Programmierlogik

Phil Szalay

19.05. - 28.05.2025

Ablauf

PHIL SZALAY

Über mich

- **Phil Szalay (30 Jahre)**
- Softwareentwickler und KI-Experte
- Selbstständig seit 2022
- Aufgewachsen in Herrenberg (Baden-Württemberg)
- Was ich mag: Gute Software, Fußball, Bergsteigen, (elektronische) Musik
- Links:
  `https://www.phil-szalay.de/`,
  `https://github.com/philszalay`,
  `https://www.linkedin.com/in/phil-szalay-2b5b36205/`

PHIL SZALAY

Laufbahn

- → **2014 - 2021:** Studium Informatik (Bachelor + Master) in Tübingen und Hamburg
- → **2017 - 2018:** Werkstudent bei Lufthansa Technik AG
- → **2018 - 2023:** Minubo GmbH
- → **2023 - heute:** Selbstständigkeit

PHIL SZALAY

Expertise

Programmiersprachen: Java, Python, Typescript, Javascript

Technologien: KI (LLMs, Machine Learning, Agents, RAG), Datenbanken (SQL, MongoDB), REST, Cloud (Azure, AWS, Google CLoud Platform)

Frameworks: Spring Boot, FastAPI, SQL Alchemy, Angular, React, Three.js

PHIL SZALAY

Beispiele

- → **Beyond the Loop:** `https://v2.beyondtheloop.ai/`
- → **Inovisco Company Manager**: `https://company-manager.inovisco.com/`
- → **Osmo Fassadenkonfigurator:** `https://www.fassadenkonfigurator.osmo.de/`
- → **Christian Horrer Portfolio Website:** `https://christian-horrer.com/`

Ziele

Was wollen wir in den kommenden 2 Wochen zusammen erreichen?

- Verständnis grundlegender Programmierkonzepte
- Einführung in die Objektorientierung
- Überblick über die klassischen Softwaretechnik-Themen
- Praktische Anwendung in Übungen
- Gemeinsam lernen und Spaß haben

PHIL SZALAY

Was mir wichtig ist

- → Offener Umgang
- → Lehren/Lernen auf Augenhöhe
- → Feedback
- → Eigenständiges Lernen
- → Spaß haben (keine Frustration)

PHIL SZALAY

Kennenlernen

- → Name
- → Vorkenntnisse
- → Warum seid ihr hier?

Checkliste

→ Täglicher Ablauf und Mittagspause

→ Input von eurer Seite?

→ Accounts & Installationen

- Entwicklungsumgebung (z.B. VS Code)
- Java 21 (SDK)
- Python3.9
- Git
- Github Account

- Github Repository Clonen

PHIL SZALAY

Kapitel 1: Vom Problem zum Programm

Was bedeutet “Programmieren?”

Definition (Programmieren)

Programmieren, auch bekannt als Codieren, ist der Prozess der Erstellung von Anweisungen, die ein Computer ausführt, um Aufgaben zu erledigen.

Was bedeutet “Programmieren?”

Definition (Programmieren)

Programmieren, auch bekannt als Codieren, ist der Prozess der Erstellung von Anweisungen, die ein Computer ausführt, um Aufgaben zu erledigen.

→ Oder einfach: Das Erstellen von Computer-Programmen.

Was bedeutet “Programmieren?”

Herausforderung: Ablauf/Problem aus der echten Welt soll in Code abgebildet werden.

Was bedeutet “Programmieren?”

Herausforderung: Ablauf/Problem aus der echten Welt soll in Code abgebildet werden.

Beispiele:

Was bedeutet “Programmieren?”

Herausforderung: Ablauf/Problem aus der echten Welt soll in Code abgebildet werden.

Beispiele:

- **Online-Bestellungssystem**:
  - Erfassen von Bestellungen
  - Verwaltung von Warenkörben
  - Abwicklung von Zahlungen

Was bedeutet “Programmieren?”

Herausforderung: Ablauf/Problem aus der echten Welt soll in Code abgebildet werden.

Beispiele:

- **Online-Bestellungssystem**:
  - Erfassen von Bestellungen
  - Verwaltung von Warenkörben
  - Abwicklung von Zahlungen
- **Verkehrssteuerungssystem**:
  - Echtzeitüberwachung von Verkehrsdaten
  - Steuerung von Ampelsystemen basierend auf Verkehrsaufkommen

PHIL SZALAY

Was bedeutet “Programmieren?”

Herausforderung: Ablauf/Problem aus der echten Welt soll in Code abgebildet werden.

Beispiele:

- **Online-Bestellungssystem**:
  - Erfassen von Bestellungen
  - Verwaltung von Warenkörben
  - Abwicklung von Zahlungen
- **Verkehrssteuerungssystem**:
  - Echtzeitüberwachung von Verkehrsdaten
  - Steuerung von Ampelsystemen basierend auf Verkehrsaufkommen
- **Kalenderanwendung**:
  - Erstellen und Verwalten von Terminen
  - Synchronisierung mit anderen Geräten

PHIL SZALAY

Was bedeutet “Programmieren?”

Herausforderung: Ablauf/Problem aus der echten Welt soll in Code abgebildet werden.

Beispiele:

- **Online-Bestellungssystem**:
  - Erfassen von Bestellungen
  - Verwaltung von Warenkörben
  - Abwicklung von Zahlungen
- **Verkehrssteuerungssystem**:
  - Echtzeitüberwachung von Verkehrsdaten
  - Steuerung von Ampelsystemen basierend auf Verkehrsaufkommen
- **Kalenderanwendung**:
  - Erstellen und Verwalten von Terminen
  - Synchronisierung mit anderen Geräten
- **Maschinelles Lernen Modelle**:
  - Sammeln und Vorverarbeiten von Daten
  - Training von Algorithmen zur Mustererkennung

Wichtige Eigenschaften von Entwickler:innen:

Wichtige Eigenschaften von Entwickler:innen:

- **Empathie und Nutzerorientierung**:
  - Fähigkeit, Nutzerbedürfnisse zu verstehen und daraus resultierende Anforderungen zu erkennen

Wichtige Eigenschaften von Entwickler:innen:

- **Empathie und Nutzerorientierung**:
  - Fähigkeit, Nutzerbedürfnisse zu verstehen und daraus resultierende Anforderungen zu erkennen
- **Analytisches Denken**:
  - Einsatz von Logik, um Probleme zu strukturieren und lösbare Teile zu identifizieren

Wichtige Eigenschaften von Entwickler:innen:

- **Empathie und Nutzerorientierung**:
  - Fähigkeit, Nutzerbedürfnisse zu verstehen und daraus resultierende Anforderungen zu erkennen
- **Analytisches Denken**:
  - Einsatz von Logik, um Probleme zu strukturieren und lösbare Teile zu identifizieren
- **Kreativität**:
  - Entwicklung innovativer Lösungen und Ansätze zur Problemlösung

Wichtige Eigenschaften von Entwickler:innen:

- **Empathie und Nutzerorientierung**:
  - Fähigkeit, Nutzerbedürfnisse zu verstehen und daraus resultierende Anforderungen zu erkennen
- **Analytisches Denken**:
  - Einsatz von Logik, um Probleme zu strukturieren und lösbare Teile zu identifizieren
- **Kreativität**:
  - Entwicklung innovativer Lösungen und Ansätze zur Problemlösung
- **Detailorientierung**:
  - Präzision beim Codieren, um Fehler zu vermeiden und Qualität zu sichern

Wichtige Eigenschaften von Entwickler:innen:

- **Empathie und Nutzerorientierung**:
  - Fähigkeit, Nutzerbedürfnisse zu verstehen und daraus resultierende Anforderungen zu erkennen
- **Analytisches Denken**:
  - Einsatz von Logik, um Probleme zu strukturieren und lösbare Teile zu identifizieren
- **Kreativität**:
  - Entwicklung innovativer Lösungen und Ansätze zur Problemlösung
- **Detailorientierung**:
  - Präzision beim Codieren, um Fehler zu vermeiden und Qualität zu sichern
- **Geduld und Ausdauer**:
  - Durchhaltevermögen bei der Fehlersuche und bei komplexen Projekten

Wichtige Eigenschaften von Entwickler:innen:

- **Empathie und Nutzerorientierung**:
  - Fähigkeit, Nutzerbedürfnisse zu verstehen und daraus resultierende Anforderungen zu erkennen
- **Analytisches Denken**:
  - Einsatz von Logik, um Probleme zu strukturieren und lösbare Teile zu identifizieren
- **Kreativität**:
  - Entwicklung innovativer Lösungen und Ansätze zur Problemlösung
- **Detailorientierung**:
  - Präzision beim Codieren, um Fehler zu vermeiden und Qualität zu sichern
- **Geduld und Ausdauer**:
  - Durchhaltevermögen bei der Fehlersuche und bei komplexen Projekten
- **Teamfähigkeit und Kommunikation**:
  - Effektive Zusammenarbeit im Team und klare Kommunikation mit Stakeholdern

PHIL SZALAY

Wichtige Eigenschaften von Entwickler:innen:

- **Empathie und Nutzerorientierung**:
  - Fähigkeit, Nutzerbedürfnisse zu verstehen und daraus resultierende Anforderungen zu erkennen
- **Analytisches Denken**:
  - Einsatz von Logik, um Probleme zu strukturieren und lösbare Teile zu identifizieren
- **Kreativität**:
  - Entwicklung innovativer Lösungen und Ansätze zur Problemlösung
- **Detailorientierung**:
  - Präzision beim Codieren, um Fehler zu vermeiden und Qualität zu sichern
- **Geduld und Ausdauer**:
  - Durchhaltevermögen bei der Fehlersuche und bei komplexen Projekten
- **Teamfähigkeit und Kommunikation**:
  - Effektive Zusammenarbeit im Team und klare Kommunikation mit Stakeholdern
- **Lernbereitschaft**:
  - Offenheit für neue Technologien und kontinuierliches Lernen

PHIL SZALAY

Was ist ISO/IEC 25010?

- International anerkannter Standard für Softwarequalität
- Umfasst verschiedene Qualitätsmerkmale für die Bewertung von Software

Was ist ISO/IEC 25010?

- International anerkannter Standard für Softwarequalität
- Umfasst verschiedene Qualitätsmerkmale für die Bewertung von Software

Ziel: Sicherstellung, dass Software den Anforderungen der Nutzer entspricht, zuverlässig funktioniert und ohne Probleme weiterentwickelt und gewartet werden kann.

Abbildung 1: Visualisierung von ISO/IEC 25010[1]

[1] Quelle: https://xitaso.com/die-richtigen-kriterien-fuer-software-qualitaet/

Funktionalität: Erfüllt die Nutzeranforderungen

Funktionalität: Fähigkeit einer Software, die geforderten Funktionen gemäß spezifizierten Anforderungen zu erfüllen.

- Hauptmerkmale:
  - Funktionale Eignung
  - Vollständigkeit
  - Korrektheit

Funktionalität: Erfüllt die Nutzeranforderungen

Funktionalität: Fähigkeit einer Software, die geforderten Funktionen gemäß spezifizierten Anforderungen zu erfüllen.

- Hauptmerkmale:
  - Funktionale Eignung
  - Vollständigkeit
  - Korrektheit

Ziel: Bereitstellung von Funktionen, die den Bedürfnissen der Nutzer gerecht werden, und sicherstellen, dass alle erforderlichen Aufgaben der Software korrekt ausgeführt werden.

PHIL SZALAY

Verlässlichkeit: Stabil und fehlerarm im Betrieb

Verlässlichkeit: Fähigkeit der Software, unter festgelegten Bedingungen zu funktionieren, ohne Fehlfunktionen zu verursachen.

- Hauptmerkmale:
  - Verfügbarkeit
  - Fehlertoleranz
  - Wiederherstellbarkeit

PHIL SZALAY

Verlässlichkeit: Stabil und fehlerarm im Betrieb

Verlässlichkeit: Fähigkeit der Software, unter festgelegten Bedingungen zu funktionieren, ohne Fehlfunktionen zu verursachen.

- Hauptmerkmale:
  - Verfügbarkeit
  - Fehlertoleranz
  - Wiederherstellbarkeit

Ziel: Sicherstellung eines stabilen Betriebs der Software über längere Zeiträume hinweg, selbst bei Fehlern oder Systemausfällen.

PHIL SZALAY

Usability: Intuitiv und benutzerfreundlich

Usability/Benutzbarkeit: Die Bequemlichkeit und Effizienz, mit der Nutzer die Software nutzen und lernen können.

- Hauptmerkmale:
  - Erlernbarkeit
  - Bedienbarkeit
  - Benutzerzufriedenheit

Usability: Intuitiv und benutzerfreundlich

Usability/Benutzbarkeit: Die Bequemlichkeit und Effizienz, mit der Nutzer die Software nutzen und lernen können.

- Hauptmerkmale:
  - Erlernbarkeit
  - Bedienbarkeit
  - Benutzerzufriedenheit

Ziel: Entwicklung einer Software mit einfacher Bedienung und klarer Benutzerführung zur Maximierung der Nutzerzufriedenheit.

PHIL SZALAY

Performance: Optimierte Ressourcennutzung

Performance: Fähigkeit der Software, mit minimalem Zeitaufwand und Ressourcenverbrauch zu arbeiten.

- Hauptmerkmale:
  - Zeitverhalten
  - Ressourcenverbrauch

Performance: Optimierte Ressourcennutzung

Performance: Fähigkeit der Software, mit minimalem Zeitaufwand und Ressourcenverbrauch zu arbeiten.

- Hauptmerkmale:
  - Zeitverhalten
  - Ressourcenverbrauch

Ziel: Bereitstellung schneller Antwortzeiten und geringer Systemauslastung, um die Gesamteffizienz der Software zu verbessern.

PHIL SZALAY

Wartbarkeit: Leicht anpassbar und erweiterbar

Wartbarkeit: Möglichkeit der Software, Änderungen zu integrieren, ohne die bestehende Funktionalität zu beeinträchtigen.

- Hauptmerkmale:
  - Modularität
  - Einfachheit bei der Anpassung

PHIL SZALAY

Wartbarkeit: Leicht anpassbar und erweiterbar

Wartbarkeit: Möglichkeit der Software, Änderungen zu integrieren, ohne die bestehende Funktionalität zu beeinträchtigen.

- Hauptmerkmale:
  - Modularität
  - Einfachheit bei der Anpassung

Ziel: Fähigkeit der Software, sich flexibel an neue Anforderungen und Technologien anzupassen, indem Erweiterungen schnell und problemlos integriert werden können.

PHIL SZALAY

Portierbarkeit: Funktioniert plattformübergreifend

Portierbarkeit: Fähigkeit der Software, auf verschiedenen Plattformen und Umgebungen ohne großen Aufwand zu laufen.

- Hauptmerkmale:
  - Anpassungsfähigkeit
  - Installierbarkeit
  - Austauschbarkeit

Portierbarkeit: Funktioniert plattformübergreifend

Portierbarkeit: Fähigkeit der Software, auf verschiedenen Plattformen und Umgebungen ohne großen Aufwand zu laufen.

- Hauptmerkmale:
  - Anpassungsfähigkeit
  - Installierbarkeit
  - Austauschbarkeit

Ziel: Entwicklung einer Software, die unabhängig von spezifischen Hardware- oder Softwareplattformen betrieben werden kann.

Sicherheit: Schutz vor Bedrohungen und Datenmissbrauch

Sicherheit: Fähigkeit der Software, bedrohliche Gefahren und unerlaubten Zugriff zu verhindern sowie den Schutz vertraulicher Daten sicherzustellen.

- Hauptmerkmale:
  - Vertraulichkeit
  - Integrität
  - Authentizität
  - Zurechenbarkeit

PHIL SZALAY

Sicherheit: Schutz vor Bedrohungen und Datenmissbrauch

Sicherheit: Fähigkeit der Software, bedrohliche Gefahren und unerlaubten Zugriff zu verhindern sowie den Schutz vertraulicher Daten sicherzustellen.

- Hauptmerkmale:
  - Vertraulichkeit
  - Integrität
  - Authentizität
  - Zurechenbarkeit

Ziel: Gewährleistung eines hohen Sicherheitsniveaus, um die Integrität und Vertraulichkeit der Daten sowie der umfangreiche Schutz gegen interne und externe Bedrohungen sicherzustellen.

PHIL SZALAY

Kompatibilität: Nahtlose Interaktion mit anderen Systemen

Definition: Fähigkeit der Software, sinnvoll und effizient mit anderen Systemen und Anwendungen zu kooperieren, um Daten auszutauschen und Funktionen zu teilen.

- Hauptmerkmale:
  - Koexistenz
  - Interoperabilität

PHIL SZALAY

Kompatibilität: Nahtlose Interaktion mit anderen Systemen

Definition: Fähigkeit der Software, sinnvoll und effizient mit anderen Systemen und Anwendungen zu kooperieren, um Daten auszutauschen und Funktionen zu teilen.

- Hauptmerkmale:
  - Koexistenz
  - Interoperabilität

Ziel: Sicherstellung, dass die Software problemlos in bestehende Systemlandschaften integriert werden kann und eine reibungslose Zusammenarbeit mit anderen Anwendungen gewährleistet ist.

PHIL SZALAY

Übung: Anwendung der ISO/IEC 25010 Qualitätsmerkmale

Übung: Verstehen und Anwenden der ISO/IEC 25010 Qualitätsmerkmale auf eine reale Softwareanwendung.

Übung: Anwendung der ISO/IEC 25010 Qualitätsmerkmale

Übung: Verstehen und Anwenden der ISO/IEC 25010 Qualitätsmerkmale auf eine reale Softwareanwendung.

Beispielsoftware: *EasyStay* - globale Hotelbuchungsplattform

- Funktionen:
  - Suche und Vergleich von Hotelzimmerpreisen
  - Bewertungen und Rezensionen von Nutzern
  - Echtzeitbuchungsbestätigung und -verwaltung
  - Sonderangebote und Rabattgutscheine

Übung: Anwendung der ISO/IEC 25010 Qualitätsmerkmale

Übung: Verstehen und Anwenden der ISO/IEC 25010 Qualitätsmerkmale auf eine reale Softwareanwendung.

Beispielsoftware: *EasyStay* - globale Hotelbuchungsplattform

- Funktionen:
  - Suche und Vergleich von Hotelzimmerpreisen
  - Bewertungen und Rezensionen von Nutzern
  - Echtzeitbuchungsbestätigung und -verwaltung
  - Sonderangebote und Rabattgutscheine

Aufgabe: Identifiziere für jeden der folgenden Qualitätsmerkmale konkrete Anwendungen oder Anwendungsbereiche innerhalb der Hotelbuchungsplattform:

1. Funktionalität
2. Zuverlässigkeit
3. Benutzbarkeit
4. Effizienz
5. Änderbarkeit
6. Übertragbarkeit
7. Sicherheit
8. Kompatibilität

PHIL SZALAY

Softwareentwicklungsprozess - Wie läuft ein Softwareprojekt ab?

1. **Analyse** : Identifizierung von Nutzerbedürfnissen und Systemanforderungen.
2. **Entwurf** : Planung der Softwarearchitektur und ihrer Komponenten.
3. **Programmierung** : Umsetzung der geplanten Funktionalitäten.
4. **Testen** : Sicherstellung der Fehlerfreiheit und Funktionalität.
5. **Wartung** : Updates und Anpassungen basierend auf Nutzerfeedback und neuen Anforderungen.

Ablauf vom Problem zur Lösung

Softwareentwicklungsprozess - Wie läuft ein Softwareprojekt ab?

1. **Analyse** : Identifizierung von Nutzerbedürfnissen und Systemanforderungen.
2. **Entwurf** : Planung der Softwarearchitektur und ihrer Komponenten.
3. **Programmierung** : Umsetzung der geplanten Funktionalitäten.
4. **Testen** : Sicherstellung der Fehlerfreiheit und Funktionalität.
5. **Wartung** : Updates und Anpassungen basierend auf Nutzerfeedback und neuen Anforderungen.

Ziel: Systematische Erstellung und Verbesserung von Software, um bestmögliche Ergebnisse für Anwender und Unternehmen zu erzielen.

PHIL SZALAY

Vorstellung Beispielprojekt: *StuddyBuddy*

Idee: Umsetzung aller 5 Phasen des klassischen Softwareentwicklungsprozesses anhand des Beispielprojekts *StuddyBuddy*.

Kapitel 2: Analysephase

Funktionale und nicht-funktionale Anforderungen

Analysephase: Tiefgehendes Verständnis der Aufgabenstellung

- **Funktionale Anforderungen** : Was soll die Software leisten?
  - Beispiele: Benutzerregistrierung, Suchfunktion, Bezahlfunktion
- **Nicht-funktionale Anforderungen** : Qualitätskriterien wie Performanz und Sicherheit
  - Beispiel: Programmiersprache, zeitlicher Rahmen, Anwendungstyp (Web oder Mobile)

PHIL SZALAY

Funktionale und nicht-funktionale Anforderungen

Analysephase: Tiefgehendes Verständnis der Aufgabenstellung

- **Funktionale Anforderungen** : Was soll die Software leisten?
  - Beispiele: Benutzerregistrierung, Suchfunktion, Bezahlfunktion
- **Nicht-funktionale Anforderungen** : Qualitätskriterien wie Performanz und Sicherheit
  - Beispiel: Programmiersprache, zeitlicher Rahmen, Anwendungstyp (Web oder Mobile)

Output: Dokumentation der Anforderungen mit klar definierten Zielen und Prioritäten.

PHIL SZALAY

Übung: Anforderungsanalyse

Übung: Anforderungsanalyse anhand der Projektbeschreibung von *StuddyBuddy*.

Aufgabe: Identifizieren Sie möglichst viele Anforderungen (funktional oder nicht-funktional).

Exkurs: Grundlagen der Programmierung

Was sind Datenstrukturen?

Definition (Datenstrukturen)

Datenstrukturen sind spezielle Formen, in denen Daten im Speicher abgelegt und organisiert werden – damit Programme effizient mit ihnen arbeiten können.

Was sind Datenstrukturen?

Definition (Datenstrukturen)

Datenstrukturen sind spezielle Formen, in denen Daten im Speicher abgelegt und organisiert werden – damit Programme effizient mit ihnen arbeiten können.

Warum sind sie wichtig?

- Sie bestimmen, **wie schnell** wir Daten suchen, einfügen oder löschen können.
- Sie helfen, komplexe Probleme logisch und strukturiert zu lösen.
- Sie ermöglichen es, Daten **übersichtlich** und **wiederverwendbar** zu verwalten.

PHIL SZALAY

Was sind Datenstrukturen?

Definition (Datenstrukturen)

Datenstrukturen sind spezielle Formen, in denen Daten im Speicher abgelegt und organisiert werden – damit Programme effizient mit ihnen arbeiten können.

Warum sind sie wichtig?

- Sie bestimmen, **wie schnell** wir Daten suchen, einfügen oder löschen können.
- Sie helfen, komplexe Probleme logisch und strukturiert zu lösen.
- Sie ermöglichen es, Daten **übersichtlich** und **wiederverwendbar** zu verwalten.

Wichtig: Gute Datenstrukturen machen Programme nicht nur korrekt, sondern auch schnell und elegant.

PHIL SZALAY

Einfache Datentypen in Java

Primitive Typen:

- `int` – Ganze Zahl (z.B. `42`)
- `float`, `double` – Kommazahlen (z.B. `3.14`)
- `boolean` – Wahrheitswert (`true` oder `false`)
- `char` – Ein einzelnes Zeichen (z.B. `'A'`)

Nicht-primitive Typen:

- `String` – Zeichenkette (z.B. `"Hallo Welt"`)

Einfache Datentypen in Java

Primitive Typen:

- `int` – Ganze Zahl (z.B. `42`)
- `float`, `double` – Kommazahlen (z.B. `3.14`)
- `boolean` – Wahrheitswert (`true` oder `false`)
- `char` – Ein einzelnes Zeichen (z.B. `'A'`)

Nicht-primitive Typen:

- `String` – Zeichenkette (z.B. `"Hallo Welt"`)

Beispiele:

```
int alter = 30;
float temperatur = 21.5f;
boolean istAngemeldet = true;
String name = "Lisa";
```

Ein Array: Sammlung von Werten desselben Typs, feste Länge.

Beispiel:

```
int[] zahlen = {1, 2, 3, 4, 5};
System.out.println(zahlen[0]); // Ausgabe: 1
```

Alltagsvergleich: Ein Regal mit festen Fächern – jedes Fach ist durchnummeriert.

Arrays

Ein Array: Sammlung von Werten desselben Typs, feste Länge.

Beispiel:

```
int[] zahlen = {1, 2, 3, 4, 5};
System.out.println(zahlen[0]); // Ausgabe: 1
```

Alltagsvergleich: Ein Regal mit festen Fächern – jedes Fach ist durchnummeriert.

Hinweis:

- Indizes beginnen bei 0!
- Nachteil: Größe nicht veränderbar

PHIL SZALAY

Listen

Liste: Kann als Array mit dynamischer Größe gesehen werden → Elemente können hinzugefügt oder entfernt werden → Komplexer als Array.

Beispiel:

```
import java.util.ArrayList;

ArrayList<String> namen = new ArrayList<>();
namen.add("Anna");
namen.add("Ben");
System.out.println(namen.get(1)); // Ausgabe: Ben
```

Listen

Liste: Kann als Array mit dynamischer Größe gesehen werden → Elemente können hinzugefügt oder entfernt werden → Komplexer als Array.

Beispiel:

```
import java.util.ArrayList;

ArrayList<String> namen = new ArrayList<>();
namen.add("Anna");
namen.add("Ben");
System.out.println(namen.get(1)); // Ausgabe: Ben
```

Vergleich: Einkaufsliste – man kann beliebig viele Dinge hinzufügen oder streichen → Man muss vorher nicht die Länge der Liste kennen.

Map: Speichert Schlüssel-Wert-Paare – wie ein Wörterbuch.

Beispiel:

```
import java.util.HashMap;

HashMap<String, Integer> telefonbuch = new HashMap<>();
telefonbuch.put("Anna", 12345);
telefonbuch.put("Ben", 67890);

System.out.println(telefonbuch.get("Anna")); // 12345
```

Map: Speichert Schlüssel-Wert-Paare – wie ein Wörterbuch.

Beispiel:

```
import java.util.HashMap;

HashMap<String, Integer> telefonbuch = new HashMap<>();
telefonbuch.put("Anna", 12345);
telefonbuch.put("Ben", 67890);

System.out.println(telefonbuch.get("Anna")); // 12345
```

Alltagsvergleich: Telefonnummern nach Namen gespeichert

Übung: Java-Datentypen im echten Leben

Aufgabe: Finde zu jedem der folgenden Datentypen ein Beispiel aus dem echten Leben. *Was könnte dieser Typ in einer App oder einem Programm darstellen?*

- `int` – Ganzzahl (ohne Nachkommastellen)
- `float` oder `double` – Zahl mit Nachkommastellen
- `boolean` – True/False, Ja/Nein
- `char` – Zeiechen (Buchstabe)
- `String` –Text
- `int[]` – Array von Integers
- `float[]` – Array von Floats
- `boolean[]` – Array von Booleans
- `String[]` – Array von Strings
- `List<String>` – Liste von Strings
- `Map<String, int>` – Zuordnung von String zu Integer
- `Map<String, String>` – Zuordnung von String zu String

Java-Programm kompilieren und ausführen

Voraussetzungen:

- Java Development Kit (JDK) ist installiert
- Terminal oder Kommandozeile verfügbar
- Eine Datei mit der Endung `.java`, z. B. `DataTypesExercise.java`

1. Wechsle im Terminal in den Ordner mit der Datei:

```
cd /Users/deinName/Projekte/JavaUebung
```

2. Kompiliere die Datei:

```
javac DataTypesExercise.java
```

(Erzeugt eine Datei: `DataTypesExercise.class`)

3. Starte das Programm:

```
java DataTypesExercise
```

Hinweise:

- Der Klassenname muss genau wie der Dateiname sein (ohne `.java`)
- Beim Starten `.java` und `.class` weglassen
- Groß- und Kleinschreibung beachten!

PHIL SZALAY

Übung: Java-Datentypen im Code

Aufgabe: Lies die Kommentare in **Datantypen.java**, bearbeite jede Aufgabe (TODO).

Operationen auf Datentypen

Was kann man mit Variablen machen?

- `int`, `float`, `double`:
  - Rechnen: `+`, `-`, `*`, `/`, `%`
  - Vergleichen: `==`, `!=`, `<`, `>`, `<=`, `>=`
- `boolean`:
  - Logische Operationen: `&&` (und), `||` (oder), `!` (nicht)
  - Vergleichsoperatoren geben oft `boolean` zurück
- `char` und `String`:
  - Zeichenketten zusammenfügen: `+`
  - Länge bestimmen: `str.length()`
  - Zeichen holen: `str.charAt(0)`
- `Arrays` und `Listen`:
  - Elementzugriff über Index: `arr[0]`, `list.get(0)`
  - Länge: `arr.length`, `list.size()`
  - Elemente hinzufügen (nur Liste): `list.add(Neu")`

PHIL SZALAY

Übung: Nutze Operatoren und Methoden in Java

Aufgaben:

1. Erstelle zwei Ganzzahlen und berechne ihre Summe, Differenz und das Produkt.
2. Teile eine Zahl durch eine andere und gib das Ergebnis als `float` aus.
3. Vergleiche zwei Zahlen: Ist die eine größer als die andere?
4. Erstelle zwei `String`-Variablen (z.B. Vorname und Nachname) und füge sie zu einem vollständigen Namen zusammen.
5. Gib die Länge eines Strings aus.
6. Erstelle ein `String[]`-Array mit mindestens 3 Städtenamen und gib den zweiten Namen aus.
7. Erstelle eine `List<String>` mit mindestens 3 Lieblingsessen und füge ein weiteres hinzu.
8. Erstelle eine `Map<String, Integer>` mit Namen und Alter. Lies das Alter einer Person aus der Map.

Was sind Kontrollstrukturen?

Kontrollstrukturen bestimmen, wie der Programmablauf gesteuert wird.

- **Verzweigungen:** Bedingungen prüfen und entsprechend handeln
- **Schleifen:** Code mehrfach ausführen

Beispiel:

```
int alter = 18;

if (alter >= 18) {
    System.out.println("Du bist volljaehrig.");
} else {
    System.out.println("Du bist noch nicht volljaehrig.");
}
```

Verzweigungen – `if / else`

Syntax:

```
if (Bedingung) {
    // wenn Bedingung wahr ist
} else {
    // wenn Bedingung falsch ist
}
```

Beispiel:

```
int stunde = 10;

if (stunde < 12) {
    System.out.println("Guten Morgen!");
} else {
    System.out.println("Guten Tag!");
}
```

Mehrere Fälle unterscheiden:

```
int note = 2;

if (note == 1) {
    System.out.println("Sehr gut");
} else if (note == 2) {
    System.out.println("Gut");
} else {
    System.out.println("Verbesserung noetig");
}
```

Hinweis: Immer nur der erste passende Block wird ausgeführt.

Mehrere Fälle einfacher schreiben (z.B. bei Zahlen oder Strings):

```
int tag = 3;

switch (tag) {
    case 1:
        System.out.println("Montag");
        break;
    case 2:
        System.out.println("Dienstag");
        break;
    case 3:
        System.out.println("Mittwoch");
        break;
    default:
        System.out.println("Unbekannter Tag");
}
```

Wiederhole etwas, solange eine Bedingung wahr ist:

```
int i = 0;

while (i < 5) {
    System.out.println("i ist: " + i);
    i = i + 1;
}
```

Achte darauf: Die Schleife muss irgendwann aufhören!

Kurzform für eine Schleife mit Zähler:

```
for (int i = 0; i < 5; i = i + 1) {
    System.out.println("i ist: " + i);
}
```

Bedeutung:

- `int i = 0;` → Zähler starten
- `i < 5;` → Bedingung prüfen
- `i = i + 1;` → Zähler erhöhen

Arrays mit Schleifen durchlaufen

Beispiel: Alle Elemente eines Arrays anzeigen

```
String[] namen = {"Ali", "Lisa", "Tom"};

for (int i = 0; i < namen.length; i++) {
    System.out.println(namen[i]);
}
```

Hinweis: `.length` gibt die Anzahl der Elemente im Array zurück.

Übungsaufgaben – Kontrollstrukturen

1. Schreibe ein Programm, das überprüft, ob eine Zahl größer als 10 ist.
2. Gib zu einer Note (1–6) eine Bewertung aus (z.B. „Sehr gut“).
3. Zähle mit einer `while`-Schleife von 1 bis 5.
4. Erstelle ein Array mit 3 Farben und gib sie mit einer `for`-Schleife aus.
5. Füge eine `switch`-Verzweigung ein, die je nach Wochentag etwas anderes ausgibt.

Tipp: Probiere jede Aufgabe in einer eigenen kleinen Java-Datei aus.

PHIL SZALAY

Werteübergabe in Java

Was passiert, wenn man einer Methode einen Wert übergibt?

In Java gibt es zwei Arten, wie Werte übergeben werden:

- **Call by Value** – Bei primitiven Datentypen
- **Call by Object** – Bei Objekten (z. B. Arrays, Listen, eigene Klassen)

Achtung: Java kennt *kein echtes Call by Reference*, wie z. B. C++.

Call by Value – Primitive Werte

Bei **Call by Value** wird eine **Kopie** des Werts übergeben. Änderungen in der Methode haben keine Auswirkung auf das Original.

Beispiel:

```
public static void verdopple(int x) {
    x = x * 2;
}

public static void main(String[] args) {
    int zahl = 5;
    verdopple(zahl);
    System.out.println(zahl); // Ausgabe: 5
}
```

Erklärung: Die Methode bekommt eine Kopie von `zahl`.

Call by Object – Referenz wird kopiert

Bei Objekten wird die Referenz übergeben – nicht das ganze Objekt.

Beispiel:

```
public static void setzeErstenWert(int[] arr) {
    arr[0] = 99;
}

public static void main(String[] args) {
    int[] zahlen = {1, 2, 3};
    setzeErstenWert(zahlen);
    System.out.println(zahlen[0]); // Ausgabe: 99
}
```

Erklärung: Beide Referenzen zeigen auf dasselbe Array.

Achtung: Referenz ist kopiert

Die Referenz selbst kann in der Methode geändert werden – aber das Original bleibt unverändert.

Beispiel:

```
public static void neuesArray(int[] arr) {
    arr = new int[] {10, 20, 30};
}

public static void main(String[] args) {
    int[] zahlen = {1, 2, 3};
    neuesArray(zahlen);
    System.out.println(zahlen[0]); // Ausgabe: 1
}
```

Warum? Die Methode hat nur eine Kopie der Referenz.

Call by Reference? Nicht in Java.

Java verwendet **immer Call by Value** – auch bei Objekten.
Aber: Wenn man Objekte übergibt, wird **die Referenz (Adresse)** kopiert.

Das führt zu einem Verhalten, das wie *Call by Reference* aussieht.

Wichtig: Die Referenz selbst wird kopiert – aber beide zeigen auf dasselbe Objekt.

Zusammenfassung – Werteübergabe

- Java verwendet **immer Call by Value**.
- Bei primitiven Datentypen (`int`, `float` usw.): Wert wird kopiert.
- Bei Objekten: **Referenz** wird kopiert → beide zeigen auf dasselbe Objekt.
- Änderungen an Objekten wirken sich aus – Änderungen an der Referenz nicht.

Übung: Java-Werteübergabe im Code

Aufgabe: Lies die Kommentare in **Werteübergabe.java**, bearbeite jede Aufgabe.

Problemstellung: Wiederholung und Berechnung

- In der Programmierung begegnen wir oft Aufgaben, bei denen wir Berechnungen mehrfach wiederholen müssen.
- Beispiel: Berechnung einer mathematischen Folge oder eines Produkts mehrerer Zahlen.
- Zwei häufige Herangehensweisen dafür: **Iteration** und **Rekursion**.
- Ziel: Verstehen, wie diese Ansätze funktionieren und wann man welchen einsetzen sollte.

PHIL SZALAY

Das Beispielproblem: Fakultät

- Die Fakultät einer Zahl n wird geschrieben als $n!$.
- Definition:

$$n! = 1 \times 2 \times 3 \times \ldots \times n$$

- Spezialfall:

$$0! = 1$$

- Beispiel:

$$5! = 1 \times 2 \times 3 \times 4 \times 5 = 120$$

- Fakultät ist ein gutes Beispiel, um Wiederholung (Schleifen oder Selbstaufrufe) zu üben.

Iteration: Was ist das?

- Iteration bedeutet, dass wir eine Anweisung oder einen Codeblock mehrfach ausführen – meist mit `for` oder `while` Schleifen.
- Man zählt oder wiederholt systematisch Schritte.
- Beispiel: Fakultät von n berechnen, indem man alle Zahlen von 1 bis n multipliziert.

Iteration: Beispiel in Java (Fakultät)

```java
// Fakultaet mit Schleife berechnen
public static int fakultaetIterativ(int n) {
    int ergebnis = 1;
    for (int i = 1; i <= n; i++) {
        ergebnis = ergebnis * i; // Multiplikation in jedem Schritt
    }
    return ergebnis;
}
```

Iteration: Vor- und Nachteile

- **Vorteile:**
  - Einfach und effizient.
  - Braucht wenig Speicher (kein Aufruf-Stack).
  - Gut für einfache Wiederholungen.
- **Nachteile:**
  - Manchmal unübersichtlich bei sehr komplexen Problemen.
  - Nicht intuitiv bei rekursiv definierten Problemen (z.B. Bäume).

PHIL SZALAY

Rekursion: Was ist das?

- Eine Methode ruft sich selbst auf, um ein Problem schrittweise zu lösen.
- Wichtig ist die **Abbruchbedingung** – damit die Methode nicht unendlich oft aufgerufen wird.
- Beispiel: Fakultät von n berechnet man durch $n \times (n-1)!$.
- Dabei wird die Methode immer wieder mit kleineren Werten aufgerufen.

Rekursion: Beispiel in Java (Fakultät)

```java
// Fakultaet mit Rekursion berechnen
public static int fakultaetRekursiv(int n) {
    if (n == 0) {
        return 1; // Abbruchbedingung
    } else {
        return n * fakultaetRekursiv(n - 1);
    }
}
```

Rekursion: Vor- und Nachteile

- **Vorteile:**
  - Elegante, verständliche Lösungen für manche Probleme.
  - Natürlich passend für rekursiv definierte Strukturen (Bäume, Folgen).
- **Nachteile:**
  - Mehr Speicherbedarf durch Aufruf-Stack.
  - Gefahr von Stack Overflow bei zu tiefen Aufrufen.
  - Manchmal langsamer als Iteration.

PHIL SZALAY

Aufgabe: Berechne die Fakultät

- Schreibe in IterationVsRekursion.java zwei Methoden in Java:
  - `fakultaetIterativ(int n)` mit einer `for`-Schleife.
  - `fakultaetRekursiv(int n)` mit rekursivem Aufruf.
- Teste deine Methoden mit verschiedenen Zahlen, z.B. $0, 1, 5, 10$.
- Achte darauf, dass deine rekursive Methode eine Abbruchbedingung hat!

Die Fibonacci-Folge: Einführung

- Die Fibonacci-Folge ist eine berühmte Zahlenfolge.
- Die Folge beginnt mit 0 und 1:

$$0, 1, 1, 2, 3, 5, 8, 13, 21, \ldots$$

- Jede Zahl ist die Summe der beiden vorherigen Zahlen:

$$F(n) = F(n-1) + F(n-2)$$

- Die ersten beiden Werte sind:

$$F(0) = 0, \quad F(1) = 1$$

- Die Folge hat Anwendungen in Mathematik, Informatik und Natur.

PHIL SZALAY

Aufgabe: Berechne die Fibonacci Zahl

- Schreibe in **IterationVsRekursion.java** zwei Methoden in Java:
  - `fibonacciIterativ(int n)` mit einer `for`-Schleife.
  - `fibonacciRekursiv(int n)` mit rekursivem Aufruf.
- Teste deine Methoden mit verschiedenen Zahlen, z.B. $0, 1, 5, 10$.
- Achte darauf, dass deine rekursive Methode eine Abbruchbedingung hat!

PHIL SZALAY

Lambda-Ausdrücke in Java – Einführung

- **Lambda-Ausdrücke** sind **anonyme Funktionen** – das heißt, sie haben keinen Namen.
- Sie ermöglichen es, kleine Funktionalitäten direkt als Parameter an Methoden zu übergeben.
- Lambda-Ausdrücke wurden ab Java 8 eingeführt, um funktionale Programmierung zu unterstützen.
- Syntax-Grundgerüst:

```
(Parameterliste) -> { Methodenrumpf }
```

Beispiel:

```
(int x) -> x * 2
```

Verdoppelt eine Zahl.

Warum Lambda-Ausdrücke?

- Ermöglichen kürzeren und lesbareren Code.
- Funktionale Schnittstellen können mit Lambda-Ausdrücken einfacher genutzt werden.
- Beispiel ohne Lambda (anonyme innere Klasse):

```
new Runnable() {
  public void run() {
    System.out.println("Hallo");
  }
};
```

- Mit Lambda:

```
() -> System.out.println("Hallo");
```

Was sind höherwertige Funktionen?

- Funktionen, die andere Funktionen als Parameter nehmen oder zurückgeben.
- Beispiele in Java: `map`, `filter`, `reduce` (meist bei `Streams`).
- Diese Funktionen erlauben es, Daten auf elegante Weise zu transformieren und zu filtern.

Beispiel: `map` mit Lambdas

- `map` wandelt jede Zahl in einer Liste um.
- Beispiel: Verdopple alle Zahlen in einer Liste.

```
List<Integer> zahlen = List.of(1, 2, 3, 4);
List<Integer> verdoppelt = zahlen.stream()
    .map(x -> x * 2)
    .toList();

System.out.println(verdoppelt); // Ausgabe: [2, 4, 6, 8]
```

Beispiel: `filter` mit Lambdas

- `filter` entfernt Elemente, die eine Bedingung nicht erfüllen.
- Beispiel: Nur gerade Zahlen behalten.

```
List<Integer> zahlen = List.of(1, 2, 3, 4);
List<Integer> gerade = zahlen.stream()
    .filter(x -> x % 2 == 0)
    .toList();

System.out.println(gerade); // Ausgabe: [2, 4]
```

Beispiel: `reduce` mit Lambdas

- **`reduce`** fasst alle Elemente zu einem einzigen Wert zusammen.
- Beispiel: Summe aller Zahlen berechnen.

```
List<Integer> zahlen = List.of(1, 2, 3, 4);
int summe = zahlen.stream()
    .reduce(0, (acc, x) -> acc + x);

System.out.println(summe); // Ausgabe: 10
```

Zusammenfassung

- Lambda-Ausdrücke sind kleine, anonyme Funktionen, die man einfach definieren und übergeben kann.
- Höherwertige Funktionen wie `map`, `filter` und `reduce` helfen, Listen und Streams elegant zu verarbeiten.
- Lambdas machen den Code kürzer, flexibler und oft leichter lesbar.

PHIL SZALAY

Aufgaben für euch

- Verwende `map`, um aus einer Liste von Strings die Länge jedes Strings zu berechnen.
- Filtere aus einer Liste von Zahlen alle Zahlen heraus, die kleiner als 10 sind.
- Berechne mit `reduce` das Produkt aller Zahlen in einer Liste.

PHIL SZALAY

Was ist ein Algorithmus?

- Ein Algorithmus ist eine eindeutige Schritt-für-Schritt-Anleitung zur Lösung eines Problems.
- Er ist unabhängig von Programmiersprachen und kann auch in Alltagssprache beschrieben werden.
- Jeder Algorithmus hat:
  - einen definierten Anfang,
  - eine endliche Anzahl von Schritten,
  - und ein definiertes Ende mit einem Ergebnis.
- Beispiele:
  - Kuchen backen (Rezept)
  - Wegbeschreibung geben
  - Zahlen sortieren

PHIL SZALAY

Beispiele für Algorithmen im Alltag

- **Zähneputzen:**
  1. Zahnbürste nehmen
  2. Zahnpasta auftragen
  3. 2 Minuten putzen
  4. Ausspülen
- **Pizza bestellen:**
  1. App öffnen
  2. Gericht auswählen
  3. Adresse eingeben
  4. Bezahlen
  5. Warten auf Lieferung
- Auch ein Computer führt solche Anweisungen aus — nur viel schneller und ohne Fehler (meistens).

PHIL SZALAY

Warum Sortieren wichtig ist

- Sortieren ist ein typisches Problem in der Informatik.
- Beispiele:
  - Eine Kontaktliste alphabetisch sortieren
  - Nachrichten nach Datum ordnen
  - Preise von Produkten vergleichen
- Es gibt viele verschiedene Sortieralgorithmen — einige sind einfach zu verstehen, andere besonders schnell.

PHIL SZALAY

Übung: Sortieren – Wie würdest du vorgehen?

- **Aufgabe:** Beschreibe in deinen eigenen Worten (oder in Pseudocode), wie du eine Liste von Zahlen sortieren würdest.
- Beispiel-Liste: `[5, 2, 9, 1, 3]`
- Du kannst Stichpunkte verwenden oder eine Schritt-für-Schritt-Anleitung schreiben.
- Denk daran: Du darfst Zahlen vertauschen, vergleichen, usw.

Insertion Sort – Einfache Idee

- Wir bauen uns eine sortierte Liste auf:
  - Gehe von links nach rechts durch die Liste.
  - Nimm jedes neue Element und schiebe es an die richtige Stelle im linken (sortierten) Teil.
- Wie beim Kartenspielen: Neue Karte wird in die sortierte Hand einsortiert.

Beispiel:

- Ausgangsliste: `[5, 2, 9, 1]`
- Schritt 1: `[2, 5, 9, 1]`
- Schritt 2: `[2, 5, 9, 1]`
- Schritt 3: `[1, 2, 5, 9]`

PHIL SZALAY

Selection Sort – Idee

- Finde immer das kleinste Element im unsortierten Teil und schiebe es ganz nach vorne.
- Wiederhole das für alle Positionen.
- Langsam, aber einfach.

Beispiel:

- Ausgangsliste: `[5, 2, 9, 1]`
- Schritt 1 (kleinstes = 1): `[1, 2, 9, 5]`
- Schritt 2 (kleinstes = 2): `[1, 2, 9, 5]`
- Schritt 3 (kleinstes = 5): `[1, 2, 5, 9]`

PHIL SZALAY

Bubble Sort – Idee

- Gehe die Liste durch und vergleiche benachbarte Elemente.
- Wenn zwei Elemente in der falschen Reihenfolge sind, vertausche sie.
- Wiederhole den Vorgang, bis alles sortiert ist.
- Die großen Zahlen „blasen sich nach oben“.

Beispiel:

- Ausgangsliste: `[5, 2, 9, 1]`
- Durchgang 1: `[2, 5, 1, 9]`
- Durchgang 2: `[2, 1, 5, 9]`
- Durchgang 3: `[1, 2, 5, 9]`

PHIL SZALAY

Übung: Bubble Sort implementieren

Aufgabe: Implementiere den Bubble Sort Algorithmus in Java!

Schritte:

- Schreibe in **Bubblesort.java** eine Methode `bubbleSort`, die ein Array von Ganzzahlen (`int[]`) sortiert.
- Gib das sortierte Array am Ende in der Konsole aus.

Hinweise:

- Verwende zwei `for`-Schleifen, um durch das Array zu gehen.
- Vergleiche jeweils zwei benachbarte Zahlen.
- Wenn sie in der falschen Reihenfolge sind, vertausche sie!
- Wiederhole das, bis die Liste vollständig sortiert ist.

PHIL SZALAY

Was ist ein Computer-Programm eigentlich?

Ein Computer-Programm ist eine Abfolge von Anweisungen, die ein Computer ausführt.

Damit der Computer das versteht, muss der Code, den wir schreiben, in Maschinensprache übersetzt werden.

Es gibt zwei Hauptwege, wie das passiert:

- **Compiler**: übersetzt das ganze Programm auf einmal
- **Interpreter**: übersetzt und führt Zeile für Zeile aus

PHIL SZALAY

Was ist ein Compiler?

Ein Compiler übersetzt den gesamten Quellcode in ausführbaren Maschinencode.

Ablauf:

1. Du schreibst Code (z.B. in Java oder C++).
2. Der Compiler übersetzt den Code in eine ausführbare Datei (Maschinensprache).
3. Diese Datei kann auf dem Computer gestartet werden – ohne den Quellcode.

Vorteile:

- Schnell in der Ausführung
- Code muss nicht mitgeliefert werden

Nachteil:

- Fehlersuche oft schwieriger, da alles auf einmal übersetzt wird

PHIL SZALAY

Abbildung 2: Compiler Visualisierung

Was ist ein Interpreter?

Ein Interpreter führt den Quellcode Zeile für Zeile direkt aus.

Ablauf:

1. Du schreibst Code (z.B. in Python oder JavaScript).
2. Der Interpreter liest den Code und führt ihn direkt aus.
3. Es gibt keine separate ausführbare Datei.

Vorteile:

- Einfachere Fehlersuche
- Code kann schnell getestet werden

Nachteil:

- Langsamer in der Ausführung
- Code muss vorhanden sein, um ihn auszuführen

PHIL SZALAY

Java: Compiler und Interpreter in einem!

Java benutzt **beides**:

- **Compiler:** Der Java-Compiler (`javac`) übersetzt Java-Code in sogenannte **Bytecode**-Dateien (.class).
- **Interpreter:** Die **Java Virtual Machine (JVM)** interpretiert den Bytecode und führt ihn auf deinem Rechner aus.

Warum dieser Aufbau?

- **„Write once, run anywhere“:** Der Bytecode ist nicht an ein Betriebssystem gebunden.
- Java-Programme laufen auf allen Plattformen, auf denen eine JVM installiert ist.
- Entwickler müssen ihren Code nicht für verschiedene Systeme neu schreiben oder anpassen.

Kompromiss

- Etwas langsamer als direkt kompilierter Code (z.B. in C oder C++).
- Dafür: **Hohe Portabilität**, **Sicherheit durch die JVM** und eine **große Community**.

PHIL SZALAY

Beispielvergleich Compiler vs Interpreter

Java (Compiler-basiert):

- Code schreiben: `HelloWorld.java`
- Kompilieren: `javac HelloWorld.java`
- Ausführen: `java HelloWorld`

Python (Interpreter-basiert):

- Code schreiben: `hello.py`
- Direkt ausführen: `python hello.py`

Erkenntnis: Beide Wege haben ihre Vorteile – je nach Einsatzzweck.

PHIL SZALAY

Exkurs: Objektorientiertes Programmieren

Was ist objektorientiertes Programmieren (OOP)?

Ein Programmierparadigma, das sich an „Objekten“ orientiert

- Objekte bilden Dinge der realen Welt ab
- Jedes Objekt hat:
  - Eigenschaften (Daten → **Attribute**)
  - Verhalten (Funktionen → **Methoden**)

Ziel: Programme strukturierter, verständlicher und wiederverwendbarer machen

PHIL SZALAY

Warum OOP nutzen?

- Modelliert Probleme „wie in der echten Welt“

Warum OOP nutzen?

- Modelliert Probleme „wie in der echten Welt“
- Wiederverwendbarkeit durch Klassen

Warum OOP nutzen?

- Modelliert Probleme „wie in der echten Welt“
- Wiederverwendbarkeit durch Klassen
- Einfachere Wartung und Erweiterung

PHIL SZALAY

Warum OOP nutzen?

- Modelliert Probleme „wie in der echten Welt“
- Wiederverwendbarkeit durch Klassen
- Einfachere Wartung und Erweiterung
- Gute Basis für Teamarbeit und große Softwareprojekte

PHIL SZALAY

Warum OOP nutzen?

- Modelliert Probleme „wie in der echten Welt“
- Wiederverwendbarkeit durch Klassen
- Einfachere Wartung und Erweiterung
- Gute Basis für Teamarbeit und große Softwareprojekte
- Klare Trennung von Daten und Verhalten

PHIL SZALAY

Wie denkt man beim Programmieren? Drei verschiedene Ansätze:

1. Prozedurale Programmierung

- Der Fokus liegt auf Abläufen und Funktionen
- Daten werden übergeben und verarbeitet
- Beispiel: `C`, frühes `Python`

Programmierparadigmen im Vergleich

Wie denkt man beim Programmieren? Drei verschiedene Ansätze:

1. Prozedurale Programmierung

- Der Fokus liegt auf Abläufen und Funktionen
- Daten werden übergeben und verarbeitet
- Beispiel: `C`, frühes `Python`

2. Objektorientierte Programmierung (OOP)

- Der Fokus liegt auf Objekten (Daten + Verhalten)
- Programm = Interaktion von Objekten
- Beispiel: `Java`, `C++`

Programmierparadigmen im Vergleich

Wie denkt man beim Programmieren? Drei verschiedene Ansätze:

1. Prozedurale Programmierung

- Der Fokus liegt auf Abläufen und Funktionen
- Daten werden übergeben und verarbeitet
- Beispiel: `C`, frühes `Python`

2. Objektorientierte Programmierung (OOP)

- Der Fokus liegt auf Objekten (Daten + Verhalten)
- Programm = Interaktion von Objekten
- Beispiel: `Java`, `C++`

3. Funktionale Programmierung

- Der Fokus liegt auf reinen Funktionen ohne Seiteneffekte
- Daten sind unveränderlich („immutable“)
- Beispiel: `Haskell`, `Scala`, modernes `JavaScript`

PHIL SZALAY

Beispiel: Objektorientierung in einem Videospiel

In einem Videospiel gibt es viele verschiedene Objekte:

- Spieler
- Gegner
- Waffen
- Hindernisse

Diese Objekte haben:

- **Attribute:** z. B. Name, Lebenspunkte, Position
- **Methoden:** z. B. `bewegen()`, `angreifen()`, `heilen()`

Objekt: Spieler

Attribute:

- `Name` = „Alex“
- `Lebenspunkte` = 100
- `Position` = (5, 10)

Methoden:

- `bewegen(x, y)`
- `angreifen(Gegner)`
- `heilen(Heiltrank)`

Objekt: Gegner

Attribute:

- `Typ` = „Zombie“
- `Lebenspunkte` = 50
- `Position` = (3, 8)

Methoden:

- `verfolgen(Spieler)`
- `angreifen(Spieler)`

Übung: Objekte

- Was wären in eurem Lieblingsspiel typische „Objekte“?
- Welche Eigenschaften und Verhaltensweisen könnten sie haben?
- Beispielantwort: „`Auto in Rennspiel`“ → `Geschwindigkeit`, `Modell`, `fahren()`

Was ist eine Klasse?

- Eine Klasse ist ein **Bauplan** für Objekte
- Sie definiert:
  - Welche Daten ein Objekt besitzt → **Attribute** (veränderbar)
  - Welche Aktionen es ausführen kann → **Methoden**
- Eine Klasse selbst ist noch kein Objekt – sie beschreibt nur, wie Objekte aussehen und sich verhalten
- Mit dem Schlüsselwort `class` wird eine Klasse definiert (z. B. in Python oder Java)

Beispiel in Python:

```python
class Auto:
    def __init__(self, marke, geschwindigkeit):
        self.marke = marke
        self.geschwindigkeit = geschwindigkeit

    def beschleunigen(self, wert):
        self.geschwindigkeit += wert

# Objekt erstellen
mein_auto = Auto("Toyota", 50)
mein_auto.beschleunigen(20)
print(mein_auto.geschwindigkeit)  # Ausgabe: 70
```

Klasse vs. Objekt – Was ist der Unterschied?

- **Klasse** = Bauplan (z. B. „Auto“)
- **Objekt** = konkrete Instanz (z. B. „Auto von Lisa“)
- Mehrere Objekte können aus einer Klasse erzeugt werden
- Beispiel: `Auto von Lisa`, `Auto von Tom` → beide aus Klasse **„`Auto`“**

Beispiel in Python:

```
class Auto:
    def __init__(self, marke, farbe):
        self.marke = marke
        self.farbe = farbe

# Zwei Objekte (Instanzen) der Klasse Auto
auto_lisa = Auto("BMW", "blau")
auto_tom = Auto("Audi", "schwarz")

print(auto_lisa.marke)   # Ausgabe: BMW
print(auto_tom.farbe)    # Ausgabe: schwarz
```

Klasse vs. Objekt – Was ist der Unterschied?

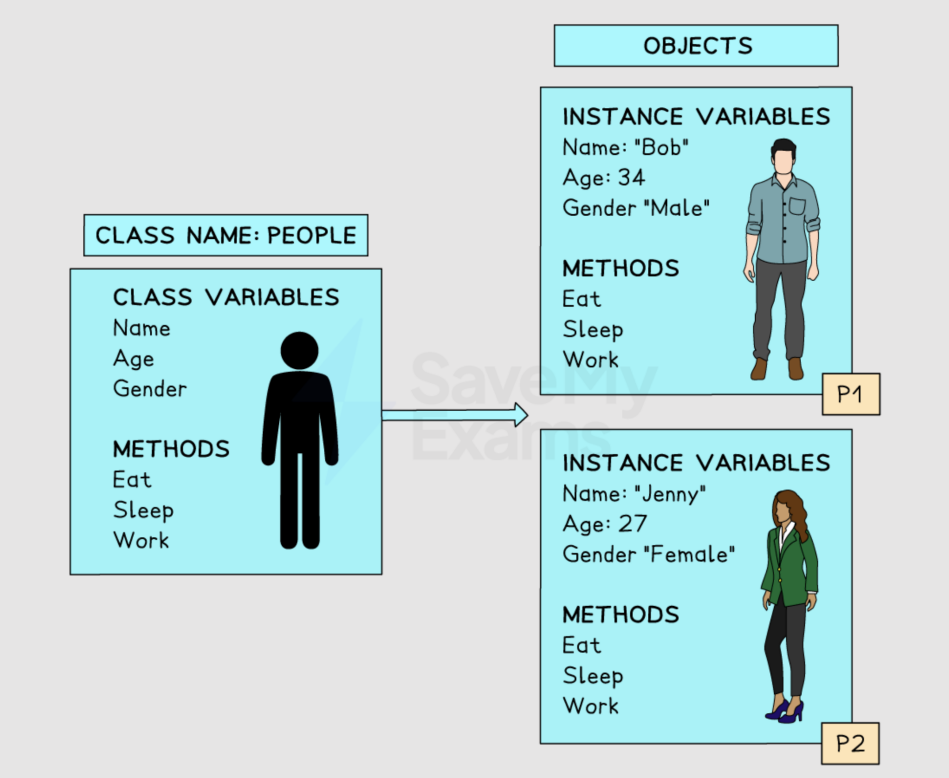

Abbildung 3: Klasse vs. Objekt[2]

[2] Quelle: https://www.savemyexams.com/a-level/computer-science/ocr/17/revision-notes/2-software-and-software-development/2-5-object-oriented-languages/classes-oop/

Übung: Programmiere dein eigenes Spielobjekt

Aufgabe:

- Implementiere eine Klasse `Spieler` mit:
  - Attributen: `name`, `lebenspunkte`, `schadenspunkte`
  - Methoden: `angreifen(gegner)`, `heilen()`
- Implementiere eine Klasse `Gegner` mit:
  - Attributen: `name`, `lebenspunkte`, `schadenspunkte`
  - Methoden: `angreifen(spieler)`
- Für alle Methoden soll geloggt werden, was passiert
- Erstelle ein Objekt vom Typ `Spieler` und ein Objekt vom Typ `Gegner`
- Spiele ein beliebiges Szenario durch mit den beiden Objekten, z.B.:
  1. `gegner.angreifen(spieler)`
  2. `spieler.heilen()`
  3. `spieler.angreifen(gegner)`
  4. `spieler.angreifen(gegner)`
  5. `spieler.angreifen(gegner)`

PHIL SZALAY

Sichtbarkeit von Attributen und Methoden

Attribute und **Methoden** können entweder von außen nutzbar sein, oder nicht:

- **public** – Jeder darf darauf zugreifen
- **private** – Nur die Klasse selbst darf darauf zugreifen
- **protected** – Nur die Klasse und ihre Unterklassen

→ Zugriff auf private/protected Felder bei Bedarf mit **Getter**/**Setter**-Methoden

Ziel: Informationen kapseln und ungewollten Zugriff verhindern.

Python:

- `_name` = protected
- `__name` = private (Name Mangling)

Beispiel: Sichtbarkeit von Attribute und Methoden

Beispiel in Python:

```
class Konto:
    def __init__(self, inhaber, kontostand):
        self.inhaber = inhaber              # public
        self.kontostand = kontostand        # public
        self.__pin = 1234                   # private

    def get_pin(self):
        return self.__pin                   # Getter fuer private Attribut

    def set_pin(self, neuer_pin):
        self.__pin = neuer_pin              # Setter

konto = Konto("Alex", 1000)
print(konto.inhaber)        # Zugriff erlaubt
print(konto.kontostand)     # Zugriff erlaubt
# print(konto.__pin)        # Fehler! Attribut ist privat
print(konto.get_pin())      # Zugriff ueber Getter
```

Übung: Privates Name-Attribut

Problem: Aktuell können die Namen von `Spieler` und `Gegner` beliebig von außen geändert werden.

Aufgabe: Das Name-Attribut in beiden Klassen soll zu **privat** geändert werden. Trotzdem soll der Name eines Objektes von außen zugänglich sein.

Statische vs. dynamische Mitglieder in Python

Was ist der Unterschied?

- **Dynamische (Instanz-)Attribute:**
  - Gehören zu einem bestimmten Objekt
  - Werden mit `self` im Konstruktor definiert
  - Beispiel: `self.lebenspunkte`, `self.name`
- **Statische (Klassen-)Attribute:**
  - Gelten für die ganze Klasse – alle Objekte teilen sie
  - Zugriff über den Klassennamen: `Spieler.max_lebenspunkte`
  - Beispiel: `Spieler.max_lebenspunkte` begrenzt Heilung für alle Spieler

Wann statisch, wann dynamisch?

- **Statisch:** wenn die Information für die ganze Klasse gilt
- **Dynamisch:** wenn jede Instanz ihren eigenen Wert haben soll

Tipp: In Python werden statische Methoden mit `@staticmethod` erzeugt

PHIL SZALAY

Beispiel: Statische vs. dynamische Attribute in Python

```python
class Spieler:
    # Statisches Attribut (Klasse)
    max_lebenspunkte = 100

    def __init__(self, name):
        # Dynamische Attribute (Instanz)
        self.name = name
        self.lebenspunkte = Spieler.max_lebenspunkte

    def heilen(self, menge):
        self.lebenspunkte = min(
            self.lebenspunkte + menge,
            Spieler.max_lebenspunkte  # Zugriff auf statisches Attribut
        )

    @staticmethod
    def begruessung():
        print("Willkommen im Spiel!")

# Verwendung:
spieler1 = Spieler("Alex")
spieler2 = Spieler("Bob")

print(spieler1.lebenspunkte)  # 100
spieler1.heilen(10)           # heilt, aber max. bis 100
Spieler.begruessung()         # Statische Methode aufrufen
```

Vererbung in der Objektorientierten Programmierung

Idee: Eine Klasse kann Eigenschaften und Verhalten einer anderen Klasse „erben“.

- **Basisklasse (Superklasse):** stellt gemeinsame Attribute und Methoden bereit
- **Abgeleitete Klasse (Subklasse):** erbt alles von der Basisklasse und kann erweitern oder überschreiben

Vorteile:

- Gemeinsamer Code muss nicht doppelt geschrieben werden
- Erleichtert Wartung und Erweiterung

PHIL SZALAY

Vererbung in der Objektorientierten Programmierung

Idee: Eine Klasse kann Eigenschaften und Verhalten einer anderen Klasse „erben“.

- **Basisklasse (Superklasse):** stellt gemeinsame Attribute und Methoden bereit
- **Abgeleitete Klasse (Subklasse):** erbt alles von der Basisklasse und kann erweitern oder überschreiben

Vorteile:

- Gemeinsamer Code muss nicht doppelt geschrieben werden
- Erleichtert Wartung und Erweiterung

Beispiele für Vererbungsstrukturen:

- `Fahrzeug` → `Auto`, `Motorrad`, `LKW`
- `Tier` → `Hund`, `Katze`, `Vogel`
- `Spieler` → `Magier`, `Krieger`, `Bogenschütze`
- `UIElement` → `Button`, `Textfeld`, `Checkbox`

PHIL SZALAY

Vererbung in der Objektorientierten Programmierung

Idee: Eine Klasse kann Eigenschaften und Verhalten einer anderen Klasse „erben“.

- **Basisklasse (Superklasse):** stellt gemeinsame Attribute und Methoden bereit
- **Abgeleitete Klasse (Subklasse):** erbt alles von der Basisklasse und kann erweitern oder überschreiben

Vorteile:

- Gemeinsamer Code muss nicht doppelt geschrieben werden
- Erleichtert Wartung und Erweiterung

Beispiele für Vererbungsstrukturen:

- `Fahrzeug` → `Auto`, `Motorrad`, `LKW`
- `Tier` → `Hund`, `Katze`, `Vogel`
- `Spieler` → `Magier`, `Krieger`, `Bogenschütze`
- `UIElement` → `Button`, `Textfeld`, `Checkbox`

Tipp: In Python wird Vererbung so geschrieben:
`class Auto(Fahrzeug):` – `Auto` erbt von `Fahrzeug`

PHIL SZALAY

Übung: Vererbung in der Objektorientierten Programmierung

Übung: Finde mindestens 10 “ist ein” Beispiele aus der Realität, die Vererbung einer übergeordneten Klasse beschreiben.

Vererbung am Beispiel

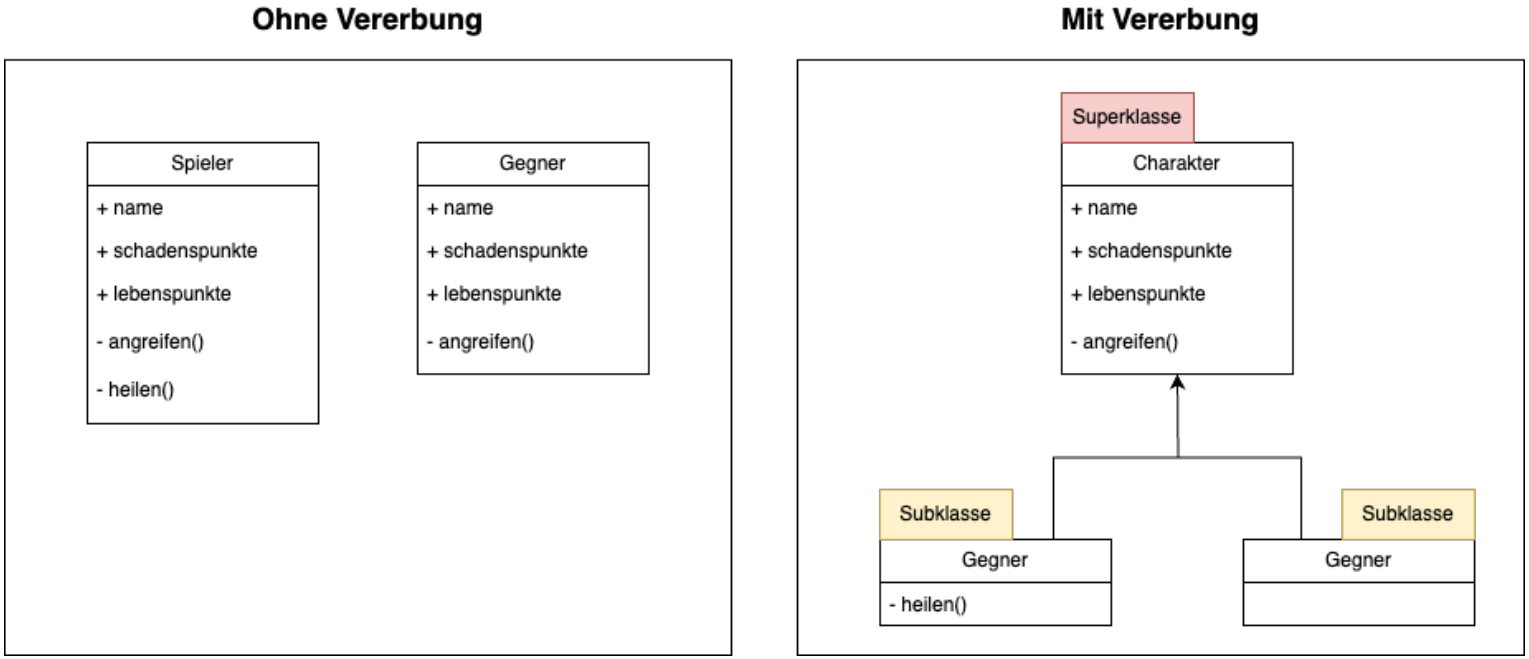

Abbildung 4: Vererbung am Beispiel Spieler und Gegner

Beispiel: Vererbung in Python

```python
# Basisklasse
class Fahrzeug:
    def __init__(self, marke):
        self.marke = marke

    def starten(self):
        print(f"{self.marke} startet den Motor.")

# Subklasse erbt von Fahrzeug
class Auto(Fahrzeug):
    def __init__(self, marke, sitze):
        super().__init__(marke)  # Aufruf Konstruktor der Basisklasse
        self.sitze = sitze

    def hupen(self):
        print(f"{self.marke} hupt: Tuuut!")

# Verwendung
vw = Auto("VW", 5)
vw.starten()    # Methode aus Basisklasse
vw.hupen()      # Methode aus Subklasse
print(vw.sitze)  # Zugriff auf neues Attribut
```

Übung: Vererbung

Die Klassenstruktur in unserer Beispielanwendung ist nicht optimal.

Aufgabe: Optimiere die Klassenstruktur wie im Beispiel beschrieben.

Polymorphie – Gleiches Interface, unterschiedliches Verhalten

Idee: Objekte verschiedener Klassen können über die gleiche Schnittstelle (Methodenname) angesprochen werden, reagieren aber unterschiedlich.

Beispiel: Alle Spielcharaktere haben eine `angreifen()`-Methode, aber jeder greift anders an.

- `Magier.angreifen()` → feuert einen Feuerball
- `Krieger.angreifen()` → schlägt mit dem Schwert
- `Bogenschütze.angreifen()` → schießt einen Pfeil

Polymorphie – Gleiches Interface, unterschiedliches Verhalten

Idee: Objekte verschiedener Klassen können über die gleiche Schnittstelle (Methodenname) angesprochen werden, reagieren aber unterschiedlich.

Beispiel: Alle Spielcharaktere haben eine `angreifen()`-Methode, aber jeder greift anders an.

- `Magier.angreifen()` → feuert einen Feuerball
- `Krieger.angreifen()` → schlägt mit dem Schwert
- `Bogenschütze.angreifen()` → schießt einen Pfeil

Vorteile:

- Flexibler und erweiterbarer Code
- Methoden können auf allgemeinem Typ aufgerufen werden (`char.angreifen()`)
- Weniger `if-else`-Konstrukte nötig

Polymorphie – Gleiches Interface, unterschiedliches Verhalten

Idee: Objekte verschiedener Klassen können über die gleiche Schnittstelle (Methodenname) angesprochen werden, reagieren aber unterschiedlich.

Beispiel: Alle Spielcharaktere haben eine `angreifen()`-Methode, aber jeder greift anders an.

- `Magier.angreifen()` → feuert einen Feuerball
- `Krieger.angreifen()` → schlägt mit dem Schwert
- `Bogenschütze.angreifen()` → schießt einen Pfeil

Vorteile:

- Flexibler und erweiterbarer Code
- Methoden können auf allgemeinem Typ aufgerufen werden (`char.angreifen()`)
- Weniger `if-else`-Konstrukte nötig

Tipp: In Python genügt es, die Methode in den Subklassen passend zu implementieren – das Prinzip heißt „Duck Typing“.

PHIL SZALAY

Beispiel: Polymorphie in Python

```
# Basisklasse
class Charakter:
    def angreifen(self):
        pass  # Wird in den Unterklassen ueberschrieben

# Subklasse 1
class Magier(Charakter):
    def angreifen(self):
        print("Der Magier schleudert einen Feuerball!")

# Subklasse 2
class Krieger(Charakter):
    def angreifen(self):
        print("Der Krieger schlaegt mit dem Schwert zu!")

# Subklasse 3
class Bogenschuetze(Charakter):
    def angreifen(self):
        print("Der Bogenschaetze feuert einen Pfeil!")

# Polymorphie in Aktion
charaktere = [Magier(), Krieger(), Bogenschuetze()]

for char in charaktere:
    char.angreifen()  # Jeder Charakter reagiert anders
```

Generische Klassen – Was ist das?

- Generische Klassen sind Klassen mit **Platzhaltern für Datentypen**.
- Sie erlauben es, eine Klasse so zu schreiben, dass sie mit verschiedenen Datentypen funktioniert.
- Dadurch müssen Klassen nicht für jeden Datentyp neu geschrieben werden – sie werden **wiederverwendbar** und **flexibler**.
- Beispiel: Eine generische `Box<T>` kann eine Box für `Integer`, `String` oder jeden anderen Typ sein.
- Vorteil:
  - Mehr Typensicherheit – der Datentyp ist beim Erstellen der Objekte klar definiert.
  - Weniger Code-Duplizierung, da gleiche Logik für verschiedene Typen gilt.
  - Erleichtert Wartung und vermeidet Fehler.
- Generische Klassen sind in vielen modernen Programmiersprachen verfügbar (z.B. Java, C++, C#, Python).

PHIL SZALAY

Beispiel: Generische Klasse `Box[T]` in Python

```python
from typing import TypeVar, Generic

T = TypeVar('T')  # Typvariable

class Box(Generic[T]):
    def __init__(self, inhalt: T) -> None:
        self.inhalt = inhalt

    def get_inhalt(self) -> T:
        return self.inhalt

    def set_inhalt(self, neues_inhalt: T) -> None:
        self.inhalt = neues_inhalt

# Verwendung
string_box = Box[str]("Hallo Welt")
print(string_box.get_inhalt())  # Ausgabe: Hallo Welt

int_box = Box
print(int_box.get_inhalt())     # Ausgabe: 123
```

Typ-Beschränkungen mit `TypeVar` und `bound`

```python
from typing import TypeVar, Generic

class Zahl:
    def wert(self) -> int:
        return 0

T = TypeVar('T', bound=Zahl)  # T muss Unterklasse von Zahl sein

class ZahlBox(Generic[T]):
    def __init__(self, zahl: T) -> None:
        self.zahl = zahl

    def zeige_wert(self) -> int:
        return self.zahl.wert()

# Beispiel-Unterklasse
class MeineZahl(Zahl):
    def __init__(self, wert: int) -> None:
        self._wert = wert

    def wert(self) -> int:
        return self._wert

zb = ZahlBox(MeineZahl(42))
print(zb.zeige_wert())  # Ausgabe: 42
```

Generische Funktionen in Python

```
from typing import TypeVar, List

T = TypeVar('T')

def drucke_liste(liste: List[T]) -> None:
    for element in liste:
        print(element)

drucke_liste([1, 2, 3])
drucke_liste(["a", "b", "c"])
```

Übung: Generische Funktionen in Python

Übung: Bearbeite alle Aufgaben in **generics.py**.

Kapitel 3: Entwurf

Kapitel 4: Programmierung

Kapitel 5: Testen

Kapitel 6: Wartung