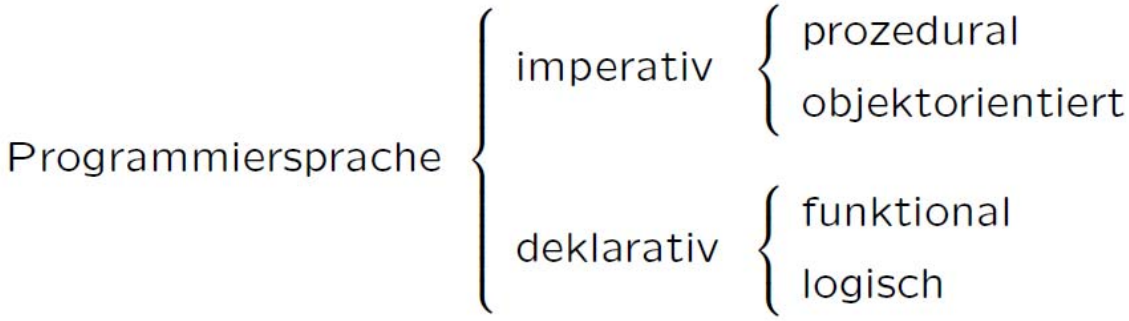
Kapitel 1

1. Was versteht man unter einem Programmierparadigma?
Unter Programmierparadigma versteht man eine bestimmte Denkweise oder Art der Weltanschauung. Entsprechend entwickelt man Programme in einem gewissen Stil.
Bei den Paradigmen unterscheidet man zwischen:



2. Wozu dient ein Berechnungsmodell?
Hinter jedem Paradigma steckt ein Berechnungsmodell, die immer einen formalen Hintergrund haben. Sie müssen konsistent und in der Regel Turing-vollständig sein.

3. Welche Berechnungsmodelle werden in Programmierparadigmen verwendet, und welche charakteristischen Eigenschaften haben sie?
Funktionen:
Prädikatenlogik:
Constraint-Programmierung:
Temporale Logik und Petri-Netze:
Freie Algebren:
Prozesskalküle:
Automatentheorie:
While, GoTo & Co:

4. Welche Eigenschaften von Berechnungsmodellen sind für deren Erfolg häufig (mit)bestimmend?
Kombinierbarkeit, Konsistenz, Abstraktion, Systemnähe, Unterstützung, Beharrungsvermögen

5. Im Spannungsfeld welcher widersprüchlichen Ziele befinden sich Programmierparadigmen? Wie äußert sich dieses Spannungsfeld?

Es ist es unmöglich, folgende Forderungen gleichzeitig und in vollem Umfang zu erfüllen:

- Flexibilität und Ausdruckskraft sollen in kurzen Texten die Darstellung aller vorstellbaren Programmabläufe ermöglichen.
- Lesbarkeit und Sicherheit sollen Absichten hinter Programmteilen sowie mögliche Inkonsistenzen leicht erkennen lassen.
- Die Konzepte müssen verständlich bleiben und es muss klar sein, was einfach machbar ist und was nicht

6. Was ist die strukturierte Programmierung? Wozu dient sie?

Die strukturierte Programmierung bringt mehr Struktur in die prozedurale Programmierung.
Jedes Programm bzw. jeder Rumpf einer Prozedur ist nur aus drei einfachen Kontrollstrukturen aufgebaut:

- Sequenz (ein Schritt nach dem anderen)
- Auswahl (Verzweigung im Programm)
- Wiederholung (Schleife, Rekursion oder Ähnliches)

7. Wie gehen unterschiedliche Paradigmen mit Seiteneffekten um?

Deklarative Paradigmen: radikaler Ansatz
strebt referentielle Transparent als Eigenschaft an Bsp: f(x) + f(x) = 2 f(x)

Objektorientierte Paradigmen: gemäßigter Ansatz
man nimmt an, dass es Querverbindungen gibt und beschränkt sie lokal auf einzelne Objekte

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 Programms dadurch zu ändern. (Bsp: 3+4 lässt sich durch 7 oder 14/2 ersetzten)

9. Wieso passt referentielle Transparenz nicht gut mit Ein- und Ausgabe zusammen, und wie kann man das Dilemma lösen?

Ein- und Ausgaben sind Seiteneffekte, die referenzielle Transparenz unmöglich macht.
Gelöst hat man es, indem man Ein- und Ausgaben nur gut sichtbar und ganz oben in der Aufrufhirachie geschoben und sie dadurch aus allen Funktionen verbannt hat.

10. Welchen Zusammenhang gibt es zwischen Seiteneffekten und der objektorientierten Programmierung?

In der OOP geht man mit Seiteneffekten ganz offensiv um. Man nimmt immer an, dass es Querverbindungen gibt. Wenn man sie lokal auf einzelne Objekte beschränkt, kann man sie jedoch überschaubar halten.

11. Was sind First-Class-Entities? Welche Gründe sprechen für deren Verwendung, welche dagegen?

First Class Entities sind Funktionen, die wie normale Daten verwendbar sind, in Variablen abgelegt werden können, als Argumente an andere Funktionen übergeben und als Ergebnisse von Funktionen zurückbekommen werden können.

- \- Häufig sehr kompliziert als vergleichbare Konzepte, weil die uneingeschränkte Verwendbarkeit eine große Zahl an zu berücksichtigenden Sonderfällen nach sich zieht.
- \+ Die uneingeschränkte Verwenbarkeit bringt Vorteile beispielsweise beim Einsatz mit Funktionen höherer Ordnung (Funktionen mit Funktionen als Parameter).Sie bringen eine neue Dimension in der Programmierung.

12. Was haben Funktionen höherer Ordnung mit einem applikativen Programmierstil zu tun?

Ein häufiger Gebrauch von Funktionen höherer Ordnung führt zu einem eigenen Paradigma, der *applikativen Programmierung.* Man schreibt quasi Schablonen von Programmteilen, die dann durch Übergabe von Funktionen (zum Füllen der Löcher in den Schablonen) ausführbar werden.

13. Welche Modularisierungseinheiten kann man unterscheiden, und was sind ihre charakteristischen Eigenschaften?

- Modul (Übersetzungseinheit, zyklenfrei, enthält Deklarationen bez. Definitionen von Variablen, Typen, Prozeduren, Funktionen, usw.)

- Objekt (zur Laufzeit erzeugt, keine Einschränkungen hinsichtlich zyklische Abhängigkeiten, kapseln Variablen und Methoden zu logischen Einheiten und schütze private Inhalte von Zugriffen // *Kapselung* und *Data-Hiding*)

- Klasse (Modul und Schablone für Objekterzeugung, gibt Variablen in Objekten vor und spezifiziert wichtigste Eigenschaften)
- Komponente (komplexere Initialisierung, eigenständiges Stück Software, dass in ein Programm eingebunden wird, ähneln Modulen)
- Namensraum (jede oben genannte Modularisierungseinheit bildet einen eigenen Namensraum, verhindert Namenskonflikte)

14. Welche Bedeutung haben Schnittstellen für Modularisierungseinheiten? Warum unterscheidet man zwischen von außen zugreifbaren und privaten Inhalten?

Klar definierte Schnittstellen zwischen Modulen sind sehr hilfreich: Einerseits braucht der Compiler Schnittstelleninformation um Inhalte anderer Module verwenden zu können, andererseits ist diese Information auch beim Programmieren wichtig um Abhängigkeiten zwischen Modulen besser zu verstehen. Meist wird nur ein kleiner Teil des Modulinhalts von anderen Modulen verwendet. Schnittstellen unterscheiden klar zwischen Modulinhalten, die von anderen Modulen zugreifbar sind, und solchen, die nur innerhalb des Moduls gebraucht werden.

Erstere werden exportiert, letztere sind privat. Private Modulinhalte sind von Vorteil: Sie können vom Compiler im Rahmen der Programmiersprachsemantik beliebig optimiert, umgeformt oder sogar weggelassen werden, während für exportierte Inhalte eine gewissen Regeln entsprechende Zugriffsmöglichkeit von außen bestehen muss. Änderungen privater Modulinhalte wirken sich nicht auf andere Module aus. Änderungen exportierter Inhalte machen hingegen oft entsprechende Änderungen in anderen Modulen nötig, die diese Inhalte verwenden.

15. Was ist ein Namensraum?

Die Namen für Objekte in einer Art Baumstruktur angeordnet und über entsprechende Pfadnamen eindeutig angesprochen. (Jede oben genannte Modularisierungseinheit bildet einen eigenen Namensraum, verhindert Namenskonflikte)

16. Warum können Module nicht zyklisch voneinander abhängen?

Weil das aufgrund der getrennten Übersetzung nicht möglich ist. Wenn ein Modul B Inhalte eines Moduls A importiert, kann A keine Inhalte von B importieren. Eine gemeinsame Übersetzung würde der Definition von Modulen wiedersprechen.

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

Datenabstraktion:	Kapselung in Kombination mit Data-Hiding
Kapselung:	Daten werden zB. in einem Objekt gekapselt (Vektor x und y Koordinate)
Data-Hiding:	private Variablen werden in einem Objekt versteckt (Var ohne getter Methode)

18. Wodurch unterscheiden sich Komponenten von Modulen?

Während ein Modul Inhalte ganz bestimmter, namentlich genannter anderer Module importiert, importiert eine Komponente Inhalte von zur Übersetzungszeit nicht genau bekannten anderen Komponenten. Erst beim Einbinden in ein Programm werden diese anderen Komponenten bekannt. Sowohl bei Modulen als auch Komponenten ist offen, wo exportierte Inhalte verwendet werden, aber bei Komponenten ist zusätzlich offen, von wo importierte Inhalte kommen. Letzteres verringert die Abhängigkeit der Komponenten voneinander. Deswegen gibt es bei der getrennten Übersetzung kein Problem mit zyklischen Abhängigkeiten.

19. Warum und inwiefern ist die Einbindung von Komponenten komplizierter als die von Modulen?

Die Einbindung von Komponenten in Programme ist aufwendiger als die von Modulen, da dabei auch die Komponenten festgelegt werden müssen, von denen etwas importiert wird. Oft werden zuerst die einzubindenden Komponenten zum Programm hinzugefügt und erst in einem zweiten Schritt festgelegt, von wo importiert wird.

20. Wie kann man globale Namen verwalten und damit Namenskonflikte verhindern?

Indem man vermehrt global eindeutige Namen zur Adressierung von Modularisierungseinheiten macht.

21. Was versteht man unter Parametrisierung? Wann kann das Befüllen von Löchern durch welche Techniken erfolgen?

Es kann zur Laufzeit erfolgen:

Konstruktor:	Beim Erzeugen eines Objekts werden Objektvariablen initialisiert
Initialisierungsmethode:	Objekte die durch Kopieren erzeugt werden.
Zentrale Ablagen:	Daten werden bei zentralen Ablagen abgelegt und bei Initialisierung abgeholt.

22. Warum braucht man zur Parametrisierung in der Objekterzeugung neben Konstruktoren gelegentlich auch Initialisierungsmethoden?

Weil auch Objekte durch Kopieren erzeugt werden können und ein Aufrufen des Konstruktors dadurch nicht möglich ist.

23. Welche Vor- und Nachteile hat die zentrale Ablage von Werten zum Zweck der Parametrisierung?

Die Abholung bei Verwendung ist auch für statische Modularisierungseinheiten verwendbar, die breits zur Übersetzungszeit feststehen.

24. Was unterscheidet Generizität von den verschiedenen Formen der Parametrisierung zur Laufzeit?
Löcher werden bereits zur Übersetzungszeit gefüllt.

25. Was sind Annotationen und wozu kann man sie verwenden? Wodurch unterscheiden sie sich von Generizität?

Annotationen sind optionale Parameter, die man zu Sprachkonstrukten hinzufügen kann. Annotationen werden von verschiedenen Werkzeugen verwendet, oder aber einfach ignoriert.
Die Löcher, die durch Annotationen befüllt werden, sind im Gegensatz zur Generizität nirgends im Programm festgelegt. Daher ist die Art und Weise, wie die mitgegebenen Informationen zu verwenden sind, ebenso unterschiedlich wie die Anwendungsgebiete.

26. Was versteht man unter aspektorientierter Programmierung?

In der aspektorientierten Programmierung kommt man in der Regel ohne Spezifikation von Löchern im Programm aus. Stattdessen fügt man zu einem bestehenden Programm von außen neue Aspekte hinzu.

27. Wodurch unterscheiden sich die verschiedenen Formen der Parametrisierung von der Ersetzbarkeit, und warum ist die Ersetzbarkeit in der objektorientierten Programmierung von so zentraler Bedeutung?

Notwendige Änderungen ist bei Modularisierungseinheiten nur schwer möglich, da dadurch auf Änderungen an anderen Stellen notwendig werden. Wenn die Löcher sich ändern, muss man auch das ändern, das zum Befüllen der Löcher benötigt wird.

28. Wann ist *A* durch *B* ersetzbar?

…, wenn ein Austausch von A durch B keinerlei Änderungen an Stellen nach sich zieht, an denen A (bez. Nach der Ersetzung B) verwendet wird.

29. Wodurch kann festgelegt sein, ob *A* durch *B* ersetzbar ist?

Die Ersetzbarkeit geht Hand in Hand mit Schnittstellen. Schnittstellen können also festgelegt sein.

30. Was ist die Signatur einer Modularisierungseinheit?

Die Signatur ist eine spezifizierte Schnittstelle, welche Inhalte der Modularisierungseinheit von außen zugreifbar macht. Diese Inhalte werden nur über ihre Namen und gegebenenfalls den Typen von Parameter und Ergebnisse beschrieben. (B muss alles enthalten und von außen zugreifbar machen, was A auch hat)

31. Wie verhält sich die Signatur einer Modularisierungseinheit zur Abstraktion, die durch diese Modularisierungseinheit gebildet wird?

Eine Abstraktion ist ein informeller Text, die zusätzlich zur Signatur beschreibt.

32. Was sind Zusicherungen, und welche Rolle spielen sie für Modularisierungseinheiten?

Sie beschreiben die erlaubten Erwartungen an eine Modularisierungseinheit. Diese Beschreibung bezieht sich auf alle nach außen sichtbaren Inhalten.

33. Wann sind Typen miteinander konsistent, und was sind Typfehler?

Wenn die Typen der Operanden mit der Operation zusammenpassen, sind die Typen konsistent. Andernfalls tritt ein Typfehler auf.

34. Wie schränken Typen die Flexibilität ein, und warum verwendet man Typen trotzdem?

Typen helfen bei der Klassifizierung dieser Werte. Viele Operationen sind nur für Instanzen bestimmter Typen definiert. Beispielsweise kann man nur ganze Zahlen oder Fließkommazahlen miteinander multiplizieren, aber keine Zeichenketten. Wenn die Typen der Operanden mit der Operation zusammenpassen, sind die Typen konsistent.

35. Was ist der Hauptgrund für den Einsatz statischer Typprüfungen?

Der Hauptgrund für statische Typprüfungen scheint die verbesserte Zuverlässigkeit der Programme zu sein. Das stimmt zum Teil, aber nicht auf direkte Weise. Typkonsistenz bedeutet ja nicht Fehlerfreiheit, sondern nur die Abwesenheit ganz bestimmter, eher leicht auffindbarer Fehler.

36. Was versteht man unter Typinferenz?

Viele Typen kann ein Compiler aus der Programmstruktur herleiten; man spricht von Typinferenz. Beispielsweise braucht man in Haskell keinen einzigen Typ hinzuschreiben, obwohl der Compiler alle Typen statisch prüft (auf Basis von Typinferenz). Zur Verbesserung der Lesbarkeit kann und soll man Typen dennoch explizit anschreiben

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

Zum Zeitpunkt der Erstellung von Module werden die meisten wichtigen Entscheidungen getroffen. Hierzu braucht man viel Flexibilität.

38. Welchen Einfluss können Typen auf Entscheidungszeitpunkte haben?

Unsichere Entscheidungen werden eher nach hinten verschoben und zu einem Zeitpunkt getroffen, zu dem bereits viele andere damit zusammenhängende Entscheidungen getroffen wurden und der Entscheidungsspielraum entsprechend kleiner ist.

39. Wie beeinflussen Typen die Planbarkeit weiterer Schritte?

Wenn man weiß, dass eine Variable vom Typ int ist, braucht man kaum mehr Überlegungen darüber anstellen, welche Werte in der Variablen enthalten sein könnten. Statt auf Spekulationen baut man auf Wissen auf. Um sich auf einen Typ festlegen zu können, muss man voraussehen (also planen), wie bestimmte Programmteile im fertigen Programm verwendet werden. Man wird zuerst jene Typen festlegen, bei denen man kaum Zweifel an der künftigen Verwendung hat.

Was ist ein abstrakter Datentyp?

Die Trennung zwischen Innenansicht und Außenansicht einer Modularisierungseinheit (Data-Hiding)
Kapselung in Kombination mit Data-Hiding

41. Was unterscheidet strukturelle von nominalen Typen?

Struktureller Typ: Typ der Modularisierungseinheit hängt nur von Namen, Parametertypen und Ereignistypen die nach außen sichtbar sind ab.
Nomialer Typ: Neben Signatur auch einen eindeutigen Namen. Der Typ eines Objekts entspricht dem Namen der Klasse.

42. Warum verwenden Programmiersprachen meist nominale Typen?
…, weil man beim Programmieren hauptsächlich abstrakt in Konzepten und nur selten in Signaturen denkt.

43. Wie hängen Untertypbeziehungen mit Ersetzbarkeit zusammen?
Untertypen werden durch das Ersetzbarkeitsprinzip definiert. Ohne Ersetzbarkeit gibt es keine Untertypen.

Faustregel: Ein Typ U ist Untertyp von Typ T wenn jede Instanz von U überall dort verwendbar ist, wo T verwendet werden kann.

44. Warum kann ein Compiler ohne Unterstützung durch Programmierer nicht entscheiden, ob ein nominaler Typ Untertyp eines anderen nominalen Typs ist?
Abstrakte Regeln und nicht zugängliche Konzepte, lassen sich nicht automatisch vergleichen.
(Bsp: Programmierbeispiel – ChangeBox ist kein Untertyp von Box, weil bei Box per definition keine Pixel mehr verändert werden dürfen. Obwohl kaum Unterschied, kann die ChanceBox NIE Untertyp sein)

45. Welche wichtige Einschränkung gibt es bei Untertypbeziehungen zusammen mit statischer Typprüfung?
Typen von Funktions- bez. Methodenparametern dürfen in Untertypen nicht stärker werden.
Bsp: boolean compare(T x) kann im Untertyp nicht zu boolean compare(U x) überschrieben sein.

46. In welchem Zusammenhang verwendet man Higher-Order-Subtyping und F-gebundene Generizität?
F-gebundene Generizität nutzt Untertypbeziehungen zur Beschreibung von Einschränkungen und wird z.B. in Java und C# eingesetzt. Higher-Order-Subtyping, auch Matching genannt, geht einen eher direkten Weg und beschreibt Einschränkungen über Untertyp-ähnliche Beziehungen, die wegen Unterschieden in Details aber keine Untertypbeziehungen sind. Dieser Ansatz wird auf unterschiedliche Weise beispielsweise in C++, aber auch in der funktionalen Sprache Haskell verwendet.

47. Wie konstruiert man rekursive Datenstrukturen?

Mittels induktiven Konstruktionen (Verketteten Listen udgl.)
data Lst = end | elem(Int,Lst)

48. Was versteht man unter Fundiertheit rekursiver Datenstrukturen?
Man muss klar zwischen M_0 (nicht-rekursiv) und der Konstruktion aller Mi mit i > 0 (rekursiv) unterscheiden, wobei M_0 nicht leer sein darf. Diese Eigenschaft nennt man *Fundiertheit*.

49. Warum wird Typinferenz in objektorientierten Sprachen meist nur lokal beschränkt eingesetzt?

Typinferenz funktioniert nicht, wenn gleichzeitig Ersetzbarkeit durch Untertypen verwendet wird. (Typinferenz und Generezität schon)

50. Wie können statisch geprüfte Typen beliebige Eigenschaften von Werten propagieren?

Eine Funktion kann nur aufgerufen werden, wenn der Typ des Arguments mit dem des formalen Parameters übereinstimmt. Dabei wird Information über das Argument an die aufgerufene Funktion propagiert. Entsprechendes gilt auch für das Propagieren von Information von der aufgerufenen Funktion zur Stelle des Aufrufs unter Verwendung des Ergebnistyps und bei der Zuweisung eines Wertes an eine Variable. Genau diese Art des Propagierens von Information funktioniert nicht nur für Typen im herkömmlichen Sinn, sondern für alle statisch bekannten Eigenschaften.

51. Erklären Sie folgende Begriffe:
- **Objekt, Klasse, Vererbung**
- **Identität, Zustand, Verhalten, Schnittstelle**
- **deklarierter, statischer und dynamischer Typ**
- **Faktorisierung, Refaktorisierung**
- **Verantwortlichkeiten, Klassenzusammenhalt, Objektkopplung**

Objekt: Ein Objekt wird als Kapsel beschrieben, in der sich Variablen und Routinen befinden

Klasse: Eine Klasse gibt die Struktur eines oder mehreren Objekten vor. Diese können mittels eines Konstruktors aus einer Klasse erzeugt werden. Die Klasse ist eine Art Schablone

Vererbung: Ist das Ableiten einer Unterklasse von einer Oberklasse. Reine Vererbung erspart meist nur Schreibaufwand, da Änderungen nur an einer Stelle vorgenommen werden müssen.

Identität: Jedes Objekt ist über eindeutige und unveränderliche Identität identifizier- und ansprechbar

Zustand: Setzt sich aus den momentanen Variablenbelegungen zusammen. Ist änderbar

Verhalten: Reaktion eines Objekts beim Erhalten einer Nachricht. Ist abhängig von
1. Der aufgerufenen Methode(Routine)
2. Den übergebenen Parametern
3. Den momentan aktuellen Zustand des Objekts (Variablenwerte)

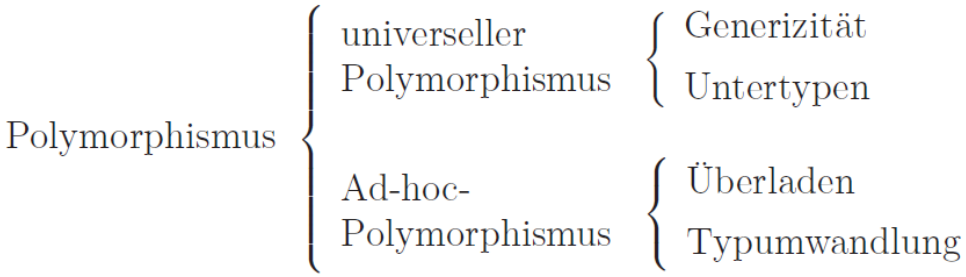
Schnittstellen: Einfach gesagt nur die Köpfe der Methoden, welche Aufschluss über Eingangs und Ausgangsparameter geben.

Instanz einer Klasse: Objekt

Instanz einer Schnittstelle: Es gibt keine direkten Instanzen von Schnittstellen. Alle Objekte welche die Schnittstelle implementieren können als Instanzen dieser Schnittstelle gesehen werden.

Instanz eines Typs: Typ = Ist eine bestimmte Sicht auf ein Objekt. Jedes Objekt kann mehrere Sichten beinhalten. Eine Sicht kann aus Schnittstellen und abstrakten Klassen bestehen. Jedes Objekt das diesen Typ implementiert ist eine Instanz des Typs

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



In der objektorientierten Programmierung sind Untertypen von überragender Bedeutung, die anderen Arten des Polymorphismus existieren eher nebenbei. Daher nennt man alles, was mit Untertypen zu tun hat, oft auch objektorientierten Polymorphismus oder nur kurz Polymorphismus.

53. Wann sind zwei gleiche Objekte identisch und wann sind zwei identische Objekte gleich?

Zustandsgleichheit: Wenn die Parameter gleich sind (wird mit equals verglichen)
Ident: Wenn es sich um das identische Objekt, also das selbe Abbild im Speicher, handelt (wird mit == verglichen)

54. Sind Datenabstraktion, Datenkapselung und Data-Hiding einander entsprechende Begriffe? Wenn Nein, worin unterscheiden sie sich?

Kapselung: Zusammenfügen von Daten und Routinen zu einer Einheit, sodass Routinen ohne die Daten nicht ausführbar sind. Bedeutung der Daten oft nur den Routinen bekannt.
Data hiding: Objekt wird als Grey- oder Blackbox gesehen. Man kennt nur die Schnittstellen, hat aber wenig Vorstellung darüber, was im inneren des Objektes abläuft.
Datenabstraktion: Kapselung zusammen mit Data hiding. Bsp: Datentypen (In welcher Datenstruktur die Daten intern gespeichert sind, ist ohne Bedeutung)

55. Was besagt das Ersetzbarkeitsprinzip? (Häufige Prüfungsfrage!)

Ein Typ U ist Untertyp eines Typs T, wenn jede Instanz von U überall verwendbar ist, wo eine Instanz von T erwartet wird.

56. Warum ist Ersetzbarkeit in der objektorientierten Programmierung so wichtig?

Aufgrund der hohen Code-Wiederverwendung

57. Warum ist gute Wartbarkeit so wichtig?
Da Wartungskosten ca. 70 % der Gesamtkosten ausmachen. Gute Wartbarkeit erspart Unmengen an Geld!

58. Wie lauten die wichtigsten Faustregeln im Zusammenhang mit Klassenzusammenhalt und Objektkopplung? Welche Vorteile kann man sich erwarten, wenn diese Faustregeln erfüllt sind?

Der Klassenzusammenhang soll hoch sein & Die Objektkopplung soll schwach sein

gute Faktorisierung, Wahrscheinlichkeit geringer, dass bei Programmänderung auch die Zerlegung in Klassen und Objekte geändert werden muss, höhere Datenabstraktion und Data-Hiding

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

- Programme
- Erfahrung (Entwurfsmuster)
- Code
- Daten
- Globale Bibliotheken
- Fachspezifische Bibliotheken
- Projektinterne Wiederverwendung:
- Programminterne Wiederverwendung

Faustregel: Code-Wiederverwendung erfordert beträchtliche Investitionen in die Wiederverwendbarkeit. Man soll diese tätigen, wenn ein tatsächlicher Bedarf absehbar ist.

60. Welche Rolle spielen Refaktorisierungen in der Wiederverwendung?

Refaktorisierungen ermöglichen das Hinführen des Projektes auf ein stabiles gut faktorisiertes Design. Gute Faktorisierung => starken Klassenzusammenhalt => gut abgeschlossene und somit leicht wiederverwendbare Klassen.

Sie ändert die Struktur eines Programms, lässt aber dessen Funktionalität unverändert.
Es wird dabei also nichts hinzugefügt oder weggelassen, und es werden auch keine inhaltlichen Änderungen vorgenommen. „Das Rad wird nicht mehr neu erfunden."

Faustregel: Ein vernünftiges Maß rechtzeitiger Refaktorisierungen führt häufig zu gut faktorisierten Programmen.

61. Wofür ist die objektorientierte Programmierung gut geeignet, und wofür ist sie nicht gut geeignet?

Faustregel: Objektorientierte Programmierung eignet sich zur Entwicklung von Systemen, deren Gesamtkomplexität jene der einzelnen Algorithmen deutlich übersteigt. Sonst sind andere Paradigmen besser geeignet.

Kapitel 2

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

1. Durch das Verwenden von Untertypbeziehungen: Untertypen können oft einen Großteil des Codes ihres Obertypen wieder verwenden.
2. Direkte Code-Wiederverwendung: (einfache Vererbung)

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!)

- In Java muss zusätzlich über „extends“ oder “implements“ abgeleitet werden

- In Java sind alle Typen Invariant außer Rückgabewerte von Methoden können kovariant sein.
Sonst wird die Methode überladen, statt überschrieben

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?

- Reflexivität Jeder Typ ist Untertyp von sich selbst
- Transitivität Wenn B gleich Untertyp von A ist und C gleich UT von B ist => C ist UT von A
- Antisymmetrie Wenn A UT von B ist und B UT von A ist => A und B sind gleich

4. Was bedeutet Ko-, Kontra- und Invarianz, und für welche Typen in einer Klasse trifft welcher dieser Begriffe zu?

Kovariant: Für jede Konstante in T (Obertyp) gibt es eine Konstante in U (Untertyp), welche ein Untertyp der entsprechenden Konstante in T sein muss.

Invariant: Für jede Variable in T (Obertyp) gibt es eine Variable in U (Untertyp), welche vom selben deklarierten Typ sein muss wie die in T.

Für jede Methode in T muss es eine Methode in U geben, welche über gleich viele Rückgabeparameter und Eingabeparameter verfügt.

Kontravariant: Eingabeparameter: Dürfen nur allgemeiner werden (muss Obertyp des Eingabeparameters derselben Methode in T sein).

Kovariant: Ergebnistypen: Dürfen nur spezifischer werden (muss Untertyp des Ergebnistypen derselben Methode in T sein).

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. Die Eigenschaft binär bezieht sich darauf, dass der Name der Klasse in der Methode mindestens zweimal vorkommt – einmal als Typ von this und mindestens einmal als Typ eines expliziten Parameters. *this.equals(Klassennamen that)*

6. Wie soll man Typen formaler Parameter wählen um gute Wartbarkeit zu erzielen?

Man soll Parametertypen vorrauschauend und möglichst allgemein wählen.

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

Weil beim Einfügen von neuen Bedingungen (neuen Anredearten z.B.) alle switch Anweisungen aktualisiert werden müssen. Es ist kaum möglich bei großem Programmen die Übersicht zu behalten und die Programmteile konsistent zu halten.

8. Welche Rolle spielt dynamisches Binden für die Ersetzbarkeit und Wartbarkeit?

Ohne die Verwendung von dynamischen Binden, müssten an vielen Stellen schwer zu wartende switch und if- Anweisungen eingesetzt werden. Soll am Programm eine Änderung vorgenommen werden, genügt es bei dynamischen Binden oft nur eine Klasse hinzuzufügen. Ohne dynamischen Binden, müssen jedoch oft an vielen Stellen im Programm Änderungen vorgenommen werden.
Ohne Dynamisches Binden hingegen müsste man die switch-Anweisung entsprechend erweitern!

Gute Erklärung siehe Skriptum WS12/13 Seite 73

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

Vorbedingungen (Pre-Conditions):	Sind vom Cilent (Methodenaufrufer) zu überprüfen und müssen vor Ausführung der Methode gegeben sein.
Nachbedingungen (Post-Conditions):	Geben den Zustand an, welcher nach Aufruf der Methode gegeben sein muss. Werden vom Server sichergestellt
Invarianten (Invariants):	Allgemeine Bedingungen, an die sich sowohl Server als auch Client halten müssen

10. Wie müssen sich Zusicherungen in Unter- und Obertypen zueinander verhalten, damit das Ersetzbarkeitsprinzip erfüllt ist? Warum? (Hinweis: Häufige Prüfungsfrage!)

Vorbedingungen (PreConditions):	Dürfen nur schwächer werden Wenn der Aufrufer nur den statischen Typ T kennt, auch nur die Vorbedingungen der in T enthaltenen Methode X kennt. Würde X aber in U aufgerufen werden, so könnte er nicht sicherstellen, dass auch die VB von der in U eventuell überschriebenen Methode X‘ erfüllt sind.
Nachbedinungen (PostConditions):	Dürfen nur stärker werden. Analog kennt der Aufrufer hier nur die Nachbedingungen von X in T, auf deren Einhaltung er vertraut. Dies wäre aber nicht gegeben wenn X als X‘ in U aufgerufen und die NB von X‘ weniger streng wären als die von X.
Invarianten (Invariants):	Dürfen nur stärker werden. Wenn in T eine Bedingung gefordert wäre und in U nicht, und der Aufrufer würde unwissentlich mit U operieren, könnte dies unerwünschte Folgen haben!

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

Stabilität der Schnittstellen/Typen ist in den obersten Klassen (ausgehend von einer Baumstruktur mit Wurzel oben) am wichtigsten, da alle anderen Klassen darauf aufbauen/diese verwenden.
Ändern sich die Schnittstellen häufig (instabil), so müssen in den darauf aufbauenden Klassen auch jedes Mal Änderungen vorgenommen werden, um sich den neuen Schnittstellen anzupassen!

12. Was ist im Zusammenhang mit allgemein zugänglichen (= möglicherweise nicht nur innerhalb des Objekts geschriebenen) Variablen und Invarianten zu beachten?

Client: Nützt einen Dienst, meist Klasse, die eine Anforderung (Methodenaufruf) an eine andere „Server" klasse schickt

Server: Stellt Dienste zur Verfügung, meist Klasse, die Methoden nach außen hin anbietet
Hat ein Client direkten Schreibzugriff auf eine Variable im Server, so kann dessen Eingreifen nur schwer, bzw. gar nicht vom Server überprüft werden. Die Einhaltung von eventuellen Invarianten, welche diese Variable betreffen, kann somit nicht mehr gewährleistet werden

13. Wie genau sollen Zusicherungen spezifiziert sein?

Unmissverständlich. Sollten aber nicht zu viele Details enthalten.
Eine zu detaillierte Zusicherung weist auf eine starke Objektkopplung hin.

14. Wozu dienen abstrakte Klassen und abstrakte Methoden? Wo und wie soll man abstrakte Klassen einsetzen?

Abstrakte Methoden: Methoden, die nur den Methodenkopf und keine Implementierung enthalten.
Sie eignen sich gut dazu um in einem Obertyp zu bestimmen, dass in einem Untertyp eine gewisse Methode implementiert werden muss. Dienen zur Schnittstellendefinition im Obertypen!

Abstrakte Klassen: Sind Klassen, von denen keine Instanzen erstellt werden können. Sie eignen sich um einen einheitlichen Obertyp (z.B.: Tier) von mehreren Untertypen (z.B.: Esel, Schwein,…) zu erstellen. Abstrakte Klassen können sowohl abstrakte Methoden, als auch vollständig implementierte Routinen enthalten!
(Point 2D und Point 3D für equals-Methode siehe Skriptum S68 & S138)

15. Ist Vererbung dasselbe wie das Ersetzbarkeitsprinzip? Wenn Nein, wo liegen die Unterschiede?

Der Compiler kennt keinen Unterschied zwischen Vererbung und dem Ersetzbarkeitsprinzip!
Ohne Kommentare wäre auch der Code nicht unterscheidbar.

Vererbung zielt rein auf Code-Wiederverwendung ab.
Beim Ersetzbarkeitsprinzip, welches zwar auf Vererbung aufbaut, müssen hingegen die Zusicherungen zwischen Unter- und Oberklasse ebenfalls erfüllt sein.

16. Worauf kommt es zur Erzielung von Codewiederverwendung eher an – auf Vererbung oder Ersetzbarkeit? Warum?

Auch wenn es auf den ersten Blick so wirkt als würde reine Vererbung eine höhere Codewiederverwendung erzielen, so ist durch das Nichtvorhandensein des Ersetzbarkeitsprinzips die Wartung eine wesentlich aufwändigere, da oft kleine Änderungen große Folgen haben!

17. Was bedeuten folgende Begriffe in Java?

- **Objektvariable, Klassenvariable, statische Methode**
- **Static-Initializer**
- **geschachtelte und innere Klasse**
- **final Klasse und final Methode**
- **Paket**

Instanzvariable (Objektvariablen): Variablen die zu einer Instanz gehören.

Klassenvariable (static): Variablen die zu einer Klasse gehören, in allen Instanzen der Klasse gleich sind. Klassenvariablen werden mit dem Modifier static erstellt.

Statische Methode: Methode die zur Klasse gehört und über den Klassennamen aufrufbar ist

static Initializer (=Klassenkonstruktor): Codeblock: static{….} , dessen Inhalt vor der ersten Verwendung der Klasse ein einziges Mal ausgeführt wird. Kann zur Initialisierung der Klassenvariablen verwendet werden.

geschachtelte Klasse: Sind Klassen innerhalb einer anderen Klasse. Sie können überall dort erstellt werden, wo auch Variablen definiert werden dürfen.

Man unterscheidet zwischen:

- geschachtelte statische Klasse: Dürfen nur auf Klassenvariablen der umliegenden Klasse zugreifen. Haben keinen Zugriff auf Instanz Variablen.
- Innere Klasse: Haben Zugriff auf die Variablen und Methoden der Umgebung. Dürfen keine statischen Variablen, Methoden oder gar statisch geschachtelte Klassen enthalten.

final Klasse: Klasse, welche keine Unterklassen haben dürfen, deren Inhalt somit auch nicht überschrieben werden darf

final Methode: Methode deren Inhalt nicht überschrieben werden darf!

Paket: Klassen können zu Paketen (Name des Verz. in dem sich die Klassen befinden) zusammengefasst werden. Beim Aufruf ist dies zu beachten.

18. Wo gibt es in Java Mehrfachvererbung, wo Einfachvererbung?

Mehrfachvererbung ist nur bei Interfaces möglich. Bei Klassen darf es nur Einfachvererbung geben.
Unterklassen können nur eine Oberklasse haben, aber mehrere Interfaces implementieren.

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

Importieren direkt im Code über: import Namespace.Package.Class

Um vorgefertigte Pakete zu importieren um beispielsweise fertige ArrayLists und HashMaps verwenden zu können.

20. Wozu benötigt man eine package-Anweisung?

Man verhindert, dass die Datei nicht einfach aus dem Kontext gerissen und in einem anderen Paket verwendet wird.

```
package paketName;
```

Ist eine solche Zeile in der Quelldatei vorhanden, muss der Aufruf von `javac` zur Kompilation der Datei oder `java` zur Ausführung der übersetzten Datei im Dateinamen den Pfad enthalten, der in `paketName` vorgegeben ist (wobei Punkte in `paketName` je nach Betriebssystem durch / oder \ ersetzt sind). Wenn die Quelldatei oder compilierte Datei in einem anderen Verzeichnis steht, lässt sie sich nicht compilieren beziehungsweise verwenden.

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

Public: Überall sichtbar, wird vererbt. Bei Methoden die von der Klasse nach außen zur Verfügung gestellt werden sollen. Variablen sollten nie public sein!

Protected: Sichtbar innerhalb des Packets, werden vererbt, auch wenn Unterklasse in anderem Paket.

Default: Wie Protected, aber nur an UK in eigenem Paket vererbt.

Private: Nur innerhalb der Klasse sichtbar. Bei Variablen, Methoden die nur den Abläufen in der Klasse dienen und nicht nach außen sichtbar sein sollen.

	`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

22. Wodurch unterscheiden sich Interfaces in Java von abstrakten Klassen? Wann soll man Interfaces verwenden? Wann sind abstrakte Klassen besser geeignet?

Interfaces entsprechen abstrakten Klassen in denen alle Methoden als abstract definiert wurden (keine Implementierung enthalten). Sie stellen sicher, dass alle in ihnen angegebenen Methoden auch in derselben Form von ihren Unterklassen verwirklicht werden. Klassen können von mehreren Interfaces erben, aber nur von einer abstrakten Klasse.

Abstrakte Klassen sollte man dann vorziehen, wenn die Unterklasse nur von einer Oberklasse (ebendiese abstrakte Klasse) erben soll und man eventuell Methoden schon im Vorhinein implementieren will, welche dann in die Unterklasse mit übernommen werden.

Kapitel 3

1. Was ist Generizität? Wozu verwendet man Generizität?

Generizität ist ein statischer Mechanismus, wo in Klassen und/oder Methoden Typparameter statt konkreten Typen verwendet werden. Man unterscheiden generische Klassen und Methoden. Generizität ermöglicht das Verwenden von Typparametern (z.B.: A), für welche bestimmte Typen(String, Person,..) eingesetzt werden können. Alle Vorkommen dieser Typparameter werden durch den angegebenen Typ ersetzt. Generizität eignet sich besonders gut bei Containerklassen, welche für mehrere Datentypen verwendbar sein sollen.

2. Was ist gebundene Generizität? Was kann man mit Schranken auf Typparametern machen, was ohne Schranken nicht geht?

Bei einfacher Generizität fehlt die Möglichkeit zu überprüfen ob ein Typ der den Typparameter ersetzt überhaupt geeignet ist (bestimmte Methoden erforderlich). Gebundene Generizität ermöglicht das Angeben einer Schranke auf den Typparameter (z.B.: A extends Compareable) Folglich können nur Typen verwendet werden, bei denen es sich um Untertypen der Schranke (hier: Compareable) handelt.

3. In welchen Fällen soll man Generizität einsetzen, in welchen nicht?

Generizität ist am besten dann einzusetzen, wenn die Verwendung die Wartbarkeit erleichtert. (Containerklassen, Iteratoren,…)

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

Der Compiler überprüft, dass eingefügte Typen dem durch den Typparameter festgelegten Typ entsprechen.

Beispiel:
```
List <Integer> x = new List < Integer > ();
x.add(„hallo“)   // Fehler, nicht vom Typ Integer!
x.add(25)        //OK
```

5. 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?

Man unterscheidet:

- homogene Übersetzung: Eine generischen Klasse wird genauso wie jede andere Klasse in genau eine JVM (JavaVirtualMachine)-Code Klasse übersetzt. Als verwendeten Typ wird falls angegeben die erste Schranke des Typparameters ersetzt, sonst (ungebunden) Object als Schranke angenommen. Rückgabetypen werden mittels Typumwandlung in den gewünschten Typ konvertiert.

- heterogene Übersetzung: Jede Verwendung der generischen Klasse mit einem neuen Typ hat die Erzeugung einer eigenen JVM Klasse zur Folge. Diese Methode ist geringfügig schneller, aber auch speicherintensiver. Java verwendet homogene Übersetzung

6. Was sind (gebundene)Wildcards als Typen in Java? Wozu kann man sie verwenden?

Generizität unterstützt keine impliziten Untertypbeziehungen. (d.h.: List<String> ist kein Untertyp con List<Objekt>, nur weil String ein Untertyp von Objekt ist)
Wildcards werden in Java verwendet, wenn man einen beliebigen Untertypen einer Klasse als Typparameter zulassen will.

Beispiel: Die Klassen Viereck und Dreieck erben von Polygon. Einer Methode drawall(List<Polygon> x) sollen in einer Liste (mit Typparameter Polygon) alle Polygone übergeben und gezeichnet werden. Da Java bei Generizität aus Sicherheitsgründen keine Untertypbeziehungen unterstützt kann man die Methode nur mit einer Liste aus Polygonen, nicht aber aus Dreiecken oder Vierecken aufrufen.

Wildcards lösen das Problem:

drawall(List<? extends Polygon> x)

Die neue Methode drawall akzeptiert durch die Verwendung von Wildcards alle Untertypen von Polygon. Alle Untertypen von Polygon bilden praktisch den Typparameter.

Achtung: bei (List<? extends Polygon> x) nur Lesezugriffe erlaubt, da Kovarianz vom Compiler gefordert.
Um nur Schreibzugriffe zu ermöglichen muss zum Beispiel (List<? super traingle> x) angegeben werden. Es können folglich alle Listen von Obertypen von Triangle übergeben werden.

7. Wie kann man Generizität simulieren? Worauf verzichtet man, wenn man Generizität nur simuliert?

Generizität kann manuell simuliert werden in dem die gewünschte Klasse nur Typen vom Typ „Object“ (oder eine andere erste obere Schranke) verwendet. Um mit einer derartigen Simulation zu arbeiten benötigt man aber etliche Typumwandlungen. Allerdings entfällt hier der Vorteil der statischen Typsicherheit. Bei der Verwendung von Generizität werden eventuelle derartige Fehler (In eine Liste mit Typparameter String will man eine Person einfügen) bereits vom Compiler erkannt. Simuliert man Generizität aber nur nach oben beschriebener Art, werden solche Fehler erst zu Laufzeit bemerkt!!

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

- homogene Übersetzung: Eine generischen Klasse wird genauso wie jede andere Klasse in genau eine JVM (JavaVirtualMachine)-Code Klasse übersetzt. Als verwendeten Typ wird falls angegeben die erste Schranke des Typparameters ersetzt, sonst (ungebunden) Object als Schranke angenommen. Rückgabetypen werden mittels Typumwandlung in den gewünschten Typ konvertiert.

- heterogene Übersetzung: Jede Verwendung der generischen Klasse mit einem neuen Typ hat die Erzeugung einer eigenen JVM Klasse zur Folge. Diese Methode ist geringfügig schneller, aber auch speicherintensiver. Java verwendet homogene Übersetzung

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

Man kann Generizität simulieren, indem man beispielsweise Collections für den Typ Object schreibt. Da alle Typen Untertypen von Object sind, ist die Klasse so verwendbar, als wäre sie eine generische Klasse. Man verzichtet dabei allerdings auf die statische Typsicherheit, der Compiler kann also nicht mehr garantieren, dass keine Laufzeitfehler bezüglich falscher Typumwandlung passieren.

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

getClass :	Liefert den dynamischen Typen der Klasse.
U instanceOf T:	Liefert true, wenn U ein Untertyp von T ist, sonst false.
Class.isInstance(Objekt):	Liefert true wenn Objekt eine Insanz von Class ist

11. Was wird bei einer Typumwandlung in Java umgewandelt – der deklarierte, dynamische oder statische Typ? Warum?

Der deklarierte Typ:
Person x = new Student (Herbert, 1234)
String y = x.getMatrikelnummer //fehler, da in Person keine derartige Methode
String y = ((Student)x).getMatrikelnummer

12. Welche Gefahren bestehen bei Typumwandlungen?

Eventuelle Fehler bei der Typumwandlung kommen erst zu Laufzeit zu Tragen oder verursachen im schlimmsten Fall fehlerhafte Veränderungen im Datenbestand.

13. Wie kann man dynamische Typabfragen und Typumwandlungen vermeiden? In welchen Fällen kann das schwierig sein?

Dynamisches Binden und(oder) die Verwendung von Generizität kann helfen Typabfragen und Typumwandlungen zu vermeiden.

14. Welche Arten von Typumwandlungen sind sicher? Warum?

Typumwandlungen sind sicher (lösen keine Ausnahmebehandlung aus), wenn:

- In einen Obertyp des deklarierten Typen umgewandelt wird
- Zuvor mittels Typabfrage sichergestellt wird, dass das Objekt einen entsprechenden dynamischen Typ hat (Ausnahme beachten!!)
- Das Programmstück so geschrieben wird, als ob man Generizität verwenden würde, und dann homogen übersetzt wird. (aufwändig!!)

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

Kovariante Probleme sind Probleme die entstehen, wenn man sich kovariante Eingangsparameter wünscht. Man sollte sie daher vermeiden. Ein Beispiel:

Kovariante Probleme:	Sei U ein Untertyp von T. In T existiert eine Methode M mit dem Eingangsparameter X. In U soll eine Methode M‘ mit dem Eingangsparameter X‘ aufgerufen werden, welcher ein Untertyp von X ist!
Binäre Methode:	Methode bei der übergebene Parametertyp dem der eigenen Klasse entspricht.

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

Multimethoden: Entstehen, wenn man auch beim Parametertyp dynamisches Binden einsetzt (von JAVA nicht unterstützt).

Beispiel:

```
Rind rind = new Rind();
Futter gras = new Gras();
rind.friss(gras); // Rind.friss (Futter x)
rind.friss((Gras)gras); // Rind.friss (Gras x)
```

Würde Java Multimethoden unterstützen, so würde rind.friss(gras);
auch einen Aufruf der Methode Rind.friss (Gras x) zur Folge haben.

Welche Methode aufgerufen wird, entscheidet sich bei Multimethoden mittels dynamischen Binden.

Überladen: Sei U ein Untertyp von T. In T gibt es eine Methode M mit dem Eingangsparameter X, welche in U mit gleichem Namen, aber einem anderem Parameter Y wieder implementiert wird. Y ist hier kein Untertyp von X.
In U existieren folglich die geerbte Methode M, als auch die ÜBERLADENE Methode M‘ mit anderen Eingangsparametern.

Im Gegensatz zu Multimethoden wird hier nicht nach dem Eingangsparameter dynamisch gebunden!

Überschreiben: Wenn man in der Superklasse T eine Methode M an die Subklasse U vererbt und diese Methode in U mit gleichem Methodenkopf aber unterschiedlichen Rumpf implementiert, so wird diese in U überschrieben

17. Wie kann man Multimethoden simulieren? Welche Probleme können dabei auftreten?

Mehrfaches dynamisches binden wie es von Multimethoden gebraucht wird, kann man durch wiederholtes einfaches Binden simulieren. Dieses Prinzip nennt man das Visitor Pattern.

Das Problem: Die Anzahl der zusätzlich nötigen Methoden, wird bei steigender Klassenanzahl schnell sehr groß!

18. Was ist das Visitor-Entwurfsmuster?

Man unterscheidet zwischen Elementklassen und Besucherklassen.
Operationen werden von den Elementklassen in externe Besucherklassen ausgelagert.

Achtung: M Futterarten
N Tiere
M*N Methoden!!

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

1) Unbewusstes Überladen statt überschreiben.
2) Wenn nicht klar entschieden werden kann welche der Methoden aufgerufen werden soll

Ein Beispiel:

```
Methode(Futter f, Gras g){ };
Methode(Gras g, Futter f){ };
Aufrufer(Gras1, Gras2);
```

Diese Problematik kann vermieden werden wenn, …
… sich die Parametertypen min. in einer Stelle unterscheiden (an einer Stelle keine Untertypbeziehungen bestehen)
… aller Parametertypen der einen Methode nur Oberklassen der anderen Methode beinhaltet.

Kapitel 4

1. Wie werden Ausnahmebehandlungen in Java unterstützt?

Ausnahmen sind in Java gewöhnliche Objekte, die über spezielleMechanismen als Ausnahmen verwendet werden. Alle Instanzen von `Throwable` sind dafür verwendbar. Praktisch verwendet man nur Instanzen der Unterklassen von `Error` und `Exception`, zwei Unterklassen von `Throwable`. Unterklassen von `Error` werden hauptsächlich für vordefinierte, schwerwiegende Ausnahmen des Java-Laufzeitsystems verwendet und deuten auf echte Fehler hin, die während der Programmausführung entdeckt wurden. Es ist praktisch kaum möglich, solche Ausnahmen abzufangen; ihr Auftreten führt fast immer zur Programmbeendigung.

2. Wie sind Ausnahmen in Untertypbeziehungen zu berücksichtigen?

Vom Programmierer geworfene Exceptions werden im Methodenkopf hinter dem Stichwort throws aufgelistet. Sie U ein Untertyp von T. In U darf keine Exceptions werfen, die in T nicht definiert sind.

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

DAFÜR …

Unvorhergesehene Programmabbrüche:	Es kommt zu einem Fehler, eine Exception wird geworfen, gefangen, eventuell behandelt und ein umfangreicher Fehlerhinweis ausgegeben.
Kontrolliertes Wiederaufsetzen:	Nach einem Fehler wird das Programm an einem anderem Punkt fortgesetzt, da das Programm auch nach einem Fehler noch laufen soll. (Haupteinführungsgrund für Exceptions)

DAFÜR NICHT…

Ausstieg aus Sprachkonstrukten:	Mit Hilfe von Exceptions kann man auch vorzeitig aus Programmblöcken, Schleifen,… aussteigen.
Rückgabe alternativer Ergebniswerte:	In Java kann eine Methode nur einen Ergebnistyp zurückliefern. Durch das Werfen einer Exception kann man aber einen alternativen Rückgabewert werfen, welcher dann an einer geeigneten Stelle gefangen wird.

Generell sind Ausnahmen sparsam und wirklich nur in Ausnahmesituationen einzusetzen.
Bevorzugter weise sollten diese noch in derselben Klasse gefangen werden, wo sie auch geworfen wurden.

4. Durch welche Sprachkonzepte unterstützt Java die nebenläufige Programmierung? Wozu dienen diese Sprachkonzepte?

Java unterstützt nebenläufige Programmierung durch die Verwendung mehrerer Threads. Threads dienen zum parallelen Bearbeiten von Ressourcen.

5. Wozu brauchen wir Synchronisation? Welche Granularität sollen wir dafür wählen?

Bei der Verwendung mehrerer Threads kann es wenn diese unkontrolliert auf Ressourcen zugreifen zu Inkonsistenzen, unerwarteten Folgen kommen, da nicht alle Threads gleichmäßig arbeiten. Eine gewisse Ressource (oder Methode) kann synchronisiert werden, damit immer nur ein Thread Zugriff darauf hat. Es ist die feinst-mögliche Granularität (Körnung) zu wählen, da Synchronisierung teuer bezüglich Laufzeit ist, da die anderen Transaktionen auf die momentan laufende warten müssen.

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

Deadlocks: Abhängigkeiten zwischen mehreren Threads
(Thread A wartet auf Thread B und B wiederum wartet auf A)

Vermeidung: Lineare Anordnung, welche ALLE Formen von Zyklen verhindert.

Lifelocks: Alle arbeiten, aber es geschieht nichts!

Kapitel 5

1. Erklären Sie folgende Entwurfsmuster und beschreiben Sie jeweils das Anwendungsgebiet, die Struktur, die Eigenschaften und wichtige Details der Implementierung: Decorator, Factory-Method, Iterator, Prototype, Proxy, Singleton, Template-Method, Visitor (siehe Abschnitt 3.4.2)

Zuerst einmal eine Einteilung:

Erzeugungsmuster:	Factory Method, Prototype, Singleton
Strukturelle Entwurfsmuster:	Decorator, Proxy
Entwurfsmuster für Verhalten:	Iterator, Template-Method, Visitor

Factory Method

Ermöglicht ein Objekt durch Aufruf einer Methode anstatt durch direkten Aufruf eines Konstruktors zu erzeugen. Die Objekterzeugung wird in Unterklassen verlagert. Für jede Klasse, welche durch eine Factory erzeugt werden soll, muss eine eigene Creator-Klasse erstellt werden. All jene Klassen sind Untertypen einer allgemeineren Creator-Klasse. Somit existiert parallel zur eigentlichen Klassen-Hierarchie eine eigene Creator-Klassen-Hierarchie. Wird eingesetzt, wenn eine Unterklasse eine Klasse erzeugen soll, welche ihr nicht bekannt ist.

Prototype

Erzeugt neue Instanzen, indem gegebene Vorlagen (Prototypen) kopiert und bei Bedarf vorher geändert werden.
Wird angewendet, wenn:

- Die zu erzeugenden Klassen sich nur in wenigen Parametern unterscheiden;
  Sie nur wenige Zustandskombinationen annehmen können.
- zu instanzierenden Klassen erst zur Laufzeit bekannt sind
- eine parallele Hirachie von Fabriken (Factory Method) vermieden werden soll

Aufbau: Alle Klassen, welche kopiert werden sollen sind Untertypen einer möglicherweise abstrakte Klasse, deren Methode clone() von ihnen überschrieben werden muss.
Diese Methode gibt eine Kopie der Klasse zurück, in der sie aufgerufen wurde.

Singleton (=einelementige Menge)

Erzeugungsmuster, welches verhindert, dass von einer Klasse mehr als eine Instanz existiert.
Auf diese eine Instanz ist jedoch globaler Zugriff möglich.

Wird angewendet, wenn:

- Es von einer Klasse nur eine Instanz geben soll
- das einzige Objekt durch Unterklassenbildung spezialisiert werden soll

Ein Singleton besteht aus einer gleichnamigen Klasse, welche über eine statische Methode instance() verfügt, die einzige Instanz der Klasse zurückgibt.

Singletons:

- Erlauben kontrollierten Zugriff auf die einzige Instanz.
- Vermeiden globale Variablen
- Unterstützen Vererbung
- Sind flexibler als statische Methoden

Decorator

Auch Wrapper genannt ermöglicht die dynamische Vergabe von zusätzlichen Verantwortlichkeiten an einzelne Objekte, ohne diese Verantwortlichkeit auf die gesamte Klasse zu übertragen.
Wird angewendet, wenn:

- Man dynamisch Verantwortlichkeiten vergeben, aber auch wieder entziehen will.
- Erweiterungen einer Klasse durch Vererbung unpraktisch sind.

Proxy

Ist ein Platzhalter (surrogate) für ein anderes Objekt und kontrolliert den Zugriff auf diesen.
Steht zwischen dem Aufrufer und dem eigentlichen Objekt.

Remote Proxies:	Platzhalter für nicht lokale Objekte, welche in anderen Namensräumen existieren (in einem anderen Netzwerk,…) Sie koordinieren die Kommunikation mit diesen (leiten Nachrichten weiter),…..
Virtual Proxies:	Wird verwendet, wenn man ein Objekt erst erstellen möchte, wenn man es wirklich braucht (da Erstellung eventuell aufwändig). Bis dahin wird ein Platzhalter verwendet.
ProtectionProxies:	Kontrollieren den Zugriff auf Objekte. Einsetzbar, wenn je nach Zugreifer und Situation unterschiedliche Zugriffsrechte gelten sollen.
Smart References:	ersetzen einfache Zeiger. Sie können bei Zugriffen zusätzliche Aktionen ausführen.

Iterator

Ein Iterator ermöglicht den sequenziellen Zugriff auf die Elemente eines Aggregats.

Bietet Möglichkeiten wie:

Next:	liefert nächstes Element
hasNext:	true, wenn noch ein Elment vorhanden, sonst false

(eventuell noch previous und hasPrevious)

Wird eingesetzt wenn folgende Eigenschaften gewünscht sind:

- Die Datenstruktur des Aggregats muss nicht offen gelegt werden, da Iterator den Zugriff steuert.
- Mehrere überlappende Zugriffe
- Vorhandensein einereinheitlichen Schnittstelle für die Abarbeitung verschiedener Aggregatstrukturen

Template Method

Definiert die Grundfunktionen einer Klasse; überlässt Einzelheiten in der Implementierung aber ihren Unterklassen.

Wird eingesetzt wenn:

- Den unveränderlichen Teil eines Algorithmus nur einmal implementieren zu müssen. Die veränderbaren Teile aber von Unterklassen bestimmen zu lassen.
- Gemeinsames Verhalten von Unterklassen in einer Oberklasse zusammenfassen zu können
- Erweiterungen in den Unterklassen kontrollieren zu können (hooks!!)

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

Man unterscheidet zwischen

interner Iterator: Kontrolliert selbst, wann die nächste Iteration erfolgt. Es ist kein Konstrukt zur Durchmusterung (wie eine Schleife) notwendig.
Dem Iterator wird hier meist nur eine Routine übergeben, welche von diesem auf alle Elemente ausgeführt wird.

externer Iterator: Anwender bestimmt, wann das nächste Element abgearbeitet werden soll.
Der Einsatz ist meist komplizierter als bei internen Iteratoren, dafür die Einsatzmöglichkeiten flexibler!

3. Wie wirkt sich die Verwendung eines Iterators auf die Schnittstelle des entsprechenden Aggregats aus?

Die Schnittstelle bleibt immer gleich. Auch eventuelle Änderungen innerhalb des Aggregats beeinflussen die Kommunikation nach außen, welche über den Iterator erfolgt nicht.

4. Inwiefern können geschachtelte Klassen bei der Implementierung von Iteratoren hilfreich sein?

Mit Hilfe von geschachtelten Klassen kann man einen Iterator in eine Containerklasse integrieren.
Dies erhöht aber die Klassenabhängigkeit zwischen Aggregat und Iterator noch weiter.

5. Was ist ein robuster Iterator? Wozu braucht man Robustheit?

Beim Verändern eines Aggregats, während es eines Aggregats (Zugriff mittels Iterator), während es von einem (anderen) Iterator durchlaufen wird kann zu Problemen (Ein Element wird zweimal gelesen, ist doppelt vorhanden,..) führen Eine Einfache Lösung ist es das Aggregat vorher zu kopieren und von einer Kopie zu lesen.

Ein *robuster Iterator* ermöglicht das Verändern eines Aggregats (Zugriff mittels Iterator), während es von einem (anderen) Iterator durchlaufen wird ohne es vorher zu kopieren.

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

Factory Method: Anzahl der Klassen nimmt zu, da parallel zur eigentlichen Klassenhierarchie eine parallele Creator- Klassenhierarchie geführt werden muss.

Prototype: hat keine (zumindest keine negativen) Auswirkungen auf die Anzahl der Klassen

Decorator: Verringert die Anzahl der Klassen, da man durch das einfache Erweitern von Objekten mittels Dekoratoren eventuelle Unterklassen einsparen kann.
(z.B.: Fenster mit Scrollbar und ohne Scrollbar)

Proxy: Für alle Objekte auf die mit einem Proxy zugegriffen werden soll, sind zusätzliche Proxy-Klassen notwendig!

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 (Creator ➔ abstrakte Klasse)
Prototype:	unverändert (in bestehende Objekte integriert)
Decorator:	erhöht (Es existiert jeweils ein Objekt mit Erweiterung und eines ohne, viele kleine Objekte)
Proxy:	vermindert (Wenn ein Objekt nie benötigt wird, wird es auch nie erzeugt)

8. Vergleichen Sie Factory-Method mit Prototype. Wann stellt welches Entwurfsmuster die bessere Lösung dar? Warum?

Die Factory Method hat den Nachteil, dass jede neue Klasse auch eine neue Erstellerklasse benötigt und so die Anzahl der Klassen rapide zunimmt. Es muss hier immer eine zur Hirachie von Klassen parallele Ersteller-Klassen Hirachie existieren. Hält sich die Anzahl der benötigten Klassen aber im Rahmen, stellt sie eine gute Möglichkeit dar.
Das Prototype Muster hingegen benötigt keine neuen Klassen, da in jeder Klasse eine entsprechende Kopiermethode besteht. Allerdings ist die Implementierung einer Methode, welche tiefe Kopien erzeugt meist sehr aufwändig!

9. Wo liegen die Probleme in der Implementierung eines so einfachen Entwurfsmusters wie Singleton?

Will man mehrere verschiedene Implementierungen von Singletons anbieten (Singleton hat Unterklassen), muss man trotzdem die Bedingung aufrechterhalten, dass immer nur eine Instanz der Oberklasse Singleton existent ist. Dies lässt sich meist nur mittel switch- Anweisungen oder ähnlichen Sprachkonstrukten erreichen, welche aber mit einer Einschränkung der Wartbarkeit und Flexibilität einhergehen.

10. Welche Unterschiede und Ähnlichkeiten gibt es zwischen Decorator und Proxy?

Ein Proxy kann dieselbe Struktur wie ein Decorator haben. Ein Proxy jedoch ist für die Zugriffssteuerung auf ein Objekt verantwortlich. Er stellt praktisch ein Bindeglied zwischen dem „realen Objekt“ und seinem Surrogat dar.
Ein Decorator erweitert hingegen die Verantwortlichkeiten eines Objekts. Er kann folglich einem Objekt Funktionalitäten hinzufügen, aber auch wieder entziehen.

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

Das Erstellen einer tiefen Kopie kann sich als sehr aufwändig gestalten. In jeder Klasse muss eine eigene Methode copy implementiert werden (unterscheidet sich von Klasse zu Klasse meist wesentlich), welche eine tiefe Kopie der Klasse erstellt.

Flache Kopie:	Die Werte der Variablen in der neu angelegten Kopie sind identisch (zeigen auf dieselben Speicherfelder) mit denen im Original. Die in der Klasse Object definierte clone Methode, welche an alle Klassen vererbt wird, erzeugt flache Kopien.
Tiefe Kopie:	Die Werte der Variablen werden mitkopiert (bekommen eigene Speicheradresse), sind also gleich denen im Original. Um tiefe Kopien zu erzeugen muss die clone Methode überschrieben werden

12. Für welche Arten von Problemen ist Decorator gut geeignet, für welche weniger? (Oberfläche versus Inhalt)

Dekoratoren eignen sich gut dazu, die Oberfläche beziehungsweise das Erscheinungsbild eines Objekts zu erweitern. Sie sind nicht gut für inhaltliche Erweiterungen geeignet. Auch für Objekte, die von Grund auf umfangreich sind, eignen sich Dekoratoren kaum.

13. Kann man mehrere Decorators bzw. Proxies hintereinander verketten? Wozu kann so etwas gut sein?

Ja, die Verkettung von mehreren Decorators als auch Proxys ist möglich!

Decorator: Man könnte zum Beispiel einem Objekt mittels Decorators mit einem Rahmen „dekorieren“. Dieser Rahmen könnte vorher mittels Dekorator eine Zierleiste bekommen haben.

Proxy: Bei Proxies würde es zum Beispiel Sinn machen, ein Virtual Proxy zu verwenden, um die Erzeugung eines Objekts zu verzögern, und dieses mit einem Protection Proxy zu verknüpfen, um etwaige Zugriffsbeschränkungen zu implementieren.

14. Was unterscheidet Hooks von abstrakten Methoden?

Hooks sind Punkte in einer sonst vorgegebenen Implementierung, die von Unterklassen ersetzt werden können. Abstrakte Methoden müssen in jedem Fall überschrieben werden.