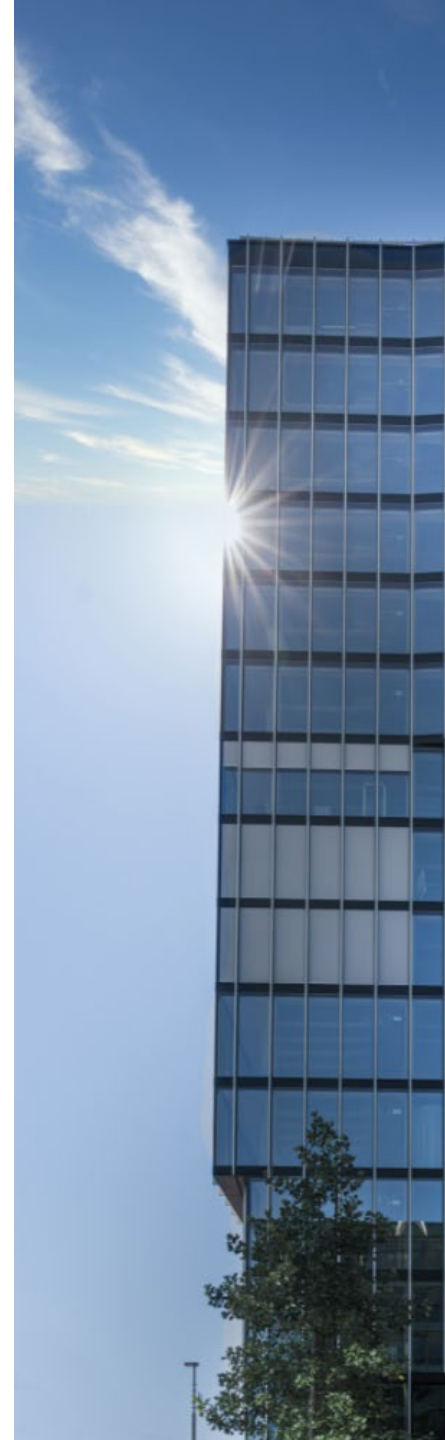


Objektorientierte Programmierung

# Schnittstellen

**Abstraktion, Modularisierung und Interfaces**

Roland Gisler



# Inhalt

- Abstraktion
- Modularisierung
- Abstrakte Klassen
- Interfaces
- JavaDoc

# Lernziele

- Sie verstehen den Vorgang der Abstraktion besser.
- Sie verstehen die grundlegende Idee der Modularisierung.
- Sie beherrschen den Entwurf von abstrakten Klassen.
- Sie sind in der Lage Schnittstellen (Interfaces) zu definieren.
- Sie können Schnittstellen einsetzen (implementieren).
- Sie können für ausgewählte Elemente eine Javadoc schreiben.

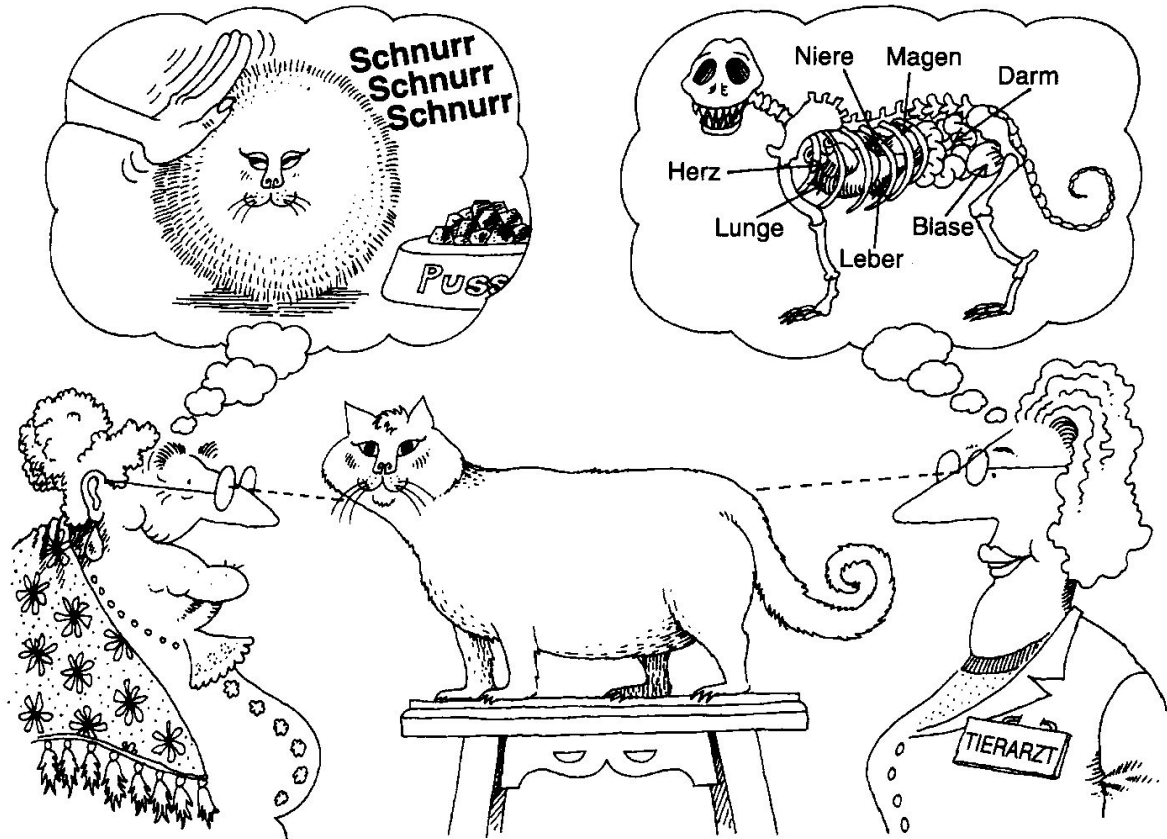
# **Abstraktion**

# Abstraktion – Vereinfachung zur Modellierung

- Einer der wesentlichsten Aspekte bei der objektorientierten Modellierung ist die Abstraktion:
  - Vereinfachung bzw. Reduktion der Realität auf das im jeweiligen Kontext unmittelbar Notwendige.
- Wenn wir direkt Klassen entwerfen, abstrahieren wir typisch aus einer «Innenperspektive»:
  - Identifikation von Attributen: Daten zur **internen** Repräsentation des Zustandes.
  - Identifikation des Verhaltens: Öffentliche als auch **private** (interne) Methoden zur Umsetzung dessen.
- In einem ersten Schritt genügt häufig die «Aussenperspektive»:
  - Nur Identifikation der Methoden (Verhalten) mit allfälligen Parametern und Rückgabewerten! ➔ **Nutzendensicht!**

# Abstraktion – Abhängigkeit vom Kontext

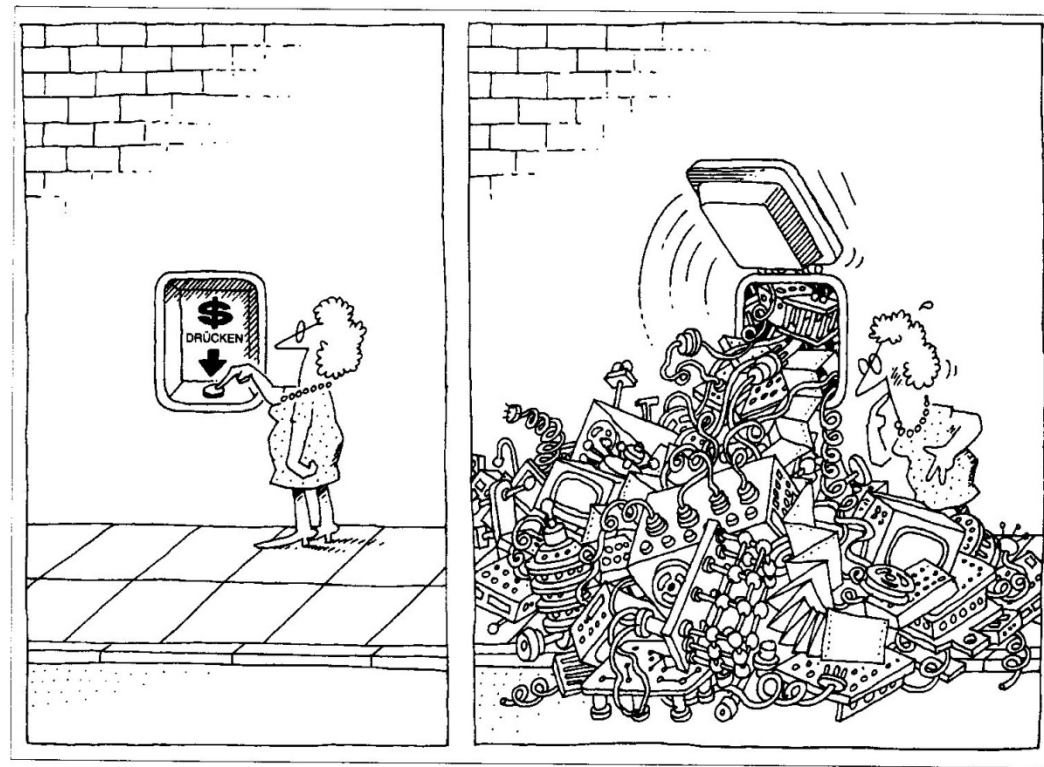
- Abstraktion ist sehr abhängig von der Perspektive der jeweiligen Betrachter\*in! ➔ Nutzendensicht versus Eigen-/Innensicht.



*Die Abstraktion konzentriert sich auf die wesentlichen Charakteristika eines Objekts, relativ zur Perspektive des Betrachters.*

# Abstraktion – Trennung zwischen WAS und WIE

- Wir können die Abstraktion zusätzlich verstärken, indem wir konsequent zwischen dem **WAS** und dem **WIE** unterscheiden.
- Beispiel: Schnittstelle für das Auslösen eines Geldbezuges
  - Hier: Einfache Taste!
- Die (hoch-)komplexe Realisation des Vorganges bleibt hinter einer vergleichsweise einfachen Schnittstelle vollständig verborgen!



*Die Aufgabe des Softwareentwickler-Teams ist, die Illusion von Einfachheit zu erzeugen*

# **Modularisierung**



# Modularisierung – Definition

- Modularisierung bezeichnet die Zerlegung einer Gesamtaufgabe in Teilaufgaben und die Definition der erforderlichen Schnittstellen, so dass die entstehenden Module möglichst unabhängig voneinander bearbeitet werden können.

[Specht, G./Beckmann, C./Amelingmeyer, J. 2002]

- „divide et impera“ – **teile und herrsche**  
Ursprünglich ein Prinzip der altrömischen Aussenpolitik: Unfrieden stiften, damit sich das Volk in Untergruppen aufteilt und besser beherrschbar ist.

# Modularisierung - Motivation

- Grundidee: Man macht grosse, komplexe Systeme (besser) beherrschbar, indem man sie in mehrere, kleinere Teile zerlegt:
  - Teile weisen ein klar definiertes Verhalten auf.
  - Teile haben eine überschaubare Komplexität und Grösse.
  - Teile sind möglichst gut in sich abgeschlossen.
  - Teile sind dadurch einzeln gut wiederverwendbar!
- Die Modularisierung ist neben der Abstraktion ein weiteres, sehr fundamentales Konzept in der Softwareentwicklung.
  - Bei der Objektorientierung sehr häufig genutzt, aber **nicht** automatisch gegeben!
- Verwendete Prinzipien:
  - **S**eparation **o**f **C**oncerns (SoC)
  - **S**ingle **R**esponsibility **P**rinciple (SRP)

# Modularisierung - Manifestation

Je nach Abstraktionsebene (hierarchisch) und verwendeter Technologie und Programmiersprache können sich «Module» sehr unterschiedlich manifestieren:

- Auf tiefer Ebene sind es in der Objektorientierung typisch Klassen, welche Module repräsentieren. → Datenkapselung
- Auf mittlerer Ebene gibt es verschiedene Technologien/Ansätze: Allgemein spricht man dort z.B. von Modulen, Komponenten, Libraries etc.
  - Nur wenige Sprachen bieten hier umfassende Ansätze an.
  - Java: «Erst» seit Version **9** wird ein eigenes Modularisierungssystem unterstützt, das sich leider nur sehr langsam durchsetzt.
- Auf höherer Ebene (Architektur) erreicht man Modularisierung z.B. durch die Implementation von [Micro-]Services.

# Modularisierung - Beispiel

- Zerlegung und Strukturierung eines komplexen Gesamtsystems in verschiedene, eigenständige Einheiten (Module).
  - Deutliche Reduktion der Komplexität (in den einzelnen Modulen).
- Die Module erfüllen jeweils eine **klar abgegrenzte** Aufgabe.
- Die Module besitzen **wohldefinierte Schnittstellen**, die möglichst schmal und einfach sind.



Bildquelle: Objektorientierte Analyse und Design; Grady Booch

*Modularität packt Abstraktionen in eigenständige Einheiten.*

# Umsetzung von Abstraktion und Modularisierung

- Auf tiefster Ebene können wir in objektorientierten Sprachen wie Java tatsächlich normale Klassen als Basis der Modularisierung betrachten.

Es gibt aber noch zwei interessante Alternativen bzw. Ergänzungen:

➔ Abstrakte Klassen

➔ Interfaces

# **Abstrakte Klassen**

# Abstrakte Klassen in Java

- In Java können wir **abstrakte Klassen** definieren!
- Diese Klassen sind «so» abstrakt, dass es **nicht** möglich ist, davon Objekte zu instanziiieren, weil abstrakte Klassen (mindestens teilweise) noch über **keine** Implementation (der Methoden) verfügen!
- Abstrakte Klassen sind somit ein Mittel zwar das **WAS** (über Methodenköpfe) zu definieren, aber noch nicht das **WIE** (keine Implementation der Methodenrumpfe).
- Es ist auch möglich, zumindest einzelne **Teile** von abstrakten Klassen bereits vollständig zu implementieren:
  - Es entsteht quasi ein **Mix** zwischen **Abstrakt** und **Konkret**.
  - Analogie: Rohbau eines Hauses!

# Abstrakte Klassen - Eigenschaften

- Schlüsselwort: **abstract** (ergänzend zu **class**)
  - Sowohl in Klassen- als auch Methodenköpfen möglich.
- Abstrakte Methoden erzwingen **implizit** eine abstrakte Klasse.
  - Haben keinen Methodenrumpf (body).
  - Bereits eine einzige abstrakte Methode macht auch die ganze Klasse (implizit) abstrakt.
- Von abstrakten Klassen können **keine** Objekte instanziiert werden, sie definieren aber einen validen **Typ**.
- Abstrakte Methoden legen somit primär eine **Schnittstelle** fest und erzwingen, dass diese anderorts implementiert wird.
  - Konkret: in einer Spezialisierung → **Vererbung** (folgt später).



# Abstrakte Klassen - Varianten

- Variante **1**: Klasse mit **mindestens** einer abstrakten Methode:
  - Führt dazu, dass die ganze Klasse **implizit** abstrakt wird, d.h. der Klassenkopf verlangt auch das Schlüsselwort **abstract**.
  - Wenn **alle** Methoden abstrakt → **vollständig** abstrakte Klasse.
- Variante **2**: Klasse wird im Klassenkopf abstrakt definiert:
  - Klasse ist somit **explizit** abstrakt.
  - Methoden können implementiert oder abstrakt sein.
- In beiden Fällen können Objekte dieser Klasse nur dann instanziiert werden, wenn man die abstrakten Klassen durch Spezialisierung konkretisiert:
  - Spezialisierung durch → **Vererbung** (folgt später) und Ergänzen der Implementation durch Überschreiben der bisher abstrakten Methoden.

# Beispiel 1: Vollständig abstrakte Klasse

- Eine **vollständig** abstrakte Klasse enthält **keine** Implementation:

```
package ch.hslu.oop.oop06;

/**
 * Abstrakte Klasse für einen Schalter.
 */
public abstract class AbstractSwitch {

    public abstract void switchOn();

    public abstract void switchOff();

    public abstract boolean isSwitchedOn();

    public abstract boolean isSwitchedOff();

}
```

<i>AbstractSwitch</i>
+ switchOn() : void + switchOff() : void + isSwitchedOn() : boolean + isSwitchedOff() : boolean

## Beispiel 2: Nicht vollständig abstrakte Klasse

- Enthält auch bereits implementierte Methoden (und ggf. Attribute):

```
public abstract class AbstractSwitchVariant {
```

```
    private boolean switchedOn = false;
```

```
    public abstract void switchOn();
```

```
    public abstract void switchOff();
```

```
    public final boolean isSwitchedOn() {
        return this.switchedOn;
    }
```

```
    public final boolean isSwitchedOff() {
        return !this.switchedOn;
    }
```

```
    protected final void setSwitchedOn(final boolean switchedOn) {
        this.switchedOn = switchedOn;
    }
}
```

### *AbstractSwitchVariant*

- switchedOn : boolean = false

+ switchOn() : void

+ switchOff() : void

+ isSwitchedOn() : boolean

+ isSwitchedOff() : boolean

# setSwitchedOn(switchedOn : boolean) : void

# Abstrakte Klassen - Namensgebung

- Abstrakte Klassen werden häufig auch direkt im Namen als solche markiert: Man setzt dazu den Prefix «**Abstract**» vor den eigentlichen Namen.
- Beispiele: **AbstractConnection**, **AbstractPerson** etc.
- Dadurch sind abstrakte Klassen schnell und einfach als solche zu erkennen.
- Bei der alphabetischen Sortierung der Klassen fallen sie dann aber leider aus dem Namenskontext.
  - Vergleiche dazu die häufige Postfix-Variante bei ➔ **Interfaces**.

# Abstrakte Klassen - Empfehlungen



- (Teil-)Abstrakte Klassen «gut» einzusetzen ist (in Java) eine **echte** Herausforderung und sollte eher zurückhaltend und wohlüberlegt erfolgen!
- Ist eine Klasse **vollständig abstrakt** (d.h. **alle** Methoden sind abstrakt) ist wiederum ein **→Interface** (folgt unten) sehr **häufig** die viel **bessere** Alternative.
  - ggf. in Kombination mit einer abstrakten Klasse.
- In der Regel sollte eine als abstrakt definierte Klasse auch **mindestens** eine abstrakte Methode enthalten.
- Will man «nur» die Instanziierung von Objekten verhindern, gibt es bessere Wege als die Klasse abstrakt zu definieren:  
Klasse **finalisieren** und **privaten** Konstruktor implementieren!

# Interfaces

# Interfaces in Java

- In Java können wir reine **Interfaces** (Schnittstellen) definieren!
- Interfaces sind vollständig abstrakt, es ist somit **nie** möglich, davon direkt Objekte zu instanziiieren, weil:
  - Interfaces enthalten **ausschliesslich** Methodenköpfe.
  - Interfaces enthalten **keine** Attribute\*.
  - Interfaces enthalten **keine** Implementation\*.
- Interfaces beschreiben ausschliesslich das «öffentliche» Verhalten von Klassen, was man gerne auch mit einer «**Rolle**» vergleicht.
  - **Keinerlei** Aussagen oder Annahmen zur Implementation!
- Interfaces sind somit ein Mittel, ausschliesslich das **WAS** (Methodenköpfe) zu definieren.

\* beides in/für Spezialfällen in Java trotzdem möglich, in der Regel aber **nicht empfohlen**.

# Interfaces - Eigenschaften

- Schlüsselwort: **interface** (anstatt **class**)
- Ein Java Interface spezifiziert **ausschliesslich** Methodenköpfe ohne Implementationen zu beinhalten, d.h. nur das **WAS!**
- **Interfaces definieren einen validen Typ.**
- Ein Interface enthält **keinen** Konstruktor und man kann keine Objekte von einem Interface instanziiieren.
- Methoden eines Interfaces sind implizit **public** und **abstract**.
  - Die entsprechenden Schlüsselwörter können somit entfallen.
- Interfaces können von Klassen **implementiert** werden:
  - Dadurch eignet sich eine Klasse den Typ (die «Rolle» welche ein Interface beschreibt) an, und ist auch darüber ansprechbar!



# Definition eines Interface - Beispiel

- Interface für einen einfachen Schalter:

```
/**
 * Schnittstellen NIE ohne Dokumentation!!!
 * Hier nur aus Platzgründen entfallen.
 */
public interface Switchable {

    void switchOn();

    void switchOff();

    boolean isSwitchedOn();

    boolean isSwitchedOff();

}
```

<<interface>>

**Switchable**

+ *switchOn() : void*  
+ *switchOff() : void*  
+ *isSwitchedOn() : boolean*  
+ *isSwitchedOff() : boolean*

Alternativ:



**Switchable**

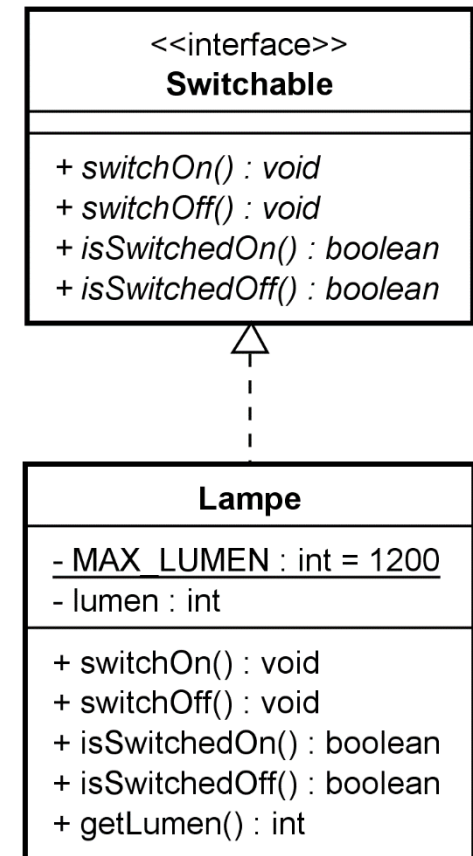
# Implementation von Interfaces

- Klassen können ein oder mehrere (1..n) Interfaces mit dem Schlüsselwort **implements** implementieren.
  - Mehrere Interfaces werden mit Komma getrennt.

```
/**
 * Modell einer Lampe mit Lichtstrom.
 */
public final class Lampe implements Switchable {

    ...

    @Override
    public void switchOn() {
        if (this.isSwitchedOff()) {
            this.lumen = MAX_LUMEN;
        }
        ...
    }
    ...
}
```



# Interfaces – Ein geniales Konzept in Java!

- Interfaces dienen als Spezifikation (das WAS).  
Die Implementation einer Funktionalität (WIE) ist somit **vollständig** von deren Schnittstelle getrennt.
- Interfaces definieren aber (wie Klassen) einen eigenständigen **Typ**.
- Interfaces können somit **überall** wo Klassen (oder elementare Typen) deklariert werden, **ebenfalls** genutzt werden!
  - Lokale Variablen, Parameter, Attribute etc.
- Die Implementation eines Interface **zwingt** die Klassen zur Implementation aller darin enthaltenen Methoden.
  - Alternative: **abstract** definieren → abstrakte Klasse
- Oft werden Interfaces verwendet, um Teile zu separieren, die nur **lose gekoppelt** sein sollen, ohne weitere Gemeinsamkeit.

# Interfaces - Namensgebung

- Es gibt sehr verschiedene Konzepte zur Namensgebung von Schnittstellen, keines davon ist aber absolut verbindlich.
  - Empfehlung: Einheitlichkeit ist in einem Projekt wichtiger als die jeweils konkrete Konvention!
- Variante **1**: «**able**»-Postfix an (fachlichen) Namen anhängen.
  - Beispiele: **Iterable**, **Printable**, **Serializable**
- Variante **2**: «**Interface**»-Postfix an fachlichen Namen anhängen.
  - Beispiele: **PrintInterface**, **EventInterface**, etc.
- Variante **3**: Man verwendet **direkt** den fachlichen Namen.
  - Beispiel: **Iterator**, **Order** (→ Implementationen müssen dann einen spezifischeren Namen bekommen).
- Variante **4**: «**I**»-Prefix vor den fachlichen Namen (C# entlehnt).

# Interface - Empfehlungen

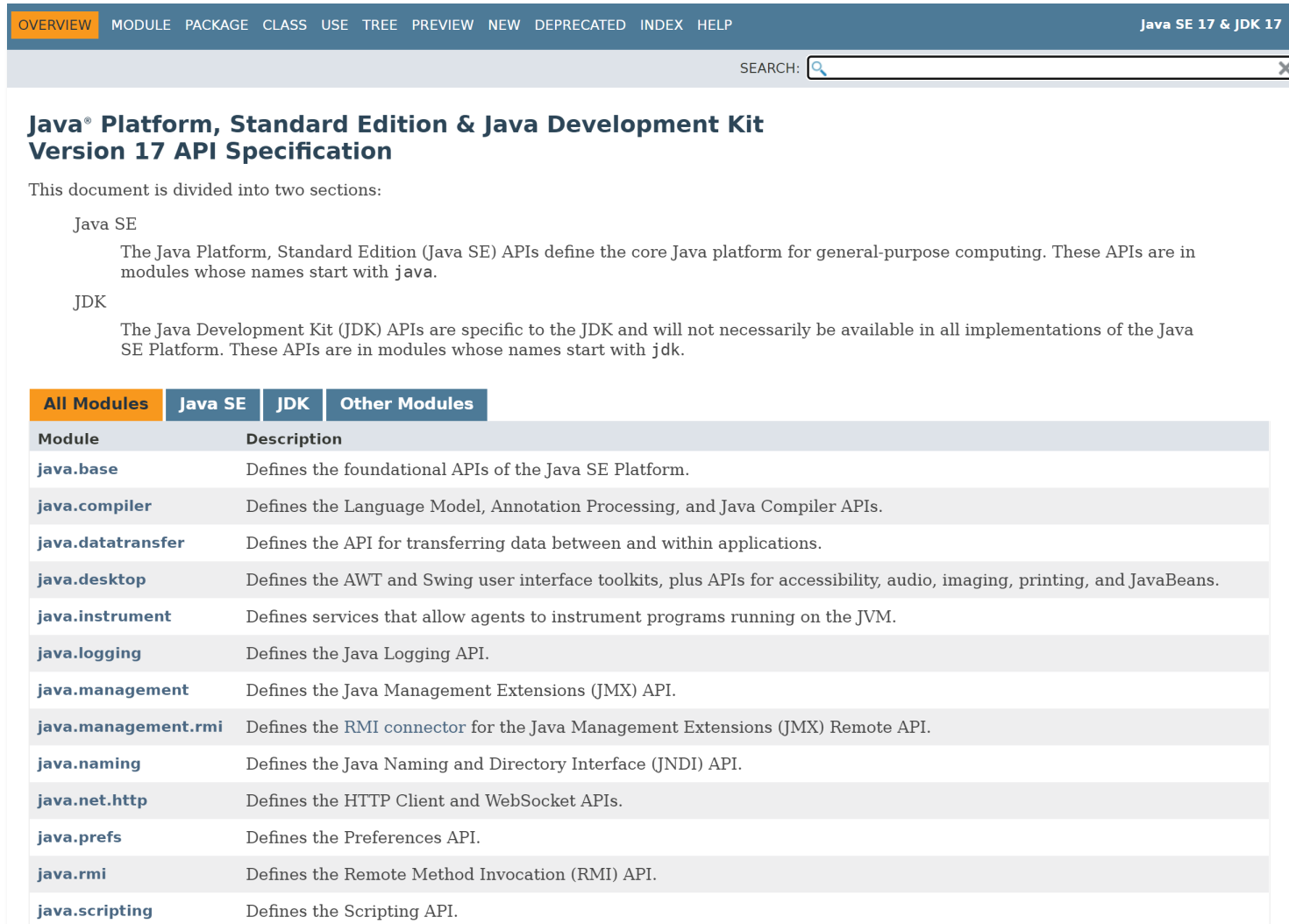


- Interfaces sind ein **sehr elegantes Konzept** und extrem nützlich zur Entkopplung von Implementationsklassen (→ Polymorphie).
- Im ersten Schritt provoziert die saubere Definition und die notwendige Dokumentation eines Interfaces zwar zusätzlichen Aufwand, dieser zahlt sich aber sehr schnell aus.
- Es gibt sogar den konsequenten Ansatz «Design by Interfaces».
- Darum: **Nutzen Sie Interfaces!**  
Erstellen Sie lieber ein Interface zu viel als zu wenig. Sie schaffen damit die Möglichkeit, verschiedene Implementationen mit minimalen Codeänderungen austauschen zu können.

# JavaDoc

# JavaDoc – Java API

## ■ Bekanntestes Beispiel: Dokumentation zu Java selber!



**Java® Platform, Standard Edition & Java Development Kit Version 17 API Specification**

This document is divided into two sections:

**Java SE**

The Java Platform, Standard Edition (Java SE) APIs define the core Java platform for general-purpose computing. These APIs are in modules whose names start with `java`.

**JDK**

The Java Development Kit (JDK) APIs are specific to the JDK and will not necessarily be available in all implementations of the Java SE Platform. These APIs are in modules whose names start with `jdk`.

All Modules	Java SE	JDK	Other Modules
Module	Description		
<code>java.base</code>	Defines the foundational APIs of the Java SE Platform.		
<code>java.compiler</code>	Defines the Language Model, Annotation Processing, and Java Compiler APIs.		
<code>java.datatransfer</code>	Defines the API for transferring data between and within applications.		
<code>java.desktop</code>	Defines the AWT and Swing user interface toolkits, plus APIs for accessibility, audio, imaging, printing, and JavaBeans.		
<code>java.instrument</code>	Defines services that allow agents to instrument programs running on the JVM.		
<code>java.logging</code>	Defines the Java Logging API.		
<code>java.management</code>	Defines the Java Management Extensions (JMX) API.		
<code>java.management.rmi</code>	Defines the RMI connector for the Java Management Extensions (JMX) Remote API.		
<code>java.naming</code>	Defines the Java Naming and Directory Interface (JNDI) API.		
<code>java.net.http</code>	Defines the HTTP Client and WebSocket APIs.		
<code>java.prefs</code>	Defines the Preferences API.		
<code>java.rmi</code>	Defines the Remote Method Invocation (RMI) API.		
<code>java.scripting</code>	Defines the Scripting API.		

# JavaDoc – Was soll/kann man dokumentieren?

- Folgende Elemente können dokumentiert werden:
  - Packages, Interfaces, Klassen, Methoden und Attribute.
- Es ist selten sinnvoll, absolut alles zu dokumentieren!
  - Kundenorientiert dokumentieren → für die Nutzer\*in.
- Absolut Pflicht: **Interfaces und deren Methoden**
  - Interfaces beschreiben den «Vertrag» und stellen, da sie mehrfach Implementiert werden können, Multiplikatoren dar!
- **Öffentliche Klassen und Methoden** (die von Dritten verwendet werden können) sind in der Regel auch zu dokumentieren.
- Potenziell weniger Sinn macht es, z.B. für private Attribute in privaten Klassen – aber es ist **nie** verboten!



# JavaDoc – Quellcode (Beispiel)

```
/**
 * Returns a string representation of the object. In general, the
 * {@code toString} method returns a string that
 * "textually represents" this object. The result
 * should be a concise but informative representation
 * suitable for a person to read.
 * It is recommended that all subclasses override this method.
 * <p>
 * The {@code toString} method for class {@code Object}
 * returns a string consisting of the name of the class of which the
 * object is an instance, the at-sign character '{@code @}', and
 * the unsigned hexadecimal representation of the hash code of the
 * object. In other words, this method returns a string equal to the
 * value of:
 * <blockquote><pre>
 * getClass().getName() + '@' + Integer.toHexString(hashCode())
 * </pre></blockquote>
 *
 * @return a string representation of the object
 */
public String toString() {
    return getClass().getName() + "@" + Integer.toHexString(hashCode());
}
```

## Markierung für Javadoc

Javadoc-Kommentare unterscheiden sich von «normalen» Inline-Kommentarblöcken durch den **Doppelstern** am Anfang des Blocks!

## Javadoc-Tags

Zur Auszeichnung von Attributen, Klassen, Parametern, Rückgabewerten etc. gibt es spezielle Javadoc-Tags mit dem @-Zeichen!

# JavaDoc – Empfehlung / Regeln



- Einfache Regel: Dokumentieren Sie das, was Sie selber zum schnelleren Verständnis des Codes (Interface/Klasse/Methode etc.) lesen möchten, wenn dieser von Dritten wäre!
- Achten Sie auf **kurze, prägnante** Kommentare. **Keine** Romane! Bringen Sie es sprichwörtlichen **auf den Punkt**.
- «Erster-Satz-und-Punkt»-Regel:  
Achten Sie darauf, dass der erste Satz eines Kommentarblocks immer kurz und prägnant ist und mit einem Punkt abschliesst!
  - Erster Satz wird von der JavaDoc z.B. in Kurzübersichten und Verzeichnissen einzeln extrahiert.
  - Nebeneffekt: Man konzentriert sich auf das Wesentliche.
- Verzichten Sie wenn möglich auf HTML-Fragmente.

# Zusammenfassung

- Begriffe Abstraktion und Modularisierung vertieft.
- Abstrakte Klassen als verstärkte Abstraktion.
- Interfaces als Abstraktion von Rollen.
- Implementation (Verwendung) von Interfaces.
- Sinn und Zweck der JavaDoc.



**Fragen?**

Fragen bitte im  
**ILIAS-Forum**

