

# Grundlagen des MethodenÜberschreibens (Override) in Java

---

- Definition: Überschreiben einer Methode der Superklasse in der Unterklasse.
- Zweck: Spezifisches Verhalten in der Unterklasse definieren.
- Nutzung von @Override Annotation zur Verbesserung der Lesbarkeit und zur Vermeidung von Fehlern.
- Überschreiben funktioniert auch ohne @Override, nur kann es dann passieren, dass durch z.B. ein Typo nicht die Funktion überschrieben wird wie beabsichtigt – z.B.:

```
public String toSting() ... // Anstatt toString
```

## Überschreiben von toString()

---

- Zweck: Rückgabe einer String -Repräsentation des Objekts.
- Wichtigkeit: Für Debugging und Logging.
- Beispiel: Anpassung zur Anzeige relevanter Objektinformationen.

## Überschreiben von equals(Object obj)


---

- Wichtigkeit: Für den Vergleich von Objekthinhalten anstelle von Objektreferenzen.
- Regeln: Reflexivität, Symmetrie, Transitivität, Konsistenz, Nicht -Null.
- Beispiel: Implementierung eines maßgeschneiderten Gleichheitsvergleichs.



## Überschreiben von hashCode()

---

- Zweck: Konsistente Erzeugung von Hashcodes für Objekte.
  - Regel: Gleiche Objekte müssen den gleichen Hashcode zurückgeben.
  - Bedeutung: Wichtig für die Verwendung in Hash -basierten Sammlungen wie HashMap, HashSet.
  - Beispiel: Implementierung in Korrelation mit equals.
- 

# Einführung in Java Records


---

- Zweck: Vereinfachung der Erstellung von Daten -Trägerklassen (data carrier classes).
- Ein Record ist eine spezielle Art von Klasse in Java, die dazu dient, unveränderliche Daten auf eine kompakte Weise zu repräsentieren.
- Syntax: `record RecordName(Type1 fieldName1, Type2 fieldName2, ...) { }`



## Hauptmerkmale von Records

---

- Daten-Immunität: Record -Instanzen sind unveränderlich (immutable).
  - Automatische Generierung: Getter -Methoden, `equals()`, `hashCode()`, und `toString()` werden automatisch generiert.
  - Kompakte Schreibweise: Reduziert Boilerplate -Code im Vergleich zu traditionellen Klassen.
- 

# Vorteile von Records


---

- Klarheit und Kompaktheit: Code ist einfacher zu lesen und zu warten.
- Reduzierung von Boilerplate -Code: Weniger manueller Code für Standardmethoden.
- Förderung der Unveränderlichkeit: Vermeidung von Seiteneffekten und Bugs.



# Einschränkungen und Überlegungen

---

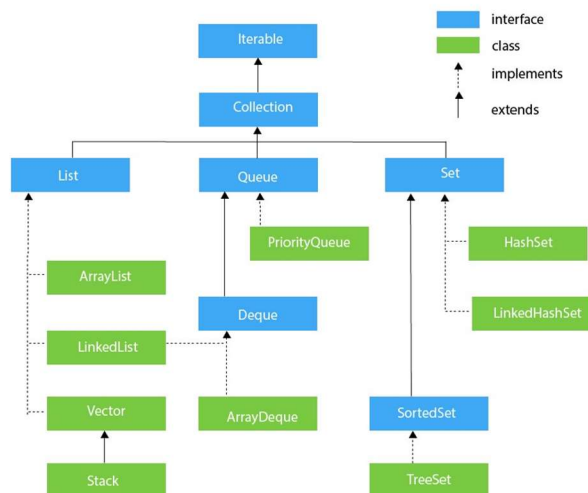
- Keine Vererbung: Records können keine anderen Klassen erweitern, außer der impliziten `java.lang.Record`.
  - Felder sind final: Keine Möglichkeit, Felder nach der Instanziierung zu ändern.
  - Verwendungszweck: Am besten geeignet für kleine, datenzentrische Klassen.
- 

# Java Collections



- In Java gibt es verschiedene Collections, um Gruppen von Objekten gemeinsam zu manipulieren und zu speichern
- Dazu bieten Collections ein bestimmtes Set an Funktionen, die alle Collection Klassen beinhalten.
- Dazu gehören zum Beispiel `add(Object e)`, `remove(Object e)`, ...

## Java Collection-Hierarchie



# Java Collections

---

- Das Iterable Interface ist das "Root"-Interface für alle Collection Klassen
- Das List Interface gibt einen Listentyp vor, in dem es möglich ist Objekte in geordneter Reihenfolge abzuspeichern
- Es können dort Elemente mehrfach vorkommen
- Es gibt 4 Klassen, die das List Interface implementieren: ArrayList, LinkedList, Vector, Stack

## ArrayList

---

- ArrayList implementiert das List Interface
- Es repräsentiert ein dynamisches Array
- Der Zugriff auf die Elemente erfolgt dynamisch

Z.B.:

```
List<String> strList = new ArrayList<>(); // or...
```

```
List<String> strList = new ArrayList(); // or...
```

```
List<String> strList = new ArrayList<String>(); // or...
```

# LinkedList

---

- LinkedList speichert Werte wie ArrayList und kann auch Duplikate beinhalten
- Der Große Vorteil von LinkedList liegt darin, dass der Zugriff auf manche Elemente einfacher gemacht wird.
- Z.B. `getFirst()`, `getLast()`, `offerFirst(E e)`, `offerLast(E e)`, ...
- Beispiel für LinkedList:  

```
LinkedList<String> linkedStrList = new LinkedList<>();
```

# Vector

---

- Vector ist einer ArrayList sehr ähnlich
- Der größte Unterschied ist, dass auf einen Vector nur synchronisiert zugegriffen werden kann
- D.h. wenn es mehrere Threads gäbe, die Werte in einen Vector eintragen würden, würden sie dieses nacheinander tun
- ```
Vector<String> strVec = new Vector<>();
```

# Stack

---

- Stack ist von Vector abgeleitet und wird durch Funktionen ergänzt, die einen Stack charakterisieren – z.B. [push\(E item\)](#), [pop\(\)](#), [peek\(\)](#), ...
- `_Stack<String> strStack = new Stack() ;`

# Queue Interface

---

- Das Queue Interface zielt darauf ab Daten nach dem First -in-First-out Prinzip zu speichern.
- Die Klassen PriorityQueue, Dequeue und ArrayDequeu implementieren das Queue Interface.
- Die PriorityQueue organisiert seine Elemente nach Ihrer Priorität. In einer Dequeue kann man Elemente von beiden Seiten der Queue anfügen.
- Die PriorityQueue erlaubt keine „null“ Elemente



# Set Interface

---

- Das Set Interface repräsentiert grundsätzlich eine Collection, die keine doppelten Werte erlaubt. ACHTUNG – bei HashSets kann sich die Reihenfolge von Elementen ändern, das bedeutet die Elemente sind dort ungeordnet enthalten
- Set wird von HashSet, LinkedHashSet und TreeSet implementiert.
- Das HashSet repräsentiert die Standardimplementierung vom Interface Set
- Das LinkedHashSet repräsentiert eine LinkedList, die nur eindeutige Werte enthalten darf.



# SortedSet Interface

---

- Das SortedSet Interface bietet eine Möglichkeit Elemente als Set in geordneter Weise zu verwalten (aufsteigend).
- Das TreeSet Implementiert das Set bzw. SortedSet Interface. Es beinhaltet also geordnete Elemente (diese werden beim hinzufügen einsortiert) und eindeutige Elemente.

# Iterable Interface

---

- Das Iterable Interface ist das Root Interface aller Collections. Es enthält nur eine abstrakte Methode: `Iterator<T> iterator()`
- Diese Methode returniert den Iterator über die Elemente vom Typ T.
- Mit dieser Methode hat jede Collection die Möglichkeit nicht nur über eine „Standard“ Schleife, sondern auch über diesen Iterator durch alle Elemente zu iterieren. Z.B.:

```
Iterator itr=list.iterator();
while(itr.hasNext()){
    System.out.println(itr.next());
}
```

## Gemeinsames Beispiel- Teilnehmerverwaltung Marathonlauf

---

- Bei einem großen Marathonlauf müssen die Daten der Teilnehmer sowie ihre Laufzeiten verwaltet werden. Deine Aufgabe ist es, ein System zu entwerfen, das diese Informationen effizient speichert und auswertet.
- Ein Läufer besteht aus einem Namen, einer Startnummer und der Laufzeit (in Minuten)
- Entwickle eine Anwendung, die eine Liste von Läufer -Records verwaltet und eine Funktion bietet, die die Durchschnittszeit aller Läufer ausgibt und eine Funktion, die die geringste Laufzeit ausgibt.

# Gemeinsames Beispiel- Bibliothekskatalogisierung


---

- In einer Bibliothek sollen Bücher zusammen mit Autoreninformationen und Verfügbarkeitsstatus katalogisiert werden. Das Ziel ist es, ein System zu entwickeln, das diese Informationen strukturiert und leicht zugänglich macht.
- Ein Buch besteht aus einem Titel, Autor, ISBN und einer Verfügbarkeit
- Erstelle ein Programm, das eine Sammlung von Büchern verwaltet, und ermögliche Funktionen wie das Suchen nach Büchern oder das Überprüfen ihrer Verfügbarkeit.



# Gemeinsames Beispiel- Verwaltung von Hotelbuchungen

---

- In einem Hotel müssen Gästeinformationen sowie Details zu ihren Aufenthalten verwaltet werden. Du bist dafür verantwortlich, ein System zu entwickeln, das diese Daten effizient speichert und abfragt.
  - Hotelbuchungen bestehen aus einem Gastnamen, Zimmernummer, Check -in-Datum und Check -out-Datum
  - Erstelle eine Anwendung, die Hotelbuchungen speichert und Funktionen wie die Überprüfung der Verfügbarkeit von Zimmern und die Historie von Gästeaufenthalten ermöglicht.
- 

# Map

---

- Eine Map bietet die Möglichkeit Werte mit eindeutigen Schlüsseln zu speichern.
- Dabei ist Map ein Interface, dass von HashMap, TreeMap, ... implementiert wird.

In einer Map kann es jeden Schlüssel nur einmal geben, allerdings können mehrere Schlüssel die gleichen Werte haben.

```
Map<Integer, String> myMap = new HashMap<>();  
myMap.put(0, "First Value with Key 0");  
myMap.put(1, "Second Value with Key 1");  
//...
```

## Gemeinsames Beispiel- Stadtbibliothek

---

- In einer Stadtbibliothek müssen die Ausleihen von Büchern effizient verwaltet werden. Jedes Buch hat eine eindeutige ID, und es muss nachvollziehbar sein, welcher Benutzer welches Buch ausgeliehen hat.
- Kreiere eine HashMap, die aus einer BuchID als Key und Benutzer als value besteht
- Entwickle ein System, um zu verfolgen, welche Benutzer welche Bücher ausgeliehen haben, und implementiere Funktionen für Ausleihen, Rückgaben und Verfügbarkeitsprüfungen.

# Lambda Ausdrücke

---

- Die Ausdrücke sind limitiert und müssen gleich einen Rückgabewert liefern.
- Um komplexeren Code mit if Statements oder Schleifen zu integrieren kann man einen Codeblock in geschwungenen Klammern angeben:  
**(Parameter1, Parameter2) -> { Codeblock }**

# Lambda Ausdrücke

---

- Lambda Ausdrücke werden normalerweise Funktionen übergeben.

Beispiel:

```
list.forEach(string -> {  
    System.out.println(string);  
});
```

- Die Funktion `forEach`, die jede Collection Klasse hat, iteriert durch alle Elemente und führt für diese die Funktion aus, die man ihr übergibt.

# Lambda Ausdrücke

---

- Lambda Ausdrücke können in Variablen gespeichert werden. Dazu muss der Variablentyp ein Interface sein, in dem es nur eine Funktion gibt.
- Der Lambda Ausdruck muss dann die gleiche Anzahl an Parametern und den gleichen Returntyp haben wie die Funktion vom Interface
- Java bietet dazu auch schon Interfaces, wie zum Beispiel das **Consumer**
- Das Consumer Interface besitzt die Funktion `accept(T t)`, die einen generischen Parameter akzeptiert. Das heißt, wenn man eine Lambda Funktion mit einem Übergabeparameter benötigt, kann man diese in einer Variablen vom Typ Consumer speichern:

```
Consumer<Integer> method = (n) -> { System.out.println(n); };
```

# Lambda Ausdrücke

---

Um Lambdaausdrücke in Methoden zu verwenden, muss es in der Methode eine Variable geben, die als Typen ein Interface besitzt, dass (genau) eine Funktion besitzt.

```
interface StringFunction {
    String makeNews(String str);
}

public static void printNews(String news, StringFunction func)
{
    String completeNews = func.makeNews(news);
    System.out.println(completeNews);
}

StringFunction fancyNews = str->"Fancy: " + str;
printNews("Finanzminister geht mit Laptop gassi.", fancyNews);
```

# Gemeinsames Beispiel- Quadratzahlen-Rechner

---

- Schreibe ein Programm, das eine Liste von Zahlen durchläuft und deren Quadratzahlen berechnet. Verwende dabei die `forEach` Funktion. Gib das Ergebnis aus.

# Gemeinsames Beispiel- Personalisierte Nachrichtengenerator

---

- Entwickle ein Programm, das personalisierte Nachrichten für eine Liste von Namen generiert.
- 1. Definiere ein funktionales Interface: Erstelle ein funktionales Interface `MessageCreator` mit einer Methode `createMessage`, die einen `String` (den Namen) entgegennimmt und einen `String` (die personalisierte Nachricht) zurückgibt.
- 2. Implementiere das Interface: Erstelle eine `Lambda` -Funktion, die das `MessageCreator` -Interface implementiert und eine einfache Begrüßungsnachricht für den gegebenen Namen generiert.
- 3. Erstelle eine Liste von Namen und verwende deine `Lambda` -Funktion, um für jeden Namen eine personalisierte Nachricht zu generieren und auszugeben.

# Gemeinsame Aufgabe- Märchenstunde



Schreib ein Programm, dass einem Kind eine von vier Gute Nacht Geschichten erzählt. Die Geschichte muss „langsam“ ausgegeben werden, als würde sie vorgelesen werden. Bei jedem Punkt gibt es eine Pause von 2 Sekunden.

Mit einer steigenden Wahrscheinlichkeit, wird das Kind die Story unterbrechen. Die Wahrscheinlichkeit wächst zwischen 1% und 10% nach 300Characktern.

Das Kind kann zufällig aus einemder folgenden Gründen die Geschichte unterbrechen:

Es muss auf die Toilette – es muss eine passende Ausgabe geben und das dauert 10 Sekunden

Es hat Hunger und möchte einen Keks– Ausgabe + 6 Sekunden

Es hat Durst – Ausgabe + 6 Sekunden

Es ist gelangweilt – in diesem Fall muss man das Kind fragen, ob es eine andere Geschichte hören will. Mit einer Wahrscheinlichkeit von 50% möchte es das, dann muss man mit einer neuen Geschichte beginnen. Wenn man bereits alle Geschichten begonnen hatte, muss man das dem Kind mitteilen und es muss so einschlafen.

## Lambda Ausdrücke



- Gibt es seit Java 8
- Ist ein kleiner Block von Code, der eine Funktion repräsentiert.
- Ein Lambda Ausdruck hat keinen Namen und kann innerhalb von Funktionen implementiert sein.
- Form eines Lambda Ausdrucks mit einem Parameter:  
**Parameter -> Ausdruck**  
\_ mit zwei Parametern:  
**(Parameter1, Parameter2) -> Ausdruck**