# Widerholungsfragen OOP WS22/23

Sebastian KLAUS
January 19, 2023

---

Die Ausarbeitung basiert auf dem Skript vom WS22. Die Fragen sind nach bestem Wissen und Gewissen ausgearbeitet. Die Rechtschreibfehler waren alle Absicht :))
gl hf beim Lernen

---

# 1 Paradigmen der Programmierung

## 1.1 Berechnungsmodell und Programmstruktur

1. **Was versteht man unter einem Programmierparadigma?**
   Ist eine bestimmte Denkweise oder Art der Weltanschauung.

2. **Wozu dient ein Berechnungsmodell?**
   Hinter jedem Programmierparadigma steckt ein Berechnungsmodell, dieses muss in sich konsistent und in der Regel Turing-vollständig sein. Es muss alles Ausdrücken können, was berechenbar ist.

3. **Welche Berechnungsmodelle werden in Programmierparadigmen verwendet, und welche charakteristischen Eigenschaften haben sie?**

   - **Funktionen**
     −primitiv-rekursive Funktionen → bildet von einer Menge einfacher Funktionen durch Komposition und Rekursion neue Funktionen. (nicht Turing-vollständig)
     −$\boldsymbol{\lambda}$-Kalkül → Turing-Vollständig
   - **Prädikatenlogik**
     −mächtiges mathematisches Werkzeug
     −heute nur noch untergeordnet in der Programmierung jedoch häufig bei relationalen Datenbanken verwendet.
   - **Constraint-Programmierung**
     −"x $<$ 5" oder "A oder B ist wahr"
     −sind mit funktionalen und imperativen Sprachen kombinierbar
     −heute werden vorwiegend fertige Bibliotheken zum Auflösen verwendet
   - **Temporale Logik und Petri-Netze**
     −zetiliche Abhängigkeiten leicht darstellbar
     −Petri-Netze gut für grafische Veranschaulichung aber temporale Logik besser für Beweise

- **Freie Algebren**
  −freie Algebren erlauben fast belibiger Strukturen (z.B.: Datenstrukturen) durch Axiome −wichtige Rolle bei Modulen und Typen
- **Prozesskalküle**
  −bekannte Prozesskalküle: CSP (Communicating Sequential Process) & $\pi$-Kalkül
  −primitive Operationen ausschließlich für senden und empfangen von Daten.
  −$\pi$-Kalkül besonders gut geeignet für reaktive Systeme, welche auf Ereignisse in der Umgebung reagieren. ($\lambda$-Kalkül kann nur transformatorische Systeme gut beschreiben → Eingabedaten in Ergebnisdaten umwandeln)
- **Automaten**
  −anschauliche Darstellung → werden oft zur Spezifikation des Systemverhaltens verwendet
- **WHILE, GOTO und Co**
  −WHILE- und GOTO-Sprache sind ganz einfache formale Sprachen (mit while Schleife bzw. goto-Anweisung) sind aber Turing-vollständgin

4. **Welche Eigenschaften von Berechnungsmodellen sind für deren Erfolg häufig (mit)bestimmend?**
   - Kombinierbarkeit
   - Konsistenz
   - Abstraktion
   - Systemnähe
   - Unterstützung (as in Leute/Firmen die an das Paradigma glauben und es pushen)
   - Beharrungsvermögen
5. **Im Spannungsfeld welcher widersprüchlichen Ziele befinden sich Programmierparadigmen? Wie äußert sich dieses Spannungsfeld?**
   −Flexibilität und Ausdruckskraft
   −Lesbarkeit und Sicherheit
   −Verständlichkeit

   z.B.: objektorientierte Sprachen haben einen hohen Grad von Punkt 1 & 2, jedoch leidet darunter häufig die Verständlichkeit
6. **Was ist die strukturierte Programmierung? Wozu dient sie?**
   strukturierte Programme bestehen aus nur drei Kontrollstrukturen:
   −Sequenz (ein Schritt nach dem anderen)
   −Auswahl (if else)
   −Wiederholung (Schleife, Rekursion)

Das Ziel der strukturierten Programmierung ist es, dass jede Kontrollstruktur nur einen wohlgeformten Einstiegs- und Ausstiegspunkt hat. → leichteres Verfolgen des Programmpfads.

7. **Wie gehen unterschiedliche Paradigmen mit Seiteneffekten um?**
   −deklarative Paradigmen: referentielle Transparenz; 3 + 4 kann durch 7 ersetzt werden (bsp für referentielle Transparenz) −objektorientierte Paradigmen: wo es keine referentielle Transparenz gibt nimmt man stets an, das es entsprechende Querverbindungen gibt. Diese werden auf einzelne Objekte beschränkt (überschaubarkeit)

8. **Was bedeutet referentielle Transparenz, und wo findet man referentielle Transparenz?**
   Ein Ausdruck ist referentiell Transparent, wenn er durch seinen Wert ersetzt werden kann, ohne die Semantik des Ausdrucks zu verändern. 3 + 4 durch 7 oder 14 / 2 erstzten, f(x) + f(x) kann auch durch 2 * f(x) ersetzt werden.

9. **Wieso passt referentielle Transparenz nicht gut mit Ein- und Ausgabe zusammen, und wie kann man das Dilemma lösen?**
   Ein- und Ausgaben sind Seiteneffekte, ohne die die deklarative Programmierung nicht auskommt. (kann ma also nd los werden)
   Lösung: Ein- und Ausgaben nur auf oberster Aufrufhierarchie erlauben und aus allen anderen Funktionen verbannen.
   → referentielle Transparenz in allen Programmteilen gewährleistet.

10. **Welchen Zusammenhang gibt es zwischen Seiteneffekten und der objektorientierten Programmierung?**
    Objekte sind Strukturierungsmittel um Prozeduren und Variablen entsprechend der Querverbindungen (Seiteneffekt) zu Einheiten zusammenzufassen.

11. **Was sind First-Class-Entities? Welche Gründe sprechen für deren Verwendung, welche dagegen?**
    Funktionen sind First-Class-Entities → Funktionen können wie Daten verwendet werden (zb.: $\boldsymbol{\lambda}$-Kalkül). (!Funktionen nicht mit Methoden verwechseln!)

    Pro: man kann Kontrollstrukturen damit erzeugen (Funktionen höherer Ordnung)
    Contra: komplizierter

12. **Was haben Funktionen höherer Ordnung mit einem applikativen Programmierstil zu tun?**
    Funktionen höherer Ordnung führen zum applikativem Programmierstil (eigenes Programmierparadigma)
    Man kann Schablonen von Programmteilen schreiben die durch Übergabe von Funktionen (zum Füllen der Löcher) ausführbar werden.

## 1.2 Programmorganisation

13. **Welche Modularisierungseinheiten gibt es, was sind ihre charakteristischen Eigenschaften, und wodurch unterscheiden sie sich?**

    - **Modul**
      Übersetzungseinheit (Compiler bearbeitet sie in einem Stück (in Java: Interface oder Klasse))
      lassen sich unabhängig voneinander entwickeln
      Datenkapselung u. Data-Hiding
    - **Objekt**
      Werden, anders als Module, erst zur Laufzeit erzeugt
      ähnlich wie Module: Datenkapselung u. Data-Hiding
      Sind fast immer First-Class-Entities
      wichtigste Eigenschaften: Identität, Zusatand und Verhalten
    - **Klasse**
      Schablone für die Erzeugung neuer Objekte
      Vererbung (to be continued)
    - **Komponente**
      eigenständiges Stück Software, welches für sich alleine nicht lauffähig ist
      ähneln Modulen sind aber flexibler: Modul importiert genannte andere Module, dies ist bei der Komponente zur Übersetzungszeit nicht bekannt
    - **Namensraum**
      Die angeführten Modularisierungseinheiten bilden jeweils einen Namensraum. Namensräume fassen mehrere Modularisierungseinheiten zu einer Einheit zusammen, ohne getrennte Übersetzbarkeit zu zerstören.

      Beispiel: a.b.C bezeichnet die Klasse C im Namensraum b, welcher im Namensraum a steht.

14. **Welche Bedeutung haben Schnittstellen für Modularisierungseinheiten? Warum unterscheidet man zwischen von außen zugreifbaren und privaten Inhalten?**
    Schnittstellen sind zusammengefasste Informationen über Inhalte des Moduls. Private Modulinhalte können vom Compiler beliebig optimiert werden. Änderungen dieser wirken sich nicht auf andere Module aus. Bei jenen die von außen zugreifbar sind aber schon $\rightarrow$ bei Änderung dieser muss oft in den anderen Modulen auch eine Änderung vogenommen werden. Zumindest müssen Module die geänderte Inhalte verwenden neu compiliert werden.

15. **Was ist und wozu dient ein Namensraum?**
    Ein Namensraum gilt pro Modul und ist dazu da Nameskonflikte abzufangen.
    Bei mehreren Namensräumen kann es zu Namenskonflikten kommen, hier muss umbenannt werden.

16. **Warum können Module nicht zyklisch voneinander abhängen, Komponenten aber schon?**
Da bei Komponenten zur Übersetzungszeit noch nicht klar ist welche Komponenten sie importieren. Erst beim Einbinden in ein Programm werden diese anderen Komponenten bekannt. Dadurch verringert sich die Abhängigkeit von Komponenten zueinander und es gibt kein Probleme mit zyklischen Abhängigkeiten bei der Übersetzung.

17. **Was versteht man unter Datenabstraktion, Kapselung und Data-Hiding?**

- Datenabstraktion: Kapselung und Data-Hiding zusammen nennt man Datenabstraktion
- Kapselung: das Kapseln von Variablen und Methoden zu logischen Einheiten
- Data-Hiding: das Schützen von privaten Inhalten vor Zugriffen von außen

18. **Warum und inwiefern ist die Einbindung von Komponenten komplizierter als die von Modulen?**
Module bilden voneinander getrennte selbständige Einheiten, Komponenten hingegen interagieren untereinander. Weiters muss beim Einbinden von Komponenten festgelegt werden, von welchen anderen Komponenten diese etwas importiert.

19. **Wie kann man globale Namen verwalten?**
-

20. **Was versteht man unter Parametrisierung? Wann kann das Befüllen von "Lücken" durch welche Techniken erfolgen?**
Es werden "Lücken" im Programm gelassen, welche erst zu einem späteren Zeitpunkt "befüllt" werden.

- **Befüllung zur Laufzeit**
−Konstruktor (klar)
−Initializierungsmethode (wenn Lücken von einem Objekt gefüllt werden, welches erst später erzeugt wird. → leerer Konstruktor und dann in der Initialisierungsmethode Wertzuweisung)
−Zentrale Ablage (z.B.: globale Variablen oder Konstanten)

- **Generizität**
Lücken werden zur Übersetzungszeit befüllt.
- **Annotationen**
Diese können vom Compiler ignoriert oder verwendet werden (je nachdem ob er was damit anfangen kann). Wirken sich also zur Übersetzungszeit aus.
- **Aspekte**
spezifiziert eine Menge von Punkten im Programm (z.B.: Punkte an denen eine bestimmte Methode aufgerufen werden soll) sowie das, was an diesen Stellen passieren soll. Da man hier den Source-Code nicht verändern muss eignen sich Aspekte gut für Debugging / Logging.

21. **Warum braucht man zur Parametrisierung in der Objekterzeugung neben Konstruktoren gelegentlich auch Initialisierungsmethoden?**
Da es sein kann, das Lücken von Objekten gefüllt werden, die erst zu einem späteren Zeitpunkt erstellt werden. Man kann in einem Konstruktor kein Objekt übergeben, welches es noch nicht gibt. Daher werden diese Lücken durch die Initialisierungsmethode, nach Objekt Erstellung befüllt.

22. **Welche Vor- und Nachteile hat die zentrale Ablage von Werten zum Zweck der Parametrisierung?**
Pro: Sie können von statischen Modularisierungseinheiten verwendet werden, da sie bereits zur Übersetzungszeit bekannt sind. Contra: Weite Sichtbarkeit (I guess)

23. **Was unterscheidet Generizität von den verschiedenen Formen der Parametrisierung zur Laufzeit?**
Die Löcher werden bereits zur Übersetzungszeit befüllt. Bevorzugt werden bei generischen Parametern keine gewöhnlichen Werte, sondern Konzepte verwendet.

24. **Was sind Annotationen und wozu kann man sie verwenden? Wodurch unterscheiden sie sich von Generizität?**
Annotationen sind optionale Parameter. Wenn der Compiler mit ihnen nichts anfängt, dann werden sie einfach ignoriert. Die Lücken werden, im Gegensatz zur Generizität nirgends im Programm festgelegt.

25. **Was versteht man unter aspektorientierter Programmierung?**
Es werden zu einem bestehenden Programm von außen neue Aspekte hinzugefügt. (siehe 20)

26. **Wodurch unterscheidet sich Parametrisierung von der Ersetzbarkeit, und warum ist die Ersetzbarkeit von so zentraler Bedeutung?**
Während bei der Parametrisierung Änderungen an den Löchern nach sich ziehen, dass auch die einzusetzenden Dinge geändert werden müssen, kann durch Ersetzbarkeit eine einfachere, nachträgliche Änderungen der Moduleinheiten erreicht werden.

27. **Wann ist A durch B ersetzbar?**
Dann wenn nach einem Austausch von A durch B keine Änderungen an Stellen nach sich zeiht, an denen A (bzw. nach Ersetzung B) verwendet wird.

28. **Wodurch kann festgelegt sein, ob A durch B ersetzbar ist?**
Durch die Schnitstellen von A und B. Wenn diese beiden das Gleiche beschreiben kann A durch B ersetzt werden. Jedoch kann die Schnittstelle von B mehr Details festlegen, welche jene von A offen lässt.

29. **Was ist die Signatur einer Modularisierungseinheit?**
Die einfachste Form einer Schnittstelle spezifiziert nur, welche Inhalte der Modularisierungseinheit von außen zugreifbar sind. Diese Inhalte werden über Namen und gegebenenfalls die Typen von Parametern und Ergebnissen beschreiben. Die Bedeutung dieser Inhalte bleibt offen. Solche Schnittstellen nennt man Signatur der Modularisierungseinheit.

30. **Wie verhält sich die Signatur einer Modularisierungseinheit zur Abstraktion, die durch diese Modularisierungseinheit gebildet wird?**
Abstraktionen sind Schnittstellen, welche die Modularisierungseinheit durch Namen und informelle Texte charakterisieren. Sie entsprechen abstrakten Sichtweisen von Objekten. Aufgrund der Alltagserfahrung, kann man durch Abstraktionen gut abschätzen ob eine solche Abstraktion für eine andere als Ersatz in Frage kommt. (Bsp. wir wissen jedes Auto und jedes Fahrrad ist ein Fahrzeug → Fahrzeug durch Auto/Fahrrad ersetzbar, jedoch ist auch klar, dass ein Auto nicht durch ein Fahrrad ersetzt werden kann)

    Ersetzbarkeit haben wir nur, wenn sowohl Signaturen als auch die Abstraktionen passen.

31. **Was sind Zusicherungen, und welche Rolle spielen sie für Modularisierungseinheiten?**
Eine genaue Beschreibung der erlaubten Erwartungen an die Modularisierungseinheit. Entspricht einem Vertrag zwischen Server und Client Stichwort *Design-by-Contract*

    **−Vorbedingung:** was sich der Server von den Clients erwarten kann.
    **−Nachbedingung:** was sich die Clients vom Server erwarten können.
    **−Invariante:** welche Eigenschaften inmkonsistenten Programmzuständen immer erfüllt sind.
    **−Histrory-Constraint:** wie und vor allem in welchen Aufruf-Reihenfolgen Clients mit dem Server interagieren können.

## 1.3 Typisierung

32. **Wann sind Typen miteinander konsistent, und was sind Typfehler?**
Wenn die Typen der Operanden mit der Operation zusammenpassen, sind die Typen konsistent. Andernfalls trutt ein Typfehler auf.

33. **Wie schränken Typen die Flexibilität ein, und warum verwendet man Typen trotzdem?**
Wir müssen uns für einen Typen entscheiden, dieser ändert sich im Laufe des Programms nicht. Wir sind also nicht so flexibel.
Jedoch verbesserte Lesbarkeit und Zuverlässigkeit.

34. **Welche Gründe sprechen für den Einsatz statischer Typprüfungen, welche dagegen?**
Pro: verbesserte Zuverlässigkeit, explizit hingeschriebene Typen erleichtern das Lesen und Verstehen der Programme.
Contra: man ist nicht so flexibel wie bei dynamischer Typprüfung.

35. **Was versteht man unter Typinferenz? Welche Gründe sprechen für bzw. gegen den Einsatz?**
Wenn Typen nicht explizit angegeben werden und der Compiler diese aus der Programmstruktur herleitet spricht man von Typinferenz.
Pro: erspart das schreiben von Typen, bis zu einem gewissen Grad bessere Verständlichkeit (kleine Programme wo klar ist welcher Typ gemeint ist)
Contra: wenn die Programmstruktur zu komplex wird sehr unverständlich

36. **Zu welchen Zeitpunkten können Entscheidungen getroffen werden (Typen und Entscheidungsprozesse)?**

- **Sprachdefinition / Sprachimplementierung**
  Es ist festgelegt das int eine ganze Zahl in 32-Bit-Zweierkomplement ist
- **Erstellung von Übersetzungseinheiten**
  die meisten wichtigsten Entscheidungen, auf die man beim Programmieren Einfluss hat werden hier getroffen.
- **Einbindung**
  wichtige Entscheidungen werden durch Parametrisierung beim *Einbinden* der Module, Klassen oder Komponenten getroffen.
- **Compiler**
  meist Optimierungen
- **Laufzeit**
  **−Initialisierungsphase**
  Einbindung von Modularisierungseinheiten und der Parametrisierung, wo dies erst zur Laufzeit möglich ist.
  **−eigentliche Programmausführung**
  if else und so i guess

37. **Welchen Einfluss können Typen auf Entscheidungszeitpunkte haben?**
Frühere Entscheidungen machen zur Laufzeit weniger Arbeit.
−Bsp.: eine Variable hat den Typ int. Der Compiler reserviert den benötigten Speicherplatz und man kann im Anschluss davon ausgehen das hier nur ein int daher kommt.
−Bsp.: eine Variable hat den Typ Object. Der Compiler reserviert Platz für eine Referenz. Wir müssen im Programm Fallunterscheidungen machen, da unterschidliche Arten von Werten daher kommen können. (eher nd so leiwand)

38. **Wie beeinflussen Typen die Planbarkeit weiterer Schritte?**
Frühe Entscheidungen erleichtern die Planbarkeit weiterer Schritte. Wenn man weiß, dass eine variable vom Typ int ist muss man kaum mehr Überlegungen daruber anstellen. Um sich auf einen Typen festzulegen muss man planen, wie bestimmte Programmteile im fertigen Programm verwendet werden.

39. **Was ist ein abstrakter Datentyp?**
bezieht sich in erster Linie auf die Trennung der Innenansicht von der Außenansicht (*Data-Hiding*). Im wesentlichen eine Schnittstelle einer Modularisierungseinheit. Man sieht von außen ein gedankliches Konzept (z.B.: int ist eine Zahl) aber die inneren Implementierungsdetails (z.B.: genau Repräsentation von int in der Maschine) bleiben verborgen. Somit ist die Modularisierungseinheit als ganzes abstrakt. Im Idealfall eine Analogie in der realen Welt.

40. **Was unterscheidet strukturelle von nominalen Typen?**

    - **strukturelle Typen**
      Typ der Modularisierungseinheit hängt nur von dem Namen, Parametertypen und Ergebnistypen der nach außen sichtbaren Inhalten ab.
    - **nominale Typen**
      Neben einer Signatur haben sie auch einen eindeutigen Namen. Der Typ eines Objekts entsrpicht dem Namen der Klasse, von dem das Objekt erzeugt wurde.

41. **Warum verwenden wir in Programmiersprachen meist nominale Typen, in theoretischen Modellen aber hauptsächlich strukturelle?**
Man muss jeder Modularisierungseinheit einen eigentlich nicht verwendeten Inhalt mitgeben, dessen Name das Konzept dahinter beschreibt. Damit werden gleiche Signaturen für unterschiedliche Konzepte ausgeschlossen. Wegen dieser Möglichket betrachtet man in der Therorie fast nur *strukturelle Typen*.
In der Praxis ist die Abstraktion von überragender Bedeutung. Man demkt beim Programmieren hautpsächlich akstrakt in Konzepten. Daher verwenden Programmiersprachen zum Großteil *nominale Typen*.

42. **Wie hängen Untertypbeziehungen mit Ersetzbarkeit zusammen?**
Untertypbeziehungen werden durch das Ersetzbarkeitsprinzip definiert: U ist Untertyp von T wenn jedes Objekt von U überall verwendbar ist wo ein Objekt von T erwartet wird. Ohne Ersetzbarkeit gibt es keine Untertypen.

43. **Warum kann ein Compiler ohne Unterstützung durch Programmierer_innen nicht entscheiden, ob ein nominaler Typ Untertyp eines anderen nominalen Typs ist?**
Abstrakte und daher fixer Regeln nicht zugängliche Konzepte lassen sich nicht automatisch vergleichen um so zu ermitteln ob ein Typ Untertyp des anderen ist.

44. **Erklären Sie Einschränkungen bei Untertypbeziehungen zusammen mit statischer Typprüfung.**
Typen von Funktions- bzw. Methoden-Parametern dürfen in Untertypen nicht stärker werden.
Obertyp T hat Methode `boolean compare(T x)`, dann kann Untertyp U diese nicht durch `boolean compare(U x)` überschreiben. Dies wird in der Praxis of benötigt aber es gibt keine Möglichkeit, entsprechende Typen statisch zu prüfen.

45. **In welchem Zusammenhang verwendet man Higher-Order-Subtyping und F-gebundene Generizität?**
**Higher-Order-Subtyping** beschreibt Einschränkungen über Untertyp-ähnliche Beziehungen, die wegen Unterschieden in Details aber keine Untertypbeziehungen sind (C++, Haskell).
**F-gebundene Generizität** nutzt hingegen tatsächliche Untertypbeziehungen zur Beschreibung von Einschränkungen(Java, C#)

46. **Wie konstruiert man rekursive Datenstrukturen?**
Man beginnt mit einer endlichen Menge z.b. $\boldsymbol{M_0 = \{end\}}$. Dann beschreibt man wie man über endlich viele Möglichkeiten aus einer Menge $\boldsymbol{M_i}$ die Menge $\boldsymbol{M_{i+1}}$ generieren kann.
−*induktive Konstruktion*

47. **Was versteht man unter Fundiertheit rekursiver Datenstrukturen? Welche Ansätze dazu kann man unterscheiden?**
Man muss klar zwischen $\boldsymbol{M_0}$ (nicht-rekursiv) und der Konstruktion aller $\boldsymbol{M_i}$ mit i $>$ 0 (rekursiv) unterscheiden, wobei $\boldsymbol{M_0}$ nicht leer sein darf.
Diese Eigenschaft nennt man *Fundiertheit*.
−In Haskell muss jede Typdefinition min. eine nicht-rekursive Alternative haben (`data Lst = end | elem(Int, Lst)`)
−In Java ist die Menge $\boldsymbol{M_0}$ schon in der Sprachdefinition vorgegeben und enthält nur **null**

48. **Warum wird Typinferenz in objektorientierten Sprachen meist nur lokal beschränkt eingesetzt?**
Typinferenz erspart das Anschreiben von Typen, ändert aber nichts an der durch statische Typprüfungen reduzierten Flexibilität. Gerade im lokalen Bereich vermeidet Typinferenz das mehrfache Hinschreiben des gleichen Typs an nahe beieinander liegenden Stellen, ohne den Dokumentationseffekt zu zerstören. Das kann die Lesbarkeit sogar verbessern.

49. **Wie können statisch geprüfte Typen beliebige Eigenschaften von Werten propagieren?**
Bei Funktionsaufrufen werden Informationen über das Argument an die aufgerufen Funktion propagiert (= verbreitet). Entsprechendes gilt auch für das Propagieren von Informationen von der aufgerufenen Funktion zur Stelle des Aufrufs unter Verwendung des Ergebnistyps.
Funktioniert nicht nur für Typen im herkömmlichen Sinn, sondern für alle statisch bekannten Eigenschaften.

## 1.4 Objektorientierte Programmierung

50. **Erklären Sie folgende Begriffe:**

- **Objekt**
Objekte sind grundlegende Modularisierungseinheit (siehe 13). Zur Laufzeit besteht die Software aus einer Menge von Objekten, die einander teilweise kennen und untereinander Nachrichten austauschen.
Daten und Methoden im Objekt sind untrennbar miteinander verbunden: Die Methodenn benötigen die Daten zur Erfüllung ihrer Aufgaben, und die genaue Bedeutung der Daten ist nur den Methoden des Objekts bekannt. Gut durchdachte Kapselung ist hier ein wichtiges Qualitätsmerkmal.

  Jedes Objekt besitzt folgende Eigenschaften:

  - **Identität**
  Objekt ist durch unveränderliche Identität eindeutig bestimmt. Über diese kann man das Objekt ansprechen. Vereinfacht stellt man sich die Identität als Speicheradresse des Objekts vor.
  Objekte sind *identisch*, wenn sie am selben Speicherplatz liegen.
  - **Zustand**
  Setzt sich aus den Werten der Variablen im Objekt zusammen (änderbar).
  Objekte sind *gleich*, wenn sie den gleichen Zustand haben und das gleiche Verhalten haben. Objekte können auch gleich sein, wenn sie nicht identisch sind; dann sind sie *Kopien* voneinander.
  - **Verhalten**
  Beschreibt wie sich das Objekt beim Empfangen einer Nachricht verhält (z.B.: Methodenaufruf).
  - **Schnittstelle**
  Beschreibt das Verhalten des Objekts in einem Abstraktionsgrad, der für Zugriffe von außen notwendig ist (kann über Zusicherungen belibig genau beschrieben werden).
  Ein Objekt *implementiert* seine Schnittstellen; legt das beschriebene Verhalten im Detail fest.

- **Klasse**
  Jedes Objekt gehört zu der Klasse, in der das Objekt implementiert ist. Die Klasse beschreibt Konstruktoren zur Initialisierung neuer Objekte.
- **Vererbung** Ermöglicht, neue Klassen aus bereits existierenden Klassen abzuleiten. In *abgeleiteten* Klassen werden nur die unterschiede zur Oberklasse beschrieben (weniger Schreibaufwand). Auch Programmänderungen werden dadurch vereinfacht.
  Unterklassen können aber nicht beliebig Verändert werden. Es gibt nur zwei Änderungsmöglichkeiten:
  - **Erweiterung**
    Die Unterklasse erweitert die Oberklasse um Variablen/Methoden/Konstruktoren
  - **Überschreiben**
    Methoden der Oberklasse werden durch neue Methoden überschrieben.
- **deklarierter, statischer und dynamischer Typ**

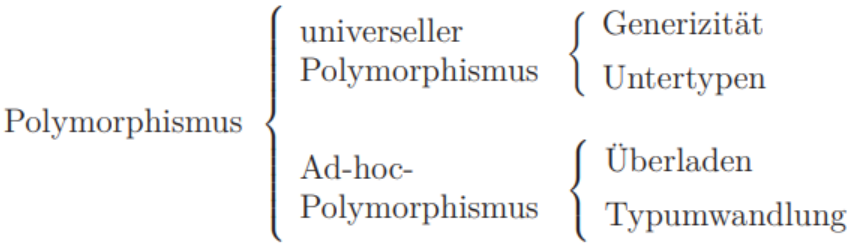
  - **Deklarierter Typ**
    Ist der Typ, mit dem die Variable deklariert wurde (existiert nur bei expliziter Typdeklaration).
  - **Dynamischer Typ**
    Ist der *spezifischste* Typ, den der in der Variablen gespeicherte Wert hat. Sind oft spezifischer als deklarierte Typen und können sich mit jeder Zuweisung ändern. Verwendet für z.B.: **dynamisches Binden**.
  - **Statischer Typ**
    Wird statisch vom Compiler ermittelt und liegt irgendwo zwischen deklariertem und dynamischem Typ. In Sprachdefinitionen kommen statische Typen nicht vor.
- **Faktorisierung, Refaktorisierung**
  *Faktorisierung* beschreibt die Zerlegung eines Programms in Einheiten mit zusammengehörigenn Eigenschaften. Statt zum Beispiel viele gleiche Codestücke zu haben, sollen diese in eine Methode zusammengefasst werden.
  **Faustregel:** Gute Faktorisierung kann die Wartbarkeit eines Programmes wesentlich verbessern.

  Bei guter Faktorisierung ist die Wahrscheinlichkeit kleiner, dass be Programmänderungen auch die Zerlegung in Klassen und Objekte geändert werden muss (*Refaktorisierung*).

- **Verantwortlichkeiten, Klassenzusammenhalt, Objektkopplung**

  - **Verantwortlichkeiten**
    (Verantwortlichkeiten einer Klasse)
    * "was ich weiß" - Beschreibung des Zustands der Objekte
    * "was ich mache" - Verhalten der Objekte
    * "wen ich kenne" - sichtbare Objekte, Klassen, etc.
  - **Klassenzusammenhalt**
    *Grad der Beziehungen zwischen den Verantwortlichkeiten der Klasse.*
    **Faustregel:** Klassenzusammenhalt soll hoch sein.
    Der Zusammenhalt ist hoch, wenn alle Variablen und Methoden eng zusammenarbeiten und durch den Namen der Klasse gut beschrieben sind. Sprich, wenn man einer Klasse mit hohem Zusammenhalt Variablen oder Methoden mopst, fehlt ihr etwas Wichtiges.
  - **Obejktkopplung**
    *Abhängigkeit der Objekte voneinander.*
    **Faustregel:** Objektkopplung soll schwach sein.
    Die Objektkopplung ist stark, wenn
    * viele Methoden und Variablen nach außen sichtbar sind,
    * im laufenden System Nachrichten und Variablenzugriffe zwischen unterschiedlichen Objekten häufig auftreten
    * und die Anzahl der Parameter dieser Methoden groß ist.

51. **Welche Arten von Polymorphismus unterscheidet man? Welche davon sind in der objektorientierten Programmierung wichtig? Warum?**




    In der objektorientierten Programmierung sind Untertypen von überragender Bedeutung. Man erzielt Gleichförmigkeit durch gemeinsame Schnittstellen für unterschiedliche Objekte.

52. **Wann sind zwei gleiche Objekte identisch und wann sind zwei identische Objekte gleich?**
    Zwei gleiche Objekte sind identisch, wenn sie am selben Speicherplatz liegen. Es handelt sich also um **ein** Objekt mit zwei Namen.
    Zwei identische Objekte sind immer gleich, da es sich ja um ein und das selbe Objekt handelt.

53. **Sind Datenabstraktion, Datenkapselung und Data-Hiding einander entsprechende Begriffe? Wenn Nein, worin unterscheiden sie sich?**
*siehe 17*

54. **Was besagt das Ersetzbarkeitsprinzip? (Häufige Prüfungsfrage!)**
Ein Typ U ist Untertyp eines Typs T, wenn jedes Objekt von U überall verwendbar ist, wo ein Objekt von T erwartet wird.

55. **Warum ist Ersetzbarkeit in der objektorientierten Programmierung so wichtig (mehrere Gründe)?**
−Dadurch werden nachträgliche Änderungen an Modularisierungseinheiten praxistauglich.
−Hohe Code-Wiederverwendung
−Ohne Ersetzbarkeit gibt es keine Untertypen

56. **Wann und warum ist gute Wartbarkeit wichtig?**
Gute Wartbarkeit erpart Zeit und Geld.

57. **Wie lauten die wichtigsten Faustregeln im Zusammenhang mit Klassenzusammenhalt und Objektkopplung? Welche Vorteile kann man sich davon erwarten, dass diese Faustregeln erfüllt sind?**
*siehe 50 (Klassenzusammenhalt & Objektkopplung)*

58. **Von welchen Kriterien hängen Klassenzusammenhalt und Objektkopplung genau ab?**
*siehe 50 (Klassenzusammenhalt & Objektkopplung)*

59. **Wie kann man überprüfen, wie hoch der Klassenzusammenhalt und wie stark die Objektkopplung ist?**
Lässt sich im eigentlichen Sinn nicht überprüfen, jedoch können wir durch Überlegungen, zusammengehörige Dinge miteinander assoziieren und somit, Klassenzusammenhalt und Objektkopplung bestimmen. (ob hoch/niedrig bzw stark/schwach)

60. **Wie wirkt sich ein hoher Klassenzusammenhalt auf die Faktorisierung und Refaktorisierbarkeit aus?**
hoher Klassenzusammenhalt → gute Faktorisierung und geringe Wahrscheinlichkeit das Refaktorisierung betrieben werden muss.

61. **Wie wirkt sich eine schwache Objektkopplung auf die Faktorisierung und Refaktorisierbarkeit aus?**
Klassenzusammenhalt und Obejktkopplung gehen Hand in Hand. Daher haben wir bei hohem Klassenzusammenhalt eine schwache Obejktkopplung (oft aber nicht immer).
Somit: schwache Obejktkopplung → gute Faktorisierung und geringe Wahrscheinlichkeit das Refaktorisierungen betrieben werden muss.

62. **Welche Arten von Software kann man wiederverwenden, und welche Rolle spielt jede davon in der Softwareentwicklung?**

    - **Programme**
      Werden meistens im Hinblick auf häufige (Wieder)Verwendung entwickelt.
    - **Daten**
      Werden auch häufig wiederverwendet. Sind meist sogar langlebiger als die Programme die sie benötigen oder manipulieren.
    - **Erfahrungen**
      Konzepte und Ideen werden in Form von Erfahrungen wiederverwendet.
    - **Code**
      Konzepte wie Untertypen, Vererbung un Generizität wurden im Hinblick auf Wiederverwendung von Code entwickelt. (Bibliotheken, Projektinterne Wiederverwendung, Programminterne Wiederverwendung)

63. **Welche Rolle spielen Refaktorisierungen in der Wiederverwendung?**
    **Faustregel:** Ein vernünftiges Maß rechtzeitiger Refaktorisierung führt häufig zu gut faktorisierten Programmen.

64. **Wofür ist die objektorientierte Programmierung gut geeignet, und wofür ist sie nicht gut geeignet?**
    **Faustregel:** Objektorientierte Programmierung eignet sich zur Entwicklung von Systemen, deren Gesamtkoplexität jene der einzelnen Algorithmen deutlich übersteigt. Sonst sind andere Paradigmen besser geeignet.

---

# 2 Untertypen und Vererbung

## 2.1 Das Ersetzbarkeitsprinzip

1. **In welchen Formen (mindestens zwei) kann man durch das Ersetzbarkeitsprinzip Wiederverwendung erzielen?**

    - Erweiterung, jedoch beibehalten der Schnittstelle
    - Ersetzbarkeit (Faktorisierung)
    - interne Code-Wiederverwendung

2. **Wann ist ein struktureller Typ Untertyp eines anderen strukturellen Typs? Welche Regeln müssen dabei erfüllt sein? Welche zusätzliche Bedingungen gelten für nominale Typen bzw. in Java? (Hinweis: Häufige Prüfungsfrage!)**
   allgemein gilt:

   - **reflexiv** - jeder Typ ist Untertyp von sich selbst.
   - **transitiv** - ist ein Typ U Untertyp eines Typs A und ist A Untertyp eines Typs T, dann ist U auch Untertyp von T.
   - **antisymmetrisch** - ist ein Typ U Untertyp eines Typs T und ist T außerdem Untertyp von U, dann sind U und T äquvalent.

   Es gilt "U ist Untertyp von T" wenn folgende Bedingungen erfüllt sind:

   - Für jede Konstante in T gibt es eine entsprechende Konstante (nach init nur lesende Zugriffe) in U, wobei der deklarierte Typ B der Konstante in U ein Untertyp des deklarierten Typs A der Konstante in T ist.
   - Für jede Variable in T gibt es eine entsprechende Variable in U, wobei die deklarierten Typen der Variablen äquivalent sind.
   - Für jede Methode in T gibt es eine entsprechende gleichnamige Methode in U, wobei
     - der deklarierte Ergebnistyp der Methode in U ein Untertyp des deklarierten Ergenistyps der Methode in T ist,
     - die Anzahl der formalenn Parameter beider Methoden gleich ist
     - und der deklarierte Typ jedes formalen Parameters in U ein Obertyp des deklarierten Typs des entsprechenden formalen Parameters in T ist (gilt nur für Eingangsparameter).

3. **Sind die in Punkt 2 angeschnittenen Bedingungen (sowie das, was Compiler prüfen können) hinreichend, damit das Ersetzbarkeitsprinzip erfüllt ist? Wenn nicht, was muss noch beachtet werden?**
   Nein, das Verhalten der Unterklasse muss zusätzlich zu jenem seiner Oberklasse passen.

4. **Was bedeutet Ko-, Kontra- und Invarianz, und für welche Typen in einer Klasse trifft welcher dieser Begriffe zu? (Hinweis: Häufige Prüfungsfrage!)**

   - **Kovarianz** - Der deklarierte Typ eines Elements im Untertyp ist ein Untertyp des deklarierten Typs des entsprechenden Elements im Obertyp. (z.b. Typen von Konstanten, Ergenisse der Methoden und Ausgangsparameter)
   - **Kontravarianz** - Der deklarierte Typ eines Elements im Untertyp ist ein Obertyp des deklarierten Typs des entsprechenden Elements im Obertyp. (z.b. deklarierte Typen von formalen Eingangsparametern)
   - **Invarianz** - Der deklarierte Typ eines Elements im Untertyp ist äquivalent zum deklarierten Typ des entsprechenden Elements im Obertyp. (z.b. deklarierte Typen von Variablen und Durchgangsparameter)

5. **Was sind binäre Methoden, und welche Schwierigkeiten verursachen sie hinsichtlich der Ersetzbarkeit?**
   Eine Methode bei der ein formaler Parametertyp stets gleich der Klasse ist, in der die Methode definiert ist, heißt *binäre Methode*.
   Bsp: `public class Haus { public boolean equal(Haus h); }`

   **Faustregel:** Kovariante Eingangsparametertypen und binäre Methoden wiedersprechen dem Ersetzbarkeitsprinzip. Es ist sinnlos, in solchen Fällen Ersetzbarkeit anzustreben.

6. **Wie soll man Typen formaler Parameter wählen um gute Wartbarkeit zu erzielen?**
   **Faustregel:** Man soll Parametertypen vorrausschauend un möglichst allgemein wählen.

7. **Warum ist dynamisches Binden gegenüber switch- oder geschachtelten if-Anweisungen zu bevorzugen?**

```
public void gibAnredeAus(int art, String name) {
    switch(art) {
        case  1: System.out.print("S.g. Frau " + name);
                 break;
        case  2: System.out.print("S.g. Herr " + name);
                 break;
        default: System.out.print(name);
    }
}
```

```
public class Adressat {
    protected String name;
    public void gibAnredeAus() {
        System.out.print(name);
    }
    ... // Konstruktoren und weitere Methoden
}
public class WeiblicherAdressat extends Adressat {
    public void gibAnredeAus() {
        System.out.print ("S.g. Frau " + name);
    }
}
public class MaennlicherAdressat extends Adressat {
    public void gibAnredeAus() {
        System.out.print ("S.g. Herr " + name);
    }
}
```

   Dynamisches Binden liefert hier bessere Lesbarkeit. Weiters kann einfach die Methode `gibAnredeAus()` aufgerufen werden und es kommt automatisch die richtige Anrede. Ein weiterer Vorteil ist das leichte Erweitern um eine dritte Klasse, welche zum Beispiel die Anrede einer Firma repräsentiert.

8. **Dient dynamisches Binden der Ersetzbarkeit und Wartbarkeit?**
   -

## 2.2 Ersetzbarkeit und Objektverhalten

9. **Welche Arten von Zusicherungen werden unterschieden, und wer ist für die Einhaltung verantwortlich? (Hinweis: Häufige Prüfungsfrage!)**

   - **Vorbedingung** - Für die Einhaltung der Vorbedingung ist jeder einzelne Client verantwortlich.
   - **Nachbedingung** - Für die Einhaltung der Nachbedingung ist der Server verantwortlich.
   - **Invariante** - Für die Einhaltung der Invarianten auf Objektvariablen sowohl vor als auch nach der Ausführung jeder Methode ist grunsätzlich der Server verantwortlich. Direkte Schreibzugriffe auf Variablen muss jedoch der Client selbst verantworten.
   - **History-Constraint** - Schränken die Entwicklung von Objekten im Laufe der Zeit ein.
     Wir unterscheiden zwei Unterarten:
     - **Server-kontrolliert** - Ähneln Invarianten, schränken naber zeitliche Veränderungen der Variableninhalte eines Objekts ein.
       (z.b. Counter kann im Laufe der Zeit größer werden, aber niemals kleiner.)
     - **Client-kontrolliert** - Man kann mit History-Constraints die Reihenfolge von Methodenaufrufen einschränken. Für diese ist der Client verantwortlich.
       (z.b. `initialize()` muss in einem Objekt als erstes aufgerufen werden, davor sind keine anderen Aufrufe erlaubt.)

   Am Ende des Tages ist aber natürlich der Programmierer für die Einhaltung der Zusicherungen verantwortlich.

10. **Wie müssen sich Zusicherungen in Unter- und Obertypen zueinander verhalten, damit das Ersetzbarkeitsprinzip erfüllt ist? Warum? (Hinweis: Häufige Prüfungsfrage!)**
    (es sein Typ U Untertyp von Typ T)

    - **Vorbedingung** - Jede Vorbedingung auf einer Methode in T muss eine Vorbedingung auf der entsprechend Methode in U implizieren.
      Das heißt: Vorbedingungen im Untertyp können schwächer, dürfen aber nicht stärker als im Obertyp sein.
    - **Nachbedingung** - Jede Nachbedingung auf einer Methode in U muss eine Nachbedingung auf der entsprechenden Methode in T implizieren.
      Das heißt: Nachbedingungen im Untertyp können stärker, dürfen aber nicht schwächer als im Obertyp sein.
    - **Invariante** - Jede Invariante in U muss eine Invariante in T implizieren.
      Das heißt: Invarianten im Untertyp können stärker, dürfen aber nicht schwächer als im Obertyp sein.

- **Server-kontrollierter History-Constraint** - Hierfür gilt das Gleiche wie für Invarianten.
- **Client-kontrollierter Histroy-Constraint** - Hierfür gilt das Gleiche wie für Vorbedingungen, jedoch bezogen auf Einschränkungen in der Reihenfolge von Methodenaufrufen.

11. **Warum sollen Signaturen und Typen stabil bleiben? Wo ist Stabilität besonders wichtig?**
    besser Wartbarkeit.

    **Faustregel:** Zusicherungen sollen stabil bleiben. Das ist für Zusicherungen in Typen nahe an der Wurzel der Typhierarchie ganz besonders wichtig.

12. **Was ist im Zusammenhang mit allgemein zugänglichen (= möglicherweise nicht nur innerhalb des Objekts geschriebenen) Variablen und Invarianten zu beachten?**
    Diese müssen in Zusicherungen bedacht werden.

13. **Wie genau sollen Zusicherungen spezifiziert sein?**
    **Faustregel:** Zur Verbesserung der Wartbarkeit sollen Zusicherungen keine unnötigen Details festlegen.
    **Faustregel:** Alle von geschulten Personen nicht in jedem Fall erwarteten, aber vorausgesetzten Eigenschaften sollen explizit als Zusicherungen im Programm stehen.

14. **Wozu dienen abstrakte Klassen und abstrakte Methoden? Wo und wie soll man abstrakte Klassen einsetzen?**
    Abstrakte Klassen sind Klassen, von denen es keine Instanz gibt. Sie werden häufig für die Festlegung von Typen verwendet (Methoden Parameter, ...).

    **Faustregel:** Es ist empfehlenswert, als Obertypen und Parametertypen hauptsächlich abstrakte Klssen (in Java vor allem Interfaces) zu verwenden.

## 2.3 Vererbung versus Ersetzbarkeit

15. **Ist Vererbung das gleiche wie das Ersetzbarkeitsprinzip? Wenn Nein, wo liegen die Unterschiede?**
    Nein.
    **Vererbung** - Erweitern und Überschreiben der Oberklasse.
    **Ersetsbarkeitsprinzip** - Das Auftreten eines Elements kann durch ein anderes ersetzt werden.

16. **Worauf kommt es zur Erzielung von Codewiederverwendung eher an, auf Vererbung oder Ersetzbarkeit? Warum?**
**Faustregel:** Wiederverwendung durch das Ersetzbarkeitsprinzip ist wesentlich wichtiger als direkte Wiederverwendung durch Vererbung.
Warum? - Unter Umständen gewinnt man zwar durch die reine Vererbung besser direkte Codewiederverwendung in kleinem Umfang, tauscht diese aber gegen viele Möglichkeiten für die indirekte Codewiederverwendung in großem Umfang, die nur duch die Ersetzbarkeit gegeben ist.

17. **Was bedeuten folgende Begriffe in Java?**

    - **Objektvariable, Klassenvariable, statische Methode**

        - **Objektvariablen** - auch Instanzvariablenn genannnt, sind Variablen, die zu Objekten gehören.
        - **Klassenvariablen** - gehören nicht zu Objekten der Klasse, sonder zur Klasse selbst (keyword *static*).
        - **statische Methode** - gehören auch zur Klasse selbst

    - **Static-Initializer**
    Sind wie Konstruktoren für Objekte nur für Klassenvariablen. Bestehen nur aus dem Schlüsselwort *static*{ ...} und einer belibigen Sequenz von Anweisungen.

    - **geschachtelte und innere Klasse**

        - **Statische geschachtelte Klassen** - gehören zur umschließenden Klasse selbst.
        Sie dürfen nur Klassenvariablen und statische Methoden der umschließenden Klasse verwenden.
        - **Inner Klassen** - gehören zu einem Objekt der umschließenden Klasse.
        Sie dürfen Objektvariablen und nichtstatische Methoden der umschließenden Klasse frei verwenden. Innere Klassen dürfen jedoch keine statischen Methoden/ statische geschachtelte Klassen enthalten, da diese von einem Objekt der äußeren Klasse abhängen würden und somit nicht mehr statisch wären.

    - **final Klasse und final Methode**

        - **final Klasse** - solche Klassen haben keine Unterklasse.
        - **final Methode** - solche Methoden können in einer Unterklasse nicht überschrieben werden.

    - **Paket, Class-Path, import-Anweisung**

        - **Paket** - Jede compilierte Java-Klasse wird in einer eigenen Datei gespeichert (*someclass.class*). Das Verzeichnis, das diese Datei enthält, entspricht dem Paket, zu dem die Klasse gehört; das ist ein Namensraum.

- **Class-Path** - Basis der Verzeichnisstruktur für Java-Klassen.
- **import-Anweisung** - *"import myclasses.examples.test;*

18. **Wo gibt es in Java Mehrfachvererbung, wo Einfachvererbung?**
Klassen in Java unterstützen nur Einfachvererbung. Interfaces hingegen erlauben Mehrfachvererbung.

19. **Welche Arten von import-Deklarationen kann man in Java unterscheiden? Wozu dienen sie?**
-

20. **Wozu benötigt man eine package-Anweisung?**
legt fest zu welchem Paket die Klasse in der Quelldatei gehört (*package paketName;*).

21. **Welche Möglichkeiten zur Spezifikation der Sichtbarkeit gibt es in Java, und wann soll man welche Möglichkeit wählen?**

| | `public` | `protected` | — | `private` |
|---|---|---|---|---|
| sichtbar im selben Paket | ja | ja | ja | nein |
| sichtbar in anderem Paket | ja | nein | nein | nein |
| ererbbar im selben Paket | ja | ja | ja | nein |
| ererbbar in anderem Paket | ja | ja | nein | nein |

**Faustregel:** Fast alle Variablen sollten *private* sein.

22. **Wodurch unterscheiden sich Interfaces in Java von abstrakten Klassen? Wann soll man Interfaces verwenden? Wann sind abstrakte Klassen besser geeignet?**
(*siehe 18*)
Interfaces haben keine konkrete Implementierung nur abstrakte Methoden und Zusicherungen.
Abstrakte Klassen haben konkrete Implementierung und Code-Vererbung, sollten verwendet werden wenn genau dies nötig ist.
**Faustregel:** Interfaces sind Klassen vorzuziehen.

---

# 3 Generizität und Ad-hoc-Polymorphismus

1. **Was ist Generizität? Wozu verwendet man Generizität?**
   Generische Klassen, Typen und Methoden enthalten Parameter, für die Typen eingesetzt werden. (auch *Typparameter* genannt). Generizität ist ein statische Mechanismus, dynamisches Binden wie bei Untertypen ist nicht nötig.
   Bsp.: Wir wollen eine Liste für `Strings` implementieren. Später kommen wir drauf, das wir auch Listen für `Integer` und `Studenten` brauchen. Also müssten wir zwei extra Klassen für `Integer` und `Student` implementieren. Hier kommt jedoch Generizität ins Spiel. Wir können statt einer Liste für `String` eine Liste mit einem generischen Typparameter implementiern und können so die Liste für alle drei Typen verwenden.

2. **Was ist gebundene Generizität? Was kann man mit Schranken auf Typparametern machen, das ohne Schranken nicht geht?**
   Manchmal braucht man über Typparameter mehr Informationen, die man bei einfacher Generizität nicht hat. (z.b. ob ein Typ eine bestimmte Methode hat). Bei der gebundenen Generizität kann man mit Schranken festlegen, dass der generische Typ Untertyp der Schranke sein muss.

3. **In welchen Fällen soll man Generizität einsetzen, in welchen nicht?**
   Gennerell ist der Einsatz von Generizität immer sinnvoll, wenn er die Wartbarkeit verbessert.
   Man soll Generizität immer verwenden, wenn es mehrere gleich strukturierte Klassen (oder Typen) beziehungsweise Methoden gibt.

   **Faustregel:** Containerklassen sollen generisch sein.
   **Faustregel:** Klassen und Methoden in Bibliotheken sollten generisch sein.

   Man soll Generizität eher nicht zur Vermeidung von dynamischem Binden verwenden, da sich dadurch die Struktur des Programms ändert und sich das auf die Laufzeiteffizientz auswirken kann.

4. **Was bedeutet statische Typsicherheit in Zusammenhang mit Generizität, dynamischen Typabfragen und Typumwandlungen?**

   - **statische Typsicherheit** - Generizität bietet statische Typsicherheit. Bereits der Compiler garantiert, dass in ein Objekt von `List<String>` nur Zeichenketten eingefügt werden können.
   - **dynamische Typabfragen** - Typabfragen zur Laufzeit mit z.b `instanceof`
   - **Typumwandlungen** - casten (sollte bekannt sein, aka ich will nd schreiben :) )

5. **Was sind (gebundene) Wildcards als Typen in Java? Wozu kann man sie verwenden?**
   Bsp.: `void drawAll (List<? extends Polygon> p) {...}`
   Das Fragezeichen steht for einen beliebigen Typ, der ein Untertyp von Polygon ist. Wenn man nur `void drawAll (List<Polygon> p)` verwendet können tatsächlich nur Objekte der Klasse Polygon in diese Liste eingefügt werden, selbst wenn `Viereck` ein Untertyp von `Polygon` wäre, wäre dies nicht möglich.

6. **Welche Arten von Generizität kann man hinsichtlich ihrer Übersetzung und ihrem Umgang mit Schranken unterscheiden? Welche Art wird in Java verwendet, und wie flexibel ist diese Lösung?**
   zwei Möglichkeiten:

   - **homogene Übersetzung** - In Java verwendet. Dabei wird jede generische Klasse, so wie auch jede nicht-generische Klasse, in genau eine Klasse mit JVM-Code übersetzt. Gebundene Typparameter werden durch erste Schranke des Typparameters ersetzt. Ungebundene durch `Object`. Ergebnisse und Argumente werden in die nötigen deklarierten Typen umgewandelt, wenn die entsprechenden Ergebnistypen und formalen Parametertypen Typparameter sind, die durch Typen ersetzt wurden.
     Compiler garantiert Typkompativilität.
   - **heterogene Übersetzung** - Hier wird für jede Verwendung einer generischen Klasse oder Methode mit anderen Typparametern eigener übersetzter Code erzeugt. Nachteil hierbei ist die große Anzahl an übersetzter Klassen und Methoden. Jedoch braucht es zur Laufzeit keine Typumwandlungen, wie es bei der homogenen Übersetztung der Fall ist.

7. **Wie kann man Generizität simulieren? Worauf verzichtet man, wenn man Generizität nur simuliert?**
   Man kann beispielsweise Collections für den Typ `Object` schreiben. Da alle Typen Untertypen von `Object` sind kann man diese Klasse so verwenden, als wäre sie generisch. Jedoch kann der Compiler hier keine statische Typsicherheit mehr garantiernen.

8. **Was wird bei der heterogenen bzw. homogenen Übersetzung von Generizität genau gemacht?**
   *siehe 6*

9. **Was muss der Java-Compiler überprüfen um sicher zu sein, dass durch Generizität keine Laufzeitfehler entstehen?**
   -

10. **Welche Möglichkeiten für dynamische Typabfragen gibt es in Java, und wie funktionieren sie genau?**

    - **getClass()** - Jedes Objekt hat diese Methode, welche die Klasse des Objekts als Ergebnis zurückgibt. Bietet die direkteste Möglichkeit des Zugriffs auf den ndynamischen Typ.
    - **instanceof** - ist ein Operator. Liefert `true` wenn das Objekt links ein Untertyp des rechten Typs ist.

11. **Was wird bei einer Typumwandlung in Java umgewandelt - der deklarierte, dynamische oder statische Typ? Warum?**
    Der deklarierte Typ eines Referenzobjekts wird umgewandelt.
    Der dynamisch ermittelte Typ reich nicht aus um auf Methoden des deklarierten Typs zuzugreifen. Daher können wir nach der dynamischen Typabfrage einen Cast ausführen um auf die gewünschten Methoden zugreifen zu können.

12. **Welche Gefahren bestehen bei Typumwandlungen?**
    Fehler in einem Programm können verdeckt werden und die Wartbarkeit erschweren. Fehler werden oft dadurch verdeckt, dass der deklarierte Typ einer Variable nur mehr wenig mit dem Typ zu tun hat, dessen Objekte wir erwarten.
    Im günstigsten Fall kommt es zur Laufzeit zu Ausnahmebehandlungen bzw. die Ergebnisse sind falsch. Es kann dadurch aber auch dazu kommen das z.b. falsche Daten in einer Datenbank gespeichert werden.
    Diese Gefahren entstehen, da durch dynamische Typabfragen und Typumwandlungen die statische Typüberprüfung des Compilers ausgeschalten wird, obwohl man wahrscheinlich nach wie vor auf die statische Typsicherheit verlässt.

13. **Wie kann man dynamische Typabfragen und Typumwandlungen vermeiden? In welchen Fällen kann das schwierig sein?**
    Man kann sie durch **dynamisches Binde**n vermeiden. Schwierig vor allem in diesen Fällen:

    - Der deklarierte Typ des Objekts ist zu allgemein.
    - Die Klassen, die dem deklarierten Typ und dessen Untertypen entsprechen, können nicht erweitert werden.
    - Die einzelnen Alternativen greifen auf private Variablen und Methoden zu. (nicht bekannt, da wo das dynamische Binden passiert.)
    - Der deklarierte Typ hat sehr viele Untertypen. Wenn `doSomething()` nicht in einer gemeinsamen Oberklasse implementierbar ist, muss `doSomething()` in vielen Klassen auf die gleiche Weise implementiert werden. → schlechte Wartbarkeit.

14. **Welche Arten von Typumwandlungen sind sicher? Warum?**

- Typumwandlungen auf elementaren Typen (z.b. `int, char, float`) sind sicher, weil hier die tatsächlichen Werte umgewandelt werden, nicht die deklarierten Typen. z.b. Fließkommazahlen werden durch Runden gegen 0 eine ganze Zahl. Es geht zwar Information verloren, aber es kommt zu keiner Ausnahmebehandlungen.
- Typumwandlung in den Obertyp
- Typumwandlung nach dynamischer Typabfrage, welche sicherstellt, dass das Objekt einen entsprechenden dynamischen Typ hat (nicht auf den else Zweig vergessen!)
- Wenn man das Programmstück so schreibt, als ob man Generizität verwenden würde, dieses Programmstück händisch auf mögliche Typfehler, die bei der Verwendung von Generizität zu Tage treten, untersucht und dann die homogene Übersetzung durchführt

15. **Was sind kovariante Probleme und binäre Methoden? Wie kann man mit ihnen umgehen oder sie vermeiden?**

- **kovariante Probleme** - Wir wissen das Eingangsparameter nur kontravariant sein können (siehe 4), sonst wäre das Ersetsbarkeitsprinzip verletzt. In der Praxis wünscht man sich jedoch manchmal kovariante Eingangsparametertypen. Diese Aufgabenstellungen nennt man *kovariante Probleme*. Diese lassen sich mittels dynamische Typabfragen und Typumwandlungen lösen. **Faustregel:** Kovariante Probleme soll man vermeiden.
- **binäre Methoden** - Ein spezialfall kovarianter Probleme (siehe 5). Diese können im Prinzip auf die gleiche Weise behandelt werden wie alle anderen kovarianten Probleme (kann aber problematisch werden. siehe Skriptum Seite 154 (Beispiel mit `equal()`)).

16. **Wie unterscheidet sich Überschreiben von Überladen, und was sind Multimethoden?**

- **Überschreiben** - Die Unterklasse überschreibt eine Methode der Oberklasse (setz vorraus das die Signatur der Methoden exakt gleich sind, sonst Überladen).
- **Überladen** - Die Methode in der Unterklasse koexistiert zur Methode in der Oberklasse
- **Multimethoden** - Die Auswahl der Methode ist nicht nur vom dynamischen Typ, in welchem die Methode aufgerufen wird abhängig (x), sondern auch von den dynamischen Typen der formalen Eingangsparameter (y). (bsp. `x.equal(y)`)

17. **Wie kann man Multimethoden simulieren? Welche Probleme können dabei auftreten?**
    In Java kann man Multimethoden durch mehrfaches dynamisches Binden simulieren, hierbei hat man jedoch schnell sehr viele Methoden (siehe Skriptum Seite 159).

18. **Was ist das Visitor-Entwurfsmuster?**
    Beruht auf mehrfachem dynamischem Binden. Man unterscheidet zwischen Visitorklassen und Elementklassen. Elementklassen rufen Methoden in den Visitorklassen mit sich selbst als Argument auf. Die Visitorklasse ruft dann die entsprechend gebrauchte Methode auf.

19. **Wodurch ist Überladen problematisch, und in welchen Fällen ergeben sich kaum Probleme?**
    -

---

# 4 Kreuz und Quer

1. **Wie werden Ausnahmebehandlungen in Java unterstützt?**
   Ausnahmen sind in Java gewöhnliche Objekte, die über spezielle Mechanismen als Ausnahmen verwendet werden. Alle Obejkte von `Throwable` sind dafür verwendbar. Praktisch verwendet man nur Obejkte der Unterklassen von `Error` und `Exception`, zwei Unterklassen von `Throwable`.
   Um Ausnahmen Abfangen zu können und entsprechend reagieren zu können gibt es `try-catch` Blöcke.

2. **Wie sind Ausnahmen in Untertypbeziehungen zu berücksichtigen?**
   Das Ersetzbarkeitsprinzip verlangt, dass die Ausführung einer Methode eines Unntertyps nur solche Ausnahmen zurückliefern kann, die bei Ausführung der entsprechenden Methode des Obertyps erwartet werden. In Untertypen dürfen Typen von Ausnahmen aber weggelassen werden.

```
class A {
    void foo() throws Help, SyntaxError { ... }
}
class B extends A {
    void foo() throws Help { ... }
}
```

3. **Wozu kann man Ausnahmen verwenden? Wozu soll man sie verwenden, wozu nicht?**

   - **Unvorhergesehene Programmabbrüche** - Wird eine Ausnahme nicht abgefangen, kommt es zu einem Programmabbruch.
   - **Kontrolliertes Wiederaufsetzten** - Nach aufgetretenen Fehlern oder in außergewöhnlichen Situationen wird das Programm an genau definierbaren Punkten weiter ausgeführt.
   - **Ausstieg aus Sprachkonstrukten** - Ausnahmen erlauben das vorzeitige Abbrechen der Ausführung von Blöcken, Kontrollstrukturen, Methoden, etc.
   - **Rückgabe alternativer Ergeniswerte** - Wenn in einer Methode eine unbehandelte Ausnahme auftritt, wird an den Aufrufer statt eines Ergenisses die Ausnahme zurückgegeben, die er abfangen kann. Damit ist es möglich, dass die Methode an den Aufrufer in Ausnahmesituationen Objekte zurückgibt, die nicht den deklarierten Ergebnistyp der Methode haben.

     **Faustregel:** Aus Gründen der Wartbarkeit soll man Ausnahmen und Ausnahmebehandlungen nur in echten Ausnahmesituationen und sparsam einsetzten.

4. **Durch welche Sprachkonzepte unterstützt Java die nebenläufige Programmierung? Wozu dienen diese Sprachkonzepte?**
   Zur Nebenläufigkeit werden mehrere Threads verwendet. Diese kann man mit `synchronized` Methoden synchronisieren. Weiters gibt es auch die Möglichkeit `synchronized` Blöcke zu schreiben. Hier ist nicht die Methode als ganzes atomar sondern nur jene Anweisung im Block.

5. **Wozu brauchen wir Synchronisation? Welche Granularität sollen wir dafür wählen?**
   Wir brauchen sie um inmkonsistente Daten zu vermeiden. Die Granularität der Synchronisation ist so zu wählen, dass kleine, logisch konsistente Blöcke entstehen, in deren Ausführung man vor Veränderungen durch andere threads geschützt ist.

6. **Zu welchen Problemen kann Synchronisation führen, und was kann man dagegen tun?**

   - **Deadlock** - zyklische Abhängigkeiten zwischen zwei oder mehr Threads. (alles steht still)
   - **Livelock** - Threads behindern sich gegenseitig bei der weiteren Ausführung. (Jeder kennts wenn man einer anderen Person ausweichen will und diese in die selbe Richtung ausweicht. Und dann nochmal und nochmal. Ist ein Livelock lol)

7. **Wozu dienen Annotationen? Wann setzt man sie sinnvoll ein?**
   Man kann damit unterschiedliche Programmteile mit Markierungeen versehen und das Laufzeitsystem sowie Entwicklungswerkzeuge prüfen das Vorhandensein bestimmter Markierungen und reagieren darauf entsprechend.

8. **Wie lange können Annotationen leben? Wofür ist welche Lebensdauer sinnvoll?**
   `@Rentation` legt fest, wie weit die Annotation sichtbar bleiben soll. Hier gibt es drei Werte:

   - `SOURCE` - Die Annotation wird vom Compiler genau so wie Kommentare verworfen. Solche Annotationen sind nur für Werkzeuge, die auf dem Source-Code operieren, von Interesse.
   - `CLASS` - Die Annotation bleibt in der übersetzten Klasse vorhanden, ost aber während der Programmausführung nicht mehr sichtbar. Das ist nützlich für Werkzeuge, die auf dem Byte-Code operieren.
   - `RUNTIME` - Die Annotation ist auch zur Laufzeit verfügbar.

9. **Wie kann man eigene Annotationen deklarieren? Welche Gemeinsamkeiten und Unterschiede zu Interfaces bestehen?**
   (Am besten zu sehen an einem Bsp :) )

```
@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.TYPE})
public @interface BugFix {
    String who();      // author of bug fix
    String date();     // when was bug fixed
    int    level();    // importance level 1-5
    String bug();      // description of bug
    String fix();      // description of fix
}
```

10. **Wie kann man zur Laufzeit auf Annotationen zugreifen?**
    Wenn man über @Retention festgelegt hat, dass die Annotation auch zur Laufzeit zugreifbar ist, dann generiert der Compiler ein entsprechendes Interface.

```
public interface BugFix extends Annotation {
    String who();
    String date();
    int    level();
    String bug();
    String fix();
}
```

Auf Annotationen kann man dann wie folgt zugreifen:

```
String s = "";
BugFix a = Buggy.class.getAnnotation(BugFix.class);
if (a != null) {  // null if no such Annotation
    s += a.who()+" fixed a level "+a.level()+" bug";
}
```

11. **Was ist aspektorientierte Programmierung? Wann setze ich sie sinnvoll ein?**
    Die aspektorientierte Programmierung kapselt Verhalten, das mehrere Klassen betrifft in Aspekten. Ein Aspekt beschreibt dabei an einer Stelle sowohl FUnktionalität, als auch alle Stellen im Programm, an denen diese FUnktionalität angewendet werden soll.

    Aspektorientierte Programmierung kann bei der Entwicklung von Frameworks (Libraries) eingesetzt werden, um z. B. Eigenschaften wie Persistenz oder Synchronisierbarkeit zu implementieren. Der Transfermechanismus bleibt dann vor dem Benutzer der Bibliothek verborgen. (*Wikipedia*)

12. **Was bedeutet Separation-of-Concerns?**
    Aufteilung der Funktionalität eines Programms auf einzelne Modularisierungseinheiten (Faktorisierung entsprechend *Separation-of-Concerns*)

13. **Was sind Core-Concerns, was Cross-Cutting-Concerns?**
    In der objektorientierten Programmierung funktioniert die Aufteilung gut für die Kernfunktionalitäten (*Core-Concerns*), die sich leicht in eine Klasse packen lassen. Oft gibt es aber Anforderungen, die alle Bereiche eines Programms betreffen (*Cross-Cutting-Concerns*, Querschnittsfunktionalitäten) → **aspektorientierte Programmierung**.

14. **Was sind Join-Points, Pointcuts, Advices und Aspekte, und wozu braucht man sie?**

    - **Join-Point** - Ist eine identifizierbare Stelle während einer Programmausführung, z.b. der Aufruf einer Methode oder der Zugriff auf ein Objekt.
    - **Pointcut** - Ist ein Programmkonstrukt, das einen Join-Point auswählt und kontextabhängige Informationen dazu sammelt, z.b. die Argumente eines Methodenaufrufs oder das Zielobjekt.
    - **Advice** - Ist jender Programmcode, der vor (`before()`), um (`arround()`) oder nach (`after()`) dem Join-Point ausgeführt wird.
    - **Aspect** - Ist wie eine Klasse das zentrale Element in AspectJ. Er enthält alle Deklarationen, Methoden, Pointcuts und Advices.

15. **An welchen Programmpunkten können sich Join-Points befinden?**
(*siehe 14*)

---

# 5 Software-Entwurfsmuster

1. **Erklären Sie folgende Entwurfsmuster und beschreiben Sie jeweils Anwendungsgebiet, Struktur, Eigenschaften und wichtige Details der Implementierung unter Verwendung vorgegebener Namen:**
**Erzeugende Entwurfsmuster** - beschäftigen sich mit der Erzeugung neuer Objekte auf eine Art und Weise, die weit über die Möglichkeiten der Verwendung von `new` in Java hinausgeht. Beispiele dafür:

   - **Factory-Method** - wird auch *Virtual-Constructor* genannt. Der Zweck ist die Definition einer Schnittstelle für die Objekterzeugung, wobei Unterklassen entscheiden, von welcher Klasse die erzeugten Objekte sein sollen. Die tatsächliche Erzeugung der Objekte wird in Unterklassen verschoben. Generell anwendbar, wenn
     - eine Klasse Objekte erzeugen soll, deren Klasse aber nicht kennt,
     - eine Klasse möchte, dass ihre Unterklassen die Art der Objekte bestimmen, welche die Klasse erzeugt,
     - Klassen Verantwortlichkeiten an eine von mehreren Unterklassen delegieren, und man das Wissen, an welche Unterklasse delegiert wird, lokal halten möchte,
     - die Allokation und Freigabe von Objekten zentral in einer Klasse verwaltet werden soll.

     Wichtige Eigenschaften:
     - Sie bieten Anknüpfungspunkte (Hooks) für Unterklassen.
     - Sie verknüpfen parallele Klassenhierarchien, die Creator-Hierarchie mit der Product-Hierarchie.
   - **Prototype** - dient dazu, die Art eines neu zu erzeugenden Objekts dur ein Prototyp-Objekt zu spezifizieren. Neu Objekte werden durch Kopieren dieses Prototyps erzeugt. Generell anwendbar, wenn ein System unabhängig davon sein soll, wie seine Produkte erzeugt, zusammengesetzt und dargestellt werden, und wenn
     - die Klassen, von denen Objekte erzeugt werden sollen, erst zur Laufzeit bekannt sind, oder
     - vermieden werden soll, eine Hierarchie von Creator-Klassen zu erzeugen, die einer parallelen Hierarchie von Product-Klassen entspricht (also Factory-Method vermieden werden soll), oder
     - jedes Objekt einer Klasse nur wenige unterschiedliche Zustände haben kann.

     Wichtige Eigenschaften:

  - Sie verstecken die konkreten Produktklassen vor den Anwendern (Client) und reduzieren damit die Anzahl der Klassen, die Anwender kennen müssen.
  - Prototypen können auch zur Laufzeit jederzeit dazugegeben und weggenommen werden.
  - Sie erlauben die Spezifikation neuer Objekte durch änderbare Werte.
  - Sie vermeiden eine übertrieben große Anzahl an Unterklassen.
  - Sie erlauben die dynamische Konfiguration von Programmen.
- **Singleton** - sichert zu, dass eine Klasse nur eine Instanz hat und erlaubt globarlen Zugriff auf dieses Objekt. Anwendbar wenn:
  - es genau ein Objekt einer Klasse geben soll, und dieses global zugreifbar sein soll,
  - die Klasse durch Vererbung erweiterbar sein soll, un Anwender die erweiterte Klasse ohne Änderungen verwenden können sollen.

  Wichtige Eigenschaften:
  - Sie erlauben den kontrollierten Zugridd auf das einzige Objekt.
  - Sie vermeiden durch Versicht auf globale Variablen unnötige Namen und weitere unangenehme Eigenschaften globaler Variablen.
  - Sie unterstützen Vererbung.
  - Sie verhindern, dass irgendwo Instanzen außerhalb der Kontrolle der Klasse erzeugt werden (Konstruktoren sind in der Regel nicht `public`).
  - Sie erlauben auch mehrere Instanzen.
  - Sie sind flexibler als statische Methoden, da statische Methoden kaum Änderungen erlauben und dynamischens Binden nicht unterstützt wird.

**Entwurfsmuster für Struktur und Verhalten** - beeinflussen die Programmstruktur.

- **Iterator** - Sollte soweit bekannt sein. Dieses Entwurfsmuster ist verwendbar, um
  - auf den Inhalt eines *Aggregats* (das ist eine Sammlung von Elementen, z.b. eine Collection) zugreifen zu können, ohne die inner Darstellung offen legen zu müssen,
  - mehrere (gleichzeitig bzw. überlappende) Abarbeitungen der Elemente in einem Aggregat zu ermöglichen,
  - eine einheitliche Schnittstelle für die Abarbeitung verschiedener Agregatstrukturen zu haben, das heißt, um polymorphe Iterationen zu unterstützen.

  Wichtige Eigenschaften:
  - Sie unterstützen unterschiedliche Varianten in der Abarbeitung von Aggregaten.
  - Iteratoren vereinfachen die Schnittstelle von Aggregaten, da Zugriffsmöglichkeiten, die über Iteratoren bereitgestellt werden, durch die Schnittstelle von Aggregaten nicht unnterstützt werden müssen.

  - Auf ein und demselben Aggregat können gleichzeitig mehrere Abarbeitungen stattfinden, da jeder Iterator selbst den aktuellen Abarbeitungszustand verwaltet.

- **Visitor** (siehe 18)
- **Decorator** - auch *Wrapper* genannt, gibt Objekten dynamisch zusätzliche Verantwortlichkeiten (z.b. einem Fenster dynamisch Scroll-Bar zu geben). Anwendbar
  - um dynamische Verantwortlichkeiten zu einzelnen Objekten hinzuzufügen, ohne andere Objekte dadurch zu beeinflussen,
  - für Verantwortlichkeiten, die wieder entzugen werden können,
  - wenn Erweiterungen einer Klasse durch Vererbung unpraktisch sind.

  Wichtige Eigenschaften:
  - Sie bieten mehr Flexibilität als statische Vererbung.
  - Sie vermeiden Klassen, die bereits weit oben in der Klassenhierarchie mit Methoden und Variablen überladen sind.
  - Sie führen zu vielen kleinen Objekten.
- **Proxy** - auch *Surrogate* genannt, stellt einen Platzhalter für ein anderes Objekt dar und kontrolliert Zugriffe darauf. Anwendbar, wenn eine intelligentere Referenz auf ein Objekt als ein simpler Zeiger nötig ist. Übliche Situationen, in denen Proxy eingesetzt werden kann:
  - **Remote-Proxies** - sind Platzhalter für Objekte, die in anderen Namensräumen existieren.
  - **Virtual-Proxies** - erzeugen Objekte bei Bedarf. Da die Erzeugung eines Objekts aufwendig dein kann, wird sie so lange hinausgezögert, bis es wirklich Bedarf dafür gibt.
  - **Protection-Proxies** - kontrollieren Zugriffe auf Objekte. Derartige Proxies sind sinnvoll, wenn Objekte je nach Zugreifer oder Situation unterschiedliche Zugriffsrechte haben sollen.
  - **Smart-References** - ersetzten einfache Zeiger.

  Wichtige Eigenschaften:
  - Verwaltet eine Referenz "realSubject", über die ein Proxy auf Objekte von "RealSubject" zugreifen kann.
  - Stellt eine Schnittstelle bereit, die der von "Subject" entspricht, da mit einem Proxy als Ersatz des eigentlichen Objekts verewendet werden kann.
  - kontrolliert Zugriffe auf das eigentliche Objekt und kann für dessen Erzeugung oder Entfernung verantwortlich sein.
  - hat weitere Verantwortlichkeiten, die von der Art abhängen.

- **Template-Method** - definiert das Grundgerüst eines Algorithmus in einer Operation, überlässt die Implementierung einiger Schritte aber einer Unterklasse. Anwendbar
  - um den unveränderlichen Teil eines Algorithmus nur einmal zu implementiern und es Unterklassen zu überlassen, den veränderbaren Teil des Verhaltens festzulegen;
  - wenn gemeinsames Verhalten mehrerer Unterklassen in einer einzigen Klasse lokal zusammengefasst werden soll, um Duplikate im Code zu vermeiden;
  - um mögliche Erweiterungen in Unterklassen zu kontrollieren, beispielsweise durch Template-Methods, die Hooks aufrufen und nur das Überschreiben dieser Hooks in Unterklassen ermöglichen.

  Wichtige Eigenschaften:
  - Stellen eine fundamentale Technik zur direkten Wiederverwendung von Programmcode dar.
  - Sie führen zu einer umgekehrten Kontrollstruktur, die manchmal als *Hollywood-Prinzip* bezeichnet wird ("Don't call us, we'll call you"). Oberklasse ruft Methoden der Unterklasse auf.

2. **Welche Arten von Iteratoren gibt es, und wofür sind sie geeignet?**

   - **interne Iteratoren** - kontrollieren selbst, wann die nächste Iterationen erfolgt (enthalten Schleife selbst). Verwendet für Stream-Operationen, in Java aber auch ein Aufruf der Methoden `foreach` in einer `Map`.
   - **externe Iteratoren** - sind flexibler als interne Iteratoren. Gut geeignet für das Vergleichen zweier Aggregate.

3. **Wie wirkt sich die Verwendung eines Iterators auf die Schnittstelle des entsprechenden Aggregats aus?**
   Iteratoren vereinfachen diese. (siehe Wichtige Eigenschaften Iterator 1)

4. **Inwiefern können geschachtelte Klassen bei der Implementierung von Iteratoren hilfreich sein?**
   Dadurch kann der Iterator auf private Details (Variablen, Methoden, ...) des Aggregats zugreifen.

5. **Was ist ein robuster Iterator? Wozu braucht man Robustheit?**
   Wenn ein Aggregat verändert wird, während der Iterator drüber rennt kann es leicht passieren das Elemente doppelt oder gar nicht abgearbeitet werden. Dieses Problem könnte man verhindern, indem man auf einer Kopie des Aggregats iteriert. Ein *robuster Iterator* erreicht dieses Ziel, ohne das ganze Aggregat zu kopieren.

6. **Wird die Anzahl der benötigten Klassen im System bei Verwendung von Factory-Method, Prototype, Decorator und Proxy (gegenüber einem System, das keine Entwurfsmuster verwendet) eher erhöht, vermindert oder bleibt sie unverändert?**

   - **Factory-Method** - steigt
   - **Prototype** - sinkt
   - **Decorator** - sinkt (not sure tho)
   - **Proxy** - steigt

7. **Wird die Anzahl der benötigten Objekte im System bei Verwendung von Factory-Method, Prototype, Decorator und Proxy (gegenüber einem System, das keine Entwurfsmuster verwendet) eher erhöht, vermindert oder bleibt sie unverändert?**

   - **Factory-Method** - unverändert
   - **Prototype** - unverändert/steigt(wenn Objekterzeugung nötig)
   - **Decorator** - steigt
   - **Proxy** - unverändert/steigt(wenn Objekterzeugung nötig)

8. **Vergleichen Sie Factory-Method mit Prototype. Wann stellt welches Entwurfsmuster die bessere Lösung dar? Warum?**
   (siehe Anwendung Prototype Punkt 2 1)

9. **Wo liegen die Probleme in der Implementierung eines so einfachen Entwurfsmusters wie Singleton?**
   Wenn Singletons Unterklassen haben wird es oft schwierig ein Programm zu schreiben, welches trotzdem Garantiert, dass es nur eine Instanz von Singleton gibt.

10. **Welche Unterschiede und Ähnlichkeiten gibt es zwischen Decorator und Proxy?**
    Die beiden Muster können ähnlich aufgebaut sein, jedoch haben sie ganz unterschiedliche Eigenschaften (siehe 1)

11. **Welche Probleme kann es beim Erzeugen von Kopien im Prototype geben? Was unterscheidet flache Kopien von tiefen?**

    - **flache Kopien** - Werte jeder Variable in der Kopie ist identisch (Default Implementierung von `clone`)
    - **tiefe Kopien** - Wenn die Werte der Variablen nicht identisch sondern nur gleich sein sollen muss eine tiefe Kopie erstellt werden. Auf jede Variable müsste `clone` aufgerufen werden. (man muss die Default Implementierung von `clone` überschreiben.)

12. **Für welche Arten von Problemen ist Decorator gut geeignet, für welche weniger? (Oberfläche versus Inhalt)**
    Gut geeignet um das Erscheinungsbild von Objekten zu verändern, aber schlecht für inhaltliche Veränderungen geeignet.

13. **Kann man mehrere Decorators bzw. Proxies hintereinander verketten? Wozu kann so etwas gut sein?**
    Ja. Bei Decorators können damit z.b. umfangreiche GUIs gebilted werden. Bei Proxies ergibt sich dadurch eine einfache Hierarchie

14. **Was unterscheidet Hooks von abstrakten Methoden?**
    Hooks haben ein Default-Verhalten, wobei abstrakte Methoden keinen Code enthalten. Beide bieten Anschlussstelle für Unterklassen.

---

if u read this, just know that i luv u ♡