extends Exception { ... }
... public void foo() throws MyException { ... } ...
```

Alle Ausnahmetypen in Java sind von der Klasse `Throwable` abgeleitet, die zwei Hauptunternübertypen besitzt:

- `Throwable`
 - `Error`:
 - Repräsentiert **schwerwiegende Ausnahmen**, nach denen eine weitere Programmausführung oft sinnlos ist (z.B. `OutOfMemoryError`).
 - Diese Ausnahmen werden **nicht überprüft (unchecked)** und müssen nicht explizit abgefangen werden.
 - `Exception`:
 - Alle anderen Ausnahmen sind von `Exception` abgeleitet.
 - `RuntimeException`:
 - Ein Untertyp von `Exception`.
 - Diese Ausnahmen werden vom Java-Laufzeitsystem geworfen (z.B. `NullPointerException`, `ArrayIndexOutOfBoundsException`).
 - Sie sind **nicht überprüft (unchecked)**. Man muss stets mit ihrem Auftreten rechnen, aber der Compiler erzwingt kein Auffangen.

- **Andere Exception -Typen (nicht RuntimeException):**
 - Diese sind **überprüft (checked).
 - Ihre Verwendung ist eingeschränkt: Der Java-Compiler erzwingt, dass sie explizit behandelt werden.

Folgende Untertypbeziehungen existieren auf Typen von Ausnahmen:



Überprüfte (Checked) vs. Nicht überprüfte (Unchecked) Ausnahmen

- **Nicht überprüfte Ausnahmen (Error und RuntimeException):**
 - Der Compiler verlangt kein explizites Abfangen.
 - Sie zeigen oft Programmierfehler oder nicht behebbare Systemprobleme an.
- **Überprüfte Ausnahmen (Exception , außer RuntimeException):**
 - Der Java-Compiler überprüft, ob jede Anweisung, die eine solche Ausnahme auslösen könnte, entweder:
 - Innerhalb eines `try`-Blocks mit einem passenden `catch`-Block liegt.
 - Oder in einer Methode auftritt, die eine `throws` Klausel in ihrer Signatur enthält.
 - **throws -Klausel:** Eine Methode, die eine überprüfte Ausnahme auslösen kann, ohne sie selbst abzufangen, muss dies in ihrer Signatur deklarieren (z.B. `public String readLine() throws IOException`). Dadurch wird die Verantwortung für die Ausnahmebehandlung an den Aufrufer delegiert.

Manuelles Werfen von Ausnahmen (throw)

Mit der Anweisung `throw x;` kann man das Auslösen einer Ausnahme manuell veranlassen, wobei `x` ein Objekt der Klasse `Throwable` sein muss.

Listing 4.3 Bei unzweckmäßiger Argument Ausnahme zum Aufrufer propagiert

```

1 public String get(int i) {
2     if (i < 0) throw new IllegalArgumentException();
3     ... // following statements unreachable if i < 0
4 }
  
```

- **Zweck von Listing 4.3:** Zeigt, wie eine `IllegalArgumentException` geworfen wird, wenn ein ungültiger Parameter übergeben wird. Dies stellt sicher, dass unerwünschte Zustände frühzeitig erkannt und kommuniziert werden.
- Für die Ausnahmebehandlung macht es keinen Unterschied, ob eine Ausnahme vom Laufzeitsystem oder explizit mit `throw` geworfen wurde.

Eigene Ausnahmetypen definieren

Man kann eigene Ausnahmetypen definieren, indem man eine neue Klasse schreibt, die von einem bestehenden Ausnahmetyp abgeleitet ist (meistens von `Exception`, wodurch sie zu überprüften Ausnahmen werden).

Listing 4.4 Initialisieren und verwenden der Objektvariablen in Ausnahmen

```

1  public void exceptionThrowingMethod(int i) {
2      if (i < 0) {
3          String s = "i >= 0 required, " + i + " found";
4          throw new IllegalArgumentException(s);
5      }
6  }
7  public void exceptionCatchingMethod() {
8      try { exceptionThrowingMethod(-3); }
9      catch(Exception e) {
10         System.out.println(e.getMessage());
11         System.out.println(e);
12         e.printStackTrace();
13         throw new RuntimeException(e);
14     }
15 }
```

- **Konstruktor-Argument (String):** Ein `String`-Argument im Konstruktor einer Ausnahme (z.B. `new IllegalArgumentException(s)`) wird in der Regel als "Message" gespeichert.
- **getMessage()**: Diese "Message" kann über die Methode `getMessage()` des Ausnahmeobjekts ausgelesen werden. Sie beschreibt üblicherweise den Fehler.
- **toString()**: Das Ergebnis von `toString()` einer Ausnahme enthält ebenfalls die "Message", falls vorhanden.
- **printStackTrace()**: Diese Methode gibt den **Stack-Trace** der Ausnahme auf der Konsole aus, genau wie bei einem Programmabbruch, auch wenn die Ausnahme abgefangen wurde. Dies ist für Debugging-Zwecke sehr nützlich.
- **Verkettung von Ausnahmen (Cause):** In Zeile 13 von Listing 4.4 wird im `catch`-Block eine neue Ausnahme geworfen, wobei die ursprüngliche Ausnahme als Argument im Konstruktor der neuen Ausnahme übergeben wird (`new RuntimeException(e)`). Dies bewirkt, dass die neue Ausnahme die ursprüngliche Ausnahme als "Cause" (Ursache) beinhaltet. Der Stack-Trace der neuen Ausnahme enthält dann auch Informationen über die ursprüngliche Ursache.
- Man kann eine gerade abgefangene Ausnahme auch direkt mit `throw e;` erneut werfen, um sie weiterzuleiten.

Aufräumarbeiten mit `finally`-Block

Um sicherzustellen, dass bestimmte Anweisungen immer ausgeführt werden – unabhängig davon, ob eine Ausnahme auftritt oder abgefangen wird – verwendet man einen **finally-Block**.

- Ein `finally`-Block folgt auf den `try`-Block und eventuelle `catch`-Blöcke: `try {...} catch {...} finally {...}.`
- Die Anweisungen im `finally`-Block werden **immer ausgeführt**:
 - Nach dem `try`-Block, wenn keine Ausnahme auftrat.
 - Nach dem `catch`-Block, wenn eine Ausnahme abgefangen wurde.

- **Vor der Propagation** einer nicht abgefangenen Ausnahme.
- Dies ist besonders nützlich, um Ressourcen (z.B. geöffnete Dateien, Datenbankverbindungen) zu schließen oder den Programmzustand aufzuräumen.

11. IO über Streams, Validierung von Eingabedaten, DbC

4.1.2 Ein- und Ausgabe über Streams

In Java basiert die **Ein- und Ausgabe (I/O)** auf **Streams**. Stell dir einen Stream wie eine Einbahnstraße für Daten vor: Informationen fließen immer nur in eine Richtung, entweder zum **Lesen** oder zum **Schreiben**. Diese Datenströme werden meist vom Betriebssystem (OS) verwaltet. Java unterstützt dich dabei, die OS-Operationen aufzurufen und bietet zusätzliche Funktionen, um die Nutzung dieser Ressourcen zu vereinfachen.

Unterscheidungsmerkmale von Streams

Arten von I/O-Streams in Java

ungepuffert: direkt mit Ziel oder Quelle verbunden, z.B. Betriebssystem

Änderungen sofort sichtbar,
aber Effizienz leidet unter vielen Interaktionen

gepuffert: zwischengeschalteter Puffer reduziert Zahl an Interaktionen

meist bessere Effizienz,
aber Änderungen werden mit Verzögerung sichtbar (Inkonsistenzen)

für **rohe Daten:** Bytes direkt vom/zum Betriebssystem übertragen

ideal für Daten, die keine Texte darstellen (z.B. Byte-Code)

für **Zeichen:** automatisch zwischen interner und externer Darstellung umgewandelt

intern immer UTF16,
externes Format wie Betriebssystemeinstellungen (z.B. UTF8, Latin1, ASCII)
oder beim Öffnen des Streams festgelegt

In Java gibt es viele verschiedene Arten von Streams. Hier sind die wichtigsten Unterscheidungsmerkmale:

Gepuffert vs. Ungepuffert

- **Ungepufferte Streams:** Wenn du auf einen ungepufferten Stream zugreifst, gehen die Daten **direkt** an das Betriebssystem.
 - Das OS kümmert sich hier selbst um die **Synchronisation**, falls mehrere Zugriffe gleichzeitig erfolgen.
 - Diese Streams bieten meist nur **minimale Funktionalität**.
- **Gepufferte Streams:** Diese Streams nutzen **Java-interne Puffer** (Zwischenspeicher) von begrenzter Größe.
 - Zugriffe modifizieren zunächst diesen Puffer. Nur wenn die benötigten Daten nicht im Puffer sind oder kein Platz im Puffer ist, werden "teure" Zugriffe auf das Betriebssystem durchgeführt.
 - Das macht gepufferte Streams **meist effizienter** als ungepufferte, da seltener mit dem Betriebssystem kommuniziert werden muss.
 - Sie bieten einen **größeren Funktionsumfang**.
 - Achtung: Bei gepufferten Streams können **Synchronisationsprobleme** auftreten, wenn mehrere Streams dieselbe Datei bearbeiten.

Zeichen vs. Bytes (Rohe Daten)

- **Rohe Daten (Bytes)**: Das sind einfach die Daten, wie sie im Betriebssystem vorliegen – ohne ein bestimmtes Darstellungsformat.
 - In Java arbeiten Untertypen von `InputStream` und `OutputStream` mit diesen rohen Daten. Beispiele sind `FileInputStream` für die ungepufferte Eingabe von Dateien oder `BufferedInputStream` für gepufferte Eingabe von rohen Daten.
- **Zeichen-Streams**: Hier werden die vom Betriebssystem empfangenen rohen Daten automatisch in **Zeichen** (im UTF-16-Format) umgewandelt. Bevor Zeichen an das Betriebssystem übergeben werden, werden sie wiederum in rohe Daten zurückverwandelt.
 - Bei der Erzeugung eines Zeichen-Streams kannst du angeben, wie die rohen Daten interpretiert werden sollen, zum Beispiel als `UTF-8`, `Latin1` oder `ASCII`. Diese Angabe der **Kodierung** ist entscheidend für die korrekte Umwandlung.
 - In Java arbeiten Unterklassen von `Reader` und `Writer` mit Zeichen. Beispiele sind `InputStreamReader` (kann die Kodierung setzen) und `FileReader` (für ungepuffertes Lesen von Zeichen aus Dateien) oder `BufferedReader` für gepuffertes Lesen von Zeichen.

Stream-Klassen für Bytes und Zeichen

Untertypen von `InputStream` und `OutputStream` für rohe Daten
 z.B. `FileInputStream` für ungepufferte rohe Daten von Datei
 z.B. `BufferedInputStream` fügt Puffer als Wrapper zu `InputStream`

Untertypen von `Reader` und `Writer` für Zeichen
 z.B. `InputStreamReader` konvertiert `InputStream` zu `Reader`
 z.B. `FileReader` für Zeichen von Datei
 z.B. `BufferedReader` fügt Puffer als Wrapper zu `Reader`

Um das Ganze etwas konkreter zu machen, hier eine Aufteilung der Stream-Klassen nach dem Typ der Daten, die sie verarbeiten:

Rohe Daten (Bytes)

- `InputStream` und `OutputStream`: Dies sind die abstrakten Basisklassen für alle Byte-Streams.
 - `FileInputStream`: Ermöglicht die **ungepufferte** Eingabe von rohen Daten aus einer Datei. Stell dir vor, du liest Byte für Byte direkt aus der Datei.
 - `BufferedInputStream`: Ermöglicht die **gepufferte** Eingabe von rohen Daten. Dieser Stream kann Daten von beliebigen Quellen (nicht nur Dateien) puffern, was den Lesevorgang effizienter macht.

Zeichen

- `Reader` und `Writer`: Dies sind die abstrakten Basisklassen für alle Zeichen-Streams.
 - `InputStreamReader`: Liest **ungepufferte** Zeichen. Das Besondere hier ist, dass du beim Erstellen dieses Streams die **Kodierung** angeben kannst (z.B. UTF-8), um zu definieren, wie die Bytes in Zeichen umgewandelt werden sollen.
 - `FileReader`: Liest ebenfalls **ungepufferte** Zeichen, aber direkt aus einer Datei. Hier wählst du einfach die Datei aus.

- **BufferedReader** : Wird für das **gepufferte** Lesen von Zeichen verwendet. Wenn du zeilenweise lesen möchtest (z.B. mit `readLine()`), ist dieser Stream eine gute Wahl, da er dafür optimierte Methoden bietet.

Verwendung von Streams & Fehlerbehandlung

öffnen — verwenden — schließen
 überprüfte Ausnahme `IOException` abfangen

Scanner fängt `IOException` des Streams ab
 z.B. jene von `System.in` (Typ ist `InputStream`)

Methoden in `PrintStream` (von `OutputStream` abgeleitet) fangen `IOException` ab:
`print`, `println` und `printf`
 bei `IOException` Flag gesetzt, über `checkError` auslesbar, meist ignoriert
 z.B. verwendet von `System.out` und `System.err`

Kommandozeilenargumente ohne Streams lesbar (Inhalt von `args`)

Streams werden in Java immer nach einem festen Muster verwendet:

1. **Öffnen**: Ein Stream wird geöffnet, sobald du ein Stream-Objekt erzeugst (also seinen Konstruktor aufrufst).
2. **Lesen/Schreiben**: Danach kannst du über verschiedene Methoden Daten vom Stream lesen oder darauf schreiben.
3. **Schließen**: **Ganz wichtig**: Nach der Benutzung musst du den Stream **unbedingt** mit der Methode `close()` schließen. Das gibt die vom Betriebssystem gebundenen Ressourcen wieder frei.

Umgang mit Ausnahmen (`IOException`)

Wenn beim Öffnen, Lesen, Schreiben oder Schließen von Streams Probleme auftreten, wird eine **überprüfte Ausnahme** vom Typ `IOException` geworfen. Die Notwendigkeit, diese Ausnahmen zu behandeln und Streams immer zu schließen, macht den Umgang mit ihnen auf den ersten Blick etwas komplex. Die grundlegende Vorgehensweise ist aber immer die gleiche, wie im folgenden Codebeispiel illustriert:

Listing 4.5 Einlesen einer Datei und Ausgeben derselben mit Zeilenummern

```

1 import java.io.*;
2 public class Numbered {
3     private static final String errMsg
4         = "Usage: java Numbered <in> <out>";
5
6     public static void main(String[] args) {
7         if (args.length != 2) {
8             System.err.println(errMsg);
9             return;
10    }
11    try {
12        BufferedReader in = null;
13        BufferedWriter out = null;
14        try {
15            Reader r = new FileReader(args[0]);
16            Writer w = new FileWriter(args[1]);
17            in = new BufferedReader(r);
18            out = new BufferedWriter(w);
19            String line;
20            for (int i = 1;
21                 (line = in.readLine()) != null;
22                 i++) {
23                out.write(String.format(
24                                "%6d: %s\n", i, line));
25            }
26        } finally {
27            if (in != null) { in.close(); }
28            if (out != null) { out.close(); }
29        }
30    } catch (IOException e) {
31        System.err.println("I/O Error: "
32                           + e.getMessage());
33    }
34}
35}

```

Warum `finally`-Blöcke und verschachtelte `try`-Blöcke?

Um sicherzustellen, dass Streams auch dann geschlossen werden, wenn während des Lesens oder Schreibens eine Ausnahme auftritt, bietet sich die Verwendung eines **`finally`-Blocks** an. Dieser Block wird immer ausgeführt, egal ob eine Ausnahme geworfen wurde oder nicht.

Das gezeigte Listing 4.5 verwendet eine spezielle, aber bewährte Struktur mit **zwei ineinander geschachtelten `try`-Blöcken**. Der Grund dafür ist:

- **Sicherstellung des Schließens bei Fehlern:** Ein äußerer `try-catch`-Block fängt die `IOException` ab, die während des gesamten I/O-Prozesses auftreten kann.
- **Schließen auch bei Ausnahmen im `close()`:** Es ist möglich, dass beim Aufruf von `close()` selbst eine `IOException` geworfen wird. Die verschachtelte Struktur hilft, auch diesen Fall abzudecken, indem der `finally`-Block den `close()`-Aufruf absichert.
- **Variablen-Deklaration und -Initialisierung:**

- `BufferedReader in = null;` und `BufferedWriter out = null;` werden **vor dem inneren `try`-Block deklariert und mit `null` initialisiert.**
- **Warum?** Nur existierende (also erfolgreich initialisierte) Streams können geschlossen werden. Da bereits beim Erzeugen der Stream-Objekte eine Ausnahme auftreten könnte, müssen wir im `finally`-Block überprüfen, ob die Variablen `in` und `out` tatsächlich auf Stream-Objekte verweisen und nicht noch `null` sind (`if (in != null)`).
- Gemäß den **Scoping-Regeln** (Sichtbarkeitsregeln für Variablen) von Java dürfen im `finally`-Block (oder in einem `catch`-Block) keine Variablen verwendet werden, die *innerhalb* des zugehörigen `try`-Blocks deklariert wurden. Durch die Deklaration **vor** dem inneren `try`-Block sind `in` und `out` auch im `finally`-Block sichtbar und zugänglich.

Analyse von Listing 4.5: Aufbau und Funktionalität

`Listing 4.5` demonstriert den typischen Umgang mit Streams in Java und verwendet dabei eine Kombination aus verschiedenen Stream-Typen, um eine Datei zeilenweise zu lesen und mit Zeilennummern versehen in eine andere Datei zu schreiben.

- **Verkettung von Streams:**
 - Das Beispiel nutzt **gepufferte Zeichen-Streams** (`BufferedReader in`, `BufferedWriter out`) für die eigentliche Lese- und Schreiboperation.
 - Diese gepufferten Streams werden aus **ungepufferten Zeichen-Streams** (`FileReader r`, `FileWriter w`) erzeugt.
 - Die Konstruktoren von `FileReader` und `FileWriter` akzeptieren direkt die Dateinamen (als `String`) der Dateien, die geöffnet werden sollen.
 - **Wichtiger Hinweis:** Beim Schließen der gepufferten Streams (`in` bzw. `out`) werden die darunterliegenden, un gepufferten Streams (`r` bzw. `w`) automatisch mitgeschlossen. Du musst dich also nicht darum kümmern, jeden einzelnen Stream explizit zu schließen.
- **Kodierung der Daten:**
 - Wenn du `FileReader` oder `FileWriter` direkt verwendest, wird die **Standard-Kodierung des Systems** angenommen. Diese muss nicht explizit angegeben werden.
 - **Wenn eine andere Kodierung benötigt wird** (z.B. UTF-8, wenn die Standard-Kodierung Latin1 ist):
 1. Zuerst erzeugst du einen **ungepufferten Stream für rohe Daten** (Bytes), z.B. `InputStream` oder `OutputStream`.
 2. Daraus erzeugst du einen **ungepufferten Zeichen-Stream** (`InputStreamReader` oder `OutputStreamWriter`) und gibst dabei die gewünschte **Kodierung** an. Dies ist der entscheidende Schritt, um die Byte-Daten korrekt in Zeichen zu interpretieren oder umgekehrt.
 3. Erst dann kannst du darauf aufbauend den **gepufferten Zeichen-Stream** (z.B. `BufferedReader`) erzeugen.
- **Vorteile gepufferter Streams:**
 - Ein wesentlicher Grund für die Verwendung gepufferter Streams ist die Verfügbarkeit **praktischer Methoden**, die in ungepufferten Streams fehlen oder weniger effizient wären. Ein gutes Beispiel ist die Methode `readLine()`, die das zeilenweise Lesen erheblich vereinfacht.

Formatierung von Ausgaben mit `String.format()`

Zur Erzeugung der auszugebenden Zeilen, insbesondere um die Zeilenummern sauber zu formatieren, kommt die statische Methode `String.format()` zum Einsatz.

- **Funktionsweise:** `String.format()` nimmt eine **Steuer-Zeichenkette** (oder Format-String) als ersten Parameter entgegen, gefolgt von beliebig vielen weiteren Parametern, deren Werte in die Steuer-Zeichenkette eingefügt werden sollen.
- **Ergebnis:** Das Ergebnis ist eine neue Zeichenkette, bei der die im Format-String enthaltenen **Steuerzeichenfolgen** durch die Werte der zusätzlichen Parameter ersetzt wurden – und zwar in der angegebenen Reihenfolge.
- **Beispiele für Steuerzeichenfolgen:**
 - `%6d` : Steht für eine **ganze Zahl** (`d` für **Dezimalzahl**), die als **sechsstellige Zahl** dargestellt wird. Falls die Zahl weniger als sechs Ziffern hat, wird sie mit Leerzeichen aufgefüllt, um die Gesamtbreite von sechs Zeichen zu erreichen.
 - `%s` : Steht für eine **Zeichenkette** (`s` für **String**) beliebiger Länge.
- **Konkretes Beispiel:**
 - `String.format("%6d: %s\n", 512, "X")`
 - Dies würde die Zeichenkette `" 512: X\n"` zurückgeben. Beachte die Leerzeichen vor der `512`, die durch die `6` in `%6d` entstehen. Das `\n` am Ende fügt einen Zeilenumbruch hinzu.

Die Standard-Streams: `System.in`, `System.out`, `System.err`

Du kennst `System.in` und `System.out` wahrscheinlich schon aus vielen deiner Java-Beispiele. Oft fällt dabei auf, dass wir bei deren Verwendung keine `IOException` abfangen mussten. Das liegt an der spezifischen Implementierung dieser Streams:

- **`System.in` (Typ: `InputStream`):**
 - Dies ist ein **ungepufferter Stream für rohe Daten (Bytes)**.
 - Obwohl `InputStream` grundsätzlich überprüfte Ausnahmen vom Typ `IOException` werfen kann, fangen die Methoden des `Scanner`-Objekts (das wir oft für die Eingabe über `System.in` nutzen) diese Ausnahmen intern ab. Deswegen siehst du sie in deinem Code nicht direkt.
- **`System.out` (Typ: `PrintStream`, Untertyp von `OutputStream`):**
 - Auch `System.out` ist ein **ungepufferter Stream für rohe Daten**.
 - Der besondere Aspekt hier ist, dass die Methoden von `PrintStream`, wie `print()`, `println()` und `printf()`, **keine Ausnahmen werfen**.
 - `printf()` : Diese Methode, die von erfahrenen Entwicklern oft bevorzugt wird, ruft im Wesentlichen `print(String.format(...))` auf. Sie ermöglicht eine formatierte Ausgabe ähnlich der `format()`-Methode von `String`.
 - **Fehlerbehandlung ohne Ausnahmen:** Falls eine Ausgabe wegen eines Problems nicht erfolgen kann, wird intern ein **Flag** gesetzt, aber die Methode terminiert normal. Der Zustand dieses Flags kann über die Methode `checkError()` abgefragt werden. Wenn du diesen `checkError()`-Aufruf weglässt, bleiben auftretende Probleme bei der Ausgabe unbemerkt.
- **`System.err` (Typ: `PrintStream`):**
 - `System.err` ist die **Standard-Fehlerausgabe** und hat denselben Typ wie `System.out` (`PrintStream`).
 - Der Hauptzweck von `System.err` ist es, Fehlermeldungen **klar vom normalen Programm-Output zu unterscheiden**. Auch wenn sie in einer IDE oft im selben Konsolenfenster angezeigt

werden, werden sie dort häufig farblich hervorgehoben oder in einem separaten Bereich dargestellt.

Trennung von Standard-Output und Fehlerausgabe

Oft sind die Standard-Streams so konfiguriert, dass Fehlermeldungen (geschrieben nach `System.err`) und normaler Output (geschrieben nach `System.out`) zwar im selben Fenster angezeigt werden, aber dennoch visuell unterscheidbar sind. In vielen Integrated Development Environments (IDEs) erkennst du das beispielsweise daran, dass Fehlermeldungen in einer anderen Farbe dargestellt werden.

Eingabe ohne Streams: Kommandozeilenargumente

Eine alternative Form der Eingabe, die ganz ohne Streams auskommt, sind **Kommandozeilenargumente**. Diese werden direkt beim Aufruf des Java-Interpreters in der Kommandozeile an dein Programm übergeben.

- **Beispiel:** Wenn du `java Numbered x y` in der Kommandozeile eingibst, werden `x` und `y` als separate Zeichenketten (Strings) an die `main`-Methode deines Programms `Numbered` übergeben.
- **Zugriff im Code:** Innerhalb der `main`-Methode stehen dir diese Argumente im `String[] args`-Parameter als Array-Elemente zur Verfügung. `args[0]` wäre in diesem Fall "`x`" und `args[1]` wäre "`y`".
- **Anwendung in Listing 4.5:** Das `Listing 4.5` nutzt diese Kommandozeilenargumente, um Dateinamen zu übergeben. Es interpretiert `args[0]` als den Namen der Eingabedatei (aus der gelesen wird) und `args[1]` als den Namen der Ausgabedatei (in die geschrieben wird). Dabei wird der Inhalt der Ausgabedatei `y` ersetzt, falls sie bereits existiert.

Dateizugriff mit `FileWriter` : Anfügen oder Überschreiben

Der `FileWriter`-Konstruktor bietet eine praktische Überladung, um das Verhalten beim Schreiben in eine bereits existierende Datei zu steuern:

- **`FileWriter(String filename, boolean append)`:**
 - Der erste Parameter ist der **Dateiname**.
 - Der zweite Parameter, ein `boolean`-Wert, bestimmt das Verhalten:
 - Wenn `false` (Standardwert, wenn nicht anders angegeben): Die Datei wird **gelöscht und überschrieben**. Der alte Inhalt geht verloren.
 - Wenn `true`: Das Geschriebene wird **an das Ende der Datei angehängt**. Der bestehende Inhalt bleibt erhalten.

Automatisches Schließen von Ressourcen (`try-with-resources`)

Seit neueren Java-Versionen (Java 7 und höher) gibt es eine elegantere Möglichkeit, Stream-Ressourcen automatisch zu schließen, ohne den manuellen `finally`-Block: das **`try-with-resources`-Statement**.

- **Voraussetzung:** Alle Klassen für Streams implementieren das Interface `AutoCloseable`. Dieses Interface enthält lediglich die Methode `close()`.
- **Funktionsweise:** Wenn du Ressourcen, die `AutoCloseable` implementieren, direkt im Kopf des `try`-Blocks deklarierst und initialisierst, sorgt Java automatisch dafür, dass diese Ressourcen am Ende des `try`-Blocks geschlossen werden. Und das geschieht **immer**, auch wenn eine Ausnahme auftritt.

- **Vorteile:** Es ist viel **lesbarer** und **reduziert Boilerplate-Code** (wiederkehrenden Code), da kein expliziter `finally`-Block mehr nötig ist.

Listing 4.6:

```
try (BufferedReader in = new BufferedReader(...);
     BufferedWriter out = new BufferedWriter(...))
{
    ...
} catch (IOException e) {
    ...
}
```

- **Ablauf in Listing 4.6:**

- Im `try`-Block werden `BufferedReader in` und `BufferedWriter out` direkt in den Klammern deklariert und initialisiert.
- Unabhängig davon, ob der Code im `try`-Block fehlerfrei durchläuft oder eine `IOException` (oder eine andere Ausnahme) geworfen wird: Am Ende des `try`-Blocks werden die deklarierten Ressourcen automatisch geschlossen.
- Die Reihenfolge des Schließens ist **umgekehrt zur Deklarationsreihenfolge**: Zuerst wird `out` geschlossen, dann `in`. Dies ist wichtig, wenn Streams voneinander abhängen.

4.1.3 Validierung von Eingabedaten

Beim Programmieren sprechen wir von einem **Fehler**, wenn Annahmen, von denen wir ausgehen, dass sie erfüllt sind, es doch nicht sind. Gute Programmierung versucht, nur auf erfüllte Annahmen zu bauen. Manchmal müssen wir jedoch "unsichere" Annahmen treffen, die aber als fast sicher gelten, z.B. dass genügend Speicher vorhanden ist, die Hardware funktioniert oder das Betriebssystem Ressourcen bereitstellt. Wenn eine solche (fast sichere) Annahme verletzt wird, ist es legitim, dass das Programm mit einer Fehlermeldung endet.

Ganz anders sieht es bei Eingabedaten aus!

Welche Daten wo zu validieren sind

alle Daten, die von außen kommen, sind vor erster Verwendung zu validieren
z.B. über Tastatur oder anderes Eingabegerät eingegebene Daten,
aus Datei oder Datenbank gelesene Daten,
über Netzwerk und sonstige Kommunikationskanäle eingelesene Daten

Validierung möglichst direkt nach Eingabe
meist einfachere Möglichkeit zur Reaktion auf invalide Daten,
Validierung wird nicht so leicht vergessen

nicht zu validieren sind Daten, die aus validierten Daten intern berechnet wurden
inkonsistente mehrfache Validierungen vermeiden

Konzept für Validierung in Softwarearchitektur einplanen

- **Keine unsicheren Annahmen bei Eingabedaten:** Wir dürfen niemals unsichere Annahmen über Daten treffen, die von außerhalb unseres Programms kommen.
- **Eingaben sind beliebig:** Externe Daten müssen immer als beliebig angesehen werden. Du musst davon ausgehen, dass der Benutzer unerwartete Eingaben tätigt, die alle deine Vorgaben ignorieren.
- **"Eingabefehler" aus verschiedenen Perspektiven:**
 - **Aus Programmsicht:** Es gibt keine "Eingabefehler", weil jede Eingabe, die das Programm erhält, technisch gesehen erlaubt ist. Das Programm verarbeitet einfach die Daten, die es bekommt.
 - **Aus Anwendungssicht:** Ein Eingabefehler liegt vor, wenn das Programm Daten erhält, die nicht den Erwartungen entsprechen und deshalb unerwartete oder falsche Ergebnisse liefert. Hier liegt es in der Verantwortung des Programmierers, die Eingaben zu validieren.

Datenvälidierung und Fehlerbehandlung

Wie Daten zu validieren sind

klare, leicht überprüfbare **Datenformate** festlegen (Syntax)

Kriterien für die **Plausibilität** von Daten festlegen (Wertebereiche beschränken)

keine zusätzliche Annahme treffen, insbesondere keine unüberprüfbare

daraus eindeutige Kriterien für die Validierung ableiten

genau diese Kriterien überprüfen, nicht mehr und nicht weniger
(strukturiert vorgehen, nicht nur intuitiv)

„Reparaturversuche“ können gefährlich sein

Wenn eingegebene Daten zur Weiterverarbeitung **bestimmte Eigenschaften** benötigen, müssen diese **überprüft (validiert)** werden. Es müssen zudem **eigene Programmpfade** vorgesehen werden, um mit Daten umzugehen, die diese Eigenschaften nicht erfüllen.

Beispiel: Dateizugriffe (Listing 4.5)

- In Listing 4.5 erfolgen Dateizugriffe nur dann, wenn das Programm mit **genau zwei Kommandozeilenargumenten** aufgerufen wurde. Andernfalls wird eine **Fehlermeldung** ausgegeben – dies ist ein **Fehler aus Anwendungssicht**.
--hier Codeausschnitt von Listing 4.5 einfügen--
- **Ohne eigenen Programmtext** ist zudem sichergestellt, dass die beiden Argumente **alle Eigenschaften erfüllen**, die von Dateinamen (mit Pfadangaben) erwartet werden.
- Sind diese Eigenschaften nicht erfüllt, werfen die Konstruktoren von `FileReader` und `FileWriter` **Ausnahmen (Exceptions)**, die abgefangen und weitergemeldet werden.
- Daten können aus `in` ohne Überprüfungen gelesen werden, da alle Daten **unabhängig von ihrer Struktur** weiterverarbeitet werden.

Listing 4.7 Überprüfung von Eingaben, Fehlermeldungen über Ausnahmen

```

1 import java.io.*;
2 import java.util.*;
3 import java.util.regex.Pattern;
4
5 public class StudInfo {
6     private final static Pattern
7         SEP = Pattern.compile(";"),
8         PAT = Pattern.compile("\d{8};\w(\w| )*\d+");
9     private String name;
10    private int points;
11
12    private StudInfo(String n) { name = n; }
13
14    public static void main(String[] args) {
15        try {
16            if (args.length < 2)
17                throw new IOException(...);
18            Map<Long, StudInfo> map = new HashMap<>();
19            Set<String> used = new HashSet<>();
20            for (int i = 0; i < args.length-1; i++) {
21                if (used.contains(args[i]))
22                    throw new IOException(...);
23                used.add(args[i]);
24                if (args[i].equals("-"))
25                    add(map, System.in);
26                else try (InputStream s =
27                            new FileInputStream(args[i]))
28                    { add(map, s); }
29            }
30            if (args[args.length - 1].equals("-"))
31                write(map, System.out);
32            else try (PrintStream p =
33                        new PrintStream(args[args.length-1]))
34                { write(map, p); }
35        } catch (IOException e) {
36            System.err.println(... + e.getMessage());
37        }
38    }
39    private static void add(Map<Long, StudInfo> map,
40                           InputStream in)
41                           throws IOException {
42        Set<Long> used = new HashSet<>();
43        Scanner lines = new Scanner(in);
44        while (lines.hasNextLine()) {
45            String line = lines.nextLine();
46            if (!line.isEmpty()) {
47                if (!PAT.matcher(line).matches())
48                    throw new IOException(...);

```

```

49         Scanner s = new Scanner(line);
50         s.useDelimiter(SEP);
51         long n = s.nextLong();
52         String name = s.next();
53         int points = s.nextInt();
54         if (used.contains(n))
55             throw new IOException(...);
56         used.add(n);
57         StudInfo i = map.getOrDefault(n,
58                                         new StudInfo(name));
59         if (!i.name.equals(name))
60             throw new IOException(...);
61         i.points += points;
62         map.put(n, i);
63     }
64 }
65 }
66 private static void write(Map<Long, StudInfo> map,
67                           PrintStream s) {
68     Long[] ns = new Long[map.size()];
69     map.keySet().toArray(ns);
70     Arrays.sort(ns);
71     for (Long n : ns) {
72         StudInfo i = map.get(n);
73         s.printf("%08d;%s;%d\n", n, i.name, i.points);
74     }
75 }
76 }
```

Überprüfung strukturierter Eingabedaten (Listing 4.7)

Listing 4.7 demonstriert eine Methode zur Überprüfung strukturierter Eingabedaten. Das Programm konsolidiert die Punkte, die Studierende in verschiedenen Beurteilungsschritten (Übungen und Tests) erreicht haben, zu Gesamtpunktzahlen.

Struktur der Eingabedateien:

Jede Eingabedatei enthält pro beurteilter Person eine Zeile mit der Struktur:

<Matrikelnummer>;<Name wie in TISS>;<Punktezahl>

Struktur und Sortierung der Ergebnisdatei:

Die generierte Ergebnisdatei soll dieselbe Struktur aufweisen und nach Matrikelnummern sortiert sein.

Bedingungen für die Erzeugung der Ergebnisdatei in Listing 4.7:

Die Ergebnisdatei wird nur erzeugt, wenn folgende Bedingungen erfüllt sind:

- **Mindestens eine Eingabedatei:** Vor der Ergebnisdatei muss mindestens eine Eingabedatei als Kommandozeilenargument angegeben sein. Dabei steht " - " als Dateiname für die Standardeingabe oder -ausgabe.
- **Eindeutige Eingabedateinamen:** Mehrere angegebene Eingabedateien müssen voneinander verschiedene Namen haben.
- **Eindeutige Matrikelnummern pro Datei:** Alle Zeilen innerhalb einer einzelnen Datei müssen unterschiedliche Matrikelnummern aufweisen.
- **Konsistente Namen bei gleicher Matrikelnummer:** Personen mit der gleichen Matrikelnummer aus unterschiedlichen Eingabedateien müssen denselben Namen haben.

- **Vorgegebene Zeilenstruktur:** Jede Zeile muss die **vorgegebene Struktur** einhalten:

- Matrikelnummern: Bestehen aus **acht Ziffern**.
- Namen: Bestehen aus **ASCII-Buchstaben und Leerzeichen**.
- Erreichte Punkte: Bestehen aus **ein oder mehreren Ziffern**.

Plausibilitätsprüfungen:

Ein großer Teil des Programmtextes ist allein für diese Überprüfungen zuständig. Dieses Vorgehen ist in realen Programmen durchaus üblich. Man spricht hier von **Plausibilitätsprüfungen**, da nicht festgestellt werden kann, ob die Daten **richtig** sind, sondern nur, ob ihre Korrektheit **plausibel** erscheint.

- **Variabilität der Details:** Die überprüften Details können in einem weiten Bereich variieren.
 - Beispiele: Erlauben von Umlauten, Akzenten und bestimmten Sonderzeichen in Namen oder Beschränken der Punkte auf drei Stellen.

Sonderzeichen zulassen oder die Punkte auf drei Stellen beschränken.

Listing 4.7 vereinfacht die Überprüfung mittels *Pattern-Matching*: Der *compilierte reguläre Ausdruck* in PAT beschreibt die gewünschte Struktur einer Zeile – siehe die Spezifikation der Klasse **Pattern**. Hat die Zeile nicht diese Form, wird eine Ausnahme geworfen. Nach nur einer Überprüfung (Programmzeile 47) ist garantiert, dass die Zeile keine andere Form haben kann. So ist es leicht, die Zeile ohne weitere Überprüfungen mittels **Scanner** in die einzelnen Teile mit den gewünschten Typen zu zerlegen, wobei die Teile nicht wie meist durch White-Space voneinander getrennt sind, sondern durch ";", wie in Programmzeile 50 über den compilierten regulären Ausdruck **SEP** angegeben.

In Listing 4.7 verwenden wir überprüfte Ausnahmen um Fehlermeldungen zu sammeln und an zentraler Stelle auszugeben. Das vereinfacht die Programmstruktur. Wir müssen aber nicht mit Ausnahmen arbeiten, sondern können auch direkt Fehlermeldungen nach **System.err** schreiben. Die zentralisierte Stelle funktioniert nur, wenn das Programm nach jeder Art von Fehler auf dieselbe Weise fortgesetzt wird – im Beispiel mit der Beendigung. Wenn für jede Art von Fehlermeldung eine unterschiedliche Programmfortsetzung nötig ist, wäre es besser, auf das Werfen von Ausnahmen zu verzichten.

Oft wird empfohlen, jede Annahme im Programm immer wieder zu überprüfen um Fehler rasch zu erkennen. In dieser Allgemeinheit geht die Empfehlung leider am wesentlichen Punkt vorbei. Wir müssen nur jene Annahmen prüfen, die unbekannte, von außen kommende Daten betreffen. Die Überprüfungen müssen sehr zuverlässig sein, möglichst nah an der Schnittstelle, über die Daten hineinkommen. Danach können wir uns auf die Annahmen verlassen. Beispielsweise wird in Listing 4.7, Zeilen 51–53 nicht überprüft, ob wirklich ein **long**-Wert, eine Zeichen-

ketten und ein int-Wert aus einer Zeile lesbar ist. Wir wissen das ja schon. Unnötige Überprüfungen erhöhen die Komplexität und können dazu führen, dass wir Überprüfungen an Schnittstellen unterlassen weil wir uns fälschlich auf spätere Überprüfungen verlassen.

Manchmal ist nicht klar, was von außen kommt und was nicht. Beispielsweise haben wir in Listing 4.3 überprüft, ob der Parameterwert im erwarteten Bereich liegt. Das ist sinnvoll, wenn die Methode außerhalb unserer Kontrolle aufgerufen wird. Jedoch ist das Werfen einer Ausnahme nur eine Notlösung. Wir wollen schon vor der Programm-ausführung sicherstellen, dass die Methode nur erwartete Parameter bekommt. Dieses Thema behandeln wir im nächsten Abschnitt.

Zentral versus dezentral

zentral: Programmtext für Validierung an nur einer Stelle

entsprechende Methode an jedem Eingang aufgerufen (meist zur Konvertierung),
nicht valide Daten führen zu Ausnahme,
diese Ausnahme am Eingang abgefangen für spezifische Reaktion,
einheitliches Konzept, relativ sicher

dezentral: Validierung an jedem Eingang getrennt

einfach und bedarfsgerecht auf nicht valide Daten reagierbar,
aber Überprüfungen häufig nicht einheitlich, meist unsicher

4.2 Statisches Programmverstehen

Stufen des Programmverstehens

1. Verstehen der Syntax und Semantik von Sprachelementen,
Verstehen des Programmaufbaus durch Zusammensetzen von Sprachelementen
2. Nachvollziehen des Programmablaufs auch für komplexe Programme,
Vorhersehen von Ergebnissen für bestimmte Eingaben
3. Statisches Verstehen beim Programmlesen,
Vorhersehen der Gesamtheit von Ergebnissen für alle Eingaben

Design-by-Contract hilft beim Programmverstehen auf der 3. Stufe

Die **wichtigste Kompetenz beim Programmieren** ist das **Verstehen von Programmen**. Man kann grob drei Arten des Programmverstehens unterscheiden, die auch den typischen Lernfortschritt widerspiegeln:

1. **Verstehen von Syntax und Semantik:** Hier geht es darum, die Bedeutung einzelner Sprachelemente und die Art und Weise, wie Programme aus diesen Elementen aufgebaut sind, zu begreifen. (Bspw. Was macht eine `for`-Schleife? Wie ist eine Klasse aufgebaut?)

2. **Nachvollziehen des Programmablaufs:** Diese Art des Verstehens beinhaltet das Nachvollziehen des Schritt-für-Schritt-Ablaufs auch bei komplexeren Programmen. Ziel ist es, die Ergebnisse für **bestimmte, vorgegebene Eingaben** vorhersagen zu können. (Bspw. Was kommt raus, wenn ich die Zahlen 5 und 7 eingebe?)
3. **Statisches Verstehen beim Programmlesen:** Dies ist die fortgeschrittenste Form. Hierbei wird die **Gesamtheit aller möglichen Ergebnisse** eines Programms vorhergesagt, **ohne Annahme bestimmter Eingaben**. Es geht also um das Verstehen des allgemeinen Verhaltens des Programms.

Bedeutung des statischen Programmverstehens:

- Es ist die **Grundvoraussetzung für eine effiziente und professionelle Softwareentwicklung**.
- Das statische Programmverstehen ist **aufwendig** zu erlernen.
- Auch sehr erfahrene Entwickler können große und komplexe Programme nur durch **klar strukturiertes Vorgehen** und die Anwendung **entsprechender Techniken** statisch verstehen.

Im weiteren Verlauf dieses Abschnitts werden einige dieser Vorgehensweisen und Techniken vorgestellt.

Vertragspartner in Design-by-Contract (DbC)

Vertragspartner: Objekte in der Software, treten als Client und Server auf

Server bietet Leistungen in Form von Methodenausführungen an
Client nimmt Leistungen durch Methodenaufrufe in Anspruch

Details der Leistungen in einem **Software-Vertrag** geregelt

Software-Vertrag ist Bestandteil der Klasse des Servers (hauptsächlich Kommentare)

Design-by-Contract (DbC)

Design-by-Contract ist eine weit verbreitete Vorgehensweise in der Programmentwicklung, die Softwareentwicklung als ein System von **Verträgen** zwischen verschiedenen Programmteilen betrachtet.

- **Objekte als Server und Clients:** Jedes Objekt wird sowohl als **Server** (der Leistungen, sogenannte Services, in Form von Methoden anbietet) als auch als **Client** (der Leistungen von anderen Servern durch Methodenaufrufe in Anspruch nimmt) gesehen.
- **Verträge (contracts):** Alle Leistungen zwischen Servern und Clients werden durch Verträge beschrieben.
 - Ein Vertrag definiert, was sich ein **Client vom Server** und der **Server vom Client** erwarten darf.
- **Vorgehen:**
 1. Zuerst werden die **Verträge festgelegt**.
 2. Danach werden die Klassen auf Basis dieser Verträge **implementiert**.
- **Fehlerbehebung:** Tritt ein Fehler auf, wird anhand der Verträge geklärt, welcher Programmteil eine **Vertragsverletzung** begangen hat und somit die Schuld trägt. Dort sollte der Fehler dann behoben werden.

- **Fehlerprävention:** Durch das Einhalten aller Vertragsdetails können viele Fehler von vornherein ausgeschlossen werden.

Softwarevertrag und Zusicherungen

Verträge folgen einer vorgegebenen Struktur und werden durch eine Menge von **Zusicherungen (Assertions)** spezifiziert. Jede Zusicherung ist eine Bedingung, die zu festgelegten Zeitpunkten erfüllt sein muss. Zusicherungen können formal oder informell (z.B. als Kommentare) dokumentiert sein.

Man unterscheidet folgende Arten von Zusicherungen:

- **Vorbedingung (Precondition):**

Vorbedingung (Precondition):

Bedingung, die vor Methodenausführung erfüllt sein muss, von Client garantiert, z.B.: „Einträge im an Methode übergebenen Array sind aufsteigend sortiert“

- Bezieht sich auf eine **bestimmte Methode (Service)**.
- Muss zu den **Zeitpunkten aller Aufrufe** der Methode erfüllt sein.
- Der **Aufrufer (Client)** muss für die Einhaltung sorgen.
- Der **Server** (genauer: die Implementierung der aufgerufenen Methode) darf sich darauf verlassen.
- Beschreibt meist die **erwarteten Eigenschaften von Methodenparametern** zum Zeitpunkt des Aufrufs.

- **Nachbedingung (Postcondition):**

Nachbedingung (Postcondition):

Bedingung, die nach Methodenausführung erfüllt sein muss, von Server garantiert, z.B.: „Methode gibt null zurück wenn gesuchter Eintrag nicht gefunden“

- Bezieht sich ebenfalls auf eine **bestimmte Methode**.
- Muss **am Ende (bei der Rückkehr aus) der Methode** erfüllt sein.
- Der **Server** (genauer: die Implementierung der aufgerufenen Methode) muss für die Einhaltung sorgen.
- Der **Client (Aufrufer)** darf sich darauf verlassen.
- Beschreibt meist, **was die Methode leistet und welche Eigenschaften der Rückgabewert hat**.

- **Invariante (Invariant):**

Invariante (Invariant):

Bedingung in Bezug auf Objektzustand, muss vor und nach jeder Methodenausführung erfüllt sein, von Server garantiert z.B. „Anzahl der Einträge stets größer oder gleich 0“

- Bezieht sich auf den **Zustand eines Objekts** bzw. die Inhalte von Objektvariablen.
- Muss **bei jedem Aufruf** und **am Ende jeder Ausführung jeder Methode** des Objekts erfüllt sein.
- Der **Server** (genauer: die gesamte Implementierung der Klasse) muss für die Einhaltung sorgen.
- Jeder **Client** darf sich darauf verlassen.
- Vereinfacht gesagt, ist eine Invariante eine Bedingung auf dem Objektzustand, die **immer gelten muss**.

Zusicherungen in Programmen und ihre Verantwortlichkeiten

Rechte und Pflichten

Rechte des Servers: darauf verlassen, dass zu Beginn jeder Methodenausführung alle Vorbedingungen der Methode und Invarianten des Objekts erfüllt

Pflichten des Servers: dafür sorgen, dass am Ende jeder Methodenausführung alle Nachbedingungen der Methode und Invarianten des Objekts erfüllt und vor jedem Methodenaufruf alle Invarianten des Objekts erfüllt

Rechte des Clients: darauf verlassen, dass nach Methodenausführung alle Nachbedingungen der Methode und Invarianten des Objekts erfüllt

Pflichten des Clients: dafür sorgen, dass vor jeder Methodenausführung alle Vorbedingungen der Methode erfüllt

Zusicherungen (Assertions) finden sich in Programmen meist als **Kommentare**.

- **Platzierung von Kommentaren:**
 - **Vor- und Nachbedingungen** werden typischerweise bei den **Methodenköpfen** platziert.
 - **Invarianten** finden sich bei den **Deklarationen von Objektvariablen**.
- **Abgrenzung zu bisheriger Verwendung:** Diese Art von Kommentaren ist nicht neu; sie wurden bereits zur Beschreibung abstrakter Datentypen verwendet. Der wesentliche Unterschied im Kontext von Design-by-Contract ist die **klare Unterscheidung zwischen den Arten von Zusicherungen** und die Sichtweise, dass jeder Kommentar **Bestandteil eines formalen Vertrags** ist.

Verantwortlichkeiten bei Vertragsverletzungen:

Aus dieser Vertragssicht ergibt sich eindeutig, wer für die Einhaltung welcher Zusicherung zuständig ist:

- **Clients** sind für die Einhaltung von **Vorbedingungen** verantwortlich.
- **Server** sind für die Einhaltung von **Nachbedingungen und Invarianten** verantwortlich.

Unterscheidung der Zusicherungsarten:

Um diese Verantwortlichkeiten klar zuzuordnen, ist es wichtig, die unterschiedlichen Arten von Zusicherungen voneinander unterscheiden zu können:

- **Invarianten** beziehen sich auf **Objektvariablen** und deren Konsistenz (den Zustand eines Objekts).
- **Vor- und Nachbedingungen** beziehen sich auf **Methoden**.
 - **Vorbedingungen** sind beim **Methodenaufruf** relevant und beschreiben meist erwartete Eigenschaften von **Parameterwerten**.
 - **Nachbedingungen** sind am Ende der Methodenausführung von Bedeutung und beschreiben, was die Methode getan hat und welche Eigenschaften der Rückgabewert besitzt.

Die Zuordnung von Kommentaren zu den jeweiligen Zusicherungsarten erfolgt anhand ihrer **Inhalte**.

Verantwortung in Design-by-Contract: Client vs. Server

Die Platzierung und Art der Zusicherungen (Assertions) im Code bestimmt, wer für deren Einhaltung verantwortlich ist.

Listing 4.8: Sparbuch mit Zusicherungen – viel Verantwortung beim Client

Listing 4.8 Sparbuch mit Zusicherungen – viel Verantwortung beim Client

```

1  public class DepositAccount {
2      private long deposit; // deposit >= 0           Invariante
3
4      // amount > 0;                                Vorbedingung
5      // deposit incremented by amount            Nachbedingung
6      public void payIn(long amount) {deposit += amount;}
7
8      // amount > 0; amount <= deposit;          Vorbedingung
9      // deposit decremented by amount          Nachbedingung
10     public void payOut(long amount) {deposit -= amount;}
11
12    public long deposit() {return deposit;}
13 }
```

- **Vorbedingung in Zeile 8 (`amount > 0; amount <= deposit;`) für `payOut()`:**
 - Diese Vorbedingung bedeutet, dass **jeder Aufrufer (Client)** der Methode `payOut` sicherstellen muss, dass der einzuzahlende Betrag (`amount`) größer als 0 ist und die Einlage (`deposit`) hoch genug ist, um die Auszahlung zu decken.
 - Der Client ist also dafür verantwortlich, diese Bedingungen **vor dem Aufruf** zu überprüfen (z.B. durch Vergleich des Arguments mit dem Ergebnis von `deposit()`).
 - **Innerhalb von `payOut` ist keine weitere Überprüfung nötig**, da der Server sich darauf verlassen darf, dass der Client die Vorbedingung erfüllt hat.
- **Invariante in Zeile 2 (`deposit >= 0`):** Beschreibt, dass der Kontostand (`deposit`) niemals negativ sein darf.

Listing 4.9: Methode in DepositAccount – viel Verantwortung beim Server

Listing 4.9 Methode in `DepositAccount` – viel Verantwortung beim Server

```

1      // if true: deposit decremented by amount      Nachbed.
2      public boolean payOut(long amount) {
3          if (amount <= 0 || amount > deposit) return false;
4          deposit -= amount;
5          return true;
6      }
```

- In dieser Variante von `payOut` liegt die **Verantwortung ganz beim Server**.
- **Clients können `payOut` ohne vorherige Überprüfung aufrufen**, da **alle notwendigen Überprüfungen innerhalb der Server-Methode selbst stattfinden** (z.B. `if (amount <= 0 || amount > deposit) return false;`).
- **Vorteilhaftigkeit:** Es ist ratsam, Überprüfungen dort anzusiedeln, wo die dafür nötige Information vorhanden ist. Dies ist sehr oft der Server, wodurch die Variante in Listing 4.9 gegenüber Listing 4.8 als vorteilhafter gilt.

Fehlerbehandlung mit Ausnahmen (Exceptions)

Listing 4.10: Werfen einer Ausnahme als Nachbedingung beschrieben

Listing 4.10 Werfen einer Ausnahme als Nachbedingung beschrieben

```

1      // deposit decremented by amount;           Nachbedingung
2      // throws IllegalArgumentException if
3      // amount <= 0 or amount > deposit       Nachbedingung
4      public void payOut(long amount) {
5          if (amount <= 0 || amount > deposit)
6              throw new IllegalArgumentException();
7          deposit -= amount;
8      }

```

- Das Werfen einer Ausnahme ermöglicht es, die **Verantwortung beim Server** zu belassen, ohne Informationen über einen Fehler über den Rückgabewert der Methode an den Aufrufer zurückzugeben zu müssen (wie in Listing 4.9 mit `boolean`).
- Die Ausnahme (`IllegalArgumentException`) wird hier als **Nachbedingung** beschrieben, die bei bestimmten ungültigen Zuständen (z.B. `amount <= 0` oder `amount > deposit`) eintritt.

Listing 4.11: Werfen einer Ausnahme trotz Vorbedingung

Listing 4.11 Werfen einer Ausnahme trotz Vorbedingung

```

1      // amount > 0; amount <= deposit;           Vorbedingung
2      // deposit decremented by amount           Nachbedingung
3      public void payOut(long amount) {
4          if (amount <= 0 || amount > deposit)
5              throw new IllegalArgumentException();
6          deposit -= amount;
7      }

```

- Obwohl hier explizit eine **Vorbedingung** formuliert ist (`amount > 0; amount <= deposit;`), wird dennoch eine Ausnahme geworfen, wenn die Bedingungen nicht erfüllt sind.
- Logik:** Laut Vertrag darf die Vorbedingung in der Methode nicht verletzt werden. Was die Methode bei einer verletzten Vorbedingung macht, ist daher definitionsgemäß "egal".
- Sicherheitsaspekt:** Durch das zusätzliche Werfen einer Ausnahme wird die **Zuverlässigkeit des Programms weiter erhöht**, da so ein Fehler aktiv gemeldet wird, selbst wenn der Client seine vertragliche Pflicht missachtet hat.

Listing 4.12: Vorbedingung und Nachbedingung – Verantwortung nicht klar

Listing 4.12 Vorbedingung und Nachbedingung – Verantwortung nicht klar

```

1      // amount > 0; amount <= deposit;           Vorbedingung
2      // otherwise IllegalArgumentException;    Nachbedingung
3      // deposit decremented by amount         Nachbedingung
4      public void payOut(long amount) {
5          if (amount <= 0 || amount > deposit)
6              throw new IllegalArgumentException();
7          deposit -= amount;
8      }

```

- Verträge mit unklaren Verantwortlichkeiten sollten vermieden werden.
- In diesem Beispiel wird versucht, die Bedingung für das Werfen der `IllegalArgumentException` sowohl als Vorbedingung als auch als Nachbedingung zu formulieren. Dies führt zu einer Verwischung der Verantwortlichkeiten zwischen Client und Server und macht den Vertrag unklar.

Allgemeine Erwartungen an Programmtexte

Allgemeine Erwartungen

auch ohne explizite Zusicherungen muss gelten, was allgemein erwartet wird

- z.B. Methode muss machen, was Methodename verspricht
- z.B. Objekt darf nicht verändert werden, außer wo dies erwartet wird
- z.B. Zusicherung nach allgemeinem Verständnis interpretiert, nicht nach Wortlaut

Verständnis für allgemeine Erwartungen erfordert Einfühlungsvermögen, erlernbar durch gemeinsames Programmieren

Neben den explizit in Verträgen beschriebenen Bedingungen gibt es auch **allgemeine, nirgends genau festgelegte, aber übliche Erwartungen** an Programmtexte:

- **Beispiel:** Eine Methode namens `insert` wird erwartet, etwas einzufügen und nicht etwa Daten zu löschen.
- **Objektzustandsänderungen:** Generell wird erwartet, dass eine Methode Objekte nicht verändert, es sei denn, der Methodename legt entsprechende Änderungen nahe (z.B. `set...`, `add...`) oder Zusicherungen (Postconditions) beschreiben explizit solche Änderungen.
- **Priorität der Erwartungen:** Allgemeine Erwartungen müssen immer erfüllt sein, es sei denn, Verträge schränken diese Erwartungen explizit ein.

Listing 4.13: Implementierungen entsprechen nicht den Erwartungen

Listing 4.13 Implementierungen entsprechen nicht den Erwartungen

```

1  // if true: deposit incremented by amount      Nachbed.
2  public boolean payIn(long amount) { return false; }
3
4  // if true: deposit decremented by amount      Nachbed.
5  public boolean payOut(long amount) {
6      deposit -= amount;
7      return false;
8  }
```

- Die Implementierungen in Listing 4.13 sind **hinsichtlich Design-by-Contract fehlerhaft**, da sie den allgemeinen Erwartungen widersprechen, obwohl die Nachbedingungen formal erfüllt sein könnten (z.B. `payIn` gibt `false` zurück, was die Nachbedingung erfüllt, aber die Methode nicht "einzhaltet").

Zusicherungen und Untertypen (Liskovsches Substitutionsprinzip)

Vorbedingung (auf einer Methode in Ober- und Untertyp):

darf im Untertyp nicht stärker, höchstens schwächer sein

Nachbedingung (auf einer Methode in Ober- und Untertyp):

darf im Untertyp nicht schwächer, höchstens stärker sein

Invariante (auf Objektzustand von Ober- und Untertyp):

darf im Untertyp nicht schwächer, höchstens stärker sein

Beim Bilden von Untertypen (Vererbung) müssen die Zusicherungen des Design-by-Contract-Ansatzes bestimmte Bedingungen erfüllen, um die Korrektheit des Programms zu gewährleisten. Dies ist eng mit dem **Liskovschen Substitutionsprinzip** verbunden, welches besagt, dass Objekte eines Obertyps durch Objekte eines Untertyps ersetzt werden können, ohne die Korrektheit des Programms zu beeinträchtigen.

Folgende Bedingungen müssen **unbedingt eingehalten** werden:

- **Vorbedingungen auf Methoden in Untertypen:**
 - Dürfen nur **gleiche oder schwächere Einschränkungen** ausdrücken als auf den entsprechenden Methoden in Obertypen.
 - Dies entspricht einer **Verknüpfung mit ODER**.
 - **Beispiel:** Ist $x > 5$ eine Vorbedingung im Obertyp, kann die Vorbedingung im Untertyp $x > 3$ sein. $x > 5 \parallel x > 3$ ergibt $x > 3$. (Der Untertyp akzeptiert also eine größere Bandbreite an Eingaben.)
- **Nachbedingungen auf Methoden in Untertypen:**
 - Dürfen nur **gleiche oder stärkere Einschränkungen** ausdrücken als auf den entsprechenden Methoden in Obertypen.
 - Dies entspricht einer **Verknüpfung mit UND**.
 - **Beispiel:** Ist $x > 3$ eine Nachbedingung im Obertyp, kann die Nachbedingung im Untertyp $x > 5$ sein. $x > 3 \&\& x > 5$ ergibt $x > 5$. (Der Untertyp garantiert also stärkere Eigenschaften des Ergebnisses.)
- **Invarianten in Untertypen:**
 - Dürfen nur **gleiche oder stärkere Einschränkungen** ausdrücken als in Obertypen.
 - Dies entspricht ebenfalls einer **Verknüpfung mit UND**.
 - **Beispiel:** Ist $x > 3$ eine Invariante im Obertyp, kann die Invariante im Untertyp $x > 5$ sein. $x > 3 \&\& x > 5$ ergibt $x > 5$. (Der Untertyp hält also strengere Objektzustandsbedingungen ein.)

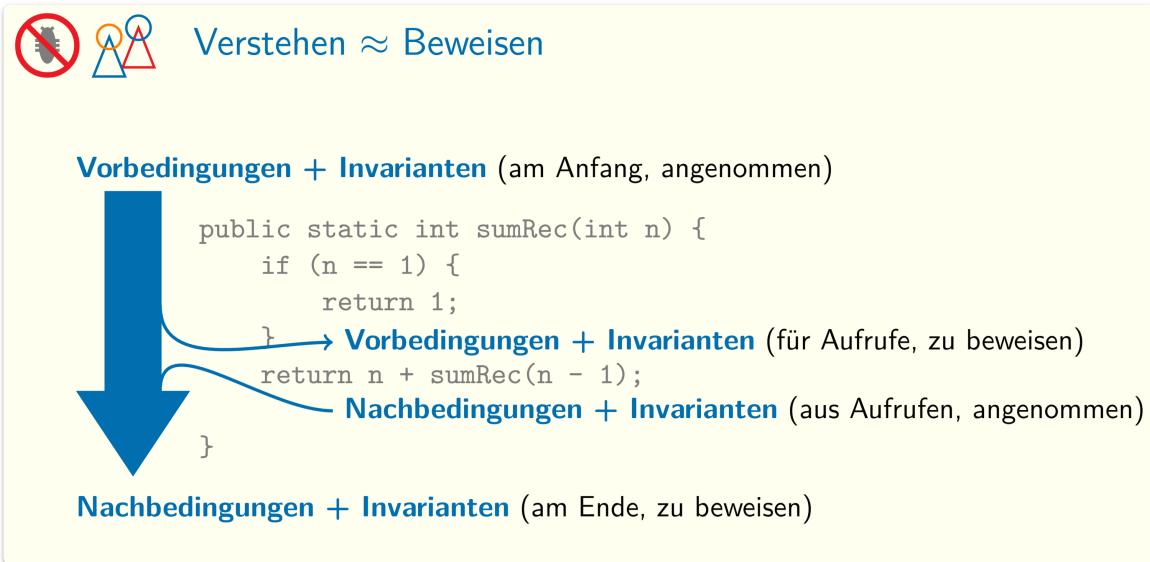
Wichtige Ausnahme bei Invarianten:

- Diese Regel (stärkere Invarianten in Untertypen) gilt **nur**, wenn die Variable, auf die sich die Invariante bezieht (z.B. x), **nicht von außerhalb des Objekts geschrieben** werden kann.

- Dies ist ein weiterer Grund, warum **Objektvariablen generell private sein sollten**, um den direkten Zugriff von außen zu verhindern und die Kontrolle über den Zustand zu behalten.
- **Ausnahme:** Kann eine Objektvariable von außerhalb geschrieben werden, muss eine Invariante, die sich auf diese Variable bezieht, in **Unter- und Obertyp genau gleich** sein. Andernfalls könnte der Obertyp nicht korrekt durch den Untertyp substituiert werden, da der Untertyp möglicherweise nicht die Invariante des Obertyps aufrechterhält, wenn die Variable extern verändert wird.

12. Statisches Programm-Verstehen, Code-Review, Programmdokumentation

Statisches Verstehen von Programmteilen



Das **statische Verstehen** eines Programmteils kann mit dem **formalen Beweisen der Korrektheit** dieses Teils verglichen werden. Das Ziel ist es, einen **Korrektheitsbeweis** zu finden.

- Findet man einen Beweis, wurde der Programmteil verstanden.
- Findet man keinen Beweis, bedeutet das entweder, dass der Programmteil nicht korrekt ist, oder dass man ihn noch nicht verstanden hat.

Die zentrale Frage ist: **Welche konkreten Eigenschaften müssen bewiesen werden**, damit ein Programmteil als korrekt gilt?

Design by Contract (DbC) und Korrektheitsbeweise

Für **Methoden** liefert **Design by Contract (DbC)** die Antwort:

- Man muss beweisen, dass **nach der Methodenausführung** die **Nachbedingungen** der Methode und die **Invarianten** der Klasse gelten.
- Dabei wird angenommen, dass zu **Beginn der Ausführung** die **Vorbedingungen** und **Invarianten** erfüllt sind.

Allerdings sind Zusicherungen (Vor-, Nachbedingungen und Invarianten) oft nicht präzise und vollständig genug formuliert, um einen formalen Beweis durchzuführen.

Stattdessen wird ein **grobes Beweisschema** angestrebt, das mit diesen Ungenauigkeiten umgehen kann. Erfahrene Entwickler können solche Beweisschemata oft schon beim Lesen des Programmcodes nebenbei entwickeln.

Hilfsmittel Hoare-Tripel

$\{V\}S\{N\}$ wobei V Vorbedingung, S Anweisung, N Nachbedingung

wenn V vor Ausführung von S erfüllt ist, dann ist N nach Ausführung von S erfüllt

z.B.: $\{x = 1 \wedge y = 2\}x=y;\{x = 2 \wedge y = 2\}$

wenn zwei von drei (V, S, N) bekannt, kann passendes Drittes gefunden werden

z.B.: aus $\{x = 1\}S\{x = 2\}$ folgt $S = x++$; oder $S = x=2*x-2$; oder ...

Hintereinanderausführung: $\{V\}S_1\{X\}, \{X\}S_2\{N\} \Rightarrow \{V\}S_1;S_2\{N\}$

Verzweigung: $\{V \wedge B\}S_1\{N\}, \{V \wedge \neg B\}S_2\{N\} \Rightarrow \{V\}\text{if}(B) S_1 \text{ else } S_2\{N\}$

Vor- und Nachbedingung in Hoare-Tripel \neq Vor- und Nachbedingung auf Methode

Ein Hoare-Tripel ist eine Kombination aus:

- **V (Vorbedingung):** Eine Bedingung, die **vor** der Ausführung des Programmsegments S erfüllt sein muss.
- **S (Programmsegment):** Ein Ausschnitt aus dem Programmcode.
- **N (Nachbedingung):** Eine Bedingung, die **nach** der Ausführung des Programmsegments S erfüllt ist.

Anwendungsfälle des Hoare-Kalküls

1. Elementare Anweisungen:

- Kennt man die Semantik einer elementaren Anweisung S , kann die Nachbedingung N aus der Vorbedingung V und S abgeleitet werden.
- **Beispiel:**

$$x = 1; y = 2 \quad x = y; \quad x = 2; y = 2$$

2. Methodenaufrufe:

- Die Auswirkungen eines Methodenaufrufs können aus den **Zusicherungen der Methode** (gemäß Design by Contract) abgeleitet werden.
- Die Vorbedingung V des Hoare-Tripels muss die **Vorbedingungen** der aufgerufenen Methode implizieren (d.h. aus V müssen die Vorbedingungen der Methode folgen).
- Die Nachbedingung N des Hoare-Tripels ergibt sich aus den **Nachbedingungen** der aufgerufenen Methode.

Regeln für Programmstrukturen

Der Hoare-Kalkül bietet Regeln zur Darstellung von Verknüpfungen von Programmsegmenten:

• Hintereinanderausführung:

- Wenn

$$VS1X$$

und

$$XS2N$$

gelten, dann gilt für die Hintereinanderausführung von $S1$ und $S2$:

$VS1S2N$

- Hierbei ist X eine Zwischenbedingung, die nach $S1$ und vor $S2$ gilt.
- **Bedingte Ausführung:** Über einfache Regeln darstellbar (genauere Regeln sind hier nicht ausgeführt).
- **Schleifen:**
 - Schleifen sind komplexer, aber handhabbar (weitere Details folgen eventuell später).

Solche Verknüpfungen von Hoare-Tripeln ermöglichen es, Korrektheitsbeweise über das gesamte Programm zu führen.

„Korrektheitsbeweis“

Vor- und Nachbedingungen entsprechend Hoare-Tripeln zwischen Programmzeilen

```
// sum of all integers from 1 to n; Nachbedingung, ist zu zeigen
// n >= 1; Vorbedingung, wird angenommen
public static int sumRec(int n) {
    {n ≥ 1}
    if (n == 1) {
        {n ≥ 1 ∧ n = 1} entspricht {n = 1}
        return 1;
        {n = 1 ∧ Ergebnis = 1 = ∑i=1n i}
    }
    {n ≥ 1 ∧ n ≠ 1} entspricht {n > 1}, impliziert {n - 1 ≥ 1} (Vorbed. für sumRec)
    return n + sumRec(n - 1); Vorbed. ist zu zeigen, Nachbed. wird angenommen
    {n > 1 ∧ Ergebnis = n + ∑i=1n-1 i = ∑i=1n i} wobei ∑i=1n-1 i aus Nachbed. von sumRec
}
```

Die `assert`-Anweisung in Java

assert-Anweisung

```
// sum of all integers from 1 to n;
// n >= 1
public static int sumRec(int n) {
    assert n >= 1 : "n = " + n;
    if (n == 1) {
        assert n == 1;
        return 1;
    }
    assert n > 1;
    return n + sumRec(n - 1);
}
```

Die `assert`-Anweisung in Java dient dazu, eine **Zusicherung** (Assertion) auszudrücken. Sie ähnelt der Funktion einer Vor- oder Nachbedingung in einem **Hoare-Triple**.

Listing 4.14 Binäre Suche, semi-formale Zusicherungen zwischen Anweisungen

```

1 // pre: sorted(a) && (l>h || (0<=l && h<a.length));
2 // post: result r with r>=0 && a[r]==v && l<=r && r<=h;
3 // post: result r== -1 if not(v,a,l,h); (a not modified)
4 static int searchR(int v, int[] a, int l, int h) {
5     assert sorted(a) && (l>h || (0<=l && h<a.length));
6     if (l > h) {
7         assert sorted(a) && l>h && not(v,a,l,h);
8         return -1;
9     }
10    assert sorted(a) && 0<=l && l<=h && h<a.length;
11    int r = (l + h) / 2;
12    assert sorted(a)&&0<=l&&l<=r&&r<=h&&h<a.length;
13    if (a[r] < v) {
14        assert not(v,a,l,r) && sorted(a) &&
15                    (r+1>h || (0<=r+1 && h<a.length));
16        return searchR(v,a,r+1,h); //not(v,a,l,h) if -1
17    } else if (a[r] == v) {
18        assert a[r]==v && l<=r && r<=h;
19        return r;
20    } else {
21        assert not(v,a,r,h) && sorted(a) &&
22                    (l>r-1 || (0<=l && r-1<a.length));
23        return searchR(v,a,l,r-1); //not(v,a,l,h) if -1
24    }
25 }
26
27 static boolean sorted(int[] a, int i) {
28     return i<=0 || (a[i-1]<=a[i] && sorted(a, i-1));
29 }
30 static boolean not(int v, int[] a, int l, int h) {
31     return l>h || (a[l]!=v && not(v,a,l+1,h));
32 }

```

Eigenschaften von `assert`-Anweisungen:

- **Keine Programmauswirkung (meistens):** Ähnlich wie Kommentare haben `assert`-Anweisungen in der Regel **keine Auswirkungen auf die Ausführung des Programms** im produktiven Code. Sie sind primär für die Entwicklungs- und Testphase gedacht.
- **Syntaxprüfung durch Compiler:** Obwohl sie oft zur Laufzeit ignoriert werden können, prüft der Compiler die **Syntax** der `assert`-Anweisungen. Das bedeutet, dass sie syntaktisch korrekt sein müssen.
- **Einschränkungen:** Für Zusicherungen, die nicht der Java-Syntax entsprechen (z.B. wenn sie nach einer `return`-Anweisung platziert werden, wo kein ausführbarer Code mehr erwartet wird), müssen stattdessen **Kommentare** verwendet werden.

Schleifeninvarianten und Terminationsbeweis

Schleifeninvariante

Schleife: $\{I \wedge B\} S\{I\} \Rightarrow \{I\} \text{while}(B) S\{I \wedge \neg B\}$

I ist **Schleifeninvariante** – bleibt bei jeder Iteration unverändert

```
assert I;
while (B) {
    assert I && B;
    ...
    assert I;
}
assert I && !B;
```

schwieriger Schritt im Beweis: geeignete Schleifeninvariante finden

Bei Schleifen ist der Nachweis der **partiellen Korrektheit** (dass das Programm das Richtige tut, wenn es terminiert) komplexer. Hierfür ist eine **Schleifeninvariante** erforderlich.

Eine **Schleifeninvariante** ist eine Bedingung, die beschreibt, **was sich während der Iterationen einer Schleife nicht ändert**. Sie ist quasi das Gegenstück zur Beschreibung des Fortschritts für den Terminationsbeweis.

- **Eigenschaften der Schleifeninvariante:**

- Muss **vor der Schleife** erfüllt sein.
- Muss **nach der Schleife** erfüllt sein.
- Muss am **Beginn jeder Iteration** erfüllt sein.
- Muss am **Ende jeder Iteration** erfüllt sein (bevor die Schleifenbedingung erneut geprüft wird).
- Die **Schleifenbedingung selbst** kann **nicht Teil der Schleifeninvariante** sein, da sich die Schleifenbedingung im Laufe der Schleifenausführung ändert (sie wird irgendwann falsch, damit die Schleife terminiert).

Beispiel: Binäre Suche mit Schleife (Listing 4.15)

Listing 4.15 Binäre Suche mit Schleife

```

1 // pre: sorted(a);
2 // post: result r with r>=0 && r<a.length && a[r]==v;
3 // post: result r== -1 if not(v,a,0,a.length-1);
4 static int search(int v, int[] a) {
5     assert sorted(a);
6     int l = 0, h = a.length - 1;
7     assert inv(v,a,l,h);
8     while (l <= h) {
9         assert inv(v,a,l,h) && l<=h;
10        int r = (l + h) / 2;
11        assert inv(v,a,l,h) && l<=r && r<=h;
12        if (a[r] < v) {
13            assert inv(v,a,l,h) && l<=r && r<=h
14                && not(v,a,0,r);
15            l = r + 1;
16        } else if (a[r] == v) {
17            assert r>=0 && r<a.length && a[r]==v;
18            return r;
19        } else {
20            assert inv(v,a,l,h) && l<=r && r<=h
21                && not(v,a,r,a.length-1);
22            h = r - 1;
23        }
24        assert inv(v,a,l,h);
25    }
26    assert inv(v,a,l,h)&&l>h&&not(v,a,0,a.length-1);
27    return -1;
28 }
29
30 static boolean inv(int v, int[] a, int l, int h) {
31     return sorted(a) && not(v,a,0,l-1)
32             && not(v,a,h+1,a.length-1);
33 }
```

- In Listing 4.15 steht `inv(v,a,l,h)` für die Schleifeninvariante.
- Die Funktionen `sorted` und `not` sind wie in Listing 4.14 definiert.
- **Definition der Invariante `inv(v,a,l,h)` (Zeilen 30-32):**

```

static boolean inv ( int v , int [] a , int l , int h ) {
    return sorted ( a ) && not ( v ,a ,0 ,l -1)
        && not ( v ,a , h +1 , a. length -1);
}
```

Diese Invariante besagt:

1. `sorted(a)`: Das Array `a` bleibt **stets sortiert**.
 2. `not(v,a,0,l-1)`: Das gesuchte Element `v` ist **nicht** im Bereich von Index 0 bis `$l-1$` enthalten (d.h., links vom aktuellen Suchbereich).
 3. `not(v,a,h+1,a.length-1)`: Das gesuchte Element `v` ist **nicht** im Bereich von Index `$h+1$` bis zum Ende des Arrays enthalten (d.h., rechts vom aktuellen Suchbereich).
- **Implikation der Invariante:** Daraus folgt auch, dass wenn `$l > h$` ist (was die Abbruchbedingung der Schleife ist), das gesuchte Element `v` im gesamten Array `a` nicht vorkommen kann.

Beweisführung

Abgesehen von der Einführung der Schleifeninvariante unterscheidet sich die Beweisführung in Listing 4.15 (mit Schleife) nicht wesentlich von der in Listing 4.14 (mit Rekursion).

Finden einer Schleifeninvariante

Das Finden einer geeigneten Schleifeninvariante ist ein **kreativer und oft schwieriger Prozess**.

- **Empfehlung:** Es ist am besten, gleich beim **Erstellen einer Schleife** nach einer passenden Schleifeninvariante, einer klaren Abbruchbedingung und einer Beschreibung des Fortschritts zu suchen.
- Diese drei Komponenten legen den **Charakter der Schleife vollständig fest**.
- Wenn man diese drei Teile kennt, hat man die Schleife **vollständig verstanden**, was bei der Implementierung der Schleife Probleme vermeidet.

„Korrektheitsbeweis“ mit Schleife

```
// sum of all integers from 1 to n, i.e. 1+...+n;
// n >= 1;
public static int sum(int n) {
    int i = n, s = n;           { $n \geq 1 \wedge i = n \wedge s = n = \sum_{j=n}^n j$ }
    // Inv: i >= 1 && i <= n && s == i+...+n
    while (i > 1) {           { $n \geq 1 \wedge i > 1 \wedge i \leq n \wedge s = \sum_{j=i}^n j$ }
        s += --i;             { $n \geq 1 \wedge i \geq 1 \wedge i \leq n \wedge s = \sum_{j=i}^n j$ }
    }                         { $n \geq 1 \wedge i = 1 \wedge i \leq n \wedge s = \sum_{j=1}^n j$ }
    return s;
}
```

zusätzlich: auf **Fortschritt** in jeder Iteration achten

Codereview

In der Softwareentwicklung spielen **Dokumente** eine zentrale Rolle bei der Gestaltung und Wartung von Software. Dabei sind **Programmtexte** besonders wichtig.

Code-Review

erfahrene Person liest Programmtext zusammen mit der Dokumentation und gibt Beurteilung hinsichtlich geprüfter Kriterien ab

Code-Review meist von externer Person durchgeführt

Kriterien geprüft, die nicht einfach automatisch beurteilbar sind

kann Fehler und Schwächen aufdecken, die anders kaum zu finden sind

erfordert (wenn richtig gemacht) hohe Konzentration, sehr zeitaufwendig

Bedeutung von Programmtexten

Kriterien für Code-Review (Beispiele)

Software-Verträge (DbC) brauchbar und eindeutig
 Software-Verträge erfüllt und Programm statisch verstehbar
 Dokumentation (Kommentare) konsistent mit Programmtexten
 Objekte von Obertypen durch alle Objekte von Untertypen ersetzbar
 passende Algorithmen und Datenstrukturen eingesetzt
 nur notwendige Eigenschaften vorausgesetzt
 Namensgebung konsistent und intuitiv
 alle Daten, die von außen kommen, sinnvoll validiert
 nicht mehr ausführbare Programmteile entfernt
 Sichtbarkeit von Programmteilen bestmöglich beschränkt

Programmtexte haben nicht nur den Zweck, Anweisungen für eine Maschine zu sein, sondern auch:

- Sie spiegeln unsere **Gedanken während der Entwicklung** wider.
- Sie helfen, Gedanken **fokussiert, organisiert und nachprüfbar konsistent** festzuhalten.
- Sie dienen dem **Gedankenaustausch mit anderen Personen**.
- Sie sind eine **Gedächtnisstütze für uns selbst**.

Heute werden fast alle wichtigen Aspekte von Programmen, oft auch Anforderungen und Testfälle, **direkt in den Programmtexten** ausgedrückt. Die Bedeutung von Programmtexten als **Kommunikationsmittel** nimmt mit der Komplexität und Langlebigkeit von Programmen zu.

Kommentare in Programmtexten

Kommentare sollen **Informationen liefern, die für die Konsistenz des Gesamtsystems wichtig sind und nicht direkt aus den Sprachkonstrukten ableitbar sind**. Sie dienen nicht dazu, komplexe Sprachkonstrukte umgangssprachlich zu beschreiben.

Dazu gehören insbesondere:

- **Zusicherungen** (Assertions) gemäß **Design by Contract (DbC)**.
- **Komplexe Schleifeninvarianten**.

Ohne diese Informationen wären Programme und Programmteile kaum zu verstehen.

- Die Entwicklung **guter Zusicherungen** kann **aufwendiger** sein als die des eigentlichen Programms. Oft beginnt die Entwicklung sogar mit den Zusicherungen, die kontinuierlich aktuell gehalten werden müssen.
- **Oberflächliche Kommentare**, die nur zur "Wahrung des Scheins" hinzugefügt werden, sind **unsinnig**.

Design-Rules und Kommentar-Regeln

Design-Rules umfassen alle Regeln, an die sich ein Softwareentwicklungsteam halten muss. Die spezifischen Regeln für Kommentare können je nach Unternehmen und Projekt variieren.

Häufig müssen folgende Informationen in Kommentaren enthalten sein:

- **Vor einer Klasse oder einem Interface:**
 - Zweck, Grobstruktur und Besonderheiten.
 - Verantwortliche für Erstellung und Wartung.
 - Anvisierte Benutzer.
- **Bei den Deklarationen von Objektvariablen:**
 - Zweck der Variablen.
 - Invarianten der Variablen (Bedingungen, die immer erfüllt sein müssen).
- **Vor Methoden und Konstruktoren:**
 - Vorbedingungen (Preconditions).
 - Nachbedingungen (Postconditions) – diese beschreiben oft auch den Zweck der Methode.
- **In Methodenrümpfen:**
 - Schleifeninvarianten.
- **Überall:** Hinweise auf erfolgte Änderungen, um zu verhindern, dass Änderungen wiederholt gemacht und rückgängig gemacht werden.

Beobachtungen zu Kommentaren

- Der Großteil der Kommentare steht typischerweise bei Deklarationen oder vor Definitionen, vergleichsweise wenig in den eigentlichen Rümpfen von Methoden und Konstruktoren.
- Die Qualität von Kommentaren hängt nicht von der Menge ab, sondern von ihrer Aussagekraft und guten Platzierung.
- Ein übermäßiger Bedarf an Kommentaren kann auf unausgereifte oder fehleranfällige Programmteile hindeuten. Gut durchdachte und stabile Teile kommen oft mit wenigen Kommentarzeilen aus.
- Die Java-Standardbibliotheken bieten viele Beispiele für hochwertige Dokumentationen von Klassen und Interfaces. Man sollte ähnliche hohe Standards anstreben.

Merkmale hochwertiger Kommentare (nach Java-Standardbibliotheken)

- Sie sind aus dem **Blickwinkel der Anwendung** geschrieben.
- Sie geben **Implementierungsdetails nur dort preis, wo es unumgänglich ist**.
- Sie sind als **allgemeinverständliche Texte** gestaltet.
- Trotzdem sind sie **unmissverständlich und präzise Zusicherungen** gemäß Design by Contract.
- **Namen** folgen einem **einheitlichen Schema** und unterstützen intuitiv die Zusicherungen.
- **Randfälle** sind **sehr genau beschrieben**.
- Die **Textlänge ist dennoch kurz gehalten**.

Statisches Verstehen und Code-Review

Qualitativ hochwertige Programmteile sind ohne großen Aufwand statisch verstehbar. Das statische Verständnis ist eine unabdingbare Voraussetzung für den Korrektheitsbeweis und somit langfristig für die Korrektheit des Programms. Ein hoher Aufwand für das Verstehen kann zu einem gewaltigen Kostenfaktor werden.

- Obwohl es ausgefeilte Werkzeuge zur **automatischen Überprüfung der Programmkorrektheit** gibt, können diese Werkzeuge **informelle Kommentare und die Intuition hinter Namen nicht verstehen**.
- Daher bleibt die **Code-Review** eines der wichtigsten Hilfsmittel zur Beurteilung der Qualität von Programmteilen.

Code-Review

Bei einer **Code-Review** liest eine Person den Programmtext und gibt Feedback.

- Oft werden Code-Reviews von **erfahrenen Personen durchgeführt, die nicht zum direkten Entwicklungsteam gehören**. Gelegentlich beurteilen auch Teammitglieder gegenseitig Programmtexte.
- Das Feedback enthält allgemeine Auffälligkeiten und Beurteilungen anhand einer **vorgegebenen Kriterienliste**, die sich je nach Fall unterscheidet.
- Die Kriterienliste orientiert sich an den **Design-Rules**. Automatisch beurteilbare Kriterien sind normalerweise nicht enthalten, um die Konzentration auf das Wesentliche zu fokussieren.

Beispiele für Kriterien bei einer Code-Review

- Sind **Software-Verträge (Design by Contract)** so beschrieben, dass kompetente Personen klar verstehen, was verlangt wird und was erwartet werden darf?
- Erfüllen **Implementierungen von Methoden und Konstruktoren** nachweislich alle ihre in den Software-Verträgen beschriebenen Pflichten (d.h., werden unter Annahme der Vorbedingungen und Invarianten alle Nachbedingungen und Invarianten eingehalten)?
- Erfüllen **Methodenaufrufe** nachweislich alle ihre in den Software-Verträgen beschriebenen Pflichten (d.h., sind alle Vorbedingungen aufgerufener Methoden erfüllt und sind zu den Zeitpunkten von Methodenaufrufen alle bekannten Invarianten erfüllt)?
- Sind **Kommentare konsistent** mit dem Rest des Programms?
- Sind **Untertypen wirklich Spezialisierungen** ihrer Obertypen (Liskovsches Substitutionsprinzip)?
- Werden **nur nötige Eigenschaften vorausgesetzt** (z.B. kein Untertyp verlangt, wo ein Oberotyp ausreichen würde)?
- Sind alle Programmteile mit **vertretbarem Aufwand statisch zu verstehen**?
- Ist die **Namensgebung in sich konsistent und intuitiv**?
- Werden **alle Daten validiert**, die von außerhalb des Programms kommen könnten?
- Sind **Überprüfungen von Daten ausreichend restriktiv**, um ein Eindringen von außen zu verhindern?
- Werden **Überprüfungen von Daten nicht unnötig wiederholt** und sind sie in sich konsistent?
- Werden **adäquate Algorithmen und Datenstrukturen eingesetzt**?
- Kommen **keine unnötigen Zusicherungen oder Programmteile** vor (möglicherweise nach Änderungen übriggeblieben)?
- Ist die **Sichtbarkeit von Programmteilen bestmöglich beschränkt** (Kapselung)?

Diese Liste kann beliebig erweitert werden.

Wert von Code-Reviews

- Code-Reviews sind **extrem fordernd** und erfordern hundertprozentige Konzentration über lange Zeiträume, was zu langsamen Fortschritten führt.
- Dennoch zahlen sich Code-Reviews aus, da nur durch sie eine **hohe Programmqualität erreichbar** ist.

- Sie bewirken unter anderem, dass bereits bei der Programmerstellung noch mehr Wert auf die überprüften Kriterien gelegt wird.

Programmdokumentation

Zweck der Programmdokumentation

Kommunikation zwischen Personen, die an Entwicklung und Wartung beteiligt

Anforderungen an zu entwickelnde Software festlegen

gemeinsame Richtlinien (Design-Rules) festlegen

festlegen, wer wofür verantwortlich (Design-by-Contract)

ermöglicht Einarbeitung in Projekt

Dokumentation heute zu einem großen Teil als Kommentare in Programmtexten

Wo welcher Kommentar steht

vor Klasse/Interface:

Zweck, Grobstruktur, Anwendungshinweise, Besonderheiten, Verantwortliche

bei Deklarationen von Objektvariablen:

Zweck der Variablen, Invarianten

vor Methoden/Konstruktoren:

Vorbedingungen, Nachbedingungen (schließen Zweck ein)

in Methodenrümpfen:

Schleifeninvarianten

betroffene Stellen (überall):

Hinweise auf erfolgte Änderungen

Hochwertige Kommentare

aus Blickwinkel der Anwendung beschrieben,
vermeiden Hinweise auf Implementierungsdetails

allgemein verständliche Texte,
dennoch unmissverständlich, präzise, Zusicherungen gemäß Design-by-Contract

Namen entsprechen einheitlichem Schema, sind intuitiv

Randfälle genau beschrieben (wenn nicht ohnehin klar)

Textlänge kurz gehalten

Erkenntnisse aus Dokumentation

Dokumentation (Kommentare) generell hochwertig → Software „vertrauenswürdig“
→ häufiger wiederverwendet (unabhängig von sonstiger Qualität)

Dokumentation zu bestimmten Aspekten (Sicherheit, Wartungsfreundlichkeit, . . .)
→ Aspekte vermutlich durchdacht → weniger fehleranfällig

häufig erfolgte Änderungen an bestimmten Stellen im Programmtext
→ Hinweis auf konzeptuelle Schwächen

Häufung von Kommentaren an bestimmten Stellen
→ Hinweis auf (möglicherweise unnötige) Komplexität

Meta-Dokumentation: Design-Rules

Regeln, an die sich alle Personen im Team halten müssen

dienen teilweise nur dem Team-Zusammenhalt (Erkennungswert fördert Kommunikation),
haben häufig einen auf Erfahrung beruhenden Hintergrund (Fehlervermeidung)

Beispiel: equals darf nicht über toString implementiert werden;
gilt auch, wo Implementierung über toString (scheinbar) funktioniert,
weil Erfahrung zeigt, dass bei Prüfung Sonderfälle übersehen werden

Sicherheitslücken

Beispiel: Pufferüberlauf

Datenmenge größer als dafür vorgesehener Speicherbereich,
typisches Beispiel: Arrayzugriffe außerhalb erlaubter Indexgrenzen

bekanntes Sicherheitsrisiko, am häufigsten für Systemeinbrüche genutzte Fehler
scheinbar kein Problem in Java, weil Index für jeden Arrayzugriff überprüft,
aber Experten finden dennoch Möglichkeiten für Pufferüberläufe,
etwa über native Methoden, Ausschalten von Laufzeitüberprüfungen, ...
vor allem: Array größer als genutzter Arraybereich

mögliche Design-Rule:

- Größe eingelesener Zeichenketten muss beschränkt sein,
- Einhaltung der Schranke ist bei Validierung eingelesener Daten zu prüfen,
- Validierung und Angemessenheit der Schranke ist durch Code-Review zu prüfen

Beispiel: SQL-Injection

eingegebene Daten könnten wie Programme ausgeführt werden,
z.B. eingegebener Begriff in SQL-Datenbank gesucht (als Teil der SQL-Abfrage),
aber vermeintlicher Begriff ist in Wirklichkeit selbst SQL-Ausdruck

bekanntes Sicherheitsrisiko für Datenleck und Manipulation von Daten

kein allgemeiner wirksamer Schutz (der nicht nur für SQL wirken würde),
aber PreparedStatement unterstützt sicheres Zusammensetzen von SQL-Abfragen

mögliche Design-Rule:

- keine eingelesenen Daten an externe Systeme (wie Datenbanken) übergeben,
- außer wenn Daten überprüft, etwa durch PreparedStatement zusammengesetzt;
- Überprüfung durch ein Werkzeug oder Code-Review sichergestellt

Beispiel: Privilege-Escalation

über sozialen Druck oder Tricks Benutzer dazu gebracht,
wenig vertrauenswürdigen Programmen hohe Sicherheitsstufen zuzuerkennen,
über diese Privilegien verschaffen sich weitere Programme Zugang zu geschützten Daten

großes Sicherheitsrisiko, wegen sozialem Aspekt schwer kontrollierbar

mögliche Design-Rule:

- nur freigegebene Werkzeuge und Softwarepakete dürfen verwendet werden,
- Freigabe erfolgt nur nach genauer Prüfung durch speziell dafür geschultes Personal

13. Testen Qualität Optimierung

Testen von Programmen

Das **Testen von Programmen** ist primär ein **Ausprobieren**. Obwohl es bei kleinen Programmen einfach erscheint, ist es bei komplexen Programmen mit Qualitätsansprüchen sehr anspruchsvoll.

Wissenswertes zum Testen

ep2-25v13Folien, p.2, Skriptum, p.154

Die folgenden Fakten widerlegen die einfache Vorstellung, die man von kleinen Programmen hat:

- **Fehlersuche vs. Fehlerfreiheit:**
 - Intensiveres Testen findet mehr Fehler.
 - **Niemals** können *alle* Fehler zuverlässig aufgedeckt werden.
- **Fehlerdichte als Indikator:**
 - Die Anzahl der gefundenen Fehler (bei bestimmter Testintensität) ist oft **proportional zur Anzahl der tatsächlich vorhandenen Fehler**.
 - Werden in einem Programmteil viele Fehler gefunden, deutet dies auf eine höhere Fehlerdichte in diesem Teil hin.
 - Testergebnisse geben **Hinweise auf die Qualität** von Programmteilen.
- **Fehlerursachen und neue Fehler:**
 - Das Aufdecken eines Fehlers durch Testen identifiziert nicht sofort die (oft mehreren) Fehlerursachen.
 - Die Behebung einer scheinbaren Ursache **garantiert nicht die Fehlerbeseitigung**.
 - Eine Änderung zur Fehlerbehebung kann **neue Fehler einführen** ("Ein Fehler ausgebessert, zehn Fehler eingeführt.").
- **Definition eines Fehlers:**
 - Es ist oft **unklar, was genau ein Fehler ist**.
 - Ein ungewöhnliches Verhalten kann aus einem Blickwinkel ein Fehler sein, aus einem anderen jedoch **gewollt**.
 - **Bewusst eingebaute "Fehler"**: Manchmal werden, entgegen den Design-Regeln, gezielt Schwachstellen eingebaut, um an anderer Stelle Vorteile zu erzielen (z.B. Lücken in Passwortabfragen für automatisiertes Testen).
- **Unentdeckte Fehler:**
 - Ein Fehler, der über die gesamte Lebenszeit eines Programms unentdeckt bleibt, **wirkt sich nicht störend aus**.
 - Man weiß jedoch nie, ob er nicht doch noch entdeckt wird.
- **Auswirkungen von Fehlern:**
 - Fehler können sich **unterschiedlich auswirken**, von harmlos bis katastrophal (sogar tödlich).
 - Die Beurteilung der Schwere ist **subjektiv**.
 - **Beispiel**: Eine vergessene Eingabeüberprüfung kann harmlos wirken, bis ein Angreifer sie für einen Systemeinbruch nutzt. Selten auftretende Fehler können hierbei eine besonders große Gefahr darstellen, da sie bewusst von Angreifern ausgenutzt werden, um in Systeme einzudringen.

Je besser versteckt die Fehler sind, desto schwieriger wird die Beseitigung.

Strategien zu Entwicklung von Testfällen

Ein gezieltes Vorgehen und **Qualitätsmanagement** sind notwendig, um die Softwarequalität durch Testen und Debuggen kontinuierlich zu verbessern.

Bewährte Maßnahmen (aus einer unvollständigen Liste) umfassen:

- **Strategische Festlegung von Testfällen:**
 - Testfälle werden nach einer bestimmten **Strategie** festgelegt.
 - Verschiedene Strategien legen unterschiedliche Schwerpunkte fest, generieren aber alle mehr und komplexere Testfälle als bloßes Ausprobieren.
 - Die bekanntesten Strategien sind:
 - **Black-Box-Testen:**
 - Leitet Testfälle aus **Anwendungsfällen** (Spezifikation, Anforderungen) ab.
 - Testfälle und Programm entstehen **unabhängig voneinander**.
 - Ziel ist es, Fehler in Situationen aufzudecken, die bei der Programmentwicklung (interner Aufbau) nicht bedacht wurden. Der Tester kennt die interne Struktur des Programms nicht.
 - **White-Box-Testen:**
 - Leitet Testfälle aus der **Implementierung** (interner Code, Struktur) ab.
 - Es wird meist versucht, mindestens einen Testfall für jeden möglichen Programmweg einzuführen.
 - Ziel ist es, eine gute **Coverage** (Abdeckung) aller Programmteile zu erreichen. Der Tester hat hierbei Kenntnis über die interne Struktur des Programms.
 - **Grey-Box-Testen:**
 - Legt Testfälle bereits **vor der Implementierung** als **Spezifikation des Programms** fest.
 - Programme orientieren sich an diesen Tests.
 - Der Charakter liegt häufig **zwischen Black-Box- und White-Box-Tests**. Es gibt teilweise Kenntnisse über die interne Struktur, aber nicht vollständig.

Automatisierte Testläufe

Regressions-Test:

sehr empfehlenswert

nach Änderungen alle Testfälle neuerlich geprüft,
sodass durch Änderungen eingeführte Fehler eher erkannt

Regressions-Tests nur sinnvoll durchführbar wenn **automatisiert**:

eigene Programmteile nur für Testen,
durch spezielle Werkzeuge unterstützt (z.B. JUnit),
sind aber auch ohne Werkzeuge einfach zu implementieren

Kategorisieren erkannter Fehler

1. Fehler, die hohes Sicherheitsrisiko darstellen
2. Fehler, die Software für viele User unbenutzbar macht
3. Fehler, die zu Sicherheitsrisiko werden könnten
4. übrige Fehler

Einteilung in Teambesprechung

vorher klären, ob Fehler im Programm oder beim Testen

Sicherheit und Benutzbarkeit testen

ep2-25v13Folien, p.7

"Normales Testen" reicht oft nicht aus, um die Qualität in bestimmten Bereichen wie **Sicherheit und Benutzbarkeit** zu gewährleisten. Hierfür sind gezielte Fehlerstrategien notwendig:

- **Sicherheits- und Benutzbarkeitstests:**
 - Das "normale Testen" ist für diese Aspekte **unzureichend**.
 - Es ist eine **gezielte Fehlersuche** erforderlich.
- **Code-Review:**
 - Durchgehende, systematische Überprüfung des Quellcodes durch andere Entwickler, um Fehler, Schwachstellen oder unsauberer Code zu identifizieren.
- **Sicherheitstest:**
 - **Experten versuchen, in das System einzudringen**, um Schwachstellen und Sicherheitslücken aufzudecken (z.B. Penetrationstests).
- **Belastungstest (Load Testing):**
 - Testet das System bei **extrem hoher, aber erwarteter Belastung** (z.B. viele gleichzeitige Benutzer, große Datenmengen), um die Performance und Stabilität unter realen Bedingungen zu prüfen.
- **Stresstest (Stress Testing):**
 - Nimmt die **schlechtestmögliche Annahme in jedem Aspekt** an und testet das System unter extremen, über die normalen Betriebsbedingungen hinausgehenden Bedingungen, um dessen Robustheit und Fehlerbehandlung zu bewerten. Das Ziel ist, die Belastungsgrenzen des Systems zu finden und zu sehen, wie es sich unter extremen Bedingungen verhält.
- **Crashtest:**
 - Ziel ist es, das System **funktionsunfähig zu machen oder zum Absturz zu bringen**, um zu verstehen, wie es auf unerwartete oder fehlerhafte Eingaben reagiert und ob es sich sicher herunterfährt oder Datenverluste verhindert.
- **Benutzeroberflächentest (Usability Testing):**
 - Untersucht, **wie die Software von Endbenutzern verwendet wird**, um die Benutzerfreundlichkeit, Effizienz und Zufriedenheit zu bewerten (z.B. durch Beobachtung von Testpersonen bei der Aufgabenlösung).

Umgang mit aufgedeckten Fehlern: Ursachenforschung und Beseitigung

Ein aufgedeckter Fehler bedeutet nicht sofort, dass die **Fehlerursache** klar ist. Es ist entscheidend, die Ursache zu finden, bevor man versucht, die **Auswirkung des Fehlers** zu beseitigen.

- **Fehler aufgedeckt ≠ Fehlerursache bekannt:**
 - Wenn ein Fehler entdeckt wird, ist zunächst nur dessen **Auswirkung** sichtbar.
 - Eine direkte Behebung der Auswirkung kann **große Probleme** verursachen oder zu neuen Fehlern führen.
- **Priorität: Ursachenforschung vor Beseitigung:**
 - Der erste Schritt nach dem Aufdecken eines Fehlers ist immer die **Ursachenforschung**.
 - Erst nachdem die genaue Fehlerursache identifiziert wurde, sollte diese **beseitigt** werden.
- **Hilfsmittel zur Fehlersuche:**
 - **Code-Review:**
 - **Immer notwendig** und sehr effektiv.
 - Systematische Überprüfung des Quellcodes durch andere Entwickler zur Identifizierung von Fehlern und Schwachstellen.
 - **Debugger:**
 - Nützlich, aber meist nur für **einfache Fälle**.
 - Ermöglicht das schrittweise Ausführen von Code, um Variablenwerte und Programmfluss zu überprüfen.
 - **Log-Dateien und Traces:**
 - Automatisch generierte Aufzeichnungen von Systemereignissen und Programmausführungen.
 - Bieten wertvolle Informationen über den Zustand des Systems und die Abfolge von Operationen vor dem Auftreten eines Fehlers.
 - **Einbeziehung relevanter Personen:**
 - **Entwickler:** Personen, die an der Software mitentwickelt haben und den Code gut kennen.
 - **Experten im betroffenen Gebiet:** Fachleute mit tiefem Wissen über den spezifischen Bereich, in dem der Fehler auftrat (z.B. Datenbankexperten bei Datenbankfehlern).
 - **Intensive Benutzer:** Personen, die die Software regelmäßig und intensiv nutzen und möglicherweise ungewöhnliche Nutzungsmuster oder Randfälle kennen, die zu Fehlern führen können.

Mehrstufiges Testen: Kostenoptimierung und Qualitätssicherung

ep2-25v13Folien, p.10

Die **Fehlerbeseitigung** wird im Laufe des Entwicklungszyklus **immer teurer**, je später ein Fehler entdeckt wird. Um Kosten zu sparen und die Qualität zu sichern, wird ein **mehrstufiges Testverfahren** angewendet:

1. Modul- bzw. Komponententest (Unit Testing):

- **Wann:** Sehr früh im Entwicklungsprozess.
- **Was:** Einzelne, **unabhängige Programmteile (Module, Komponenten oder Funktionen)** werden isoliert getestet.
- **Ziel:** Sicherstellen, dass jede Komponente für sich korrekt funktioniert.

2. Schnittstellentest (Interface Testing):

- **Was:** Überprüft das **Zusammenspiel und die Kommunikation zwischen mehreren miteinander verbundenen Programmteilen.**
- **Ziel:** Sicherstellen, dass Daten korrekt übergeben werden und die Interaktion der Module fehlerfrei ist.

3. Integrationstest (Integration Testing):

- **Was:** Testet das **Zusammenspiel aller oder einer größeren Gruppe von Programmteilen in einer realitätsnahen Umgebung.**
- **Ziel:** Überprüfen, ob die integrierten Komponenten als Einheit funktionieren und die Anforderungen erfüllen.

4. Systemtest (System Testing):

- **Was:** Testet das **vollständige Softwaresystem in einer möglichst realen Anwendungsumgebung.**
- **Wann:** Nach Abschluss der Integrationstests.
- **Ziel:** Überprüfen, ob das Gesamtsystem die spezifizierten Anforderungen erfüllt, einschließlich funktionaler und nicht-funktionaler Aspekte (Performance, Sicherheit, etc.).
- Wird auch als **Abnahmetest** bezeichnet, wenn dabei die **vereinbarten Qualitätskriterien** mit dem Kunden oder Endbenutzer geprüft werden.

Empfehlungen für effektives Testen und Verbesserungen

ep2-25v13Folien, p.11

Um die Qualität von Software nachhaltig zu verbessern, sollten über das reine Testen hinaus weitere Maßnahmen ergriffen werden:

- **Identifizierung und Überarbeitung fehleranfälliger Stellen:**
 - **Testergebnisse** dienen dazu, **fehleranfällige Programmstellen** zu erkennen.
 - Es ist oft **effizienter, besonders fehlerträchtige Bereiche zu überarbeiten oder komplett auszutauschen**, anstatt viele einzelne Fehler aufwändig zu beseitigen. Dies kann zu einer grundlegenden Verbesserung der Code-Qualität führen.
- **Nutzung von Praxishinweisen für Testfälle:**
 - **Hinweise aus der praktischen Anwendung** und **erkannte Fehlerursachen** sind wertvolle Quellen.
 - Diese Informationen sollten genutzt werden, um **bei Bedarf zusätzliche Testfälle einzuführen**, die spezifische Probleme oder Anwendungsfälle abdecken, die bisher nicht ausreichend getestet wurden.
- **Überprüfung der Testqualität:**
 - Die **Qualität des Testprozesses selbst muss regelmäßig überprüft werden.**
 - Eine Methode hierfür ist, absichtlich Fehler in den Code einzuführen und dann die **Zuverlässigkeit der Testsuite bei der Erkennung dieser Fehler zu messen**. Dies hilft, Lücken in den Testverfahren aufzudecken und die Effektivität der Tests zu bewerten (z.B. durch Mutationenstests).

Qualität

Kriterien für Programmqualität: Eine Konfliktsituation

ep2-25v13Folien, p.13

Programmqualität wird durch eine Vielzahl von Kriterien bestimmt, die oft in **Konflikt** zueinander stehen können (z.B. hohe Sicherheit vs. hohe Effizienz).

Brauchbarkeit (Usability)

Ein Programm ist brauchbar, wenn es die Bedürfnisse der Benutzer erfüllt und einfach zu handhaben ist:

- **Aufgabenerfüllung:** Das Programm erfüllt seinen vorgesehenen Zweck und unterstützt den Benutzer bei seinen Aufgaben.
- **Bedienbarkeit:** Das Programm ist einfach zu erlernen und intuitiv zu bedienen.
- **Konfigurierbarkeit:** Benutzer können das Programm an ihre spezifischen Bedürfnisse und Vorlieben anpassen.
- **Spaß:** Die Nutzung des Programms bereitet Freude oder ist zumindest nicht frustrierend.

Zuverlässigkeit und Sicherheit (Reliability & Security)

Diese Kriterien stellen sicher, dass das Programm korrekt und geschützt arbeitet:

- **Korrektheit:** Das Programm liefert immer die richtigen Ergebnisse gemäß den Spezifikationen.
- **Datenintegrität:** Daten sind konsistent, korrekt und vollständig.
- **Seltene Ausfälle:** Das Programm stürzt selten ab oder hängt sich auf.
- **Klare Hinweise auf Fehler:** Bei Fehlern gibt das Programm verständliche Informationen.
- **Intuitives Verhalten:** Das Programm verhält sich vorhersehbar und logisch.
- **Antwortzeiten im erwarteten Bereich:** Das Programm reagiert schnell genug auf Benutzereingaben.
- **Nichts Unerwartetes:** Keine unvorhergesehenen oder unerwünschten Verhaltensweisen.
- **Keine Daten an/von Unberechtigten:** Sensible Daten sind vor unautorisiertem Zugriff geschützt.
- **Sicherheitslücken rasch geschlossen:** Bekannte Schwachstellen werden schnell behoben.

Effizienz (Efficiency)

Effizienz bezieht sich auf den sparsamen Umgang mit Ressourcen und die Leistung des Programms:

- **Sparsamer Ressourceneinsatz:** Das Programm nutzt CPU, Speicher, Festplatte und Netzwerk effizient.
- **Andere Programme wenig gestört:** Das Programm beeinträchtigt die Leistung anderer Anwendungen auf demselben System nicht wesentlich.
- **Wenig Interaktionen nötig:** Aufgaben können mit minimalen Schritten oder Eingaben erledigt werden.
- **Entwicklungszeiten bleiben im Rahmen:** Die Entwicklung erfolgt innerhalb eines akzeptablen Zeitrahmens.

Wartbarkeit (Maintainability)

Wartbarkeit ist entscheidend für die langfristige Pflege und Weiterentwicklung der Software:

- **Einfachheit:** Der Code ist einfach zu verstehen und zu ändern.
- **Gut lesbar:** Der Code ist klar strukturiert und kommentiert.
- **Softwareverträge klar:** Schnittstellen und Abhängigkeiten sind gut definiert.
- **Änderungen einfach:** Neue Funktionen können leicht hinzugefügt oder bestehende geändert werden, ohne unerwünschte Nebenwirkungen.

- **Schwache Abhängigkeiten:** Module sind so entworfen, dass Änderungen in einem Modul minimale Auswirkungen auf andere haben.
- **Entwicklungsteam auch für Wartung:** Das Team, das die Software entwickelt hat, ist auch für deren Wartung verantwortlich, was das Wissen über den Code und die schnellen Fehlerbehebung verbessert.

Maßnahmen zur Qualitätssteigerung in der Softwareentwicklung

ep2-25v13Folien, p.14

Die Steigerung der Softwarequalität erfordert einen umfassenden Ansatz, der verschiedene Maßnahmen über den gesamten Entwicklungszyklus hinweg integriert:

- **Qualitätsbewusstsein schaffen:**
 - **Design-Rules (Entwurfsregeln):** Festlegung von Richtlinien und Standards für Design und Implementierung, die konsistente und qualitativ hochwertige Ergebnisse fördern.
 - **Schulungen:** Regelmäßige Weiterbildung der Entwickler zu Best Practices, neuen Technologien und qualitätssichernden Maßnahmen.
 - **Zertifizierungen:** Externe Zertifizierungen (z.B. nach ISO-Standards) können das Qualitätsbewusstsein und die Einhaltung von Standards fördern.
 - **Sehr effektiv:** Diese Maßnahmen verankern das Qualitätsdenken im Team.
- **Anforderungsanalyse fokussieren:**
 - Klar definieren, **auf welche Qualitätskriterien es ankommt**. Dies umfasst funktionale und nicht-funktionale Anforderungen (z.B. Performance, Sicherheit, Usability).
 - Eine präzise Anforderungsanalyse bildet die Grundlage für zielgerichtete Qualitätssicherung.
- **Ständige Kontrolle im Entwicklungsprozess:**
 - Qualitätssicherung muss **in den gesamten Entwicklungsprozess integriert** sein, nicht nur am Ende.
 - **Beispiel: Pair-Programming:** Zwei Entwickler arbeiten gemeinsam an einem Code, wobei einer schreibt und der andere beobachtet und Feedback gibt. Dies führt zu sofortiger Fehlererkennung und besserem Design.
- **Regelmäßige Besprechungen:**
 - **Regelmäßige Meetings für das gesamte Team** sind entscheidend.
 - Auch **Einschulungen** zu neuen Tools, Prozessen oder Problemen.
 - Fördert den Wissensaustausch und die gemeinsame Verantwortung für Qualität.
- **Analyse-Werkzeuge einsetzen:**
 - Verwendung von Tools zur **automatisierten Bewertung der Code-Qualität nach vordefinierten Modellen** (z.B. für Code-Komplexität, Duplikierung).
 - Dies ist eine Ergänzung zum **Code-Review**, da automatische Tools menschliche Fehler finden, wo Menschen diese übersehen könnten.
- **Formale Beweise:**
 - **Notwendig bei sehr hohen Sicherheits- oder Zuverlässigkeitssanforderungen** (z.B. in der Luft- und Raumfahrt, Medizintechnik).
 - Mathematische Methoden werden eingesetzt, um die Korrektheit des Programms zu beweisen.
 - Für **große Programme** ist dies oft mit **extrem hohem Aufwand** verbunden und daher nicht immer praktikabel.
- **Aufeinander Verlassen (Teamvertrauen):**

- **Teambildende Maßnahmen** und eine Kultur des Vertrauens sind

Optimierung

Grundsätze der optimierung

für Nichtexperten: „Verzichte auf Optimierungen.“



für Experten: „Warte, bis das Programm fertig ist.“

Optimierung von Programmen

ep2-25v13Folien, p.17

Optimierung bezeichnet die Steigerung der **Laufzeit-, Speicher- und/oder Energieeffizienz** eines Programms. Dies kann auf verschiedenen Ebenen erfolgen:

- **Compiler-Optimierung:** Der Compiler analysiert den Quellcode und generiert effizienteren Maschinencode.
- **Laufzeitsystem-Optimierung:** Das Laufzeitsystem (z.B. Java Virtual Machine) kann zur Laufzeit weitere Optimierungen durchführen (z.B. Just-In-Time-Kompilierung).
- **Verändern von Programmtexten (Manuelle Optimierung):** Direkte Änderungen am Quellcode durch den Entwickler.

Argumente gegen Optimierung durch Verändern von Programmtexten

Obwohl manuelle Optimierungen manchmal notwendig sein können, sprechen oft folgende Gründe dagegen, sie als primäres Mittel einzusetzen:

- **Einseitige Fokussierung:** Manuelle Optimierungen konzentrieren sich oft nur auf einen Aspekt (z.B. Laufzeit), während andere **Qualitätskriterien wie Lesbarkeit, Einfachheit und Wartbarkeit leiden** können.
- **Wettbewerb mit dem Compiler:** Ohne **Expertenzwissen** über Compiler-Interna ist manuelle Optimierung häufig **in Konkurrenz zu den Optimierungen des Compilers**. Der Compiler kann oft effizientere Entscheidungen treffen, da er den gesamten Code besser überblickt und auf Maschinenebene optimiert. Eine manuelle Optimierung kann die Compiler-Optimierung sogar behindern.
- **Häufige Hinfälligkeit:** **Programmänderungen machen Optimierungen häufig hinfällig.** Eine manuell optimierte Stelle könnte nach einer kleinen Code-Änderung ihre Effizienz verlieren oder sogar ineffizienter werden.
- **Große Fehlerwahrscheinlichkeit:** Manuelle Optimierungen sind **fehleranfällig**. Der Eingriff in den Code zur Leistungssteigerung kann unbeabsichtigt neue Fehler einführen oder bestehende Funktionalität beeinträchtigen, da die Änderungen oft komplex und schwer zu überblicken sind.

Sinnvolle Optimierungsmaßnahmen

ep2-25v13Folien, p.18

Statt sich auf manuelle Code-Anpassungen zu konzentrieren, sollten Optimierungsmaßnahmen an einem früheren Punkt im Entwicklungsprozess ansetzen und eine ganzheitliche Perspektive einnehmen:

- **Auswahl geeigneter Algorithmen und Datenstrukturen:**
- **Komplexe Lösungen durch einfache ersetzen:**
 - Oft sind **einfachere Implementierungen** nicht nur lesbarer und wartbarer, sondern auch effizienter.
 - Weniger Code, weniger Verzweigungen und weniger komplexe Logik führen oft zu besserer Performance und geringerer Fehleranfälligkeit.
- **Data-Hiding (private) anwenden:**
 - Die konsequente Nutzung von **Data-Hiding (z.B. durch `private`-Schlüsselwörter)** ist eine grundlegende Praxis der objektorientierten Programmierung.
 - Es schützt interne Implementierungsdetails einer Klasse vor dem Zugriff von außen.
 - Dies ermöglicht es, die **interne Implementierung zu ändern (z.B. die zugrunde liegende Datenstruktur)**, ohne die externen Nutzer der Klasse zu beeinflussen. Dadurch können Optimierungen vorgenommen werden, ohne das gesamte System zu refaktorieren.
- **Ganzheitliche Betrachtung statt Klein-Kariertheit:**
 - Nicht nur einzelne Codezeilen oder kleine Funktionen betrachten.
 - **Über mehrere Ecken denken und die Gesamtheit des Systems betrachten.**
 - **Beispiel:** Eine `Klasse` als Typ kann in manchen Fällen eine Spur effizienter sein als ein `Interface`, da bei einem `Interface` zusätzliche Indirektionen (Methodentabellen) entstehen können.
 - **ABER:** Ein `Interface` ermöglicht einen einfachen Austausch der zugrunde liegenden **Datenstruktur**.
 - Durch die **bessere Wahl der Datenstruktur (hinter dem Interface)** kann der Effizienzgewinn oft **deutlich größer** sein als der geringe Performance-Verlust durch das `Interface` selbst. Dies ist ein Paradebeispiel dafür, wie Flexibilität und bessere Design-Entscheidungen zu einer größeren Gesamtoptimierung führen können.

Vorgehensweise beim Optimieren

ep2-25v13Folien, p.20

Umgangssprachlich bedeutet "Optimieren" nicht, "optimal zu werden", sondern lediglich, sich **schrittweise einem bekannten Ziel zu nähern** und zu "verbessern".

Vorgehensweise zur Optimierung (iterativer Prozess)

Wiederhole die folgenden Schritte, solange die geforderte Programmeffizienz noch nicht erreicht ist:

1. **Analyse:**
 - Analysiere, **welche Programmteile zur unzureichenden Effizienz beitragen**. Hierfür werden oft Profiling-Tools eingesetzt, die Engpässe identifizieren.
2. **Auswahl:**
 - Wähle den gefundenen Programmteil mit dem **größten Optimierungspotenzial** aus (der sogenannte "Hot Spot" oder "Flaschenhals"). Konzentriere dich auf die Bereiche, die den größten Einfluss auf die Gesamtperformance haben.
3. **Optimierung:**

- Optimiere diesen Programmteil durch Anwendung einer **sinnvollen Maßnahme** (z.B. Algorithmuswechsel, Datenstrukturänderung, Refactoring).

Problem der Optimierung

- **Schnelle Ausschöpfung des Potenzials:** Ein Großteil des Optimierungspotenzials ist oft **bald ausgeschöpft**.
- **Aufwendige weitere Schritte:** Weitere Optimierungsschritte werden **zunehmend aufwendiger**, bringen aber **verhältnismäßig wenig** zusätzlichen Gewinn. Es tritt ein Punkt auf, an dem der Aufwand für weitere Optimierungen den Nutzen übersteigt (Gesetz des abnehmenden Grenzertrags).