**Zusammenfassung**

Lokale-, Instanz- und Klassenvariablen

Lokal: nur in Block verfügbar in dem sie deklariert wurden (keine Standardwerte)

Instanz / Objektvariablen: Anfang einer Klasse deklariert, in ganzer Klasse sichtbar (Individuelle Attribute eines Objekts)

Klassen / statische Variable: Anfang einer Klasse deklariert, an Klasse gebunden -> für alle gleich

Gültigkeitsbereich: Block / Compound Statement

Lokale gelten nur innerhalb der Klammern eines Blocks und in inneren davon; automatische Löschung

Typinferenz: var Geht nur für lokale Variablen! (Selbst ableiten des Datentyps durch den Compiler; Deklaration + Initialisierung muss selbe Zeile; kann Lesbarkeit steigern (z. B. lange Codezeilen mit Wdh. Oder viele Objekte an einer Stelle instanziieren)

Verwendung var: lokale Variablen in Methoden / Konstruktor; for Schleifen, try, Lambda Variablen; Typ bleibt trotzdem nach Initialisierung unverändert

Instanzvariablen: Was es hat; Attribute (Zustände, Eigenschaften, Komponenten) der Objekte

Direkt unterhalb Klassendefinition; Initialisierung mit Standardwerten (null, 0, false); Explizite im Konstruktor

Stehen jeder Instanz individuell zur Verfügung -> unterschiedliche Werte

Zustände: Geschwindigkeit, Blinker an, …

Eigenschaften: Marke, Farbe, …

Komponenten: Komposition (Objekt besteht aus anderen Objekten (Motor, Karosserie, Türen, …)

Aggregation (Objekt mit anderen Objekten assoziiert (Auto hat Fahrer und Ladung)

Lebensdauer: solange wie Objekt, Löschung automatisch

Sichtbarkeit: Innerhalb Klasse in jeder Methode und Block

Außerhalb: Über Modifikatoren, Zugriff über Objektname

Klassenvariablen: direkt unter Klassendefinition; static

Umgehen Objektorientierung (für alle Objekte gleich, speichern keine Objektattribute

Initialisierung: automatisch mit Standardwerten; Explizit sollte sofort am Anfang der Klasse stattfinden

Lebensdauer: Von Anfang bis Ende des Programms

Sichtbarkeit: Innerhalb Klasse überall, außen über Modifier  
Zugriff über Klassennamen

Klassenvariablen als Konstanten: final und Großschreibung

Lokale Variablen überdecken Instanz und Klassenvariablen, äußere bleibt unverändert -> Vermeiden!

Lokale gegenseitig können sich nicht überdecken

Instanz- und Klassenvariablen in UML. Static Quadrat. Final Pin-Nadel

Beziehungen als Kante. Ein Objekt: 1…1, Mehrere (z. B. Array): 1…\*

Nicht öffentliche werden nicht in Liste angezeigt, Kante bleibt aber bestehen

Methoden und Konstruktoren

Methoden (Objektmethoden): Was kann es

Zugriff auf alle Instanzvariablen (Manipulieren)

Dient der Benutzung / Bedienung des Objekts

Signaturgestaltung: Name muss Aufgabe erkennbar machen, keine geheimen Nebenaufgaben

Möglichst wenig Parameter; lieber Instanzvariablen manipulieren (Zustand verändern)

Geheimnisprinzip

Klassenmethoden (statische Mehtoden)

Static -> gehören keinem Objekt

Kein Zugriff auf Instanzvariablen und this, nur andere statische Elemente

Aufruf über Klassennamen

Umgehen Objektorientierung (mit Vorsicht einsetzen)

Anwendung: Hilfsmethoden ohne Objektbezug (Math)

Überladen: Gleicher Methodenname, aber unterschiedliche Parameterliste (Datentyp, Reihenfolge, Namen egal)

Unterscheidungsmerkmal darf nicht Rückgabewert sein (ggf. unsicher welche Methode verwendet wird)

geht über Klassen- und Objektmethoden

Häufiger Einsatz: Standardwerte setzen

Konstruktor: Wie wird Objekt erzeugt

Initialisierung der Instanzvariablen (z. B. mit Standardwerten)

Aufruf immer, wenn new verwendet wird

Parameter, um Objekte einfacher zu erzeugen -> Aufrufer kann Attribute bei Erzeugung festlegen

This, um Namenskonflikte aufzulösen

Überladen: Mehrere Konstruktoren möglich, aber unterschiedliche Parameterliste (Auswahl bei new Anhand der Parameterliste)

Problem: Erzeugt häufig redundanten Code -> Fehleranfällig (alle redundanten Stellen müssen gleichzeitig geändert werden)

-> Gegenseitiger Aufruf this(…)

Mit super() Aufruf Konstruktor der Oberklasse

Konstruktor wird zwingend aufgerufen (implizit super() -> wenn es diesen nicht gibt muss explizit angegeben werden)

Aufruf anderer Konstruktor immer in erster Zeile (egal ob super oder this)

Reihenfolge: Konstruktoren rufen sich gegenseitig auf, letzter ruft Oberklasse auf

Auto(String name){this(name, 0)} -> Auto(String name, int ps){this(name, “No-Name”, ps)} -> Auto(String name, String marke, int ps){super(name);this.marke=marke;this.ps=ps;}

Alternativ: Hilfsmethoden

Ohne Angabe von Konstruktor im Code: Standard Konstruktor (kein Parameter, nur super() Aufruf)

Daher Problem, wenn Oberklasse keinen parameterlosen Konstruktor hat

Implizite Standardkonstruktor verschwindet beim Anlegen eines beliebigen expliziten

Sofortige Initialisierung für Konstanten

Klasseninitialisierung mit statischen Initialisierungsblöcken (static {…}) (Initialisierung von Klassenvariablen

Exemplarinitialisierer: Initialisierung von Instanzvariablen bei anonymen inneren Klassen {…}

Reihenfolge:

1. Statische Init. (laden der Klasse (meist Anfang von Programm))
2. Objektbezogene Init. (Erzeugen des Objekts -> erst mit super zuerst Instanzvar der Oberklasse

Exemplarinitialisierer werden nach super() an den Anfang eines jeden Konstruktors kopiert

UML: Darstellung mit Name und Parameterliste. Je Konfiguration ggf. nur öffentliche. Static: Quadrat

Packages, Module, Java Archive:

Modularisierung: Zerlegung in zusammengehörige Sinneinheiten

Geheimnisprinzip: Implementierungsdetails Klasse versteckt; Nur zur Benutzung wichtige Informationen nach außen sichtbar (Schnittstellen)

Package: Sammlung zusammengehöriger Typen (BigDecimal, BigInter in java.math)

Jede datei.class gehört in genau ein Package

Package Deklaration immer in erster Zeile

Hierarchischer Aufbau: vollständiger Klassenname: Package-Namen + Klassenname (java.util.Random

Aufbau häufig: Domänenname rückwärts

Package auf Festplatte: Ordner (src Ordner wird erstellt -> alle Packages zum Projekt / Modul da rein)

Kompilieren erzeugt selbe Paket-Hierarchie

Src ist Default-Package, sollte nicht verwendet werden -> jede Klasse in echtem Package

Verwendung anderer Klassen: voller qualifizierter Name:

java.util.Random random = new java.util.Random()

oder import: import java.util.Random; -> Random random = new Random()

imports direkt nach Package

Import aller Klassen eines packages: .\*; einzelne Klasse über Name

Effizienz ist gleich (Compiler sucht nur benötigte Packages); aber ggf. Namenskonflikte (selber Name) -> wieder Verwendung voll qualifizierter Namen

Alle statischen Variablen und Methoden einer Klasse (statischer import) import java.lang.Math.\*;

Einzelne statische Variable oder Methode einer Klasse (statischer import) import java.lang.Math.PI;

-> Zugriff: einfach PI (nur maßvoll verwenden wegen Namenskonflikte durch Überdeckung der Imports durch eigene Variablen

Java Lang steht immer zur Verfügung (Wrapper, Object, String, …)

UML: Package als Rahmen um Klassen; Oben links Package-Name

Module: Programm lädt nur noch Module der Java-Plattform, die es benötigt -> Speicher- + Performancevorteil

Aufteilung in Module: stärkere Kapselung

Problem Ohne Module: Alle öffentlichen Elemente nach eines Packages nach außen sichtbar, auch wenn sie nur zur Unterstützung für anderes Packet dienen

Lösung: JPMS -> Packages werden Modul zugeordneet, Sichtbarkeit eines Packages innerhalb und außerhalb des Moduls kann im Modul festgelegt werden

Alle Packages in Modul

Modul entscheidet: Welche Packages nach außen exportiert; an wen genau (allgemeiner Export oder nur an bestimmtes Modul (qualifizierter export)); required Modules (auf welche anderen es zugreifen möchte) -> module-info.java im src-Ordner

Modularisierung optional -> falls nicht verwendet: module-info.java löschen

Java Archive: komprimiertes ZIP-Archiv; dient Zusammenfassung und Auslieferung aller Anteile eines Programms)

Modifikatoren:

Modularisierung: zusammengehörige Methoden und Variablen in Klassen zusammengeführt

Geheimnisprinzip: Implementierungsdetails einer Klasse verstecken; nur zur Benutzung wichtige Informationen nach außen sichtbar machen (durch Zugriffsmodifikatoren)

Public geht bei allem (bei Interfaces geht nichts anderes)

Protected: Var, Methode, innere Klassen

<Standard>: Var, Methode, Konstruktor, Klasse

Private: Var, Methode, Konstruktor, innere Klasse

Getter und Setter: Zugriff auf private Variablen

set…(); get…() (bei boolean: is…())

Getter machen lesbar; Setter machen schreibbar

Entweder nur lesbar, nur schreibbar oder beides aber Setter mit Rahmenbedingung

Private Methoden: Hilfsmethoden

Private Konstruktoren: Erzeugung nur noch innerhalb Klasse -> erzeugen von außen über z. B. Methoden oder Singeltons private Singleton(){} public static Singleton getInstance(){ if (instance == null){instance = new Snigelton();} return instance;}

Private (innere) Klassen dienen Strukturieren Klasse

UML: oft nur public Variablen in Liste, Linien bleiben aber bestehen

static: Macht aus Variablen und Methoden Klassenvariablen /-methoden

Statische innere Klassen können nicht auf Instanzvariablen der umgebenden Klasse zugreifen; Instanziierung von außen ohne Objekt der umgebenden Klasse

Klasseninitialisierung mit statischem Initialisierungsblock

final: unveränderliche Anteile

Klassen: nicht mehr vererbt (können schneller verarbeitet werden, da keine dynamische Bindung); Wrapper, String, Math, … final

Methoden: können nicht in Unterklasse überschrieben werden -> für alle Unterklassen gleich

Variablen: nach Initialisierung nicht mehr änderbar (Objektzugriffe bleiben möglich!)

Aufgeschobene Initialisierung möglich

final geht auch für lokale Variablen, auch in Parameterlisten, Klassenvariablen, Instanzvariablen

Lokale Variablen häufig unveränderlich -> guter Stil: solche auch mit final zu modifizieren

-> Programmierfehler verhindert (versehentliches ändern), schlechter Programmierstil verhindert (Mehrfachnutzung, Umbelegung und Wiederverwendung von Variablen unterbunden)

Klassenvariablen mit final sind Konstanten -> Großschreibung

strictfp vor nicht abstrakten Methoden; Klassen und Interfaces -> gleiches Ergebnis auf allen Plattformen, aber ungenauer

Javadoc und Kommentare:  
Kommentare: Warum und Wozu

Einzeilger (Anmerkung)

Mehrzeilig (Beschreibung Code-Abschnitt)

Javadoc (automatische Auswertung; besondere Bedeutung in Java)

Gute Kommentare:

Warum; nicht wie

Wichtige Hintergrundinformationen

Gut lesbar, verständlich

Aktuell und passen zum Code

Schlechte Kommentare kontraproduktiv:

Werden oft gelesen

Dürfen nicht irreführend sein

Kommentare zu schlechtem Code; verrottete Kommentare; überflüssige Kommentare

-> so wenig Kommentare wie möglich

Schlechten Code (Nebenwirkungen, Methodennamen, Komplexität) nicht kommentieren, sondern umschreiben und verbessern

Code verrotet durch Änderungen (Features, Bugfix, …)

Kommentare noch schneller (bei jeder Änderung, sofern Kommentar nicht aktualisiert wird -> mit der Zeit falsch, irreführend oder überflüssig)

Überflüssig: Geschwätz, Humor, Offensichtliches

ToDo nur wenn etwas nicht erledigt werden kann (fehlende Schnittstelle, …) nicht bei schlechtem oder fehlendem Code)

Wichte Kommentare:  
Grundlegende Informationen zum Programm, Klarstellungen, Warnungen vor Konsequenzen, juristische Informationen

Javadoc für nicht private Anteile von Klassen

Kapselung durch ADT inspiriert

Konkreter Datentyp (z. B. boolean, short, …)

ADT (formale oder informelle Spezifikation (Daten und alle zulässigen Funktionen)

Ohne Beug auf bestimmte Implementierung oder Programmiersprache

Stack, Queue, Liste, …

ADT:

Signatur (alle öffentlichen Methodensignaturen -> Namen, Typen Parameter, Typen Rückgabe; Wie benutzt?),

Semantik: Javadoc-Dokumentation (Verhaltenscbeschreibung; Was macht er? Nicht wie)

Klassenbeschreibung, Konstruktorbeschreibung, Methodenbeschreibung

/\*\*

\*Das Auto wird um den gegebenen Wert beschleunigt.

\*

\* @param kmh Wert um den das Auto beschleunigt warden soll.

\*

\* @return Aktuelle Geschwindigkeit des Autos.

Für überschriebene Methoden nicht nötig

@throws IllegalArgumentException wenn … (nach return) v.a. bei Unchececked Exceptions

Clean Code, Namen und Kommentare:

Code verrottet -> wird mit Zeit chaotischer, komplexer, schlechter lesbar und schwieriger wartbar

Grund: neue Features, Änderungen (Bugfixes, …), Im Laufe eines Projekts

Der beste Code verrottet mit der Zeit, wenn man nichts dagegen macht

Produktivität sinkt mit Grad der Verrottung -> Änderungen führen zu defekten an andere Stelle

Endgültig verrottet: Jede Änderung erzeugt neue Bugs -> gar nicht mehr sinnvoll verändern

Sauberen Code erstellen: Pfadfinder-Regel; ständiges Refactoring; Unit Tests

Aussagekräftige Namen; Richtiger Umgang mit Kommentararen und Javadoc

Gut strukturierete Code Implementierung -> Lesbar; Formatierung + Code Conventions; bekannte Design Patterns

Erstellen + pflegen Testfälle

Gute Namen brauchen Zeit, dürfen sich auch mal ädern

Namen sollte wichtige Fragen beantworten (Warum existiert, Was tut, Wie benutzt)

Falsch: i,j; falsche Informationen BMW als Name vw speichern; keine Datentypen figurenArray

Nicht IMessbar; Humor; Codierung oder Abkürzung (nur SI-Norm); Plural für Listen

Aussprechbare / Suchbare Namen

Keine Synonyme (ein Wort pro Konzept)

Passender Kontext (Leser weiß wo er ist -> in Klasse Kreis nicht kreisRadius)

Länge: Langer Geltungsbereicht -> langer Name, kurzer Geltungsbereich kurzer Name

Fach-Termini aus Problemdomäne richtig verwendne

Englisch

Jeder Variable erfüllt einen Zweck (aus Namen ersichtlich)

Benennung: Klassenname für allgemeine Objekte; Rolle wenn Objekt bestimmte Rolle erfüllt

Methodenname: Eine einzige Aufgabe, aus Namen eindeutig ersichtlich (keine zusätzlichen Kommentare)

Verb

Prinzip der Geringsten Überraschung: Überrasche niemals Benutzer; bei Aufruf sollte passieren was Benutzer erwartet; Keine geheimen Nebeneffekte die nicht aus Namen hervorgehen

date.add()?? -> date.increaseByDays(); oder date2 = date.yearsLater();

Parameternamen auch Zweckbeschreibend

get/is; set

Single-Responsibiltiy-Prinzip: Klasse hat nur eine einzige Verantwortlichkeit (aus Namen ersichtlich)

Name beantwortet, was es ist -> Methoden erahnbar

Substantiv; nicht Manager, Processor, Data (Meta-Begriffe) -> nur in Verbindung mit anderen Begriffen (LevelManger)

Packagename: Domain rückwarts -> Weltweite Eindeutigkeit (ggf. auch Abteilung und Teamname)

Kleinbuchstaben

Refactoring ändert nicht die Kommentaer

Vererbung mit extends:

Grund: Vermeidung redundanter Code

Gemeinsamkeiten in Oberklasse realisieren

Nur Unterschiede in Unterklassen implementieren

public class Student extends Person{

Oberklasse / Elternklasse /Superklasse

Unterklasse / Kindklasse / Subklasse

Erbt auch alles was Oberklasse geerbt hat

Keine Mehrfachvererbung

Vererbt: Alle Datentypen der Oberklasse (inklusive Interfaces), Exceptions in Methoden (optional)

Nicht: Private Elemente, Konstruktoren

Vererbung: Is-A Beziehung

Alle sind implizit Object

Objekte mit gemeinsamen Datentypen können gemeinsam abgelegt werden (Listen; Zugriff aber nur auf Variablen und Methoden des genutzten Datentypens)

Geerbte Elemente können verändert werden

Erweitern: Unterklasse besitzt geerbte Funktionalität und neu hinzugefügte Funktionalität

Unterschied beim Aufruf nicht erkennbar

Inhalt von geerbten Methoden darf verändert werden -> @Override; gleiche Signatur

Methode der Oberklasse ist somit verdeckt

Redundanter Code kann durch super.aktuellerVerbrauch() \* 1.1; behoben werden

Vorteil überschreiben: Alle erben selbe Methode -> Aufruf in Schleife möglich

Änderung durch Kindklassen werden durch dynamische Bindung berücksichtigt.

Dynamische Bindung: Überschreiben führt zu mehreren Varianten -> passende wird zur Laufzeit gewählt -> keine if / switch nötig

@Override nicht zwingend aber sehr hilfreich (Tippfehler, Parameter -> überladen, private)

Da hier keine dynamische Bindung stattfindet

Variablen können nicht überschrieben werden (keine dynamische Bindung nur überdeckt -> Vermeiden)

Final bei Klassen -> nicht vererbbar

Final bei Methoden -> nicht überschreibbar (Schutz einer Implementierung)

Zugriffsmodifikator Methode darf öffentlicher gemacht werden

Kovarianter Rückgabetyp -> Rückgabetyp darf spezieller gemacht werden (durch den Typ ersetzen, der weiter unten ist, z. B. Auto durch BMW und Opel)

Konstruktoren nicht vererbt -> Zugriff über super(…)

Frühestmöglicher Aufruf (alle Variablen der Oberklasse initialisiert bevor Subklasse darauf zugreift)

-> super(…) immer in erster Zeile eines Konstruktors

Implizit (nur wenn Standardkonstruktor in Elternklasse existiert)

Impliziter Standardkonstruktor verschwindet beim Erstellen eines expliziten Konstruktors

Listen:

Listen implemtieren Collections (direkt oder indirekt)

Fertige Implementierungen: ArrayList, LinkedList

Listen sind in größe Veränderlich; Beim löschen werden Lücken geschlossen

LinkedList: doppelt verkettete Liste mit Vorwärts und Rückwärts Zeigern

Elemente werden mit Zeigern zu Liste verkettetet

Einfügen, Löschen: Zeigeroperationen automatisch -> sehr schnell

Zugriff ggf. Langsam (Listendurchlauf)

LinkedList<Auto> autos = new LinkedList<>();

Andere Collection kann im Kontruktor übergeben werden (Typ Auto)

Elemente werden der Reihe nach hinzugefügt

Oder List.of() -> LinkedList<Auto> autos = new LinkedList<>(List.of(new Auto(“BMW”), …));

Hinzufügen:

addFirst() Beginn; add; addLast ans Ende

add(4, new Auto(…)) -> am index 4 (5. Stelle) einfügen -> IndexOutOfBounds wenn weniger als 4 Elemente

addAll(…) andere Collection

Zugriff: getFirst() OutOfBounds wenn leer; get(7) -> Auto an 8. Stelle; OutOfBounds wenn weniger als 8;

getLast() outofbounds wenn leer

Löschen: removeFirst(), remove(7), removeLast(), geben je Objekt zurück

removeAll(collection) -> Alle löschen, die in anderer Liste sind; equals nötig; boolean remove(auto) -> erstes Vorkommen

set(4, new Auto(…)) -> ersetzt Stelle 5

autos.conatains(auto); autos.clear(); autos.size(); autos.isEmpty()

autos.sublist(2,5) -> 3–5 Element; Operationen wirken auf beide -> autos.sublist(2,5).clear() löscht 3-5

LinkedList<Auto> clone = (LinkedList<Auto>) autos.clone(); Flache Kopie

autos.retainAll(collection) Schnittmenge

ArrayList: Einfügen+Löschen: Inhalte umsortiert; Mehr Speicher -> vergrößert -> daher ggf. aufwändig

Zugriff: sehr schnell (Arrayzugriff)

Durch gemeinsames Interface List, gibt es viele Methoden in beiden

Konstruktor aber verschieden

Erzeugen: Vergrößern ggf. sehr aufwändig -> mit Konstruktoren optimierbar

Default: 10, verdoppeln; new ArrayList<>(15) -> 15 Elemente, verdoppeln; new ArrayList<>(20,5) -> 20 Elemente je um 5 vergrößern

Auswahl: LinkedList: schnelleres hinzufügen + löschen; langsamerer Zugriff

ArrayList: schnellerer Zugriff; Langsames Hinzufügen und Löschen

Gleichzeitige Nutzung: Zwischen beiden Listen wechseln -> effizienteste für bestimme Operationen (aber zusätzliche Kosten)

Zugriff: 399.999 Element: ArrayList: EinSchritt (Startadresse + Größe \* 399.999)

LinkedList: 399.999 Schritte (vorantasten)

Löschen 2. Element von 400.000: ArrayList Umkopieren aller folgenden Elemente (399.998 Schritte)

LinkedList: Schnelles und einfaches Umbelegen der Zeiger -> zwei Schritte

Umwandeln zwischen Listen mit Konstruktor und andere Liste einfach übergeben

In Array umwandeln: myList.toArray(new Integer[0]);

Array als Liste: List<Integer> zahlen = Arrays.asList(myArray); LinkedList<Integer> zahlenLL = new LL<>(zahlen); \* import java.util.Arrays.

Schleifen: for -> flexibel; Achtung LInkedList -> Zugriff kostet Ziet

Foreach: effizient; weniger Möglichkeiten als for

ListIterator: effizient, auch rückwärts, Hinzufügen und entfernen möglich

LIstIterator<Auto> it = autos.listIterator(); while(it.hasNext()){Auto auto = it.next(); it.remove();}

Listen warden umsortiert bei einfügen und entfernen (links oder rechts verschoben)

Daher muss Reihenfolge stabil bleiben in Schleifen -> Entferen und hinzufügen nur mit ListIterator sonst undefinierter Zustand (Falsches Ergebnis oder Absturz)

Speichern in zusätzlicher Liste und danach addAll / removeAll

Oder remove(); add(…) (an aktueller Stelle einfügen); set(…) (ersetzen) im Iterator

Polymorphismus und dynamische Bindung:

Polymorphismus: Vielgestaltigkeit (mehrere Datentypen pro Objekt)

Unterklasse erbt von Oberklasse + Interfaces (Object implizit)

-> Verberung: Is-A Beziehung

Ohne extends implizit Object

Objekt kann mit jedem seiner Datentypen verwendet werden variable Tier kann ein Zebra zugewiesen werden, aber nicht umgekehrt oder in verschiedenen Vererbungszweigen (Typumwandlung)

Nach oben: Generalisierung, nach unten: Spezialisierung

Von unten nach oben: automatische Typumwandlung: Tier otto = new Zebra(„Otto“)

Objekte können einfach allgemeinere Datentypen zugewiesen werden

Bei der Umwandlung gehen Objektspezifische Eigenschaften verloren -> Zugriff nur noch auf Variablen und Methoden der Oberklasse

Vorteil: Gemeinsame Verarbeitung von verschiedenen Objekten mit gemeinsamen Datentypen (z. B. in Schleifen und Arrays. Aber nur die Methoden und Variablen des gemeinsamen Datentyps stehen zur Verfügung -> Kompakte Programmierung ohne überflüssige Verzweigungen)

Typumwandlung nach oben nur mit Cast-Operator -> Zebra zebra = (Zebra) object; Spezifische Eigenschaften dadurch wieder verfügbar

Sinnlose Casts werden auch zugelassen vom Compiler -> Achtung Laufzeitabsturz -> Weitgehend vermeiden

zebra instanceof Tier -> true sobald Tier einer der Datentypen von Zebra ist

Bei null kommt false

Mit getClass() überprüfen, ob es genau die Klasse hat (egal welcher Typ es aktuell ist.

hugo.getClass() == Zebra.class

Im selben Vererbungszweig / ähnlichen? Kommt false; Bei ganz anderem z.b Apfel: Compilerfehler

Objekt ist null führt zu Laufzeitabsturz

Typ-Prüfung generell vermeiden, Dynamische Bindung

Keine dynamische Bindung bei Methoden die private, static oder final sind -> können nicht überschrieben werden. (Auch wenn kein Modifikator aber anderes Package)

Falls trotzdem gleicher Name erstellt wird -> nur überdeckt, nicht überschrieben -> keine dynamische Bindung!

Vermeidbar mit @Override -> compiler erkennt den Fehler

Override optional, aber sinnvoll da dynamische Bindung nicht aus Versehen nicht stattfindet (falscher Methodenname, falsche Parametertypen)

Variablen: keine Dynamische Bindung; werden überdeckt -> Fehleranfällig -> vermeiden

Abstrakte Klassen und Interfaces:

Abstrakte Klassen: abstract -> Basis Klasse für extends (Unterklassen bildbar)

Instanziierung nicht möglich (Keine Objekte)

Inhalt: Normale Methoden, Variablen, Konstruktoren und abstrakte Methoden (ohne Implementierung)

Abstrakte Methoden immer public abstract , nicht final, static

Keine Implementierung; statt {…} nur ;

Müssen von Unterklasse überschrieben werden (ausschließlich für dynamische Bindung angelegt)

Sobald Klasse abstrakte Methode hat muss die Klasse abstract sein

Rein abstrakte Klassen nur abstrakte Methodden; Parteill abstrakte Klassen (auch konkrete Implementierung)

Umfang 2D Figur muss berechnbar sein, aber für alle unterschiedlich -> abstrakte Methode

In Vererbungshierarchie muss abstrakte Methode nur einmal überschrieben werden, dann nicht mehr abstrakt (danach wieder optional)

Erbende Klassen dürfen abstrakt sein, dann kann die abstrakte Methode überschrieben werden, aber muss nicht; Unterklassen müssen trotzdem

Auch hier keine Mehrfachvererbung

Interfaces: public interace Messbar (immer public)

Ähnlich abstrakte Klassen, aber keine Variablen, Objektmethoden oder Konstruktoren

Nur: Konstanten (public static final), Abstrakte Methoden (public abstract), Klassenmethoden (public static oder private static), Geschachtelte Typen und Default Methoden

public, abstract und public static final sind müsse nicht angegeben werden

Implementierung mit implements (erbt Datentyp, Methoden und Konstanten)

Wichtigste Elemente in interfaces: Abstrakte Methoden (nur für dynamische Bindung); erzwingt überschreiben in Klassen, die interface implementieren

Wenn Klasse Methode nicht implementieren möchte, muss die abstrakt sein

Erste nicht abstrakte Unterklasse muss abstrakte Methode implementieren

Konstanten: Keine Zustände in interfaces, auch keine Klassenvariablen (nur Konstaten)

Nur unveränderliche oder primitive Datentypen (String, int, …), da Objekt sonst verändert werden könnte

Statische (Utility) Methoden können verwendet werden, müssen aber implementiert werden, können nicht überschrieben werden. Standard Modifier public -> private kann explizit angegeben werden (Hilfsmethode zur Vermeidung von Redundanz)

Zugriff über Interfacenamen, nicht über Klassennamen der implementierenden Klasse

Interfaces erlauben Mehrfachvererbung (da keine Kollision zwischen abstrakten Methoden

Mehrere Interfaces durch Komma trennen

Interfaces werden weitervererbt; Überschreibung nur einmal nötig pro Vererbungslinie

Interfaces können von anderen Interfaces mit extends erben (Subinterfaces sind Unterklassen). Auch hier ist Mehrfachvererbung erlaubt

Verwendung häufig: Eigenschaften von Klassen verschiedener Vererbungshierarchien ausdrücken

Namen: substantiviere Eigenschaftswörter (-ble)

Beispiel: Comparable, Iterable

Interfaces oft als Flags für Eigenschaften (leicht mit instanceof z.B. Clonable)

Keine abstrakten Methoden in dem Fall

Seit @Override eigentlich nicht mehr nötig

Wenn Klasse Interface implementiert hat sie eine neue Rolle -> kann auch aus Sicht der Rolle betrachtet werden (z. B. bei Listen)

-> Unterschiedliche Klassen bekommen gemeinsame Eigenschaften -> können unter diesem Blickwinkel verarbeitet werden mit anderen solchen Objekten

UML / Vererbungshierarchie: Links oben bei abstrakten Klassen „A“ und bei Interfaces „I“

Leere Pfeilspitze (Dreieck) von erbender zur Oberklasse.

In IntelliJ Interface grünes I, abstrakte Klassen blaues C mit Klammern drum (Eher Rechteckig)

Implementierung von Clean Code:

Ansätze: Aussagekräftige Namen, Umgang mit Kommentaren und sinnvolle Verwendung JavaDoc

Richtiger Umgang mit fremden Schnittstellen

Variable hat nur einen Zweck (aus Namen erkennbar)

Instanzvariablen für Objekt-Attribute

Klassenvariablen für Klasseneigenschaften

Lokale Variablen zum Speichern von Zwischenergebnissen

Keine Magic Numbers -> lokale Variable mit lesbarer Benennung

Benennen von Zwischenergebnissen (z. B. bei Booleans)

Methode erfüllt eine einzige Aufgabe (aus Namen erkennbar, keine geheimen Nebeneffekte)

Dient der Benutzung von Objekten; Klassenmethoden für Hilfsfunktionen

Eine Methode sollte eine Aufgabe erledigen. Sie sollte sie gut erledigen. Sie sollte nur diese eine Aufgabe erledigen.

Prinzip der geringsten Überraschung -> überrasche niemals Benutzer; Es sollte beim Aufruf das passieren, was Benutzer erwartet

Methoden: Eine Abstraktionsebene pro Methode (beschreiben durch Namen der Methode)

Anordnung: Von oben nach unten -> wie Erzählung; Hilfsmethoden direkt unter der Methode in der sie das erste Mal verwendet werden

Bedingungen, die nicht sofort verständlich sind müssen in Hilfsmethode ausgelagert werden -> lesbar

Grundbedingungen einkapseln (Schwer Kontrollierbar -> lieber mitte = anzahl / 2 -> if (mitte < 2) statt if(anzahl / 2 < 2). Auch bei Schleifen

Negative Bedingungen vermeiden (positive Benennung bei booleans)

Umfang Methode: weniger als 10, besser 5 Zeilen; (wie Geschichte; Eigenheiten Programmiersprache treten in Hintergrund)

Viele kleine Methoden mit gut gewählten Namen führen zu sehr guter Lesbarkeit -> nur selten in Untermethoden eintauchen um zu verstehen. Details müsse nicht gelesen werden -> Fehlersuche schneller

Getter + Setter -> nur lesbar oder nur schreibbar, oder setter mit Rahmenbedingung

Abfrage von Anweisung trennen, nicht aus setter boolean zurück geben

Problem 1: Interne Variablen bringen Nutzer oft nichts -> lieber Methode einparken statt setter für Auto und Parkplatz

Problem 2: Änderungen nicht mehr ohne weiteres möglich (Abwärtskompatibilität) -> interne Variablen sollen offen und austauschbar bleiben

Übergabeparameter: Möglichst wenig (0 oder 1)

Mehr nur wenn es semantisch passt (x,y,z) bei 3D-Koordinate

Reduzierung: Kapseln in Klasse -> Umwandlung Parameter in Instanzvariablen (ggf. mehr davon)

Keine Selektor-Argumente (boolean bei Methoden) um weiteres verhalten zu bestimmen (oft unklar) -> lieber mehrere Methoden mit eigenen Namen

Null ist niemals ein gültiger Übergabeparameter (überflüssig, fehlleitend) und niemals ein gültiger Rückgabewert (jeder muss null prüfen, hunderte Null Prüfungen, NullPointerExceptions)

Lieber leere Liste, leeres Array oder ganz anders implementieren oder Optional<T> zurückgeben

Try Catch Blöcke schlecht lesbar -> Auslagern der Inhalte von Try-Catch-Blöcken in eigene Methoden

Fehlerbehandlung ist eine Aufgabe – eine Methode hat immer genau eine Aufgabe -> Methode die Fehler behandelt sollte nichts anderes tun

public void benutzerEingabeMitFehlerbehandlung() {

try {

benutzerEingabe();

} catch (IOException e) {

… // Fehlerbehandlung

}

}

Aufgaben von Klassen: Single-Responsibility-Prinzip -> Eine Klasse hat nur eine einzige Verantwortlichkeit (aus Namen eindeutig erkenntlich)

Es gibt nur einen Grund zur Änderung

Umfang: Weniger als 10 Methoden, lieber 5

Lesbarkeit: Viele kleine Klassen mit klar umrissener Aufgabe vs. Eine riesige Klasse mit unklarerer Aufgabe

Aufteilen: Finden von kleinen Klassen anhand von Verantwortlichkeiten; Gruppieren von Instanzvariablen; Gruppieren von Methoden

Delegation: Neue Klasse als Instanzvariable nutzen

Vererbung: Klasse in Ober- und Unterklasse aufteilen

Kohäsion erhöhen -> höchste (100%) -> jede Methode verwendet alle Instanzvariablen

Hoch gut, 100% nicht realistisch

-> Reduzierung der Übergabeparameter

-> Möglichst wenig Instanzvariablen, dafür intensive Nutzung (kleinere Klassen)

Vorgehen beim Aufräumen:

Methoden aufteilen (5-10 Zeilen pro Methode)

Magic Numbers entfernen

Zwischenergebnisse und Grenzfälle kapseln

Abstraktionsebene in Methoden anpassen

Variablen neu benennen (Sichtweise des Benutzers)  
Übergabeparameter eliminieren (am besten 0)

In Instanzvariablen umwandeln (sinnvolle Benennung)

Übergabeparameter in neue Klassen Zusammenfassen (Punkt(x,y,))

Klassen aufteilen in 5-10 Methoden

Zusammenpassende Instanzvariablen in spezialisierte Methoden auslagern

Delegation oder Vererbung

Die Klasse Object:

Gemeinsame Oberklasse -> alle erben direkt oder implizit -> alle Objekte sind Typ Object

11 Methoden (6 direkte Verwendung, 5 zum Überschreiben)

Nebenläufigkeit: wait(); wait(long timeout); wait(int nanos) -> warten eines Threads

Aufwecken: notify(); notifyAll()

Feststellen des Typs: getClass(); Verleich mit == Fahrzeug.class

toString: Text Repräsentation -> aktueller Zustand als String (wichtigste Instanzvariablen)

Standard Implementierung von Object: auto.Auto@.... (Hexadezimal des Hashwerts)

@Override

public String toString() {

return "Auto[" + marke + ", " + ps + "PS] ";

}

equals: Inhaltlicher Vergleich eines Objekts mit einem anderen; == prüft nur Referenz (Standard in Object)

Vergleich der wichtigsten Instanzvariablen

Durführung des Vergleichs: Ganzzahlige (Byte, Int, …) ==; float, double: Double.compare(d1,d2) oder Float (geht auf für NaN); 0 bei Gleichheit;

Referenzdatentypen: java.util.Objects: Objects.equals(o1, o2) -> false einer null; true beide null; verwendet equals der Objekte;

Arrays: java.util.Arrays: Arrays.equals(a1, a2) -> für eindimensionale Array mit deren equals; deepEquals für mehrdiemensionale -> Elementweise mit deren equals

Regeln:

@Override

public boolean equals(Object o) {

if (o == this) {

return true;

}

if (o == null || getClass() != o.getClass()) { -> keine NullPointer

return false;

}

Auto other = (Auto) o;

return ps == other.ps

&& Objects.equals(marke, other.marke);

}

Equals erfordert auch überschreiben von HashCode

hashCode(): Berechnet zu Objekt Hashwert (möglichst eindeutige Ganzzahl; positiv oder negativ)

Dient Identifizierung (z. B. in HashSet oder HashMap) -> daher müssen inhaltlich gleiche Objekte auch gleichen Hashwert liefern

Objects liefert nur für identische Elemente gleichen Hashwert

Überschreiben:

Java.util.Objects: hash(Object.. values); geht auch bei null; bei einer: .hashCode()

@Override

public int hashCode() {

return Objects.hash(ps, marke);

}

x.equals(y) == true x.hashCode() == y.hashCode() (also selbe Variablen verwenden)

Wenn hashCode() überschrieben wird, muss auch equals() überschrieben werden

clone: vollwertige Kopie (= geht nicht, da nur Referenz kopiert wird -> selbes Objekt (außer String, Wrapper, …)

Selbe Eigenschaften

Clone von Object erzeugt nur flache Kopie; erzeugt ggf. CloneNotSupportedException

Clone() muss public sein (in Object protected)

Regeln:

x.clone() != x

x.clone().getClass() == x.getClass()

x.clone().equals(x)

Wenn clone überschrieben wird, muss auch equals und hashCode überschrieben werden

Interface java.lang.Clonable muss implementiert warden (zeigt an ob Klasse klonen unterstützt, hat keine Methoden oder Variablen); Aufruf clone() ohne implements führt zu CloneNotSupportedException

Rückgabetyp: Klassentyp -> kein Cast nötig; Exception wird entfernt (klonen möglich)

class Auto implements Cloneable {

@Override

public Auto clone() {

Auto other = null;

try {

other = (Auto) super.clone();

} catch (CloneNotSupportedException ignored) {

}

return other;

}

super.clone() erzeugt flache Kopie; cast nötig; Referenzdatentypen muss noch kopiert werden

Wenn clone in einer Oberklasse implementiert ist, ist kein try catch mehr nötig

Genercis, Comparable und Comparator:

Java typsicher -> kein falscher Datentyp kann verwendet werden (Warnung durch Compiler oder spätestens Absturz zur Laufzeit)

Generics verbessern Typsicherheit -> besser wenn Fehler zur Compilezeit auftritt (Code kann nicht ausgeliefert werden)

Generics liefern zusätzliche Informationen zu verwendeten Datentypen (erhöhen Typsicherheit des Compilers -> mehr Fehler in Compilezeit findbar)

Beim Kompilieren werden alle Informationen gelöscht

Generischer Typ ist Datentyp der durch Typparameter<T> parametrisierbar ist

Typ mit Typargument ist parametrisierter Typ

ArrayList autos vs ArrayList<Auto> autos

Parametrisierter Typ ist fast vollwertiger Datentyp und kann als solcher benutzt werden (Einschränkungen z. B. keine Arrays aus parametrisierten Typen)

Wenn Datentyp parametrisierbar ist sollte immer Typargument angewendet werden (Compiler kann sonst Typfehler nicht entdecken)

Ohne Typargument wie als würde man Object verwenden als Argument

Instanziierung: ArrayList<Auto> autos = new ArrayList<Auto>();

Typinferenz -> meist reicht ArrayList<aAuto> autos = new ArrayList<>();

Verschachtelung möglich: ArrayList<ArrayList<Integer>> …

Mehrere Typargumente: HashMap<String, String> …

Verwendung: Oft wo Objekt andere Objekte verwaltet (z. B. Container, Liste, …) -> alle Collection-Typen -> daher immer Angabe von Typparameter (sonst Liste vom Typ Object) -> man kann in Liste mit Strings auch noch ein Auto adden -> Absturz erst zur Laufzeit

Generische Klasse / Interface: Typparameter für beliebigen Datentyp am Klassennamen

public class Klassenname<T>{…} (steht Klasse dann frei zur Verfügung)

Namenskonventionen: T für Type, E für Element, N für Number, K für Key, V für Value

T kann beliebig verwendet in der Klasse werden für Instanzvariablen, Übergabeparameter, Rückgabewerte, im Rumpf von Methoden und Kontruktor (private T t;)

Compiler prüft bereits beim Aufruf des Kontruktors den Typ

Mehrere Typparameter unabhängig voneinander mit verschiedenen z. B. E1, E2

Generische Methoden: public static <T> T flipCoin(T t1, T t2){…} T hier ggf. unabhängig vom Typparameter der Klasse

Einschränken des Typparameters mit extends -> auf bestimmte Eigenschaften beschränkt (nur noch von Basisklasse abgeleitete Typen erlaubt)

Auch mehrere gleichzeitig möglich: <T extends Typ1 & Type 2>

Wildcards: <?> -> beliebiger Typparameter; <? Extends Typ> Typ der Klasse Typ und alle Subklassen von Typ; <? super Typ> Typ und alle Basisklassen von Typ

ArrayList<? extends Number> liste3 = new ArrayList<Double>();

ArrayList<? super Double> liste5 = new ArrayList<Number>();

Comparable -> natürliche Ordnung (paarweise Vergleichbar);

Direkt in der Klasse, die eine Ordnungsrelation bekommen soll

Interpretation: x.compareTo(y) < 0 x kleiner y; > 0 x größer y; == 0 x gleich y

Größte / kleinste Element einer Menge finden (z. B. Auto mit meisten PS); Menge nach Größe sortieren (Auto nach PS)

Anwendung:

Auto implements Comparable<Auto>{

@Override

public int compareTo(Auto other){

return Integer.compare(this.ps, other.ps);

}

Kein cast nötig wegen Typ

Hilfsmethoden der Wrapper(Double.compare() -> geht auch mit NaN…)

Comparable überschreiben -> equals und hashCode überschreiben

(compareTo == 0 -> equals true) -> geprüfte Variablen müssen auch in equals geprüft werden

String implementiert es

Comperator: Kann Eigenschaften beliebiger Objekte vergleichen

compare(x,y) -> < 0: x < y; >0 -> x > y; == 0 -> x gleich y

Keine natürliche Ordnung, sondern beliebiger Vergleich

Unabhängig von equals

Zwei Objekte als Parameter int cmpare(x,y)

Implementierung in separaten Klasse -> z. B. GewichtsVergleich

Anwendung:

public class GewichtsVergleich implements Comparator<Auto> {

@Override

public int compare(Auto a, Auto b) {

return Integer.compare(a.kg, b.kg);

}

}

Auto bmw = new Auto("BMW", 180, 1300);

Auto opel = new Auto("Opel", 120, 950);

if (new GewichtsVergleich().compare(bmw, opel) > 0) {

System.out.println("Der BMW hat mehr Gewicht!");

}

Java Utility Klassen mit Sortiermethoden, die nach natürlicher oder spezieller Ordnung sortieren

java.util.Arrays.sort(Object[] a) -> natürlich

java.util.Array.sort(T[] a, Comparator<? Super T> c) -> speziell

java.util.Collections.sort(List<T> a) -> natürlich

java.util.Collections.sort(List<T> a, Comparator <? Super T> c) -> spezielle Ordnung

Beispiel:

Collections.sort(autos);

Collections.sort(autos, new GewichtsVergleich());

TreeSet:  
Duplikate werden anhand des Vergleichs ermittelt

 Die Methode equals(…) wird nicht für den Vergleich genutzt!

 Beispiel „Größe“ in einer Klasse Person:

 Es kann keine zwei gleich großen Personen geben

 Nur die erste von mehreren gleich großen Personen wird zum Set hinzugefügt

Abstrakte Methoden UML: Oben links „A“

Final: Oben links „F“