

# 1. Einführung

Quelle: ep2-01\_Einführung.pdf

Beinhaltet: Einführung

## Klassifizierung und Modularisierung

### 1. Beispielcode

```
public class TestFactorial {
    public static void main(String[] args) {
        System.out.println(Factorial.fact(3));
    }
}

public class Factorial {
    public static long fact(int n) {
        return n < 2 ? (long)n : n * fact(n - 1);
    }
}
```



### 2. Erklärung des Codes

#### TestFactorial.java

- **Zweck:** Startpunkt der Programmausführung.
- In der `main`-Methode wird die Methode `fact(3)` aus der Klasse `Factorial` aufgerufen.

#### Factorial.java

- **Methode:** `public static long fact(int n)`
  - Berechnet rekursiv die Fakultät von `n`.
  - Bedingung:
    - Wenn  $n < 2$ , wird `n` direkt zurückgegeben.
    - Sonst wird  $n \cdot fact(n - 1)$  rekursiv berechnet.

### 3. Verweise auf Klassen und Methoden

#### Verweise auf andere Klassen

- Der Aufruf `Factorial факт(3)` in `TestFactorial` ist ein **Verweis auf eine Methode in einer anderen Klasse**.
- Aber auch `System` und `String[]` sind Verweise auf Methoden von anderen Klassen
  - Klassenname als Präfix:** Da `fact` statisch ist, wird sie über den Klassennamen aufgerufen: `Factorial. факт(...)`.

## Verweis auf Methode innerhalb derselben Klasse

- Innerhalb der Klasse `Factorial` ruft `fact` sich selbst auf mit `fact(n - 1)`.
  - Dies ist ein **impliziter Verweis auf eine Methode innerhalb derselben Klasse**.
  - Da es sich um dieselbe Klasse handelt, ist kein Präfix notwendig.

## 4. Fazit

- Interner Methodenaufruf:** `fact(n - 1)` → innerhalb von `Factorial`, daher kein Klassennamen nötig.
- Externer Methodenaufruf:** `Factorial. факт(3)` → von `TestFactorial` aus, daher mit Klassennamen.



### Klassifizierung und Modularisierung

```
public class TestFactorial {
    public static void main(String[] args) {
        System.out.println(Factorial. факт(3));
    }
}

public class Factorial {
    public static long fact(int n) {           Verweis auf Methode innerhalb der Klasse
        return n < 2 ? (long)n : n * fact(n - 1);  entspricht Factorial. факт(n - 1)
    }
}
```

Verweise auf andere Klassen



## Verwendungsbeispiele abstrakter Datentypen

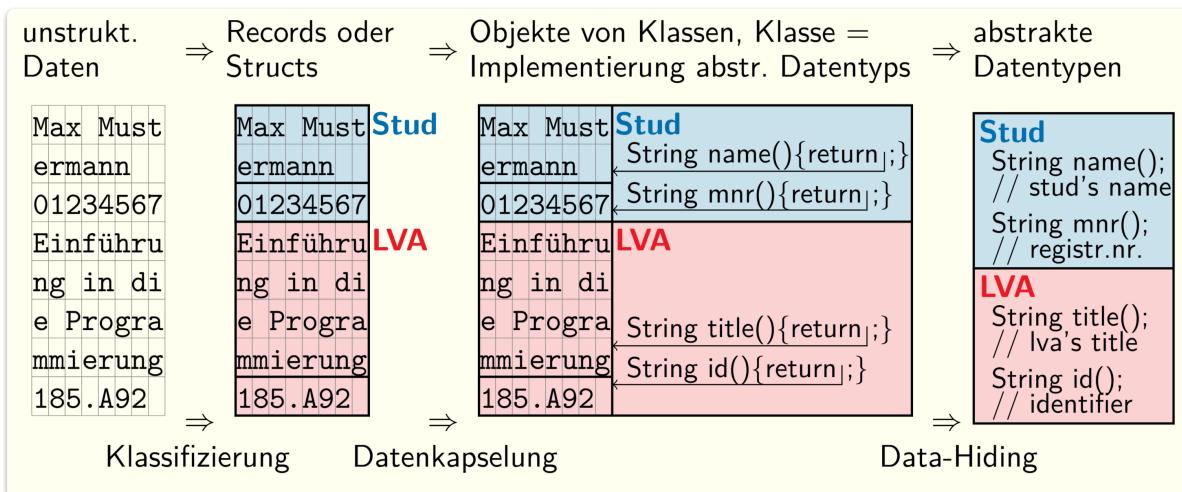
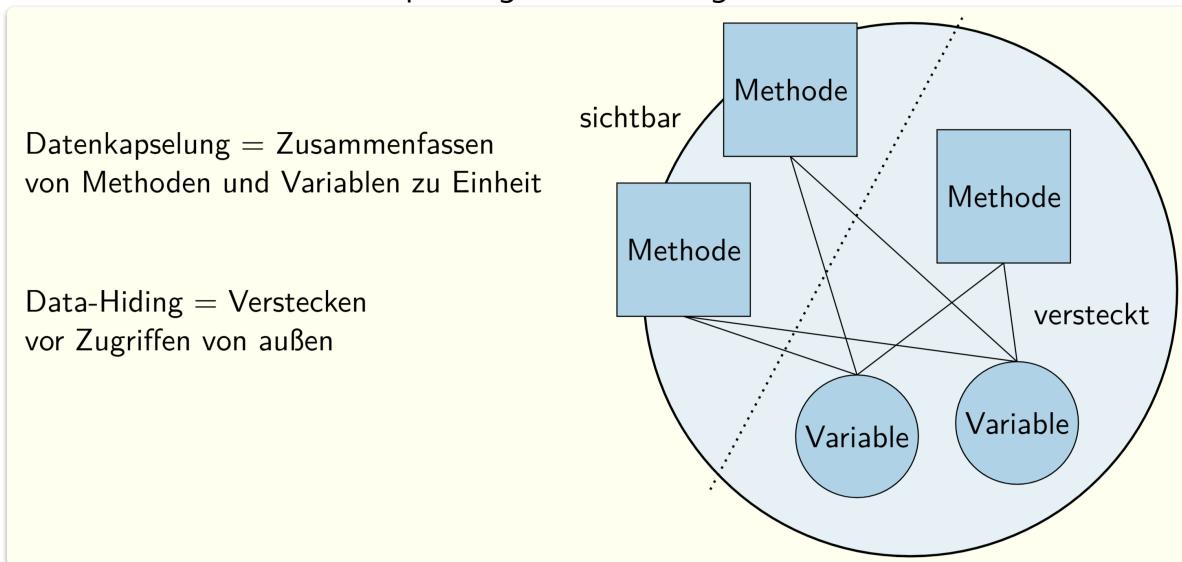
```
private static void printFifthChar(String s) {  
    if (s != null && s.length() > 4) {  
        System.out.print(s.charAt(4));  
    }  
}  
  
private static void echo() {  
    Scanner scanner = new Scanner(System.in);  
    while (scanner.hasNextLine()) {  
        System.out.println(scanner.nextLine());  
    }  
}
```

**String**  
**Scanner**

Referenz auf Objekt

# Datenabstraktion

Datenabstraktion = Datenkapselung + Data-Hiding



# Abstrakte Datentypen

## Entwerfen eines abstrakten Datentyps:

```
*****
class BoxedText: Rectangular text within border lines.
public methods:
    void newDimensions(int width, int height);
    void setLine(int index, String txt);
    void print();
    String toString();
*****
private static void testBoxedText() {
    BoxedText t = new BoxedText();
    t.newDimensions(10, 3);
    t.setLine(1, "Das ist ein Text");
    t.print();
}
```

BoxedText als abstrakter Inhalt einer Objektreferenz

## Implementierung des abstrakten Datentyps

```
public class BoxedText {
    // Deklarationen von Objektvariablen
    private int textWidth = 0;
    private int textHeight = 0;
    private char[][] text = new char[0][];

    // Objektmethode
    public void newDimensions(int width, int height) {
        // Zugriffe auf Objektvariablen
        textWidth = width;
        textHeight = height;
        text = new char[height][width];

        for (char[] line : text) {
            fill(line, 0);
        }
    }

    // Private Objektmethode
    private void fill(char[] line, int i) {
        for (; i < textWidth; i++) {
            line[i] = ' ';
        }
    }
}
```

```
}
```

Um so eine Klasse zu erstellen siehe hier [2. Data Hiding und co > Klassen erstellen](#)

---

# Arten von Methoden

## 1. Objektmethode

- **Definition:** Ohne den Modifier `static`
- **Zugehörigkeit:** An ein bestimmtes Objekt gebunden
- **Zugriff auf Objektvariablen:** Direkter Zugriff möglich
- **Nutzung:**
  - Häufig in abstrakten Datentypen
  - Ermöglicht enge Zusammenarbeit zwischen Methoden und Objektzustand
- **Beispiele:**
  - Aufruf auf Objekt: `x.newDimensions(10, 3)` (*wenn x vom Typ `BoxedText`*)
  - Aufruf im selben Objektkontext: `fill(line, 0)` (*vereinfachter Aufruf ohne Objektpräfix*)

## 2. Klassenmethode

- **Definition:** Mit dem Modifier `static`
  - **Zugehörigkeit:** An die Klasse, **nicht** an ein Objekt
  - **Zugriff auf Objektvariablen:** Kein direkter Zugriff möglich
  - **Nutzung:**
    - Nicht für abstrakte Datentypen im engen Sinn geeignet
    - Sollte eher sparsam eingesetzt werden, vor allem wenn Objektbezug notwendig ist
  - **Beispiele:**
    - Aufruf über Klasse: `Factorial.factor(n - 1)`
    - Aufruf innerhalb derselben Klasse: `factor(n - 1)` (*vereinfachter Aufruf ohne Klassennamen*)
-

# Arten von Variablen

## 1. Parameter

- **Deklaration:** In der Parameterliste einer Methode
- **Sichtbarkeit:** Nur innerhalb des Methodenrumpfs
- **Lebensdauer:** Nur während der Ausführung der Methode
- **Beispiel:** `void methode(int x) { ... }`

## 2. Lokale Variablen

- **Deklaration:** Innerhalb des Methodenrumpfs
- **Sichtbarkeit:** Nur in dem Block, in dem sie deklariert wurden
- **Lebensdauer:** Nur während der Ausführung des Blocks
- **Initialisierung:** Muss manuell erfolgen – keine automatische Initialisierung

## 3. Objektvariablen (Instanzvariablen)

- **Deklaration:** In der Klasse, **ohne static**
- **Zugehörigkeit:** Zu einem bestimmten Objekt
- **Existenz:** Einmal pro erzeugtem Objekt
- **Zugriff:** In Objektmethoden derselben Klasse direkt möglich
- **Initialisierung:** Automatisch mit Standardwerten (`0`, `0.0`, `null`)

## 4. Klassenvariablen (statische Variablen)

- **Deklaration:** In der Klasse, **mit static**
- **Zugehörigkeit:** Zur Klasse, **nicht** zu einem Objekt
- **Existenz:** Nur **einmal** im gesamten Programm
- **Zugriff:** In Objekt- und Klassenmethoden zugreifbar
- **Initialisierung:** Automatisch mit Standardwerten (`0`, `0.0`, `null`)
- **Hinweis:** Verwendung möglichst vermeiden (insbesondere bei Objektbezug)

## 2. Data Hiding und co

Quelle: ep2-02\_Data-Hiding\_Objekterzeugung\_Datensatz.pdf

Beinhaltet: Data-Hiding, Objekterzeugung, Datensatz

---

## Data Hiding

### 1. Außen- und Innensicht

- **Außensicht:**
  - Definition des **abstrakten Datentyps (ADT)** aus Anwendersicht
  - Sichtbar ist nur, was für die Verwendung **notwendig** ist
  - Fokus auf **Benutzerfreundlichkeit** und **Schnittstelle**
- **Innensicht:**
  - Interne **Implementierung** des ADT
  - **Alle Details sichtbar** (Variablen, Methoden, Algorithmen etc.)
  - Fokus auf **Effizienz** und **Wartbarkeit**
- **Unterschiedliche Sichtbarkeiten → Data-Hiding**
  - Ziel: **Trennung** von Schnittstelle (Außensicht) und Implementierung (Innensicht)
  - Bessere **Modularität** und **Wartbarkeit**

### 2. Data-Hiding

- **Zugriffsmodifikatoren:**
  - `public`:
    - Gehört zur Außen- **und** Innensicht
    - Überall zugreifbar
  - `private`:
    - Gehört nur zur **Innensicht**
    - Nur innerhalb der **eigenen Klasse** zugreifbar
- **Änderung der Innensicht bei gleichbleibender Außensicht:**
  - Anwendungen bleiben **unverändert**
- **Änderung der Außensicht:**
  - Anwendungen müssen ggf. **angepasst** werden
- **Praxisempfehlung:**
  - Möglichst viele Methoden und Variablen als `private` deklarieren
  - Dadurch **bessere Wartbarkeit**, auch wenn es anfangs als Nachteil empfunden werden kann

### 3. Sichtbarkeit auf Klassenebene

- Zugriff zwischen Objekten derselben Klasse:

- Auch `private` Mitglieder eines anderen Objekts sind zugreifbar
- Beispiel:

java

KopierenBearbeiten

```
public class A { private int x; public int add(A a) { return x + a.x; //  
Zugriff auf privates x von a erlaubt } }
```

- Erklärung: `a` ist vom Typ der Klasse `A`, daher ist Zugriff auf dessen private Felder innerhalb von `A` erlaubt

- Fazit:

- Außen-/Innensicht → objektbezogen
- `public` / `private` → klassenbezogen
- Dies kann zu **scheinbar widersprüchlichem Verhalten** führen, ist aber durch das Klassenmodell gerechtfertigt

# Klassen erstellen

## Sichtbarkeit von Klassen: `public` Modifier

- `public class` :
  - Klasse ist allgemein verwendbar
  - Normalfall: genau eine `public` Klasse pro Datei
  - Klassename = Dateiname (bis auf Dateiendung)
- **Ohne `public` vor `class`:**
  - Klasse ist nur im selben Ordner (Package) sichtbar
  - Dient als Hilfsklasse
- **Ausnahme:**
  - Bei Data-Hiding kann von der Standardregel abgewichen werden

## Objekterzeugung mit `new`

- Ausführung von `new A()` :
  - Speicherbereich für Objektvariablen und Identität wird reserviert
  - Speicher wird mit Null-Werten vorinitialisiert
  - Ein Konstruktor der Klasse `A` wird zur Initialisierung ausgeführt
  - Eine Referenz auf den Speicherbereich (das Objekt) wird zurückgegeben
- **Identität von Objekten:**
  - Wenn  $x == y$  wahr, dann referenzieren  $x$  und  $y$  dasselbe Objekt

# Konstruktoren

- Konstruktor ist ähnlich wie Methode, hat:
  - **gleichen Namen wie die Klasse**
  - **keinen Ergebnistyp**
  - **Parameter** zur Initialisierung der Objektvariablen
- **Beispiel:**

```
public class Point {
    private int x, y;

    public Point(int initX, int initY) {
        x = initX;
        y = initY;
    }
}
```

- Wird durch `new Point(3, 5)` aufgerufen
- Initialisiert Objekt mit  $x = 3, y = 5$

## Überladene Konstruktoren und Default-Konstruktor

- Mehrere Konstruktoren mit unterschiedlicher Parameterliste möglich (Überladung)

```
public class Point {
    private int x, y;

    public Point(int initX, int initY) {
        x = initX;
        y = initY;
    }

    public Point() {}
}
```



- `new Point()` entspricht `new Point(0, 0)`
- **Default-Konstruktor:**
  - Wird automatisch erzeugt, wenn **kein anderer Konstruktor** vorhanden ist

## Konstruktoraufruf mit `this(...)`

- Konstruktor kann andere Konstruktoren derselben Klasse aufrufen

```
public class Point {
    private int x, y;

    public Point(int initX, int initY) {
        x = initX;
        y = initY;
    }

    public Point() {
        this(1, 1);
    }

    public Point(Point p) {
        this(p.x, p.y);
    }
}
```



- Konstruktor-Aufruf mit `this(...)`:

- Nur als erste Anweisung im Konstruktor erlaubt

- Beispiele:

- new Point(3, 5)
- new Point()
- new Point(new Point())

## Selbstreferenz mit this

- this referenziert das aktuelle Objekt, in dem sich der Code gerade befindet
- Wird oft zur Unterscheidung von Parameter- und Attributnamen genutzt

```
public class Point {
    private int x, y;

    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }

    public Point(Point p) {
        this(p.x, p.y);
    }

    public Point copy() {
        return new Point(this);
    }
}
```



- this ist eine Pseudovariable:
  - Nur lesbar, nicht überschreibbar
- In this(...) handelt es sich um einen Konstruktoraufruf, nicht um eine Selbstreferenz

# Datenstruktur != abstrakter Datentyp

## Datenstruktur

- Beschreibt, wie Daten zusammenhängen, wie sie auffindbar sind und wie Operationen darauf zugreifen
- Offene Aspekte:
  - verwendete Programmiersprache
  - konkrete Datentypen
  - mögliche Größenbeschränkungen

## Abstrakter Datentyp (ADT)

- **Außensicht:** wie Objekte verwendet werden können
- Blendet **Implementierungsdetails** aus
- Lässt offen:
  - konkrete Algorithmen
  - Datenstrukturen
  - sonstige interne Details

## Implementierung eines abstrakten Datentyps

- Umfasst:
  - konkrete Algorithmen
  - verwendete Datenstrukturen
- Klärt offene Punkte aus Sicht von ADT und Datenstruktur
- **Übergang zwischen ADT und Datenstruktur ist fließend**

## Datensatz als Datenstruktur

- Sehr einfache Datenstruktur
- Besteht aus **zusammengehörenden Variablen**, die bei Bedarf gelesen oder geschrieben werden
- Beispiel:

```
Student:
  regNumber
  name
  mail
```



- In dieser Form **relativ uninteressant**

## Datensatz als abstrakter Datentyp

- Abstraktionsebene höher als einfache Datenstruktur
  - Fragestellungen zur Abstraktion:
    - Wie sind Werte der Variablen eingeschränkt?
    - Welche Variablen sind wann lesbar, wann schreibbar?
    - Bleiben Variablen hinter der Abstraktion sichtbar?
    - Welche Abstraktion ermöglicht eine einfache Verwendbarkeit?
-

# Getter und Setter

## Datensatz mit Gettern und Settern

- **Getter und Setter möglichst vermeiden**
  - Grund: lassen interne Variablenstruktur nach außen durchscheinen
  - Verstoßen gegen Prinzip der Datenkapselung
- **Beispiel:**

```
public class Student {
    private final int regNumber;
    private String name;

    public Student(int regNumber, String name) {
        this.regNumber = regNumber;
        setName(name);
    }

    public int regNumber() {
        return regNumber;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }
}
```



- **regNumber:**
  - `final`, d.h. **nach Initialisierung nicht mehr veränderbar**
  - Aber: auch **nicht-finale** Variablen können nach außen **nur lesbar** gemacht werden
- Einschränkungen durch Typen auch in der **Außensicht** sichtbar
- Getter/Setter **übertragen wesentliche Funktionalität nach außen**
  - führen zu Verlust von Kontrolle und Abstraktion

## Umgang mit zusammenhängenden Daten

- Zugriff erfolgt **indirekt** über Methoden, nicht direkt über Variablen
- Beispiel: Suche in einem Array von `Student`-Objekten

```
private static Student find(Student[] studs, int reg) {  
    for (Student stud : studs) {  
        if (stud.regNumber() == reg) {  
            return stud;  
        }  
    }  
    return new Student(reg, "Max Mustermann");  
}
```



- Vergleich über **indirekten Zugriff** (hier `stud.regNumber()`)
  - **Rückgabe eines vollständigen Datensatzes** (mit allen Variablen)
-

# Funktionalität angereicherter Datensatz

## Student -Klasse ohne Getter und Setter

- **Wesentliche Funktionalität in die Klasse verschoben:**
  - Statt Getter und Setter gibt es nun Methoden, die innerhalb der Klasse verwendet werden.
  - Vermeidet das Offenlegen der internen Datenstruktur nach außen und wahrt die Kapselung.
- **Beispiel:**

```
public class Student {
    private final int regNumber;
    private String name;
    private String mail;

    public Student(int regNumber, String name) {
        this.regNumber = regNumber;
        this.name = name;
        mail = "e" + regNumber + "@student.tuwien.ac.at";
    }

    public void showPersonalData() {
        // Anzeige der persönlichen Daten
    }

    public void editPersonalData() {
        // Bearbeiten der persönlichen Daten
    }

    public void mail(String head, String text) {
        // Funktion zum Senden einer E-Mail
    }
}
```



- **Vorteile:**
  - **Keine Getter und Setter notwendig**, da alle Zugriffe und Operationen innerhalb der Klasse bleiben.
  - **Verborgene Datenstruktur**: Die Variablen `regNumber`, `name` und `mail` sind nur innerhalb der Klasse zugänglich.
- **Funktionalitäten:**
  - `showPersonalData()` : zeigt die persönlichen Daten an.
  - `editPersonalData()` : ermöglicht das Bearbeiten der persönlichen Daten.

- `mail()` : sendet eine E-Mail mit dem angegebenen Betreff und Text.

## Prinzip der Datenkapselung

- **Getter und Setter vermeiden:** Durch das Verschieben der wesentlichen Funktionalität innerhalb der Klasse werden externe Zugriffe auf die Variablen vermieden, was die **Datenkapselung** fördert.
  - Wenn **alle zugreifenden Methoden innerhalb der Klasse** sind, ist der Zugriff auf die Variablen kontrolliert und sicher.
-

# Idee hinter objektorientierter Programmierung

- **Schwerpunkt auf Funktionalität**, nicht auf dem Datensatz:
  - Der **Datensatz** bleibt **hinter der Funktionalität** gänzlich abstrakt.
  - Fokus liegt darauf, wie **Operationen und Funktionen** auf den Daten ausgeführt werden, nicht auf der reinen Speicherung der Daten.
- **Software-Objekt simuliert ein „reales Objekt“:**
  - Ein Software-Objekt muss **nicht nur konkrete, materielle** Objekte abbilden, sondern auch **immaterielle** Objekte oder Konzepte.
  - Es geht darum, **nur die in der Software relevanten Eigenschaften** eines „realen Objekts“ zu simulieren.
- **Modellierte Objekte** sind:
  - Häufig **mit Funktionalität angereicherte Datensätze**:
    - Das bedeutet, die Daten sind nicht isoliert, sondern sie haben eine **funktionale Bedeutung**, die es ermöglicht, Operationen oder Methoden darauf anzuwenden.

# Zusätzliche Informationen aus dem Skriptum:

## 1. Datenabstraktion und Data-Hiding

- **Datenabstraktion** ist die kombinierte Anwendung von **Datenkapselung** und **Data-Hiding**. Data-Hiding bezeichnet das „Verstecken“ der Implementierungsdetails eines abstrakten Datentyps vor externen Zugriffen. Es ermöglicht, zwischen zwei verschiedenen Sichten zu unterscheiden:
  - **Außensicht**: Diese ist für den Anwender sichtbar und definiert, wie der abstrakte Datentyp verwendet wird (z.B. über Methoden wie `newDimension`, `setLine`, `print`).
  - **Innensicht**: Diese beschreibt die Implementierung des abstrakten Datentyps und enthält die erforderlichen Variablen und Methoden, die zur internen Funktionsweise notwendig sind. Sie ist nicht direkt zugänglich.

## 2. Private vs. Public Deklaration

- Bei der **Deklaration von Variablen und Methoden** eines abstrakten Datentyps ist es wichtig, zu unterscheiden, welche Elemente **public** (von außen zugänglich) und welche **private** (nur innerhalb der Klasse zugänglich) sind.
  - **private** Variablen und Methoden sind Implementierungsdetails und sollen vor externem Zugriff geschützt werden.
  - **public** Variablen und Methoden sind für den externen Gebrauch erforderlich, z.B. um die Funktionalität des abstrakten Datentyps zu ermöglichen.

## 3. Konstruktoren und Objekterzeugung

- **Konstruktoren** dienen der Initialisierung von Objekten. Ein Objekt wird mit dem **new-Operator** erzeugt, und der Konstruktor wird sofort nach der Erstellung des Objekts ausgeführt, um die Objektvariablen zu initialisieren.
  - Wenn eine Klasse keinen expliziten Konstruktor hat, fügt der Compiler automatisch einen **Default-Konstruktor** hinzu, der keine Argumente benötigt.
  - Konstruktoren sind nicht **static** und können unterschiedliche Parameterlisten haben. Sie können auch überladen werden, d.h., eine Klasse kann mehrere Konstruktoren mit unterschiedlichen Parametern haben.

## 4. Verwendung von `this` in Konstruktoren

- In Konstruktoren wird `this` verwendet, um zwischen den Parametern und den Objektvariablen zu unterscheiden, wenn sie denselben Namen haben.
  - `this.x` greift auf die Objektvariable zu, während `x` auf den Konstruktorparameter verweist. Dies ist wichtig, um Klarheit zu schaffen, insbesondere wenn Parameter und Objektvariablen gleich benannt sind.

## 5. Getter- und Setter-Methoden

- **Getter- und Setter-Methoden** werden verwendet, um auf **private** Variablen zuzugreifen. Sie ermöglichen es, den Wert einer privaten Variable zu lesen (Getter) oder zu ändern (Setter).
  - Diese Methoden bieten eine Möglichkeit, private Variablen von außen zugänglich zu machen, sind jedoch aus Wartungsgründen nicht ideal, da sie zusätzliche Komplexität einführen können.
  - Ein **Getter** gibt den Wert einer Variablen zurück, während ein **Setter** den Wert einer Variablen setzt. Auch wenn diese Methoden praktisch sind, kann es oft besser sein, **public** Variablen ganz zu vermeiden, wenn möglich.

## 6. Datenstruktur vs. Abstrakter Datentyp

- Eine **Datenstruktur** beschreibt, wie Daten **repräsentiert** und miteinander in Beziehung gesetzt werden. Sie definiert die **Verknüpfung** der Daten und wie diese durch **Operationen** zugänglich gemacht werden.
- Ein **abstrakter Datentyp** (ADT) ist eine spezifizierte Sammlung von Operationen, die auf einer Datenstruktur arbeiten. Der ADT abstrahiert die Implementierung und konzentriert sich auf das Verhalten der Operationen. **Beispiel:**
  - **Datenstruktur:** Ein Array ist eine einfache lineare Datenstruktur, bei der die Elemente direkt hintereinander im Speicher liegen und über Indizes zugänglich sind.
  - **Abstrakter Datentyp:** Ein Array-ADT könnte die Operationen `set()`, `get()` und `size()` umfassen, ohne sich darum zu kümmern, wie die Daten im Array gespeichert werden oder welche Programmiersprache verwendet wird.

## 7. Datensätze

- Ein **Datensatz** ist eine einfache Datenstruktur, die eine feste Menge von Variablen (oft als **Objektvariablen** bezeichnet) zusammenführt. Diese Variablen sind über Getter- und Setter-Methoden zugänglich.
  - **Beispiel:** Ein `Student`-Objekt könnte Felder wie `regNumber`, `name` und `mail` enthalten, die durch Getter- und Setter-Methoden abgerufen oder geändert werden.

In der Praxis enthalten Datensätze häufig nur Variablen und die Operationen zum Zugreifen und Bearbeiten dieser Variablen. Der Datensatz selbst enthält keine komplexen Methoden, die auf den Daten arbeiten.

## 8. Übergang von einfachen zu funktionalen Datensätzen

- **Einfache Datensätze:** Einfache Datensätze, wie im **Listing 2.10** gezeigt, bestehen aus den grundlegenden Daten und Getter-Setter-Methoden, die jedoch keine tiefere Funktionalität bieten.
- **Funktionale Datensätze:** Ein fortgeschrittenerer Datensatz, wie im **Listing 2.12** gezeigt, enthält nicht nur Datenfelder, sondern auch **spezifische Methoden** zur Bearbeitung der Daten, wie z.B. `showPersonalData()`, `editPersonalData()`, oder `mail()`. Diese

Methoden bieten mehr Funktionalität und reduzieren den Aufwand außerhalb der Klasse, um mit den Daten zu arbeiten.

## 9. Datenabstraktion

- **Datenabstraktion** bezeichnet den Prozess, bei dem die Details der **Implementierung** einer Datenstruktur verborgen und nur die wesentlichen Merkmale für die Nutzung bereitgestellt werden.
  - Ein **abstrakter Datentyp** abstrahiert von der konkreten Implementierung der Datenstruktur und stellt nur die Operationen zur Verfügung, die mit der Struktur durchgeführt werden können.
  - **Datenstruktur** und **abstrakter Datentyp** können als zwei Perspektiven auf dasselbe Objekt betrachtet werden. Die Datenstruktur beschreibt die internen Beziehungen und die Repräsentation der Daten, während der abstrakte Datentyp sich auf die Nutzung und die verfügbaren Operationen konzentriert.

## 10. Getter- und Setter-Methoden vs. funktionale Methoden

- **Getter- und Setter-Methoden** bieten direkten Zugriff auf private Datenfelder eines Datensatzes. Diese Methoden sind jedoch häufig nicht die beste Wahl, weil sie:
  - Wenig oder keine spezifische Funktionalität bieten.
  - Oft als **anwendungsneutral** betrachtet werden.

Stattdessen ist es ratsam, **anwendungsbezogene Methoden** wie `showPersonalData()` oder `editPersonalData()` zu verwenden, die spezifische Funktionalitäten bieten und das Verhalten der Klasse in einem größeren Kontext kapseln.

## 11. Veränderbarkeit und Einschränkungen bei Datentypen

- Ein wichtiger Punkt bei der **Abstraktion von Datentypen** ist, dass die Außensicht die Verwendung von **bestimmten Datentypen** festlegt. Zum Beispiel wird im Fall des `Student`-Objekts im **Listing 2.10** der `regNumber` als `int` deklariert. Diese Entscheidung kann jedoch später zu Problemen führen, wenn beispielsweise die Matrikelnummer auf acht Stellen erweitert werden soll.
  - Die **Datenstruktur** abstrahiert von solchen Implementierungsdetails, während die Außensicht des ADT den Wertebereich und die genaue Implementierung explizit macht.

## 12. Anwendung von Datensätzen in Programmen

- In realen Anwendungen sind Datensätze oft nicht isoliert, sondern werden innerhalb von **Arrays** oder **anderen Datenstrukturen** verwendet. Beispiel: Ein Array von `Student`-Objekten könnte verwendet werden, um eine Liste von Studierenden zu verwalten, wie im **Listing 2.11** dargestellt.
- **Verwendung in Methoden:** Datensätze können auch als Argumente in Methoden übergeben werden. In einem Beispiel wie **Listing 2.11** gibt eine Methode `find()` ein

Student -Objekt zurück, das aus einem Array von Studierenden gesucht wird.

### 13. Abstraktion durch Algorithmen vs. Abstraktion durch Datentypen

- **Abstraktion durch Algorithmen** bezieht sich darauf, wie bestimmte Operationen auf den Daten ausgeführt werden, unabhängig von der zugrunde liegenden Datenstruktur.
- **Abstraktion durch abstrakte Datentypen** konzentriert sich auf das Verhalten der Datenstruktur als Ganzes und auf die Operationen, die darauf ausgeführt werden können, ohne sich mit den Details der Implementierung zu befassen.

### 3. Lineare Zugriffe und co

Quelle: ep2-03\_lineare-Zugriffe\_Arbeitsblatt-Zugriffe\_Implementierungsdetails.pdf

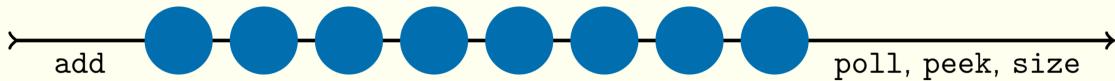
Beinhaltet: Lineare Zugriffe, Assoziative Zugriffe, Implementierungsdetails

---

## Lineare Zugriffe auf Daten

- Prinzip des linearen Zugriffs:
  - Ein Element wird hineingegeben, die Daten werden in eine Sammlung aufgenommen.
  - Nächstes Element wird herausgeholt, und so weiter.
  - Kein aufwendiges Adressieren oder Indexieren erforderlich.
- Eigenschaften:
  - Einfache Verwendung:
    - Daten können sequentiell durchlaufen und bearbeitet werden, ohne komplexe Operationen oder Berechnungen.
  - Häufig keine Größenfestlegung nötig:
    - Bei vielen linearen Datenstrukturen muss die Größe nicht im Voraus definiert werden, was eine flexible Handhabung der Daten erlaubt.

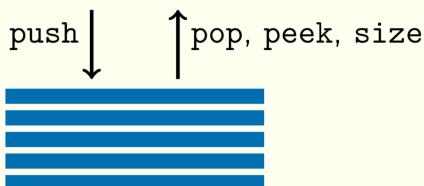
## Queue



FIFO-Verhalten (first-in, first-out)

- add** Einfügen sollte in der Regel funktionieren (mitwachsende Datensammlung)
- poll** Lesen und Entfernen, Ergebnis `null` wenn keine Eintrag vorhanden
- peek** Lesen ohne Entfernen, Ergebnis `null` wenn keine Eintrag vorhanden
- size** Anzahl der Einträge, nötig wenn `null` als Eintrag erlaubt

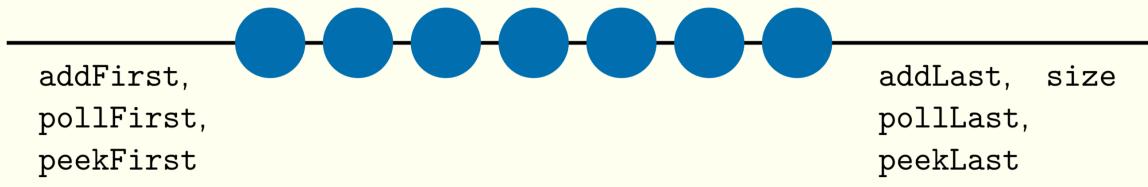
## Stack



LIFO-Verhalten (last-in, first-out)

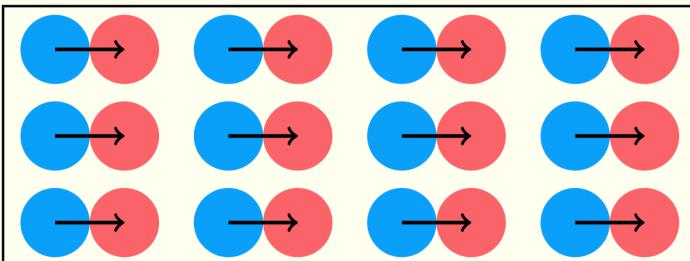
- push** Einfügen sollte in der Regel funktionieren (mitwachsende Datensammlung)
- pop** Lesen und Entfernen, Ergebnis `null` wenn keine Eintrag vorhanden
- peek** Lesen ohne Entfernen, Ergebnis `null` wenn keine Eintrag vorhanden
- size** Anzahl der Einträge, nötig wenn `null` als Eintrag erlaubt

## Double-Ended-Queue



symmetrische Zugriffe auf beiden Seiten → Verhalten variabel (FIFO, LIFO, ...)

# Assoziative Datenstruktur



**put(key, value)**  
**get(key)**  
**remove(key)**  
**containsKey(key)**  
**containsValue(value)**  
**size()**

beliebig viele unterschiedliche **Schlüssel** mit je einem **Wert** assoziiert

**Eintrag** ist Kombination aus Schlüssel und assoziiertem Wert

wahlfreier Zugriff auf Werte über Schlüssel (beliebig oft zugreifbar)

Datensammlung wächst meist mit der Anzahl der Einträge

## Methoden einer AD

|                         |  |
|-------------------------|--|
| <b>put(k, v)</b>        | assoziiert Schlüssel k mit neuem Wert v,<br>gibt alten Wert zurück (oder null wenn Eintrag neu)                  |
| <b>get(k)</b>           | gibt mit Schlüssel k assoziierten Wert zurück<br>(oder null wenn k mit keinem Wert assoziiert ist)               |
| <b>remove(k)</b>        | entfernt Eintrag mit Schlüssel k (falls er existiert),<br>gibt vorher mit k assoziierten Wert zurück (oder null) |
| <b>containsKey(k)</b>   | true wenn ein Eintrag mit Schlüssel k existiert  |
| <b>containsValue(v)</b> | true wenn es einen Eintrag mit Wert v gibt   |
| <b>size()</b>           | Anzahl der Einträge  |

--> [1. Einführung > Arten von Methoden](#)

## Assoziative Datenstruktur vs. Array

### Array

- **Indexierung:**
  - Der **Index** im Array ist eine **ganze Zahl** in einem **fortlaufenden Indexbereich**.
- **Größe:**
  - Die **Größe** des Arrays muss bereits **beim Anlegen bekannt sein**.
- **Indexbereich:**
  - Der **Indexbereich** des Arrays bleibt **über die gesamte Lebensdauer des Arrays**

konstant.

- **Zugriffseffizienz:**

- Zugriff auf das Array ist sehr effizient, da die Elemente durch ihre Position im Speicher direkt adressierbar sind.

## Assoziative Datenstruktur

- **Indexierung:**

- Der **Schlüssel** in einer assoziativen Datenstruktur kann **beliebigen Typ** haben, nicht zwingend eine fortlaufende Zahl.
- **Kein fortlaufender Bereich** nötig, der Schlüssel kann beliebig gewählt werden (z. B. Strings, Objekte).

- **Größe:**

- Eine assoziative Datenstruktur **kann nach Bedarf mitwachsen** und muss nicht vorab eine feste Größe haben.

- **Flexibilität:**

- Einträge in einer assoziativen Datenstruktur (Schlüssel + Wert) sind **hinzufügbar und entferbar**. Sie können dynamisch angepasst werden.

- **Zugriffseffizienz:**

- Der Zugriff auf eine **assoziativen Datenstruktur** ist im Allgemeinen weniger effizient als der Zugriff auf ein Array, da intern oft eine Hash-Tabelle oder ein ähnliches Verfahren verwendet wird, um die Zuordnung zwischen Schlüssel und Wert zu finden.

## Vorteile der assoziativen Datenstruktur

- **Einfache Handhabung:**

- Der Umgang mit assoziativen Datenstrukturen ist oft einfacher und flexibler, da man nicht mit festen Indexen oder Größen arbeiten muss.

- **Dynamische Anpassung:**

- Sie wachsen und schrumpfen nach Bedarf und bieten mehr **Freiheit** bei der Handhabung der Daten.

# Ziele bei Implementierung

## Korrektheit:

Implementierung entspricht vorgegebener Außensicht

## Einfachheit ( $\approx$ Wartbarkeit):

- so wenige Fallunterscheidungen und Schleifen wie möglich
- mehrfahe Vorkommen gleicher und ähnlicher Programmtexte vermeiden
- nur leicht nachvollziehbare Annahmen treffen
- schwer verständliche Textteile vermeiden

## Effizienz:

- effiziente Programmerstellung (wichtiger Kostenfaktor)
- ausreichend effizienter Programmablauf

# Wrapper

## SQueue als Wrapper auf DEQueue

- **Definition:**
  - SQueue ist ein **Wrapper** auf die DEQueue -Klasse.
  - Der Begriff "Wrapper" bezeichnet eine Klasse, die eine andere Klasse **einbettet** und deren Funktionalität weiterverwendet.
- **Delegierung:**
  - Die Methoden von SQueue delegieren ihre Aufgaben an das Objekt der Klasse DEQueue .
  - **Delegieren** bedeutet, dass die Methodenaufrufe von SQueue an die entsprechenden Methoden des eingebetteten DEQueue -Objekts weitergegeben werden.
- **Beispiel:**

```
public class SQueue {
    private final DEQueue q = new DEQueue(); // DEQueue wird als internes Objekt
    verwendet

    public void add(String e) {
        q.addLast(e); // Delegierung an DEQueue
    }

    public String poll() {
        return q.pollFirst(); // Delegierung an DEQueue
    }
    // Weitere Methoden könnten folgen
}
```



## Funktionsweise des Wrappers

- **Wrapper erzeugt neue Außensicht:**
  - Der Wrapper ( SQueue ) bietet eine **neue Sicht** auf die bestehende Klasse DEQueue .
  - Dies kann z.B. durch:
    - **Andere Namen oder Parameterreihenfolgen** von Methoden,
    - **Vorgegebene Werte für bestimmte Parameter** (z. B. Standardwerte),
    - **Weglassen von Methoden** oder das **Hinzufügen neuer Methoden** geschehen.
  - In diesem Beispiel delegiert SQueue die Aufgaben der Methode add an addLast von DEQueue und die Methode poll an pollFirst .

## Vorteile des Wrappers

- **Anpassung der API:**

- Mit einem Wrapper kann man die API einer bestehenden Klasse anpassen, ohne die ursprüngliche Klasse zu verändern.
- Der Wrapper bietet eine **vereinfachte oder angepasste Schnittstelle** für bestimmte Anwendungsfälle.
- **Verstecken von Details:**
  - Details der inneren Implementierung können vor dem Benutzer verborgen werden. Beispielsweise könnte `SQueue` zusätzliche Logik oder Fehlerbehandlung einführen, ohne dass der Benutzer die Details der `DEQueue` kennt.

## Zusammenfassung

- **Delegation** bedeutet, dass ein Objekt die Verantwortung für die Ausführung bestimmter Aufgaben an ein anderes Objekt weitergibt.
  - Ein **Wrapper** bietet eine angepasste Außensicht auf eine bestehende Klasse und verändert die Art und Weise, wie Methoden aufgerufen oder verwendet werden, ohne die ursprüngliche Klasse zu ändern.
-

# Index als Modulo-Wert

## Index als Modulo-Wert

```

public class DEQueue {
    private int mask = (1 << 3) - 1; Zweierpotenz vorteilhaft,  
ermöglicht Modulo durch Maskieren
    private String[] es = new String[mask + 1];
    private int head, tail; Maske = alle gültigen Bits des Index  
Einträge von es[head] bis es[tail-1] gültig,  
durch Modulo auch bei tail < head,  
eine Grenze inklusiv, eine exklusiv
    public void addFirst(String e) {
        es[head = (head - 1) & mask] = e;
        ... zuerst Index dekrementieren (modulo mask + 1), dann zugreifen
    }
    public void addLast(String e) {
        es[tail] = e; zuerst zugreifen, dann Index inkrementieren
        tail = (tail + 1) & mask;
        ...
    }
}

```

## Schnelle Bitshift Wiederholung:

### Bitwise Shift Operatoren in Java

Bitwise Shift Operatoren manipulieren die einzelnen Bits einer Zahl. In Java gibt es drei Haupttypen:

## Arten von Bitwise Shift Operatoren

- **Left Shift (`<<`)**: Verschiebt die Bits nach links. Freie Stellen auf der rechten Seite werden mit Nullen aufgefüllt.
- **Signed Right Shift (`>>`)**: Verschiebt die Bits nach rechts. Das Vorzeichenbit (das höchstwertigste Bit) bleibt erhalten und wird in die frei werdenden Stellen auf der linken Seite kopiert. Dies erhält das Vorzeichen der ursprünglichen Zahl.
- **Unsigned Right Shift (`>>>`)**: Verschiebt die Bits nach rechts. Freie Stellen auf der linken Seite werden immer mit Nullen aufgefüllt, unabhängig vom Vorzeichenbit.

## Funktionsweise

### Left Shift (`<<`)

Ein Left Shift um `n` Stellen (`x << n`) ist äquivalent zur Multiplikation der Zahl `x` mit  $2^n$ .

### Beispiel:

```

int a = 5;      // Binär: 00000101
int b = a << 2; // Binär: 00010100 (entspricht 20)

```

```
System.out.println(b); // Ausgabe: 20
```



## Signed Right Shift ( `>>` )

Ein Signed Right Shift um `n` Stellen (`x >> n`) ist äquivalent zur Division der Zahl `x` durch  $2^n$ , wobei das Ergebnis auf die nächste ganze Zahl abgerundet wird und das Vorzeichen erhalten bleibt.

**Beispiel:**

```
int a = 20;      // Binär: 00010100
int b = a >> 2; // Binär: 00000101 (entspricht 5)
System.out.println(b); // Ausgabe: 5

int c = -20;     // Binär (als 32-Bit Zweierkomplement): ...11101100
int d = c >> 2; // Binär: ...11111011 (entspricht -5)
System.out.println(d); // Ausgabe: -5
```

## Unsigned Right Shift ( `>>>` )

Ein Unsigned Right Shift um `n` Stellen (`x >>> n`) verschiebt die Bits nach rechts und füllt die linken Stellen immer mit Nullen. Das Vorzeichen geht dabei verloren.

**Beispiel:**

```
int a = -20;      // Binär (als 32-Bit Zweierkomplement): ...11101100
int b = a >>> 2; // Binär: 00...00111011 (ein großer positiver Wert)
System.out.println(b);
```



## Implementierung von Bitshifts ohne Operatoren

Bitshifts können mithilfe von logischen Operationen und Division durch 2 simuliert werden.

### Left Shift ( `<< n` ) ohne Operator

Ein Left Shift um eine Stelle ist äquivalent zur Multiplikation mit 2. Ein Left Shift um `n` Stellen kann durch `n` Multiplikationen mit 2 erreicht werden.

```
int x = 5;
int n = 3;
int result = x;
for (int i = 0; i < n; i++) {
    result = result * 2;
```

```
}
```

```
System.out.println(result); // Ausgabe: 40 (5 << 3)
```

## Right Shift (`>> n` oder `>>> n`) ohne Operator

Ein Right Shift um eine Stelle ist äquivalent zur Integer-Division durch 2. Ein Right Shift um `n` Stellen kann durch `n` Integer-Divisionen durch 2 erreicht werden.

Für den **Signed Right Shift (`>>`)** funktioniert die einfache Integer-Division für positive Zahlen. Bei negativen Zahlen ist zu beachten, dass die Integer-Division in Java immer in Richtung Null rundet. Um das Verhalten des Signed Right Shifts exakt nachzubilden, insbesondere bei negativen Zahlen, ist es komplexer und erfordert zusätzliche Logik, um das Vorzeichenbit korrekt zu behandeln.

Für den **Unsigned Right Shift (`>>>`)** ist die einfache Integer-Division nicht ausreichend, da sie das Vorzeichenbit beachtet. Um einen Unsigned Right Shift ohne den Operator zu implementieren, müsste man die Zahl als vorzeichenlose Größe behandeln (was in Java mit primitiven `int` nicht direkt möglich ist) und die Bits manuell verschieben und Nullen einfügen. Dies wäre deutlich aufwändiger und würde typischerweise bitweise logische Operationen erfordern, die den Shift-Operator im Grunde nachbilden.

**Vereinfachte Simulation des Right Shifts (ähnlich `>>` für positive Zahlen):**

```
int x = 20;
int n = 2;
int result = x;
for (int i = 0; i < n; i++) {
    result = result / 2; // Integer-Division
}
System.out.println(result); // Ausgabe: 5 (20 >> 2)
```



**Wichtiger Hinweis:** Die exakte Nachbildung des `>>>`-Operators ohne den Operator ist in Java mit reinen arithmetischen Operationen nicht trivial und würde den Einsatz bitweiser logischer Operationen erfordern, was den Sinn der Übung, den Operator zu vermeiden, untergräbt. Die gezeigte Division simuliert eher das Verhalten des `>>`-Operators für positive Zahlen.

## Anwendungsfälle

Bitweise Shift Operatoren sind nützlich für:

- Effiziente Multiplikation und Division mit Zweierpotenzen.
- Manipulation einzelner Bits in Datenstrukturen (z.B. Flags, Bitssets).
- Low-Level-Programmierung und Hardware-Interaktion.
- Kompakte Speicherung von Zuständen.

## Auf null setzen

```
public class DEQueue {
    ...
    public String pollFirst() {
        String result = es[head];
        head = (head + 1) & mask;
        if (tail != head) {
            es[head] = null;
            head = (head + 1) & mask;
        }
        return result;
    }
    public String peekFirst() {
        return es[head];
    }
}
```

Ergebnis merken  
ursprünglichen Eintrag auf null setzen  
gut für Programmhygiene, Speicherbereinigung,  
spart Fallunterscheidungen  
head == tail wenn ganz leer oder voll,  
voll nicht möglich weil vorher Array vergrößert,  
null als gültiger Eintrag möglich  
es[head] == null wenn leer, keine Fallunterscheidung nötig

Kurz gesagt, das explizite Setzen von Referenzen auf `null`, wenn sie nicht mehr benötigt werden, dient zwei Hauptzwecken:

1. **Vereinfachung der Logik:** Durch das Setzen auf `null` kann man sich in vielen Fällen Sonderbehandlungen für nicht gefundene oder ungültige Einträge sparen. Eine `null`-Referenz ist ein klar definierter Zustand.
2. **Verbesserung der Speicherverwaltung (indirekt):** Obwohl die Java Garbage Collection nicht sofort bei `null`-Zuweisung aktiv wird, signalisiert es dem Garbage Collector, dass das referenzierte Objekt potenziell freigegeben werden kann, da keine aktiven Referenzen mehr darauf existieren (außer möglicherweise im Debugger). Dies kann indirekt zur effizienteren Speichernutzung beitragen, insbesondere bei Objekten, die in langlebigen Datenstrukturen gehalten werden.

Es ist wichtig zu verstehen, dass das Setzen auf `null` nicht *immer* notwendig ist, aber in bestimmten Situationen (wie in dem genannten Beispiel mit Array-Einträgen) die Code-Wartbarkeit und potenziell die Speichernutzung verbessern kann.

## Arrays vergrößern

```

public void addFirst(String e) {
    es[head = (head - 1) & mask] = e;
    if (tail == head) { doubleCapacity(); } —sofort verdoppeln wenn voll,
} —Zweierpotenz beibehalten

private void doubleCapacity() {
    mask = (mask << 1) | 1; —ein Bit mehr
    String[] newes = new String[mask + 1]; —neues Array
    int i = 0, j = 0; —gilt noch immer: tail == head
    while (i < head) { newes[j++] = es[i++]; }
    j = head += es.length; —ab hier: tail != head, Lücke nicht gefüllt
    while (i < es.length) { newes[j++] = es[i++]; }
    es = newes;
}

```

Zusammenfassend lässt sich sagen, dass der Code eine dynamisch vergrößerbare Datenstruktur implementiert, die das Hinzufügen von Elementen am Anfang optimiert, indem sie einen Ringpuffer und eine Verdopplungsstrategie für die Kapazität nutzt.

Das Array wird vergrößert, wenn beim Einfügen eines neuen Elements der `tail` den `head` erreicht.

## Vergleich wenn null als Wert zählt

```

public class SimpleAssoc {
    private int top; —jeder Array-Index i mit i < top ist gültig
    private String[] ks = new String[8]; —zwei getrennte Arrays für Schlüssel und Werte
    private String[] vs = new String[8]; —kleine Zweierpotenz
    private int find(String s, String[] a) { —Schlüssel oder Werte
        int i = 0;
        while (i < top && !(s==null ? s==a[i] : s.equals(a[i]))) —Vergleich der Identität für null, Gleichheit sonst
            i++; —wegen komplexem Vergleich zahlt sich Methode aus
        return i;
    }
    ...
}

```

# Eintragen und Entfernen

## Eintragen – Fallunterscheidungen vermeiden

```

public String put(String k, String v) {
    int i = find(k, ks);           i == top → kein Eintrag vorhanden → einfügen
    if (i == top && ++top == ks.length) {
        String[] nks = new String[top << 1];
        String[] nvs = new String[top << 1];
        for (int j = 0; j < i; j++) {
            nks[j] = ks[j];   nvs[j] = vs[j];
        }
        ks = nks;   vs = nvs;
    }
    ks[i] = k;                   Schlüssel neu eingetragen, egal ob nötig oder nicht
    String old = vs[i];
    vs[i] = v;                  Wert muss sowieso immer neu eingetragen werden
    return old;
}

```

Arrays verdoppeln wenn voll,  
einfaches Umkopieren reicht  
weil alle Einträge gültig sind

## Entfernen mit Verschieben eines Eintrags

```

public String remove(String k) {
    int i = find(k, ks);
    String old = vs[i];
    if (i < top) {             i == top wenn Schlüssel nicht gefunden
        ks[i] = ks[--top];
        ks[top] = null;
        vs[i] = vs[top];
        vs[top] = null;
    }
    return old;
}

```

Anzahl gültiger Einträge verringern (--top),  
Einträge von neuem Index top nach Index i  
(unnötig wenn i == top für neues top),  
Einträge an Index top auf null setzen

# Design-Entscheidungen sparen Programmtext

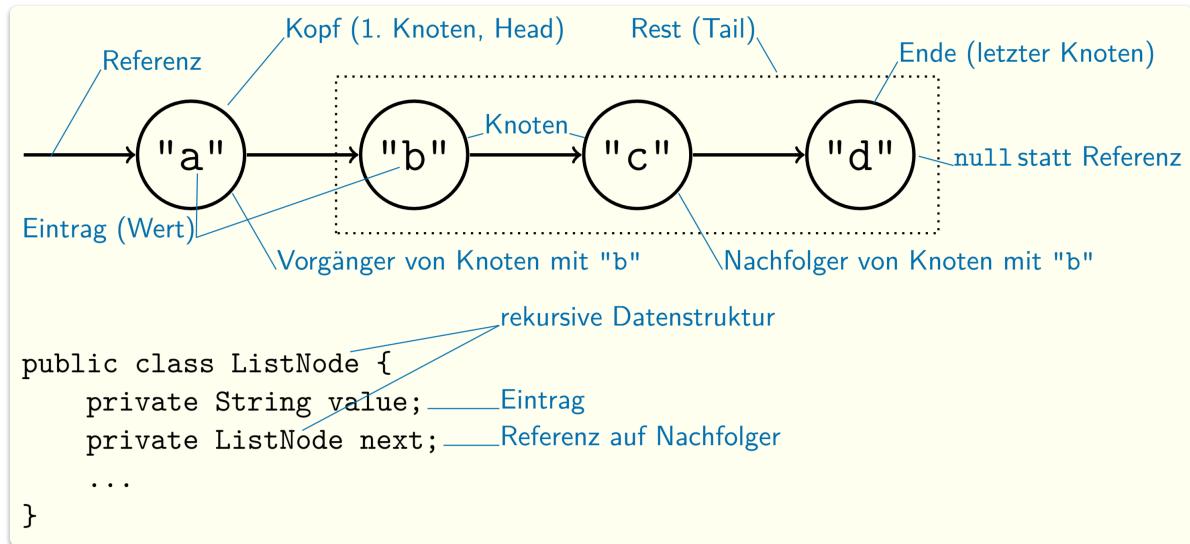
```
public String get(String k) {  
    return vs[find(k, ks)]; keine Fallunterscheidung: gefunden/nicht gefunden  
}  
public boolean containsKey(String k) {  
    return find(k, ks) < top; Gültigkeit eines Eintrags einfach feststellbar  
}  
public boolean containsValue(String v) {  
    return find(v, vs) < top; gleiche Methode für Suche nach Schlüssel und Wert  
}  
public int size() {  
    return top; Anzahl der Einträge entspricht Index in top  
}
```

## 4. Lineare Liste und co

Quelle: ep2-04\_lineare-Liste\_Binäre-Bäume.pdf

Beinhaltet: Lineare Liste, Linearer Baum

### Lineare Liste

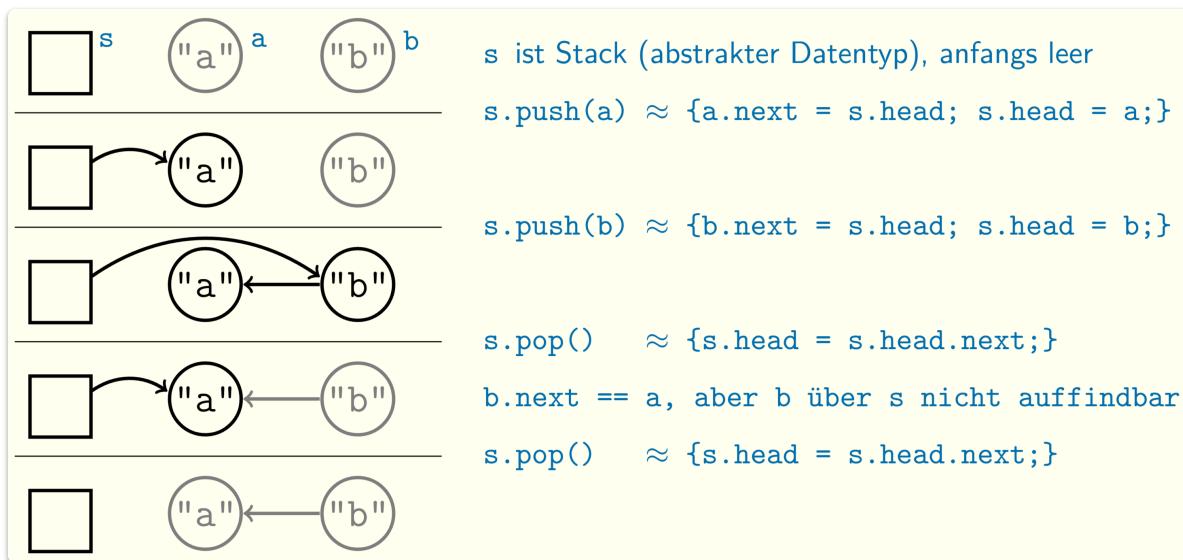


das ist aber falsch weil das mit Tail ist nur der letzte gemeint und nicht der ganze Kasten

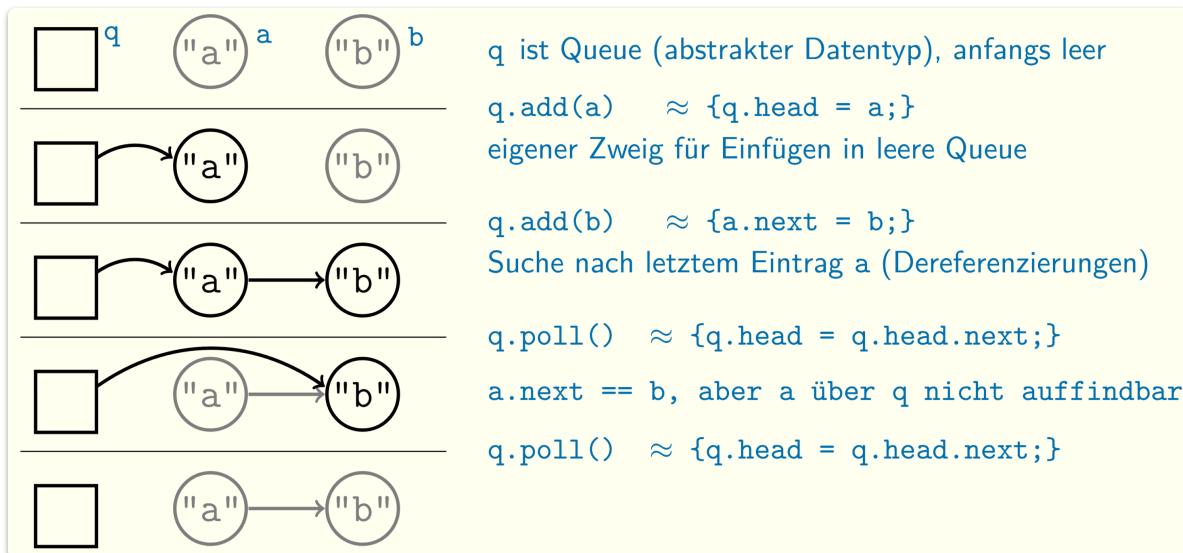
ist ein 1. Einführung > Abstrakte Datentypen

# Arten von Listen

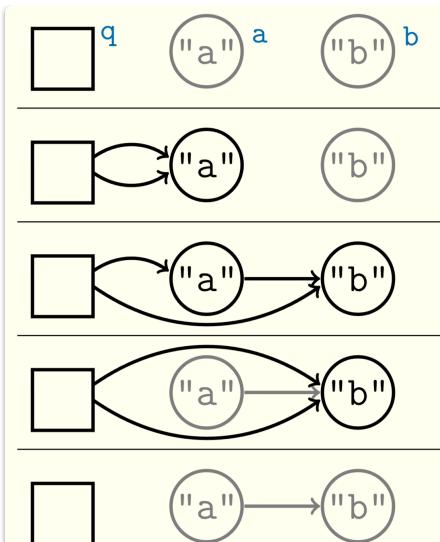
## Lineare Liste als Stack



## Lineare Liste als Queue



## Lineare Liste als Queue (mit last)



q ist Queue (abstrakter Datentyp), anfangs leer

q.add(a)  $\approx \{q.\text{head} = a; q.\text{last} = a;\}$

eigener Zweig für Einfügen in leere Queue

q.add(b)  $\approx \{q.\text{last} = q.\text{last}.\text{next} = b;\}$

keine Suche nach letztem Eintrag

q.poll()  $\approx \{q.\text{head} = q.\text{head}.\text{next};\}$

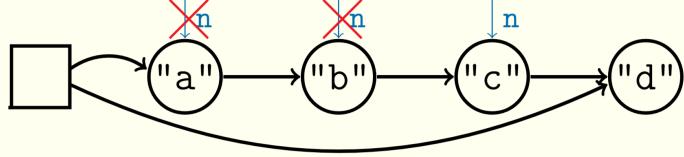
a.next == b, aber a über q nicht auffindbar

q.poll()  $\approx \{q.\text{head} = q.\text{last} = \text{null};\}$

eigener Zweig für Entfernen des letzten Knotens

# Methoden auf Listen

## Traversieren einer Liste (Suche)



```

ListNode n = head;           n könnte bereits zu Beginn gleich null sein
while (n != null && ...) {   Suchbedingung
    ...
    n = n.next();           was für jeden besuchten Knoten gemacht werden soll
}                           dereferenzieren (weiterschalten)

```

## Einfügen in eine Liste



### Einfügen in eine Liste

```

if (... /* insert at begin */) {
    head = new ListNode(v, head);
    if (last == null) {
        last = head;
    }
} else { /* not at begin */
    ListNode n = ...; /* insert after n */
    n.setNext(new ListNode(v, n.next()));
    if (last == n) {
        last = n.next();
    }
}

```

new ...  
 n  
 "v"  
 "a"  
 "b"

Sonderbehandlung für ersten Eintrag  
 Referenz auf Vorgänger  
 Zusatzaufwand für last

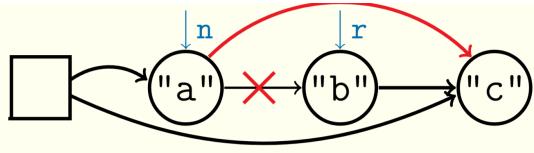
## Entfernen aus einer Liste

nur wenn zu entfernender Knoten existiert

```

if (... /* remove first node */) {
    head = head.next();
    if (head == null) {
        last = null;
    }
} else {
    ListNode n = ...; /* predecessor of r */
    ListNode r = n.next(); /* to be removed */
    n.setNext(r.next());
    if (last == r) {
        last = n;
    }
}

```



Sonderbehandlung für ersten Eintrag

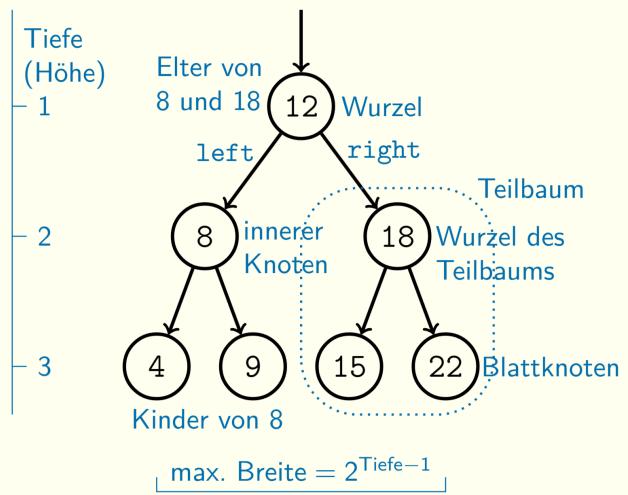
Referenz auf Vorgänger

Zusatzaufwand für last

# Linearer Baum

jeder Knoten hat bis zu 2 Kinder

```
public class TreeNode {
    private int value;
    private TreeNode left;
    private TreeNode right;
    ...
}
```



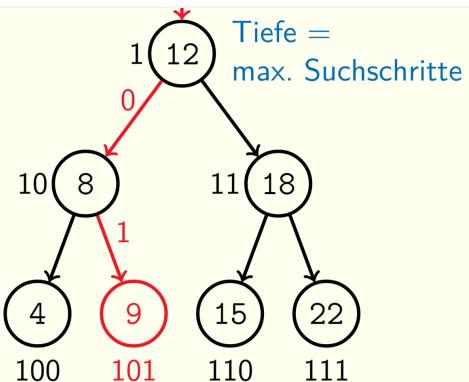
ist ein 1. Einführung > Abstrakte Datentypen

## Binärer Suchbaum

Binärer Baum, Einträge sortiert  
(z.B. links kleiner, rechts größer/gleich)

Analogie zu Binärzahlen – effiziente Suche

```
public class TreeNode {
    private int value;
    private TreeNode left, right;
    ...
    public boolean contains(int v) {
        return v == value ||
               (v < value ? left!=null && left.contains(v)
jeweils nur ein Zweig betrachtet : right!=null && right.contains(v));
    }
}
```

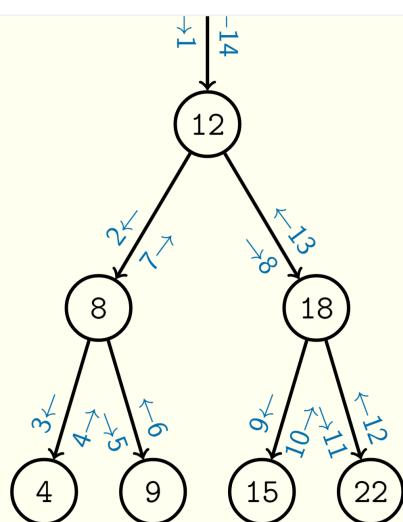


## Einfügen in binären Suchbaum

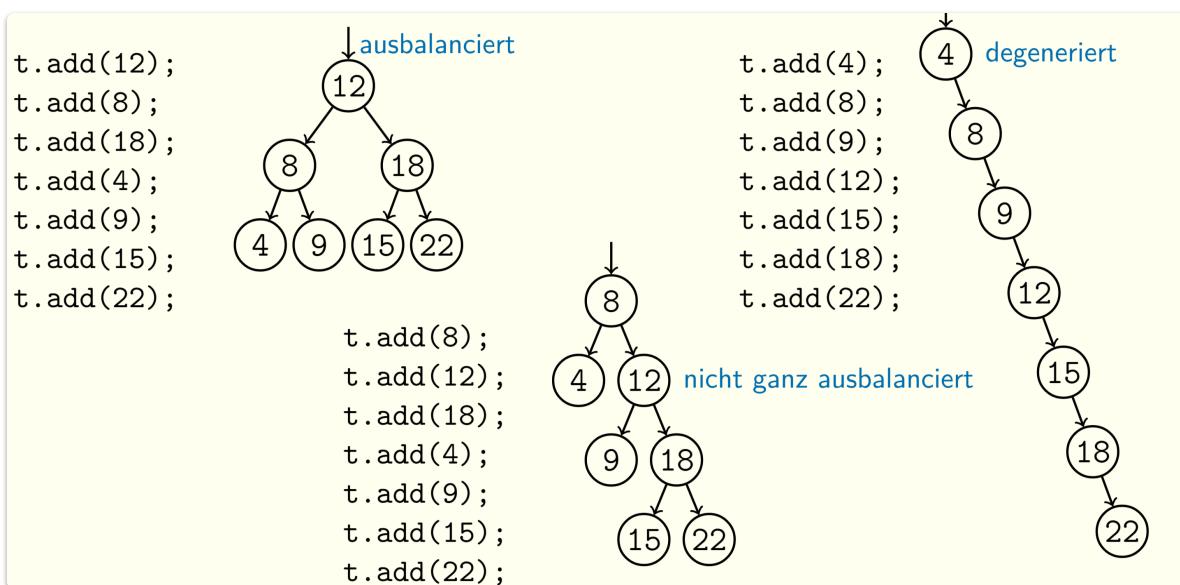
```
public class TreeNode {
    private int value;
    private TreeNode left, right;
    ...
    public void add(int v) {
        if (v < value) { // links oder rechts?
            if (left != null) { // wenn Teilbaum existiert, rekursiver Aufruf
                left.add(v); // sonst Platz zum Einfügen gefunden
            } else { left = new TreeNode(v); }
        } else if (v > value) { // nicht einfügen wenn schon vorhanden,
            if (right != null) { right.add(v); }
            else { right = new TreeNode(v); }
        }
    }
}
```

## Traversieren des gesamten Baums

```
public class TreeNode {
    private int value;
    private TreeNode left, right;
    ...
    public void visit() {
        ... in Reihenfolge ↓ (12,8,4,9,18,15,22)
        if (left != null) { left.visit(); }
        ... in sortierter Reihenfolge (4,8,9,12,15,18,22)
        ... Vertauschen kehrt Sortierreihenfolge um
        if (right != null) { right.visit(); }
        ... in Reihenfolge ↑ (4,9,8,15,22,18,12)
    }
}
```



## Baumstruktur von Reihenfolge des Einfügens abhängig



## Binärer Suchbaum benötigt Sortierbarkeit

```
public class TANode {
    private String key, value;
    private TANode left, right;
    ...
    private int compare(String k) { Vergleich nur mit key, nicht mit value
        if (k == null) {
            return key == null ? 0 : -1;
        }
        if (key == null) { x.compareTo(y) → -1 wenn x kleiner y
            return 1;
        }
        0 wenn x gleich y
        1 wenn x größer y
        return k.compareTo(key);
    }
}
```

# Rekursion

## Rekursion: Grundprinzipien

- **Fundierung:**
  - Rekursion muss immer eine **Abbruchbedingung** haben, um die unendliche Rekursion zu verhindern.
  - **Abbruchbedingung:** Ein Zustand, in dem die Rekursion nicht weiter fortgesetzt wird.  
Beispiel: Ein Basisfall in einer rekursiven Methode (z. B. bei der Berechnung einer Fakultät, wenn der Wert 0 oder 1 erreicht wird).
- **Fortschritt:**
  - In jedem rekursiven Aufruf müssen **andere Parameterwerte** verwendet werden, die dem Problem näher kommen.
  - Dies bedeutet, dass sich der Zustand in jedem Schritt der Rekursion verändern muss, damit der Basisfall irgendwann erreicht wird.

## Beispiel einer rekursiven Methode:

```
public int factorial(int n) {
    if (n <= 1) {
        return 1; // Fundierung (Abbruchbedingung)
    }
    return n * factorial(n - 1); // Fortschritt durch andere Parameterwerte
}
```



- **Fundierung:** Wenn `n <= 1` erreicht wird, hört die Rekursion auf.
- **Fortschritt:** Der Parameter `n` wird in jedem rekursiven Aufruf um 1 verringert, bis die Abbruchbedingung erreicht wird.

# Rekursive Datenstrukturen

## Rekursive Datenstrukturen: Fundierung und Fortschritt

- **Fundierung:**
  - Die **Fundierung** einer rekursiven Datenstruktur erfolgt durch **null** oder durch einen **spezifischen Knoten**, der als Basisfall dient.
  - In vielen Fällen ist `null` der Ausgangspunkt, der das Ende einer rekursiven Struktur markiert (z. B. das Ende einer verknüpften Liste).
- **Fortschritt:**
  - **Fortschritt** erfolgt durch **induktiven Aufbau**, das heißt durch **schrittweises Hinzufügen von Knoten und Referenzen**.

- Bei der Rekursion werden Elemente schrittweise hinzugefügt, bis der Basisfall erreicht wird (z. B. ein `null`-Knoten).

## Beispiel einer rekursiven Datenstruktur (z. B. verkettete Liste):

```
public class Node {
    int data;
    Node next;

    public Node(int data) {
        this.data = data;
        this.next = null; // Fundierung: der letzte Knoten verweist auf null
    }
}

public void printList(Node head) {
    if (head == null) {
        return; // Fundierung (Abbruchbedingung)
    }
    System.out.println(head.data);
    printList(head.next); // Fortschritt (rekursiver Aufruf auf das nächste
Element)
}
```



- Fundierung:** Der `null`-Wert am Ende der Liste stellt den Abbruch der Rekursion dar (kein weiteres Element mehr).
- Fortschritt:** Der rekursive Aufruf geht von einem Knoten zum nächsten, wobei `head.next` auf den nächsten Knoten verweist, bis der Endknoten erreicht wird.

## Rekursive Methoden und Datenstrukturen gekoppelt

- Verknüpfung von rekursiven Methoden und rekursiven Datenstrukturen:**
  - Oft sind **rekursive Methoden** und **rekursive Datenstrukturen** miteinander gekoppelt.
  - Die Abbruchbedingung in der Methode wird durch das Erreichen von `null` oder eines speziellen Knotens in der Datenstruktur ausgelöst.
  - Der **Fortschritt** erfolgt durch **Dereferenzierung** der Knoten bei rekursiven Aufrufen (z. B. das Weitergeben der Referenz zum nächsten Knoten in einer Liste).

## Beispiel für verknüpfte rekursive Methode und Datenstruktur:

- Bei der rekursiven Traversierung einer Liste wird der Fortschritt durch Dereferenzieren der `next`-Referenz im aktuellen Knoten erreicht.



# 5. Zyklen und co

**Quelle:** ep2-05\_Zyklen\_doppelte-Verkettung\_Abstraktionshierarchien\_Objektschnittstellen.pdf

**Beinhaltet:** Zyklen, doppelte Verkettung Abstraktionshierarchien Objektschnittstellen

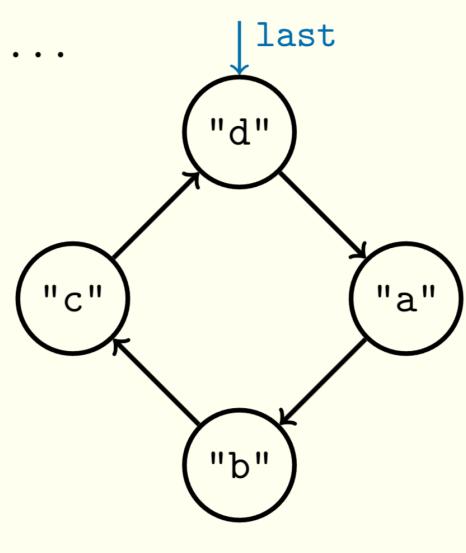
---

## Ringlisten

sind eine Erweiterung von [4. Lineare Liste und co > Lineare Liste](#) mit der Extra Eigenschaft, dass das letzte Element entweder auf nil oder auf das erste Element referenziert.

```
public class RingQueue { private ListNode last; ...
    public String poll() {
        if (last != null) {
            ListNode n = last.next();
            if (n == last) {
                last = null;
            } else {
                last.setNext(n.next());
            } return n.value();
        } else {
            return null;
        }
    }

    public void add(String v) {
        if (last == null) {
            last = new ListNode(v, null);
            last.setNext(last);
        } else {
            last.setNext(last = new ListNode(v, last.next())));
        }
    }
}
```



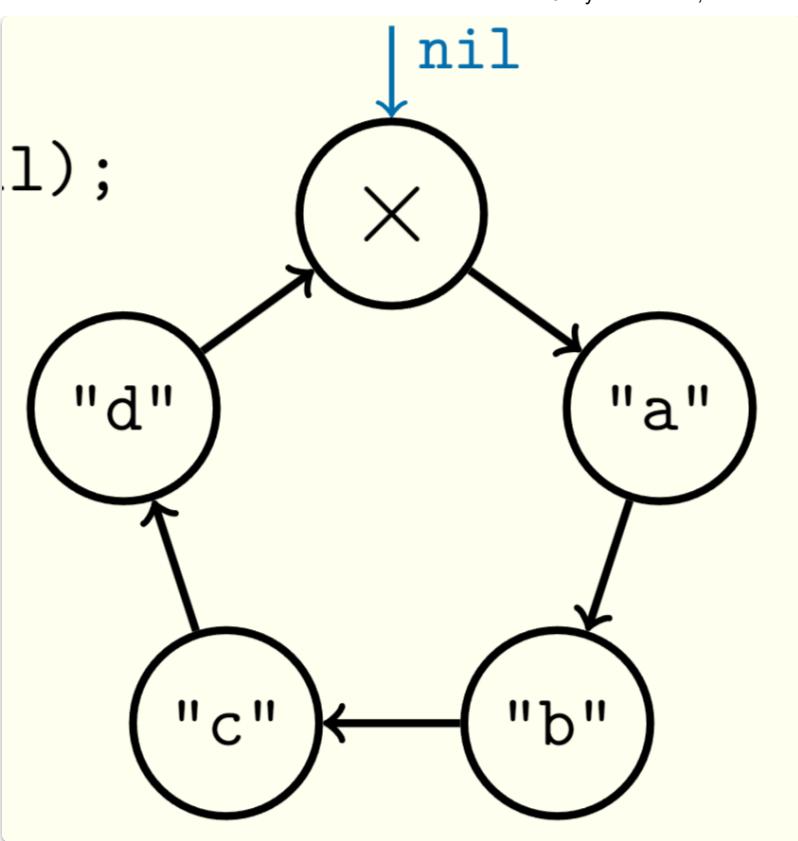
## Ringliste mit speziellem Knoten nil

```

public class RingQueue {
    private ListNode nil = new ListNode(null, null);
    public RingQueue() {
        nil.setNext(nil);
    } ...

    public String poll() {
        ListNode n = nil.next();
        nil.setNext(n.next());
        return n.value();
    }
    public void add(String v) {
        nil.setValue(v);
        nil.setNext(nil = new ListNode(null, nil.next()));
    }
}
    
```

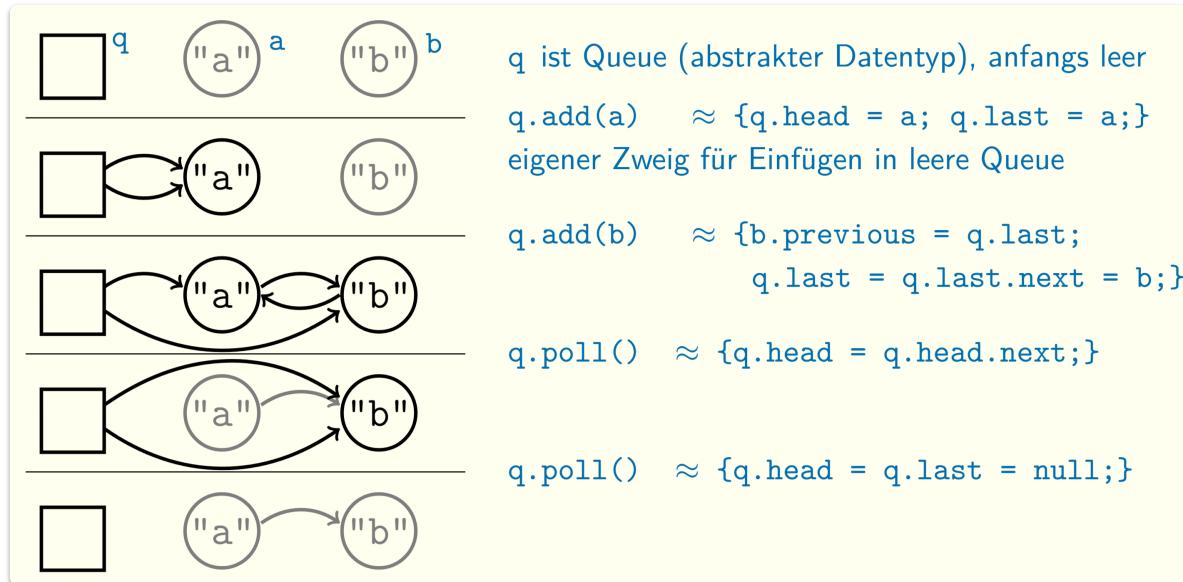




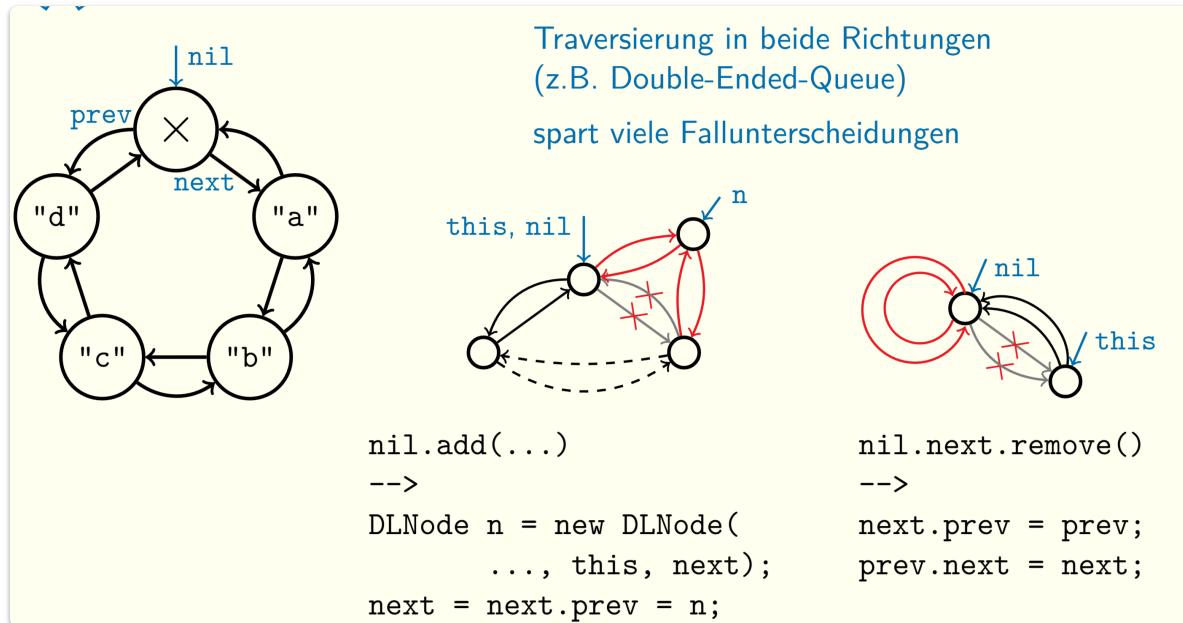
# Double linked lists

Sind eine weitere Erweiterung von [5. Zyklen und co > Ringlisten](#) oder [4. Lineare Liste und co > Lineare Liste](#), mit dem Unterschied, dass jedes Objekt seinen Nachfolger **aber auch den Vorgänger** referenziert.

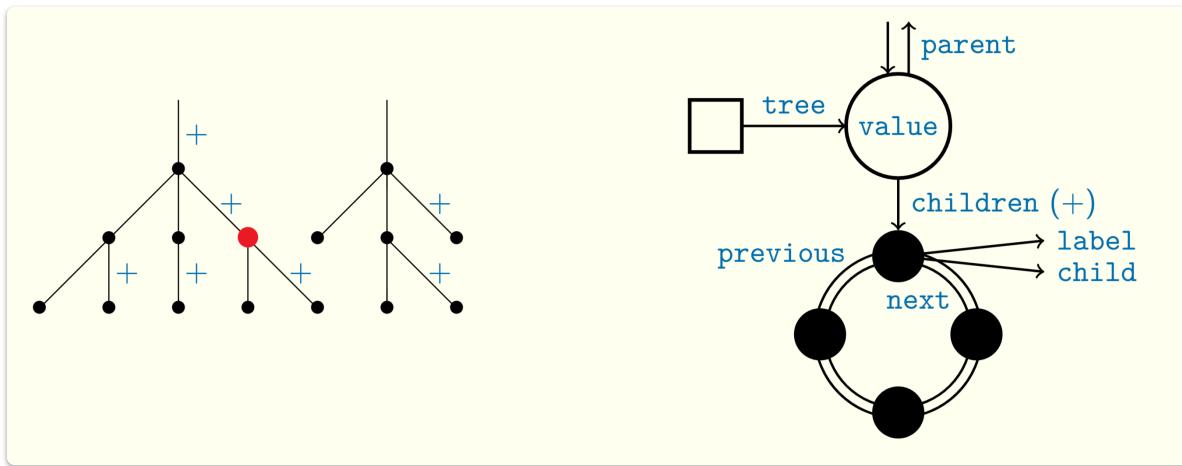
## ohne nil



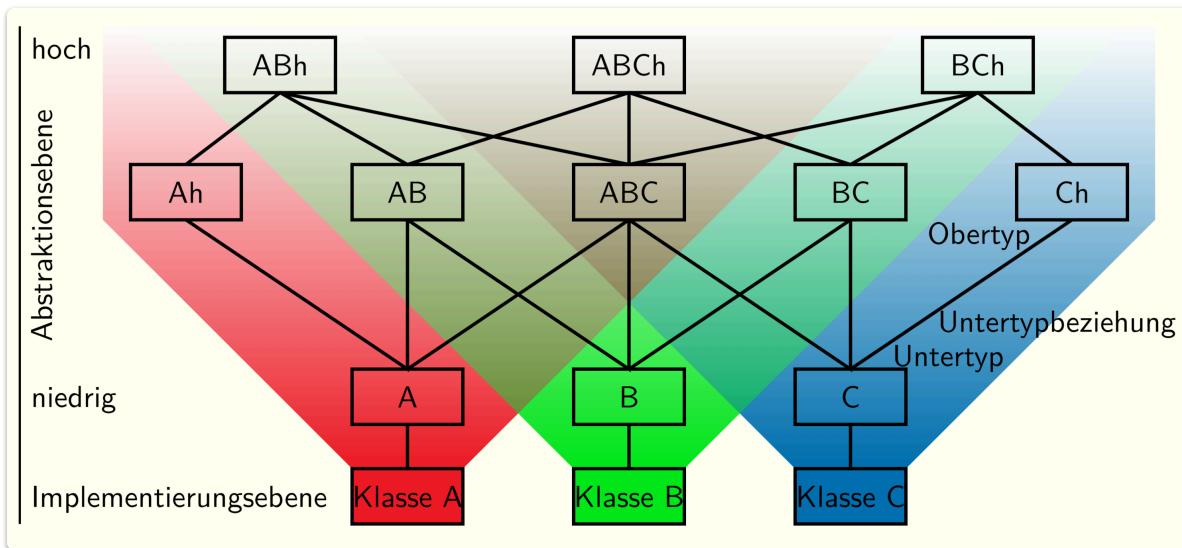
## Mit nil



# Navigieren durch verallgemeinerten Baum



## Abstraktionshierarchie – Beziehungen zw. abstrakten Datentypen



A, B und C sind *unterschiedlich* und haben nichts miteinander zu tun. Aber es kann sein, dass die Gemeinsamkeiten haben. Und all das was miteinander zu tun hat liegt in einer neuen Abstraktionsebene. Natürlich müssen da auch die Beschreibungen die da zusammengehören zusammenpassen.

# Java-Interface zur Beschreibung einer Objektschnittstelle

```
*****
BoxedText: Rectangular text within border lines.
public methods:
    void newDimensions(int width, int height);
    void setLine(int index, String txt);
    void print();
    String toString();
*****
public interface AbstrBoxed {
    void newDimensions(int width, int height);
    void setLine(int index, String txt);
    void print();
    String toString();
}
```



Hier stehen die Methodenköpfe drinnen. Die Methoden sind dann automatisch Public. Also Interface ist eine Sichtfläche von einzelnen Methoden nach außen.

## Java-Interface: Definition, Implementierung und Verwendung

```
public interface AbstrBoxed {
    void newDimensions(int width, int height);
    void setLine(int index, String txt);
    void print();
    String toString();
}
public class BoxedText implements AbstrBoxed {
... // defines at least all methods of AbstrBoxed
}
AbstrBoxed ab = new BoxedText();
ab.newDimensions(5, 3);
ab.setLine(1, "ABCDE");
ab.print();
```

## Abstraktion auf höherer Ebene

```
public interface SetBoxed {
    void newDimensions(int width, int height);
    void setLine(int index, String txt);
}
public class BoxedText implements SetBoxed {
... // all methods including print and toString
}
    BoxedText bt = new BoxedText(); bt.print(); // OK, BoxedText specifies print
    SetBoxed sb = bt; // OK, BoxedText is subtype of SetBoxed
    sb.newDimensions(5, 3); // OK, SetBoxed specifies newDimensions
    sb.setLine(1, "ABCDE"); // OK, SetBoxed specifies setLine
    sb.print(); // ERROR, SetBoxed does not specify print bt = sb; // ERROR,
SetBoxed is no subtype of BoxedText
```



Da es das print nicht in sb gibt, kann man es hier nicht verwenden. Wenn man aber ein neues Interface hinzufügt, welches print public macht, ist das möglich.

---

## Klasse implementiert mehrere Interfaces

```
public interface Print {
    void print();
}

public class BoxedText implements SetBoxed, Print {
    ... // defines at least all methods of SetBoxed and Print
}

BoxedText bt = new BoxedText();
SetBoxed sb = bt;           // OK
Print p = bt;              // OK
sb.newDimensions(5, 3);   // OK
sb.setLine(1, "ABCDE");  // OK
p.print();                 // OK
sb.print();                // ERROR
p.setLine(1, "ABCDE");   // ERROR
```

---

## Interface erweitert mehrere Interfaces

```
public interface SetBoxed {
    void newDimensions(int width, int height);
    void setLine(int index, String txt);
}

public interface Print {
    void print();
}

public interface AbstrBoxed extends SetBoxed, Print {
    void newDimensions(int width, int height);
    void print();
    String toString();
}

public class BoxedText implements AbstrBoxed {
    ...
}
```

---

## Nominales Subtyping

Untertypbeziehungen beruhen auf Typdefinitionen (implements und extends),  
Vorhandensein von Methoden nicht hinreichend (`Print`  $\neq$  `PrintBoxed`)

```
public interface Print {  
    void print(); // prints 'this'  
}  
public interface PrintBoxed extends Print {  
    void print(); // prints 'this' as text in a box  
}  
  
Print p = ...;  
PrintBoxed pb = ...;  
p = pb;           // OK  
pb = p;          // ERROR
```

# Vererbung auf Klassen

**Grundprinzip:** Jede Klasse in Java erbt von *genau einer* anderen Klasse.

**extends -Schlüsselwort:**

- Wird verwendet, um die Oberklasse (Superklasse oder Elternklasse) einer Klasse festzulegen.
- **Syntax:** `public class Unterklasse extends Oberklasse { ... }`
- **Implizite Oberklasse:** Wenn keine `extends`-Klausel angegeben wird, erbt die Klasse automatisch von der Klasse `Object`.

**Beispiel 1:**

```
public class BoxedText extends Object implements AbstrBoxed {
    // ...
}
```



- `BoxedText` erbt von `Object`.
- Die `extends Object`-Klausel ist hier explizit, aber **redundant**, da `Object` die Standard-Oberklasse ist.
- `BoxedText` implementiert zusätzlich das Interface `AbstrBoxed`.

**Verfügbarkeit von Methoden:**

- Alle `public`-Methoden der Oberklasse sind automatisch in der Unterklasse verfügbar (werden vererbt).

**Subtyping:**

- Eine Unterklasse ist ein **Subtyp** ihrer Oberklasse. Das bedeutet, ein Objekt der Unterklasse kann überall dort verwendet werden, wo ein Objekt der Oberklasse erwartet wird.

**Beispiel 2:**

```
public class BoxedTextReset extends BoxedText {
    public void reset() {
        newDimensions(0, 0);
    }
}
```

- `BoxedTextReset` erbt von `BoxedText`.
- Alle `public`-Methoden von `BoxedText` (und somit auch die geerbten `public`-Methoden von `Object`) sind in `BoxedTextReset` verfügbar.

- `BoxedTextReset` fügt eine eigene `public` -Methode `reset()` hinzu.
- `BoxedTextReset` ist ein Subtyp von `BoxedText`. Das heißt, eine Variable vom Typ `BoxedText` könnte ein Objekt vom Typ `BoxedTextReset` referenzieren.

### Zusammenfassend:

Vererbung ermöglicht die Schaffung neuer Klassen, die auf bestehenden Klassen aufbauen, wodurch Code-Wiederverwendung gefördert und eine hierarchische Struktur von Klassen geschaffen wird. Die Unterklasse erbt die öffentlichen Eigenschaften und Verhaltensweisen ihrer Oberklasse und kann diese erweitern oder spezialisieren.

---

## Ersetzbarkeit

### Objekt eines Untertyps überall verwendbar wo Objekt eines Obertyps erwartet

jede Referenzvariable (auch Parameter) hat gleichzeitig

**deklarierten Typ:** Typ in der Deklaration der Variablen

**dynamischen Typ:** Klasse des Objekts, das gerade in der Variablen steht

→ jeder Ausdruck hat:

statisch vom Compiler ermittelten deklarierten Typ

zur Laufzeit abfragbaren, sich mit Zuweisungen ändernden dynamischen Typ

mehr zu Ersetzbarkeit: [6. Ersetzbarkeit und co > Ersetzbarkeit](#)

---

## getClass, class, instanceof

```
Print p = new BoxedText();      // declared as Print, dynamic type BoxedText
Class cp = p.getClass();       // representation of dynamic type of p
Class cBT = BoxedText.class;   // representation of class BoxedText
Class cP = Print.class;        // representation of interface Print
Class cA = Print[].class;      // representation of array of Print
Class ci = int.class;          // representation of int (no reference type)

cp == cBT                      -> true  (BoxedText is dynamic type of p)
cp == cP                        -> false (Print is not dynamic type of p)
p instanceof Print              -> true  (BoxedText is subtype of Print)
p instanceof BoxedText          -> true  (every type is subtype of itself)
p instanceof SetBoxed           -> true  (BoxedText is subtype of SetBoxed)
p instanceof BoxedTextReset     -> false (not subtype of BoxedTextReset)
null instanceof Print            -> false (null is not subtype of any type)
```



# Cast auf Referenztypen

```
Print p = new BoxedText();
p.print();                      // OK
p.setLine(0, "");                // syntax error, no setLine in Print
((BoxedText)p).setLine(0, "");    // OK, cast changes declared type
((SetBoxed)p).setLine(0, "");    // OK
((BoxedTextReset)p).print();     // error at runtime, dynamic type
                                // BoxedText no subtype of BoxedTextReset
((Print)null).print();          // null.print -> error at runtime
p = (Print)null;                // OK, null can be cast to each ref. type
```

## 6. Ersetzbarkeit und co

Quelle: ep2-06\_Ersetzbarkeit\_Dynamisches-Binden\_Methoden-aller-Klassen\_toString().pdf

Beinhaltet: Ersetzbarkeit Dynamisches Binden Methoden aller Klassen `toString()`

---

## Eigenschaften von Untertypbeziehungen

### 1. Reflexivität:

- **T ist Untertyp von T:** Jede Klasse oder jedes Interface ist ein Untertyp von sich selbst.

### 2. Transitivität:

- **Wenn R Untertyp von S und S Untertyp von T, dann ist auch R Untertyp von T:**

Wenn eine Klasse oder ein Interface ein Untertyp von einem anderen ist, und das andere ein Untertyp von einem weiteren, dann ist der erste auch ein Untertyp des dritten.

### 3. Antisymmetrie:

- **Wenn S Untertyp von T und T Untertyp von S, dann sind S und T gleich:** Zwei Typen sind nur dann gleich, wenn sie sich gegenseitig als Untertypen haben.

### 4. Interface und Untertypbeziehungen:

- **Wenn S ein Interface ist und Untertyp von T, dann ist auch T ein Interface:** Wenn ein Interface ein Untertyp eines anderen ist, muss das andere ebenfalls ein Interface sein.

### 5. Klassen und Untertypbeziehungen:

- **Wenn T eine Klasse ist und S ein Untertyp von T, dann ist auch S eine Klasse:** Ein Untertyp einer Klasse muss ebenfalls eine Klasse sein.

### 6. Vererbung von `Object`:

- **T ist eine Klasse, dann ist T Untertyp von Object:** Jede Klasse ist ein Untertyp von `Object`, der Wurzelklasse in Java.

### 7. Null und Untertypbeziehungen:

- **Null ist kein Objekt von T, obwohl vom Compiler so akzeptiert:** `null` wird vom Compiler als gültiger Wert für jede Referenz akzeptiert, ist aber kein tatsächliches Objekt eines bestimmten Typs.

# Ersetzbarkeit

## Wesentliche Aspekte der Ersetzbarkeit

- **Ersetzbarkeit** bedeutet, dass ein Objekt eines Untertyps (`s`) dort verwendet werden kann, wo ein Objekt des Obertyps (`T`) erwartet wird. Dies passiert hauptsächlich bei:
  - **Zuweisung** (z. B. einem Wert vom Typ `T` kann ein Wert vom Typ `s` zugewiesen werden),
  - **Parameterübergabe** (z. B. ein Parameter vom Typ `T` kann durch ein Argument vom Typ `s` ersetzt werden, wenn `s` ein Untertyp von `T` ist).

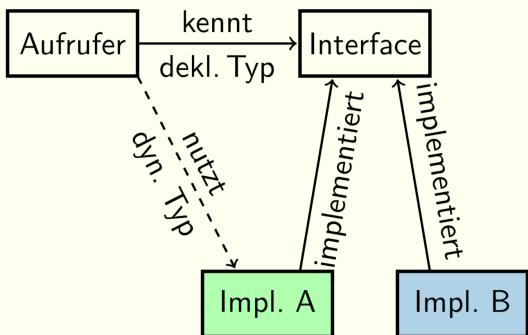
## Methode und Ersetzbarkeit

- Wenn **S Untertyp von T ist** und **T eine Methode beschreibt**, dann beschreibt auch **S eine Methode mit (vereinfacht) gleicher Signatur**. Die Methode kann in Objekten beider Typen (`s` und `T`) aufgerufen werden.
- **Voraussetzung:**
  - Die **Methodenbeschreibungen** (Kommentare) in den Klassen und Interfaces von `s` und `T` müssen übereinstimmen.
  - **Konsistenz der Kommentare:**
    - Kommentar in `s` beschreibt dieselbe Methode wie der Kommentar in `T`.
    - Kommentar in `s` kann konkreter sein, der Kommentar in `T` kann abstrakter sein.
    - **Compiler prüft nicht die Konsistenz von Kommentaren** – diese müssen vom Entwickler sichergestellt werden.

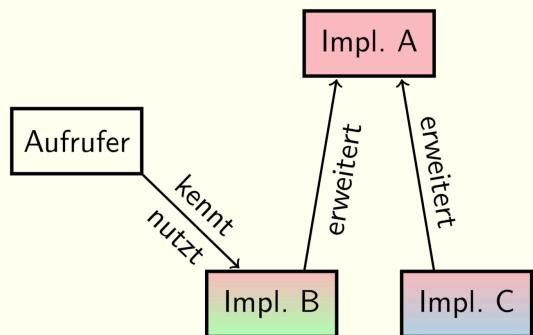
## Wozu Ersetzbarkeit?

- **Wiederverwendung durch Bilden konsistenter Versionen:**
  - Ein Teil der Software kann durch eine neue Version ersetzt werden, wenn die neue Version ein Untertyp der alten Version ist.
  - Der Rest der Software bleibt dabei unverändert und benötigt keine Anpassungen.
- **Wiederverwendung innerhalb einer Anwendung:**
  - Ein Programmteil kann mit verschiedenen Datenstrukturen umgehen, wenn diese Datenstrukturen Untertypen eines gemeinsamen Obertyps sind.
  - Dadurch wird die Flexibilität und Wartbarkeit des Codes verbessert, da man verschiedene Implementierungen des gleichen Typs verwenden kann.

### Untertypbeziehung



### Vererbungsbeziehung



# Welche Methode wird ausgeführt?

## Deklarierter Typ vs. Dynamischer Typ

### 1. Deklarierter Typ:

- Der **deklarierte Typ** bestimmt, welche **Methoden** aufgerufen werden können.
- Es handelt sich um den Typ, der bei der **Deklaration** einer Variablen oder eines Parameters angegeben wird.
- Beispiel: In der Methode `testAssoc(Assoc assoc)` ist der deklarierte Typ der Parameter `assoc` der Typ `Assoc`.

### 2. Dynamischer Typ:

- Der **dynamische Typ** (auch zur Laufzeit bestimmter Typ genannt) bestimmt, in welcher **Klasse** eine Methode zur Laufzeit ausgeführt wird.
  - Der dynamische Typ ist der Typ des Objekts, das zur Laufzeit der Methode zugewiesen wird, nicht der deklarierte Typ.
  - Beispiel: Wenn in `testAssoc(new TreeAssoc())` das Objekt `new TreeAssoc()` übergeben wird, ist der dynamische Typ `TreeAssoc`, obwohl der deklarierte Typ `Assoc` ist.
-

# Dynamisches Binden

## 1. Was ist dynamisches Binden?

- **Dynamisches Binden** bedeutet, dass die Auswahl der Methode, die zur Laufzeit ausgeführt wird, basierend auf dem **dynamischen Typ** des Objekts erfolgt.
- Das bedeutet, dass bei einem Methodenaufruf nicht sofort festgelegt wird, welche Implementierung ausgeführt wird. Die tatsächliche Implementierung wird zur Laufzeit anhand des **dynamischen Typs** des Objekts bestimmt.

## 2. Vorteil von dynamischem Binden:

### • Vermeidung von bedingten Anweisungen:

- Ohne dynamisches Binden müsste man manuell prüfen, welcher Typ das Objekt hat, und dann die Methode entsprechend aufrufen, wie im Beispiel:

```
if (assoc.getClass() == SimpleAssoc.class) {
    // execute assoc.put(...) in SimpleAssoc
} else if (assoc.getClass() == TreeAssoc.class) {
    // execute assoc.put(...) in TreeAssoc
}
```



- Dynamisches Binden ermöglicht es, solche **bedingten Anweisungen** zu vermeiden. Stattdessen wird die korrekte Methode zur Laufzeit automatisch aufgerufen.

## 3. Beispiel für dynamisches Binden:

```
class Assoc {
    public void put(String key, String value) {
        System.out.println("Assoc put");
    }
}

class TreeAssoc extends Assoc {
    @Override
    public void put(String key, String value) {
        System.out.println("TreeAssoc put");
    }
}

class SimpleAssoc extends Assoc {
    @Override
    public void put(String key, String value) {
        System.out.println("SimpleAssoc put");
    }
}
```

```

public static void testAssoc(Assoc assoc) {
    assoc.put("key", "value"); // Dynamisches Binden
}

public static void main(String[] args) {
    testAssoc(new TreeAssoc()); // Gibt "TreeAssoc put" aus
    testAssoc(new SimpleAssoc()); // Gibt "SimpleAssoc put" aus
}

```



- In diesem Beispiel wird beim Aufruf von `assoc.put("key", "value")` die Methode zur Laufzeit basierend auf dem **dynamischen Typ** des Objekts (`TreeAssoc` oder `SimpleAssoc`) ausgeführt.

## Statisches Binden

### 1. Wann tritt statisches Binden auf?

- Statisches Binden passiert, wenn die Methode zur **Kompilierungszeit** (also vor der Ausführung) festgelegt wird. Es gibt mehrere Szenarien, in denen statisches Binden verwendet wird:

### 2. Fälle, in denen statisches Binden auftritt:

- **Die Methode ist `static`**: Statische Methoden sind zur Kompilierungszeit bekannt, und der Methodenaufruf wird nicht zur Laufzeit aufgelöst.
- **Die Methode ist `private`**: Private Methoden sind ebenfalls nicht vererbbar und werden zur Kompilierungszeit aufgelöst.
- **Die Methode ist `final`**: Wenn eine Methode als `final` deklariert ist, kann sie nicht überschrieben werden, und der Methodenaufruf ist zur Kompilierungszeit bekannt.
- **Der deklarierte Typ hat keine Untertypen**: Wenn der deklarierte Typ keine Untertypen hat (es gibt keine Vererbung), kann der Compiler genau bestimmen, welche Methode aufgerufen wird.
- **Der Compiler kann den Aufruf eindeutig zuordnen**: In manchen Fällen kann der Compiler ohne Vererbung und ohne Polymorphismus den Methodenaufruf eindeutig festlegen.

### 3. Beispiel für statisches Binden:

```

class Assoc {
    public static void put(String key, String value) {
        System.out.println("Assoc static put");
    }
}

class TreeAssoc extends Assoc {
    @Override
    public void put(String key, String value) {

```

```

        System.out.println("TreeAssoc put");
    }

public static void main(String[] args) {
    Assoc assoc = new TreeAssoc();
    assoc.put("key", "value"); // Hier wird statisches Binden angewendet,
    weil put() statisch ist
}

```



- Da die Methode `put()` hier **statisch** ist, wird die Methode zur Kompilierungszeit festgelegt, unabhängig vom dynamischen Typ des Objekts.

## Hauptunterschiede:

| Merkmal               | Dynamisches Binden                                       | Statisches Binden   |
|-----------------------|--|---|
| Zeitpunkt der Bindung | Zur Laufzeit   | Zur Kompilierungszeit   |
| Basierend auf...      | Dem <b>dynamischen Typ</b> des Objekts                   | Dem <b>statischen (deklarierten) Typ</b>  |
| Verwendet bei...      | Instanzmethoden (die überschrieben werden können)        | <code>static</code> , <code>final</code> , <code>private</code> Methoden, oder ohne Vererbung |
| Polymorphismus        | Wird unterstützt   | Wird <b>nicht</b> unterstützt   |
| Flexibilität          | Hoch (ermöglicht polymorphes Verhalten)                  | Gering (kein Laufzeitverhalten abhängig vom Objekttyp)  |
| Performance           | Geringfügig langsamer (wegen Laufzeitentscheidung)       | Schneller (da zur Compile-Zeit entschieden)   |
| Beispelausdruck       | <code>assoc.put(...)</code> bei überschriebenen Methoden | <code>assoc.put(...)</code> bei <code>static</code> Methoden                                  |
| Typische Verwendung   | Vererbung, Polymorphie, Laufzeit-Entscheidungen          | Utility-Methoden, Klassenmethoden   |

# Erben und Überschreiben von Methoden

```
class X extends Y { ... }
```

X ist Unterklasse von Y,  
Y ist Oberklasse von X

X **erbt** nicht-statische Methoden und Variablen von Y:  
was nicht-statisch in Y definiert ist, ist automatisch auch in X definiert;  
wenn private, dann nur über geerbte Methoden sichtbar

ererbare Methoden können **überschrieben** werden:  
wird eine in Y definierte Methode in X neu definiert,  
dann existiert in X die neu definierte Methode, sie wird nicht von Y übernommen

**final** Methoden dürfen nicht überschrieben werden

## java.lang.Object

jede Klasse ist Untertyp von Object

→ Methoden von Object existieren in jeder Klasse:

|                            |  |               |
|----------------------------|--|---------------|
| Class getClass()           | dynamischer Typ von this                   | Themen in EP2 |
| boolean equals(Object obj) | vergleicht this mit obj                    |               |
| int hashCode()             | Hash-Code von this                         |               |
| String toString()          | String-Darstellung von this                |               |
| Object clone()             | erzeugt eine Kopie von this                |               |
| void notify()              | Aufwecken eines auf this wartenden Threads |               |
| void notifyAll()           | Aufwecken aller auf this wartender Threads |               |
| void wait(...)             | aktueller Thread muss auf this warten      |               |
| void finalize()            | Destruktor, kaum verwendet                 |               |

## Überschreiben der Methoden von Object

getClass ist final → darf nicht überschrieben werden

toString, equals und hashCode werden häufig überschrieben

komplexe Einschränkungen auf equals und hashCode in Object,  
die durch Subtyping auf Untertypen übertragen werden;  
equals und hashCode müssen gemeinsam überschrieben werden

toString durch Sonderbehandlung tief in Java integriert

# toString()

## Besonderheiten von `toString()`

### 1. Verwendung von `toString()`:

- Wenn `x` eine Variable eines Referenztyps ist, dann entspricht:

```
"" + x
```



dem Aufruf:

```
"" + x.toString()
```



- `System.out.print(x)` entspricht `System.out.print(x.toString())`.
  - Dies bedeutet, dass `x.toString()` automatisch aufgerufen wird, wenn ein Objekt als Argument an `System.out.print()` übergeben wird.

### 2. Debugging:

- Der Wert von `x` wird im Debugger durch `x.toString()` dargestellt.

### 3. Standardimplementierung von `toString()`:

- Die Implementierung in `Object` liefert einen String im Format:

```
Klassenname@874
```



- Dies gibt den Klassenname und den Hashcode des Objekts zurück.

### 4. Bedingung:

- Der Ausdruck:

```
x.toString().equals(x.toString())
```



ergibt `true`, wenn `x` unverändert bleibt.

## Verwendbarkeit von `toString()`

### 1. Verwendbarkeit:

- `x.toString()` ist für jeden Ausdruck eines Referenztyps ausführbar, sofern `x != null`.

- Dies funktioniert nur, weil **jedes Objekt durch eine Klasse erzeugt wird** (mit der Ausnahme von **Arrays**).

## 2. Vererbung:

- Jeder Ausdruck eines Referenztyps hat eine **Klasse als dynamischen Typ**, und jede Klasse ist ein **Untertyp von Object**.
- Die Methode `toString()` ist in `Object` definiert.

## 3. Überschreiben von `toString()`:

- Die Methode `toString()` ist nur dann wirklich sinnvoll, wenn sie in jeder Klasse überschrieben wird.
- Der Überschreibungszweck ist, dass `toString()` dann einen sinnvollen **String** liefert, der das Objekt beschreibt.

# Informationsgehalt von `toString()`

## 1. Informationsgehalt im Ergebnis:

- Der Informationsgehalt des Ergebnisses von `toString()` ist frei wählbar, aber es gibt gängige Konventionen:
  - "**an object**": Versteckt die Informationen vollständig vor dem Benutzer.
  - "**a Tdynamischen Typ** des Objekts zurück, nützlich für Debugging.
  - "**T@874toString() in `Object`. Zeigt den **dynamischen Typ** und eine eindeutige Nummer (oft Hashcode) des Objekts.**
  - "**[1, 2, 3]Inhalt einer Datenstruktur**, was eine inhaltliche Gleichheit darstellt.

## 2. Automatische Generierung:

- `toString()` kann automatisch generiert werden, um die inhaltliche Gleichheit von Objekten (insbesondere von Collections) widerzuspiegeln.
- **Automatisch generierte `toString()`-Methoden** kennen jedoch keine **Semantik** und liefern daher oft **falschen Informationsgehalt**.
  - Zum Beispiel wird bei einer Collection ohne semantische Bedeutung der Inhalt als String ausgegeben, aber es könnte in manchen Fällen nicht den gewünschten Informationsgehalt liefern.

## 3. Aussehen:

- Die Standardimplementation von `toString()` entspricht nur den **minimalen Anforderungen**, die die Darstellung eines Objekts in Form eines Strings betrifft.

# Zusammenfassung

- `toString()` ist eine Methode, die in jeder Klasse überschrieben werden sollte, um sinnvolle Informationen über ein Objekt bereitzustellen.
- Die **Standardimplementierung von `toString()`** gibt nur den Klassennamen und den Hashcode zurück, was oft nicht aussagekräftig ist.

- Der **Informationsgehalt von `toString()`** ist flexibel und kann für Debugging, Fehlerverfolgung oder die Darstellung des Inhalts von Datenstrukturen angepasst werden.
- Automatisch generierte `toString()`-Methoden liefern in der Regel nur grundlegende Informationen und sind nicht immer semantisch korrekt.