**Abstrakte Klasse** – Eine abstrakte Klasse wird durch das vorangestellte Schlüsselwort *„abstract“* vereinbart. Von abstrakten Klassen können keine Exemplare erzeugt werden Eine abstrakte Klasse kann abstrakte Methoden besitzen. Jede Klasse, die von einer a.K. erbt, muss die geerbten abstrakten Methoden überschreiben und implementieren. Eine Klasse kann nicht als *„final“* und *„abstract“* deklariert werden.

**Abstrakte Methode** – Eine abstrakte Methode weist nur eine Signatur und einen Ergebnistyp oder *„void“* auf und wird durch das vorangestellte Schlüsselwort *„abstract“* vereinbart. Wenn in einer Klasse abstrakte Methoden sind, dann ist die Klasse auch abstrakt. Als *„private“* deklarierte Methoden können nicht abstract sein (weil sie in einer abgeleiteten Klasse nicht überschrieben werden können), statische Methoden können nicht abstract sein. Die Vereinbarung einer abstrakten Methode wird mit einem Semikolon abgeschlossen: *„Zugrifsmodifikator abstract Ergebnistyp Name (Parameterliste);“*

**Akteur** – Anwender und externe Systeme, die an AF teilnehmen, aber selbst nicht Teil des zu realisierenden Systems sind werden als Akteure (*actor*) bezeichnet.

**Akteur/AF-Beziehung** – Die Teilnahme eines Akt. an einem AF wird durch seine Aoosz. zwischen beiden ausgedrückt. Andere Beziehungen zwischen Akt. und AF gibt es nicht.

**Aktivität** – In der UML ist eine Aktivität ein benanntes Verhalten. Ein Aktivitätsdiagramm spezifiziert eine solche Aktivität.

**Algorithmus** – Ein Algorithmus ist eine wohldefinierte Verfahrensbeschreibung, die aus einen oder mehreren Eingabewerten einen oder mehrere Ausgabewerte mit bestimmten Eigenschaften produziert. Ein A. löst dabei immer eine Klasse von Problemen. Ein A. hat die folgenden Eigenschaften: **Terminierung** (Ein A. muss bei allen möglichen Eingaben nach endlich vielen Schritten beendet sein und ein Ergebnis liefern), **Finitheit** (Die Beschreibung des A. muss endlich sein, ebenso wie der zur Ausführung benötigte Speicher), **Effektivität** (Alle Schritte eines A. müssen eindeutig und einer endlichen Zeit ausführbar sein), **Determiniertheit** (gleiche Eingabe = gleiches Ergebnis), **Determinismus** (zu jedem Zeitpunkt ist der nächste Schritt eindeutig festgelegt).

**Algorithmus-Komplexität** – Mindestaufwand, den ein A. zur Lösung der Aufgabe benötigt (Rechenaufwand, Speicherbedarf).

**Algorithmus-Robustheit** – Eigenschaft, auch in ungewöhnlichen Situationen definiert zu arbeiten (fehlerhafte Eingaben, etc.).

**Algorithmus-Anpassbarkeit / Wiederverwendbarkeit** – Maß für Aufwand, den A. an eine geänderte Spezifikation anzupassen.

**Anweisungsüberdeckung -** vollständige Anweisungsüberdeckung, wird gefordert das jede Anweisung mindestens einmal ausgeführt wird. Auch prozentual möglich.

Als **Knoten** zählen alle Aktion und jede Verzweigung jedoch **nicht** Verschmelzungsknoten

**Anwendungsfall** – Ein A. (*use case*) bezeichnet eine typische Interaktion zwischen einem Anwender und einem System aus der Sicht des Anwenders. Ein Anwendungsfall umfasst eine Folge von Aktionen, die reguläre, aber auch von normalen Verhalten abweichende Abläufe beschreiben.

**Anwendungsfall Beziehungen** – *„Enthält-Beziehungen“:* Wenn ein AF das Verhalten eines anderen miteinschließt *<<include>>*

*„Erweitert-Beziehung“:* Ein AF wird an einer bestimmten Stelle unter bestimmten Bedingungen durch einen anderen erweitert (optionales Verhalten) *<<extend>>*

**Äquivalenzklassentest**

Äquivalenzklassen bilden und eine hohe Fehlerentdeckungsrate mit einer möglichst geringen Anzahl von Testfällen zu erreichen.

gültige/ungültige Äquivalenzklassen - gültige/ungültige Eingaben

Grenzwertanalyse

**Arithmetischer Ausdruck** – Ein a.A. (*arithmetic expression*) wird gemäß folgender Regeln gebildet: Jede Variable eines numerischen Typs und jedes Literal eines numerischen Typs ist ein arithmetischer Ausdruck; wenn *x* ein arithmetischer Ausdruck ist, dann ist auch (x) ein arithmetischer Ausdruck; wenn *x* ein arithmetischer Ausdruck und *eop* ein einstelliger arithmetischer Operator ist, der vor dem Operanden steht, dann ist auch *eop x* ein arithmetischer Ausdruck; wenn *x* ein arithmetischer Ausdruck und *eop* ein einstelliger arithmetischer Operator ist, der nach dem Operanden steht, dann ist auch *x eop* ein arithmetischer Ausdruck. Wenn *x* und *y* arithmetische Ausdrücke sind und *zop* ein zweistelliger arithmetischer Operator, dann ist auch *x zop y* ein arithmetischer Ausdruck.

**Arithmetische Ausdrücke (Auswertung) -** Ein arithmetischer Ausdruck wird in einer strikten Reihenfolge gemäß folgender Regeln berechnet: Ausdrücke, die in Klammern eingeschlossen sind, werden zuerst berechnet; im Falle von Verschachtelungen von innen nach außen. Klammerausdrücke erhalten somit die höchste Priorität bei der Auswertung; Zweistellige Operatoren werden von links nach rechts ausgewertet; Einstellige Operatoren werden von rechts nach links ausgewertet. Anschließend werden die Operatoren ihrem Vorrang entsprechend ausgewertet; Wenn ein Ausdruck mehrere Operatoren des gleichen Vorranges enthält, werden die Operatoren von links nach rechts berechnet.

**Assoziation und Verknüpfung -** Eine Assoziation zwischen zwei Klassen drückt aus, dass es zwischen Exemplaren dieser Klassen eine Beziehung geben kann. Eine Assoziation besitzt einen Name. Die konkrete Beziehung zwischen zwei Exemplaren wird Verknüpfung genannt. Zu jeder Verknüpfung zwischen zwei Exemplaren muss es immer eine zugehörige Assoziation zwischen den beiden Klassen geben.

**Attribut -** Die Eigenschaften, die alle Exemplare einer Klasse besitzen, werden durch Attrib. dargestellt. Attrib. werden durch Substantive bezeichnet. Ein A. beschreibt lediglich, welche Eigenschaften ein Exemplar einer Klasse hat, jedoch nicht den konkreten Wert.

**Attributwert –** bez. den Wert, den ein Exemplar zu einem bestimmten Attribut besitzt.

**Aufbau von Variablennamen -** Namen von Variablen bestehen aus einer Folge von Zeichen. Als Zeichen sind alle Unicodezeichen zugelassen. Das erste Zeichen darf jedoch keine Ziffer (0-9) sein. Ein Variablenname darf nicht mit einer reservierten Schlüsselwort der Sprache Java oder den Literalen „*true“*, „*false“* und „*null“* übereinstimmen.

**Auswertung einer Zuweisung** Eine Zuweisung der Form *v = e;* wird in folgenden Schritten ausgewertet: Der Type der Variablen *v* wird ausgewertet; Der Wert *w(e)* des Ausdrucks e wird ausgewertet; Der berechnete Wert wird der Variablen zugewiesen, vorausgesetzt, dass der Typ des Ausdrucks e mit dem Typ der Variablen verträglich ist. Falls die Typen des Ausdrucks und der Variablen nicht identisch aber miteinander verträglich sind, wird der Typ des Ausdrucks in den Typ der Variablen umgewandelt.

Bsp.: double x = 1 / 4; (x = 0) double x = 1.0 / 4; (x = 0.25)

**AVL-Baum** – binärer Baum, bei dem sich die Höhen der Unterbäume jedes Knotens um 1 unterscheiden und jeder Unterbaum ein AVL-Baum ist.

**Backtracking -** Versuch-und-Irrtum-Prinzip

**Baum** – Ein B. ist ein gerichteter, schwach zusammenhängender, azyklischer Graph. Jeder Knoten besitzt einen maximalen Eingangsgrad von 1. Die Wurzel hat als einziger Knoten einen Eingangsgrad von 0.

**Balancierter Baum** – Alle Unterbäume haben möglichst gleiche Tiefe.

**Beziehungen** →Assoziation und Verknüpfung

**Binärbaum** – Jeder Knoten hat nur 2 Nachfolger.

**Binden** – Die Festlegung, welcher Aufruf mit welcher Implementierung verknüpft ist, wird als Binden bezeichnet. Wird diese B. während der Übersetzung festgelegt, so nennt man das statische Bindung. Der Typ einer Variable oder eines Ausdrucks wird oftmals erst zur Laufzeit bestimmt. Eine Bindung zur Laufzeit wird dynamische Bindung genannt.

**Block -** Eine Folge von Anweisungen, die in geschweiften Klammern „{“ und „}“ eingefasst sind, nennt man einen Block. Der Aufbau und die Abarbeitung eines Blocks werden schematisch im folgenden Aktivitätsdiagramm dargestellt. Die Anweisungen eines Blocks werden von oben nach unten sequentiell abgearbeitet.

**Casting** – explizite Typenanpassung

**Datenabstraktion** – nur die auf ein Objekt anwendbaren Operationen werden nach außen sichtbar gemacht; Implementierung dieser Operationen bleibt verborgen.

**Datenlexikon** – Eine nach festen Regeln aufgebaute präzise Bestimmung aller Datenelemente, die für ein Anwendungsgebiet wichtig sind. -> siehe Kasten rechts

**Datentyp -** Ein Datentyp ist eine Menge von Werten und eine Menge von Operatoren, um diese Werte zu verknüpfen, zu vergleichen oder ineinander zu überführen.

**Datentyp *boolean -*** Der Datentyp *boolean* umfasst die beiden Literale *true* und *false* sowie die folgenden booleschen Operatoren: den einstelligen Präfixoperator !, der eine logische Negation beschreibt; fünf zweistellige Operatoren &&, ||, &, | und ^, um logische Ausdrücke aufzubauen; zwei zweistellige Vergleichsoperatoren == und !=, die die logische Gleichheit bzw. Ungleichheit beschreiben.

**Debugger** – Hilfsprogramm zur Fehlersuche, gewährt Einblick in das Verhalten des Programms. Funktionen: Programmstopp setzen, schrittweiser Durchlauf des Codes, Untersuchen von Attributen, Klassenvariablen, lokalen Variablen zur Laufzeit.

**Dynamisches Binden** – Bei dem d.B. legt nicht der statische Typ der Variable, sondern die Klasse des Objekts, auf das die Variable zur Laufzeit verweist fest, welche Methodenimplementierung ausgeführt wird. Bei Klassenmethoden gibt es nur statische Bindung.

**Dynamisches Testen -** Dynamisches Testen bedeutet, ein Programm(-teil) stichprobenartig auszuführen, um Unterschiede zwischen seinem erwarteten und tatsächlichen Verhalten zu finden.

**Dynamischer Typ einer Verweisvariablen -** Unter dem dynamischen Typ einer Verweisvariablen verstehen wir den Typ des Objekts, auf das die Variable zu diesem Zeitpunkt verweist. Da eine Verweisvariable im Laufe eines Programms auf verschiedene Objekte verweisen kann, kann sich auch der dynamische Typ ändern. Der dynamische Typ ist immer ein Subtyp des statischen Typs. Der dynamische Typ kann immer mit Hilfe von instanceof überprüft werden

**Einkapselung** – Unterstützt die Modularität von Programmen, vereinfacht Pflege und Weiterentwicklung, möglich durch Bündelung und Geheimnisprinzip.

**Erzwungene Typumwandlung -** Java unterstützt eine erzwungene Typumwandlung, um den Typ eines numerischen Werts während der Laufzeit in einen numerischen Typ ähnlichen Werts zu ändern. Die Syntax einer erzwungenen Typumwandlung ist: *(typename)expression* wobei *typeName* ein primitiver Typ und *expression* ein Ausdruck ist. Die Typumwandlung wird noch rechts nach links durchgeführt und liefert einen Wert vom Typ *typeName*, vorausgesetzt, dass die Konversion keine Kompatibilitätsregeln verletzt.

**EVA-Prinzip** – Eingabe🡪Verarbeitung (Zustand)🡪Ausgabe

**Exemplar** –🡪Objekt

**Fehler** – Im Quelltext vorhandene Ursache eines Fehlverhaltens

**Feld – Ein F. (engl. *array*) gruppiert mehrere Elemente des gleichen Typs zu** einer einzelnen Einheit vorgegebener Größe. Eine Variable für ein Feld wird wie folgt vereinbart: *ElementTyp[] Feldname*; Der Feldtyp ist durch den Datentyp der Elemente (*ElementTyp*), die in einem Feld enthalten sind, bestimmt. Feldtypen sind Objekttypen. Jedes Feld ist somit ein Objekt. Eine Feldvariable ist ein Verweis auf ein Feldobjekt, für das erst dann Speicherplatz angelegt wird, wenn das Feld erzeugt wird.

**Feld, Erzeugung** - Einem Feldnamen wird mittels Schlüsselwort new ein Feldobjekt fester Größe zugewiesen: Feldname = new ElementTyp[IntAusdruck]; Mit dem Ausdruck auf der rechten Seite. → Zuweisung wird ein konkretes Feld mit Elementen des Typs ElementTyp der Größe IntAusdruck angelegt. Die Feldgröße wird durch einen (nicht-neg.) Ganzzahlausdruck bestimmt. Die Referenz auf das erzeugte Feld wird der Feldvariablen Feldname, die vom gleichen Typ sein muss, zugewiesen. Die Feldelemente sind von 0 bis IntAusdruck-1 indiziert.

**Final –** Variable kann als endgültig (*final*) erklärt werden. Der Wert eine f. Variablen kann, nachdem ihr zum 1. Mal ein Wert zugewiesen wurde, nicht mehr verändert werden. Um eine Variable als finale Variable zu deklarieren, wird final dem Variablentyp vorangestellt. Finale Attribute müssen entweder direkt bei der Deklaration initialisiert werden, oder in allen Konstruktoren der Klasse (sonst Übersetzungsfehler). Wenn nicht gewünscht wird, dass von Klassen Subtypen gebildet werden, kann diese Klasse als final deklariert werden (*final class …)* → Methode (final)

**Geheimnisprinzip** – Trennung des Verhaltens von der konkreten Implementierung. Anwender eines Objektes muss nichts weiter als die Signatur und den Ergebnistyp der Methoden kennen, um es benutzen zu können. Darstellung seiner Daten und Implementierung seiner Methoden bleiben verborgen.

**Generalisierung -** Eine Generalisierung zwischen zwei Klassen drückt aus, dass die speziellere Klasse die Eigenschaften, das Verhalten und die Beziehungen der Allgemeineren erbt und dabei erweitern oder überschreiben kann. Ein Exemplar der spezielleren Klasse kann überall dort verwendet werden, wo ein Exemplar der allgemeineren Klasse verwendet werden kann.

**Generalisierungsbeziehung (zw. Akteuren)** - Sagt aus, dass der speziellere Akteur anstatt des allg. Akteurs an einen Anwendungsfall teilnehmen kann. Er kann darüber hinaus aber auch an Anwendungsfallen teilnehmen, an denen der Allgemeinere nicht beteiligt ist. Darstellung als Pfeil (Spezieller → Allgemeiner Akteur). Bsp.: Premiumkunde → Kunde, Chefin → Verkäufer

**Geschützte (*protected*) Klassenelemente -** Ein Klassenelement, das als *protected* deklariert ist, ist innerhalb des Pakets der Klasse sichtbar. Eine Unterklasse *U* hat auch außerhalb des Pakets der Oberklasse *O* Zugriff auf dieses Element. Dazu muss jedoch *O* als öffentlich deklariert sein und der Zugriff muss an einem Element vom Typ U oder einem Subtyp von *U* ausgeführt werden.

**Graph** - Ein Graph ist ein Tupel: *G = (V, E)* wobei V, die Menge der Knoten ist und E ⊆ V x V, die Menge der Kanten ist, wobei eine Kante ein Paar von Knoten ist. (Gerichteter Graph: Kanten sind ein geordnetes Paar (a,b) und eine Kante verlauft von a nach b (ein/ausgehende Kanten, Anzahl eingehende Kanten eines Knoten = Eingangsgrad, Anz. ausgehende Kanten e. Knoten = Ausgangsgrad). „stark zusammenhangend“, wenn es zw. allen Knoten einen gerichteten → Pfad gibt. „schwach zusammenhangend“, wenn es zw. allen Knoten einen ungerichteten → Pfad gibt.) (Ungerichteter Graph: Kanten weisen keine Richtung auf. Anz. aller Kanten des Knotens = Grad. Zusammenhangender G.“ wenn es von jedem Knoten ein Pfad zu jedem anderen Knoten gibt.)

**Import** - Einbindung von Klassen aus einem anderen Paket. Importanweisung (import) mit Platzhalter erstreckt sich nicht automatisch auf Unterpakete.

**Interpolationssuche** - Bei der Suche wird Wissen über die Verteilung der Werte berücksichtigt.

I**nterpreter** - Programmsystem, das Ausdrucke e. Programmiersprache in eine maschinennahe Sprache umsetzt und ausführt.

**Interface (Schnittstelle)** - Ein I. zwingt abgeleitete Klassen, bestimmte Methoden zu implementieren. Im Unterschied zu →abstrakten Klassen besitzen I. keine Exemplarattribute und keine Implementierungen der

Methoden. Ein I. wird wie folgt vereinbart: *interface Name {Attributdeklarationen… Methodendeklarationen…}* Attributdeklarationen können nur öffentliche finale Klassenattribute enthalten. Die Schlüsselwörter *public, static, final* sind zur Deklaration nicht erforderlich. Alle Methodendeklarationen sind öffentlich, nicht-statisch und abstract. Die Schlüsselwörter *public, abstract* sind zur Deklaration nicht erforderlich. Der Rumpf der Schnittstelle darf leer sein. Eine Sch. kann mit Hilfe des Schlüsselworts *extends* mehrere (!) andere Schnittstellen erweitern. Eine Klasse kann mehrere I. implementieren. Schnittstellenmethoden müssen in einer Klassenimplementierung als *public* vereinbart werden, da alle Schnittstellenmethoden implizit *public* sind. Java benutzt das Schlüsselwort *implements* um anzuzeigen, dass eine Klasse ein I. implementiert. Eine Klasse muss jede Methode ihrer Schnittstellen mit den vereinbarten Ergebnistypen und Parametern implementieren oder, falls sie nicht alle Schlüsselwörter implementiert, als *abstract* kennzeichnen. Treten Attribute gleichen Namens in verschiedenen Schnittstellen auf, die von einer Java-Klasse implementiert werden sollen, so müssen sie innerhalb der Klasse mit ihren qualifizierten Namen angesprochen werden, anderenfalls weist der Java-Übersetzer das Programm zurück.

**Kanten** - → Zweigüberdeckung

**Klasse -** Eine Klasse beschreibt in der objektorientierten Programmierung einen Bauplan für viele ähnliche, aber individuell unterscheidbare Objekte. Klassen werden durch Substantive bezeichnet. Wir bezeichnen Objekte auch als Exemplare. Klassen beschreiben die Eigenschaften, die alle Exemplare dieser Klasse besitzen und auch deren Verhalten und Fähigkeiten (Attribute). → Final (Klasse)

**Klassenvariable -** Eine Klassenvariable wird durch das Schlüsselwort *static*, das der Attributvereinbarung vorangestellt wird, nach folgendem Muster deklariert: *static Typ Name;.* Werden immer statisch gebunden.

**Klassenmethode -** Klassenmethoden werden wie Methoden vereinbart, mit dem Unterschied, dass ihnen das Schlüsselwort *static* vorangestellt ist. Werden auch als Klassenattribute bezeichnet.

Knoten → Anweisungsüberdeckung

**Kompatibilitätsregeln** (→ Typumwandlung, explizite) - Jeder Typ kann in sich selbst umgewandelt werden. Jeder num. Typ kann in jeden anderen num. Typ umgewandelt werden. Typ char kann in jeden num. Typ umgewandelt werden und umgekehrt. Typ boolean kann in keinen anderen Typ umgewandelt werden u. umgekehrt.

**Komplexes Objekt -** Objekt, dessen Attribute selbst wieder Objekte sind.

**Korrektheit -** bedeutet, dass es keinen Widerspruch zwischen den vorgegebenen Anforderungen und dem Programm gibt und dass alle Anforderungen erfüllt sind.

Die Korrektheit eines Algorithmus versucht man durch Testen nachzuweisen.

**Konstruktor -** K. sind spezielle Methoden, um Exemplare einer Klasse zu erzeugen. Die Konstruktordeklaration besteht aus einem Kopf und einem Rumpf. Der Name des K. muss mit dem Namen der Klasse identisch sein. Ein K. hat keinen expliziten Ergebnistyp und der Rumpf muss keine return-Anweisung enthalten. Exemplarvariablen werden vor der Ausführung des K. initialisiert. Besitzt die direkte Oberklasse keinen Standardkonstruktor, so muss es sich bei der ersten Anweisung eines Konstruktor der Unterklasse entweder um einen this()- oder einen super()-Aufruf handeln. Es ist jedoch nur einer der beiden möglich. Besitzt die direkte Oberklasse einen Standardkonstruktor und die erste Anweisung im Konstruktor der Unterklasse ist kein this()- oder super()-Aufruf, so wird implizit ein super()-Aufruf für den Standardkonstruktor eingefügt.

**Konvergenz -** mit zunehmender Rekursionstiefe an den Basisfall annähern

**Literale -** Literale (*literal*) sind Folgen von Zeichen, die Werte symbolisieren. Sie stellen Konstanten dar, da ihr Wert festgelegt ist und sich während der Ausführung eines Programmes nicht verändert.

**Lineare Datenstruktur -** L.D. zeichnen sich dadurch aus, dass die Elemente oder Knoten der Datenkollektion in einer Folge angeordnet sind. Den informationstragenden Bestandteil eines Elements nennen wir Eintrag (Bsp. Feld, Stapel, Warteschlange).

**Maschinensprache** - Die im Rechner verarbeiteten Instruktionen orientieren sich begriffl. an den Mogl. der Maschine und werden daher als M. bezeichnet. M. ist f. menschl. Leser schwer zu interpretieren.

**Mehrfachvererbung** - Wenn eine Klasse Spezialisierung mehrere anderer Klassen ist, so nennen wir dies Mehrfachvererbung. Wird von Java nicht direkt unterstützt. Problem: Vererbungskonflikt (bei gleichen Signaturen in mehreren Oberklassen unklar, welche Implementierung ziehen soll). Falls in Java eine Klasse mehrere Schnittstellen implementiert, die Methoden mit gleicher Signatur definieren, so entsteht kein Vererbungskonflikt, denn es erfolgt nur eine einzige Methodenimplementierung in der Klasse selbst.

**Methode** - Das gemeinsame Verhalten aller Objekte einer Klasse wird durch die Methoden der Klasse bestimmt. Jede Methode beschreibt eine Fähigkeit oder mögliche Form des Umgangs mit einem Objekt der Klasse. Methoden werden durch Verben bezeichnet und in der Regel durch ein nachstehendes Klammernpaar „(“ und „)“ abgeschlossen. 🡪Signatur

**Modell** - Ein Modell ist die Abstraktion eines Realitätsausschnitts oder eines Systems. Ein Modell wird mit Hilfe einer Modellierungssprache beschrieben.  
Eine Methode kann nicht überschrieben werden, wenn sie als finalvereinbart ist.

**Nachricht -** Zusammenarbeit von Objekten lauft mit Hilfe von N. ab. Ein Obj. Kann eine N. an ein anderes O. oder sich selbst verschicken. Eine solche N. enthalt Informationen, welche → Methode ausgeführt werden soll, und wenn nötig noch weite Informationen, die das Objekt benötigt, um die Nachricht zu verarbeiten.

**Narrowing** → Typumwandlung, explizite. Casten von Typ u in Typ t, wobei der Wertebereich vom Typ t keine Teilmenge des Wertebereichs des Typs u ist, können Verlust an Informationen hervorrufen. Wenn eine Gleitkommazahl in eine ganze Zahl umgewandelt wird, so wird nicht gerundet, sondern die Nachkommastellen werden abgeschnitten.

**Objekt -** Die OOP bezeichnet die Abbilder konkreter individuell unterscheidbarer Gegenstände und Träger individueller Rollen in Entwurfsdokumenten und Programmen als Objekte (individuell und identifizierbar).

**Operator -** Ein Operator führte eine Funktion (wie Negation, Addition, Subtraktion, Multiplikation, usw.) auf seinem oder seinen Operanden aus und liefert einen Wert zurück. Der Type des Ergebniswerts wird durch den oder die Typen der Operanden bestimmt. Der Ergebniswert hängt vom Operator und von den Werten seiner Operanden ab.

**Öffentliche (*public*) Klassenelemente -** Ein Klassenelement, das als *public* deklariert ist, ist überall dort sichtbar, wo auch die Klasse sichtbar ist.

**Package** - Ein Paket (engl. package) wird wie folgt eingeführt: *package packageName*; Diese Anweisung muss am Anfang einer Übersetzungseinheit (z. B. einer einzelnen Quelldatei) vor die Klassendeklaration gesetzt werden, die zum Paket *packageName* gehören soll. Ohne den Einsatz von Zugriffsmodifikatoren können nur Klassen, die sich in demselben Paket befinden, aufeinander zugreifen. UML: “ ~“

**Persistenz** - Fähigkeit eines Objekts, über Laufzeit eines Programms zu existieren.

**Permutation -** Umordnung einer Folge von Elementen in einer Weise, dass kein Element hinzugefügt wird oder verloren geht und auch kein Element verändert wird.

**Pfad → Graph** - Ein gerichteter P. zw. 2 Knoten ist die Folge von Kanten, wobei der Zielknoten jeder Kante mit dem Startknoten der nachfolgenden Kante identisch ist. Der Startknoten der ersten Kante ist der Anfang des P. und der Zielnoten der letzten Kante das Ende des Pfades. Berücksichtigt man bei einem P. die Richtung der Kanten nicht, erhalt man einen ungerichteten Pfad.

**Pfadüberdeckung** - misst, wie viele der möglichen Pfade durch das Aktivitätsdiagramm abgedeckt sind. Problematisch wird eine vollständige Pfadüberdeckung, wenn der Quelltext Schleifen enthält, da dann sehr viele, manchmal sogar unendlich viele Pfade existieren.

**Pflichtenheft** - Enthalt alle organisatorischen und technischen Vorgaben zur Erstellung der Software. Grundlage für die technische Realisierung der Software. 2 Hauptzwecke: 1. Muss so aufgefasst sein, dass es von Auftraggeber und Fachabteilungen verstanden wird. 2. Muss für Softwareentwickler hilfreich sein.

**Polymorphie** - Objekt einer Unterklasse kann dort benutzt werden, wo ein Objekt der Oberklasse verlangt ist

**Public** - Ein Klassenelement, das als *public* deklariert ist, ist überall dort sichtbar, wo auch die Klasse sichtbar ist.UML: “+“ → Zugriffskontrolle

**Private** - Ein Klassenelement, das als private deklariert ist, ist nur innerhalb der Klasse sichtbar. → Zugriffskontrolle UML: “-“

**Protected** - Ein Klassenelement, das als *protected* deklariert ist, ist innerhalb des Pakets der Klasse sichtbar. Eine Unterklasse U hat auch außerhalb des Pakets der Oberklasse O Zugriff auf dieses Element. Dazu muss jedoch O als öffentlich deklariert sein, und der Zugriff muss an einem Element vom Typ U oder einem Subtyp von U ausgeführt werden. Methoden und Attribute, die nicht jedem Benutzer zur Verfügung stehen sollen, jedoch von Unterklassen

benötigt werden, werden i.d.R. als *protected* deklariert. UML: “# “ → Zugriffskontrolle.

**Referenzvariablen** (Verweisvariablen) - An der Speicherstelle einer Variablen befindet sich nicht ihr Wert sondern lediglich eine Referenz auf einen anderen Speicherbereich, der den eigentlichen Wert enthalt. Objektvariablen werden standardmäßig „by reference“ zugewiesen.

**Rekursion -** Ein Alg. ist rekursiv, wenn er Methoden/Funktionen enthalt, die sich selbst aufrufen. Jede rek. Losung umfasst 2 grundlegende Teile: a) den Basisfall, für den das Problem auf einfache Weise gelöst werden kann und b) die rekursive Definition. Die rek. Def. besteht aus 3 Facetten: 1) die Aufteilung des Problems in einfachere Teilprobleme, 2) die rekursive Anwendung auf alle Teilprobleme und 3) die Kombination der Teillosungen in eine Lösung für das Gesamtproblem. Bei der rek. Anwendung ist darauf zu achten, dass wir uns mit zunehmender Rekursionstiefe an den Basisfall annähern (= **Konvergenz**)

**Rückgabeanweisung *return -*** Eine *return*-Anweisung im Rumpf einer Methode dient zwei Zwecken: Sie definiert einen Wert, dessen Typ verträglich sein muss mit dem Ergebnistyp der Methode in der sie vorkommt; dieser Wert ersetzt nach Ausführung der Methode den Methodenaufruf; sie gibt die Kontrolle an die Aufrufstelle zurück.

**Sequenzieller Zugriff (sequential access)** - Beim s.Z. wird auf die Kollektion in einer vorbestimmten, geordneten Abfolge zugegriffen, so dass der Aufwand für verschiedene Elemente einer Kollektion variiert. (Bsp.: stack, queue)

**Signatur -** Eine S. besteht aus Methodennamen sowie den Typen ihrer formalen Parameter in der Reihenfolge der Vereinbarung. In einer Klasse darf es niemals mehrere Methoden mit der gleichen Signatur geben. Es ist aber möglich, mehrere Methoden mit dem gleichen Namen, aber mit verschiedenen Parametertypen existieren. (überladene Methodennamen)

**Signatur eines Konstruktors** - Die Signatur eines Konstruktor besteht aus dem Konstruktornamen sowie den Typen seiner formalen Parameter.

Spezialisierung - Eine Klasse B kann nur dann eine Spezialisierung der Klasse A sein, wenn gilt, dass jedes Objekt der Klasse B ein A ist. (Bsp: Jeder PKW ist ein Fahrzeug, nicht umgekehrt). Standardzugriff bei Klassenelementen Ein Klassenelement, dessen Deklaration keinen Zugriffsmodifikator enthält, ist innerhalb des Paketes sichtbar, in dem die zugehörige Klasse enthalten ist.

**Signatur einer Methode -** In Java besteht die Signatur einer Methode aus dem Methodennamen sowie den Typen ihrer formalen Parameter in der Reihenfolge der Vereinbarung.

**Statischer Test -** Bei einem statischen Test wird das Programm nicht ausgeführt, sondern einer Analyse unterzogen. Möglichkeit: Übersetzen des Programms, wodurch auf syntaktische Fehler untersucht wird. Möglichkeit: Inspektion des Quelltextes.

**Statischer Typ** - Der statische Typ einer Verweisvariablen ist immer der Typ, der bei der Deklaration vereinbart wurde.

Er kann sich nicht im Laufe eines Programms verändern. → Dynamischer Typ.

**Stapel (stack)** - Zuletzt hinzugefügt. Element als 1. entnommen (LIFO = Last In First Out)

**Softwareentwicklung** - Entwerfen, Formulieren, Dokumentieren, Überprüfen eines Prog.

**Substantivanalyse** - Bei der S. werden Substantive als mögliche Kandidaten für Klassen, Attribute oder Assoziationen und Verben als Kandidaten für Methoden identifiziert.

**Substitutionsprinzip** - Vererbung bedeutet, dass jedes Objekt der Unterklasse auch eines der Oberklasse ist (→Vererbung). Ein Obj. der UK kann somit überall verwendet werden, wo ein Obj. der OK erwartet wird. Objekte der UK substituieren Objekte der OK. Das ist immer möglich, da die UK alle Attribute und Methoden der OK erbt. Man kann deshalb einer Verweisvariablen vom Typ der OK ein Objekt der UK zuweisen: Artikel a = new Pflanze();

**Super** - führt den Konstruktor der Superklasse aus, darf nur in Konstruktoren aufgerufen werden und muss dort erste Anweisung sein

**Szenario** – ist eine einzelne Abfolge von Aktionen, d.h. einen Ablaufpfad in einem Anwendungsfall. Ein Szenario, das die Sicht der Anwender wiedergibt, fasst das zu entwickelnde System als Blackbox auf. Die innere Arbeitsweise des Systems bleibt den Anwendern verborgen, von Interesse ist allein das externe Verhalten. Das Augenmerk liebt bei dieser Perspektive auf typischen Interaktionen zwischen Benutzern und dem Anwendungssystem.

**Testen** - Programm(teil) stichprobenartig ausführen, um Unterschiede zwischen seinem erwarteten und tatsächlichen Verhalten zu finden. Ein Test kann nur Anwesenheit von Fehler zeigen nicht ihre Abwesenheit. → **Verifikation** → **dynamisches/statisches Testen**

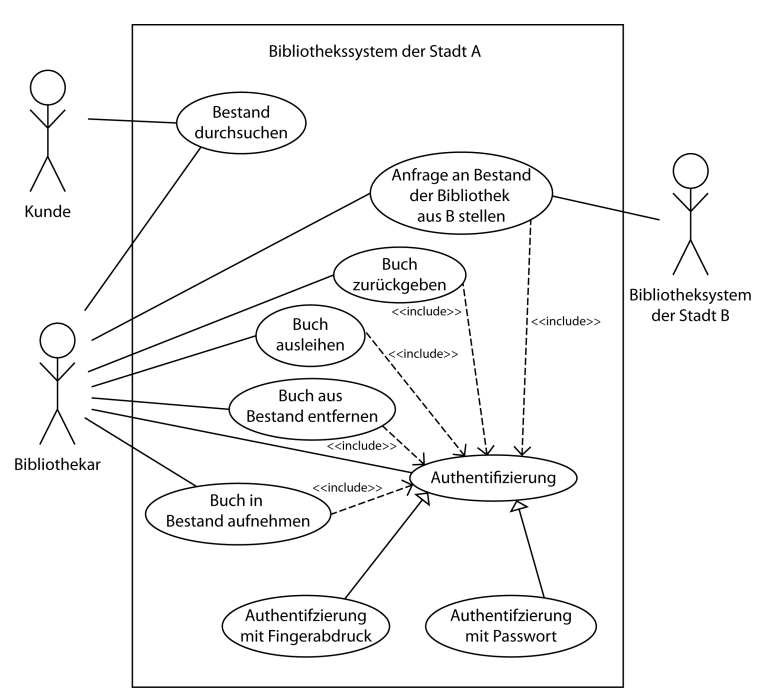
**Blackbox-Test** (funktionsorientiertes Testen, Spezialfalle der Spez. werden überprüft, Code unbekannt, nur Methodensignatur und Ergebnistyp werden betrachtet.), **Whitebox-Test** (Code wird geprüft, Programmstruktur wird analysiert und zur Konstruktion von Testfallen herangezogen), **Pos./Neg. Test** (Test mit gültigen/ungültigen Eingaben), Systemtest (Test des fertigen Programms gegen Anforderungen), **Integrationstest** (Test mehrerer Klassen, Test der Zusammenarbeit d. Programmteile), **Feldtest** (Beim Kunden) , **Komponententest** (= Klassentest, Modultest, nur eine kleine Einheit (Klasse) wird getestet), Abnahmetest )Unter Beisein des Kunden, Akzeptanz wird analysiert), **Regressionstest** (Laufende Entwicklung, Test nach jeder Änderung)

**Testfall** - Stichprobe, die Bedingungen über Eingabedaten und erwartete Ausgaben, die einem bestimmten Zweck dienen, miteinander verknüpft

**Teststrategie** - Ein systematischer und wiederholbarer Weg, um eine Klasse oder ein Programm zu testen. Beinhaltet einen Testplan, Testdaten, sowie erwartete und tatsachliche Ergebnisse. Der Testplan fasst die durchgeführten Tests zusammen, enthalt: Eingabedaten, erwartete Ergebnisse und tatsachliche Ergebnisse.

**Typanpassung, explizite (casting)** - Der Typ eines numerischen Werts wird während der Laufzeit in einen numerischen Typ ähnlichen Werts geändert. Syntax: (*typeName*) expression wobei typeName ein primitiver Typ und expression ein Ausdruck ist. Die T. wird von rechts nach links durchgeführt und liefert einen Wert vom Typ *typeName*, vorausgesetzt, die Konversion

verletzt keine Kompatibilitatsregeln. 2 Kategorien: Erweiterung (widening) und Einengung (→ narrowing). Wenn ein Cast auf einen Verweis angewendet wird, so wird lediglich der statische Typ verändert. Der dynamische Typ bleibt unverändert. So ist zum Beispiel der statische Typ des Ausdrucks ((Artikel) *new Pflanze())* der Objekttyp Artikel, der dynamische Typ hingegen Pflanze. Der dynamische Typ kann immer mit Hilfe von *instance of* überprüft werden. Der Übersetzer kann immer nur den statischen Typen von Variablen und Ausdrucken bestimmen und überprüfen, da der dynamische Typ häufig erst zur Laufzeit feststeht.



**Typanpassung, implizite** - Immer dann, wenn ein Typ gefordert ist, ist auch ein Untertyp erlaubt. Der Compiler führt eine implizite Typanpassung durch. →Typenvertraglichkeit, →Typerweiterung

**Typenverträglichkeit -** Ein Datentyp t ist vertraglich mit einem anderen Datentyp u (t<u) falls die Werte des Typs t in Werte von Typ u ausgeweitet (→ Typerweiterung) werden können.

**Typerweiterung -** Durch die Erweiterung einer Zahl des Typs t wird diese in eine Zahl des Typs u umgewandelt. Der Wertebereich von t muss eine Teilmenge des Wertebereichs von u sein.

**UML-Anwendungsfalldiagramm -** Ein UML-Anwendungsfalldiagramm beschreibt grafisch die Zusammenhänge zwischen einer Menge von Akteuren und Anwendungsfällen. Ein solches Diagramm besteht aus: dem System, dargestellt als Rechteck, wobei der Name des Systems im Rechteck steht; Anwendungsfällen, die als Ellipsen innerhalb des Systems erscheinen und die Bezeichnung des Anwendungsfalls enthalten; Akteure, die als Strichmännchen außerhalb des Systems dargestellt werden, wobei der Name des Akteurs unter das Symbol geschrieben wird; Assoziationen zwischen Akteuren und Anwendungsfällen, die durch Linien dargestellt werden;

Beziehungen zwischen Akteuren und Anwendungsfällen untereinander.

public class ListenElement {

public int wert ;

public ListenElement naechstes ;

}

public class Liste {

ListenElement erstes ;

public Liste () {

erstes = null ;

}

public void fuegeWertHinzu ( int wert ) {

ListenElement e = new ListenElement ();

e. wert = wert ;

if ( erstes == null ) {

// wenn Liste leer

erstes = e;

} else {

// wenn mind . ein Element vorhanden

ListenElement sucher = erstes ;

// letztes Element suchen

while ( sucher . naechstes != null ) {

sucher = sucher . naechstes ;

}

// neues Element anh¨angen

sucher . naechstes = e;

} }

**UML-Klassendiagramm -** Ein UML-Klassendiagramm beschreibt grafisch die Attribute, Methoden sowie Assoziationen und Generalisierungen zwischen Klassen.

**UML-Objektdiagramm -** Zur Beschreibung konkreter Situationen werden einzelne Objekte dargestellt.

**UML-Aktivität -** In UML ist eine Aktivität ein bestimmtes Verhalten, z.B. einer bestimmten Methode oder ganze Geschäftsprozesse. → UML-Aktivitätsdiagramm

**Überdeckung** → Zweigüberdeckung→ Pfadüberdeckung → Anweisungsüberdeckung

**Überladener Methodenname** → Signatur

**Überschreiben von Methoden** Eine Unterklasse *U* kann eine geerbte Methode *m()* durch Definition einer neuen Methode *m()*, die die gleiche Signatur hat wie die geerbte Methode, überschreiben (*override*). Der Ergebnistyp in der Unterklasse muss identisch oder verträglich mit dem in der Oberklasse deklarierten Ergebnistyp sein. Wird an einem Objekt der Unterklasse eine überschriebene Methode aufgerufen, so wird die in der Unterklasse implementierte Methode ausgeführt. Statische Methoden können nicht überschrieben werden. Eine Methode kann nicht überschrieben werden, wenn sie als *final* vereinbart ist.

**Übersetzungsfehler vs. Laufzeitfehler -** Fehler, die zum Zeitpunkt der Übersetzung auftreten, sind syntaktische Fehler. Bei der Ausführung können sowohl F. auftreten, die das Prog. zum Abbruch zwingen, als auch F. im Sinne von falschen Ergebnissen oder fehlerhaftem Verhalten.

**Variablen -** Eine Variable besteht aus einem Name und einem Wert.

**Variablenvereinbarung -** Eine Variablenvereinbarung oder Variablendeklaration ist eine Anweisung, die einen Name mit einem Datentyp verknüpft. Die einfachste Form einer Variablendeklaration verbindet einen Namen *name* mit einem Datentyp *type*. Jede Deklaration wird durch ein Semikolon abgeschlossen: *type name;*

**Variablennamen -** Folge von Unicodezeichen, 1. Zeichen keine Ziffer, V. darf nicht mit einem reservierten Schlüsselwort oder den Literalen *true, false, null* übereinstimmen. In jedem speziellen Umfeld darf eine Variable nur einmal vereinbart werden.

**Datenlexikon**

Zeichenkette = Buchstabe + {(Buchstabe | -)}

GanzeZahl = Ziffer + {Ziffer}

Premiumkunde = Vorname + Nachname + Anschrift + Kundennummer

Vorname = Zeichenkette

Nachname = Zeichenkette

**Vereinbarung eines Pakets -** Ein Paket (*package*) wird wie folgt eingeführt: *package paketName;* Diese Anweisung muss am Anfang einer Übersetzungseinheit vor die Klassendeklaration gesetzt werden, die zum Paket *paketName* gehören soll.

**Vererbung -** Umstand, dass eine Unterklasse von der Oberklasse Eigenschaften (Attribute) und Verhalten (Methoden) übernimmt. Mittels V. wird ausgedruckt, dass eine Klasse U ähnlich einer Klasse O ist, mit gewissen Unterschieden (neue Attribute oder neue/veränderte Methoden). Durch die V. wird zwischen Unter- und Oberklasse eine **„ist-eine-Beziehung“** erzeugt. Schlüsselwort in Java: „*extends*“, Bsp.: *class Pflanze extends Artikel* {…} → Generalisierung

**Verifikation -** Bei der V. wird im Ggs. zum Testen mit math. Verfahren bewiesen, dass der Quelltext mit dem formalen Modell oder formalen Spez. übereinstimmt. Verifizierende Verfahren können – im Ggs. zum Testen – allg. Aussagen über Korrektheit und andere Programmeigenschaften erzeugen (sind aber in der Realität schwer anwendbar).

public int berechne ( int n) {

// Basisfall

if (n == 0) { return 3; }

// rekursiver Aufruf

int g = berechne (n - 1);

//wenn ungerade %2 !=0

if (g % 2 != 0) { return 3 \* g + 1; }

else { return g /2; }

}

**Verkettete Liste -** Eine einfach verkettete List ist eine (möglicherweise leere) Menge von Knoten, wobei jeder Knoten einen Eintrag und einen Verweis auf den nächsten Knoten enthält. Der ersten Knoten der Liste wird Kopf (*head*), der letzte Schwanz (*tail*) genannt. Der Verweis des letzten Elements hat, außer bei einer zirkulären Liste, den Wert *null*.

**Verminderung der Genauigkeit -** Ganzzahlige Typerweiterungen führen niemals zu einem Informationsverlust. Die Erweiterung von ganzen Zahlen in Gleitkommazahlen kann eine Verm. d. G. bewirken. Zwar wird der Wertebereich erweitert, doch können bei der Umwandlung eines *int* oder *long* zu *float* oder *double* einige der weniger signifikanten Ziffern verloren gehen. Beispiel: float b = 100000027 ⇒ b hat den Wert 1.0E9

Implementieren Sie die Invertierung iterativ in der Java Methode

public int[] invertiere(int[] binZahl).

Dabei entspricht jeder Feldeintrag einer Ziffer einer Binärzahl.

**Verweisübergabe (call by reference)** - Wird einer Methode als Param. ein Obj. übergeben, so handelt es sich, exakt gesprochen, um eine Referenz auf dieses Obj. Sofern weitere Ref. auf dieses Obj. existieren, sind Veränderungen des Obj. an allen Ref. gleichermaßen bemerkbar. Wird dem Param. Eine andere Ref. zugewiesen, so hat dies keine Auswirkung auf den Aufrufer. Bei Java gibt es nur *call by reference*!

//Invertiere ein Array aus 000 und 1111

public int [] invertiere ( int [] binZahl ) {

// Fehlerabfrage

if ( binZahl == null || binZahl . length == 0) {

throw new IllegalArgumentException ();

}

// Ergebnisfeld erzeugen

int [] erg = new int [ binZahl . length ];

for ( int i = 0; i < binZahl . length ; i ++) {

// Fehlerabfrage

if ( binZahl [i] > 1 || binZahl [i] < 0) {

throw new IllegalArgumentException ();

}

if ( binZahl [i] == 1) { erg [i] = 0; }

else { erg [i] = 1;}

} return erg ; }

**Wahlfreier Zugriff (random access)** - Jedes beliebige Element einer Kollektion ist mit gleichem Aufwand erreichbar (Bsp. Feld)

**Warteschlage (queue)** - Immer das Element entnommen, das als erstes eingefügt wurde (FIFO = First in First Out)

**Wertübergabe (call by value)** - Wird als Parameter einer Methode ein Wert eines prim. Datentyps übergeben, so hat die weitere Verarbeitung dieses Wertes durch die Methode keinerlei Auswirkungen auf die aufrufende Stelle.

**Wertvariablen -** Der Wert, der einer Variable zugewiesen wurde, befindet sich an ihrer Speicherstelle. → Referenzvariablen

**Widening** → Typumwandlung, explizite

**Zeichen (*char*) -** Zeichen erscheinen zwischen einzelnen Anführungszeichen (`). Steuerzeichen beginnen mit dem Rückwärts-Schrägstrich (*backslash*) „\“.

**Zeichenketten in Java -** Eine Folge von Zeichen, die in Anführungsstrichen eingeschlossen sind, wie z.B. *„Der höchste Berg Javas ist der Vulkan Semeru.“* Erzeugt in Java automatisch ein Objekt der Klasse *String*, dessen Wert mit der Zeichenfolge übereinstimmt. Ein auf diese Weise erzeugtes *String*-Objekt wird *String*-Literal genannt.

**Zugriffskontrolle bei eigenständigen Klassen -** Eine Klasse ist öffentlich (*public*) zugänglich, falls in der Klassendefinition dem Schlüsselwort *class* das Schlüsselwort *public* vorangestellt ist. Eine öffentliche Klasse ist in allen Paketen sichtbar.

**Zugriffskontrolle bei Attributen -** Bei der Attributdeklaration kann dem Typen noch ein

Zugriffsmodifikation vorangestellt werden: *Zugriffsmodifikator Typ Name;*

**Zugriffskontrolle bei Methoden -** Bei der Attributdeklaration kann dem Ergebnistyp noch ein Zugriffsmodifikation vorangestellt werden: *Zugriffsmodifikator Ergebnistyp Name(Parameterliste){}*

**Zugriffskontrolle bei Konstruktoren -** Bei der Konstruktordeklaration kann dem Namen noch ein Zugriffsmodifikation vorangestellt werden: *Zugriffsmodifikator Name(Parameterliste) { }*

**Zustand eines Objekts -** duch Attrib.werte eines Obj. wird sein innerer Zustand festgelegt.

**Zuweisung -** Eine Zuweisung ist ein Ausdruck und zugleich eine Anweisung der Form: *varName = expression;* Wenn *w(expression)* den Wert des Ausdrucks *expression* bezeichnet, dann gilt unmittelbar nach der Ausführung der Zuweisung: *w(varName = expression) == w(varName) == w(expression)*.

**Zweigüberdeckung –** misst wieviele der Kanten des → Aktivitäsdiagramms ausgeführt wurden – Kanten von Start- und Endknoten zählt man nicht

**Zyklus** → Graph, → Pfad - Ein Pfad, dessen Start- und Zielknoten identisch ist. Gerichtete Graphen ohne Zyklen werden als gerichtete, azyklische Graphen bezeichnet.

public class Rechnung {

private double betrag;

private double rabatt;

Kunde rechnungsEmpfaenger; }

public static void main(String[] args) {

Rechnung r1 = new Rechnung(150.50, 30.0);

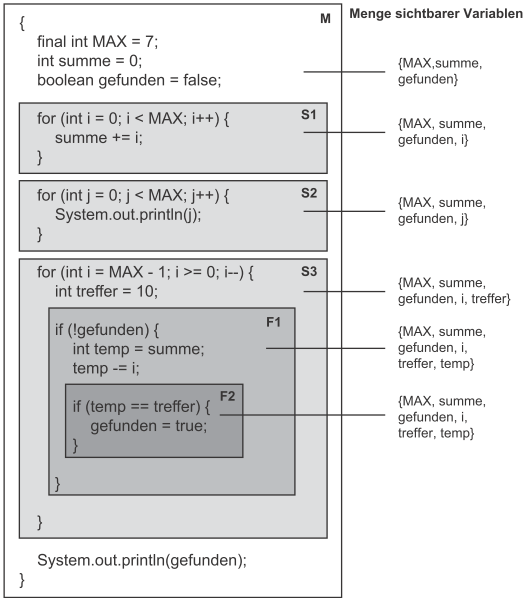
Kunde k1 = new Kunde("Denis ", "Im Heap 1 …");

r1.rechnungsEmpfaenger = k1;

r1.setBetrag(r1.getBetrag() - (r1.getBetrag() \* r1.getRabatt() / 100));

System.out.println("Rechnungsbetrag: " + r1.getBetrag());

System.out.println(r1.rechnungsEmpfaenger.kundeName + " …");}



abstract

assert

boolean

break

byte

case

catch

char

class

const

continue

default

do

double

else

enum

extends

final

finally

float

for

goto

if

implements

import

instanceof

int

interface

long

native

new

package

private

protected

public

return

short

static

strictfp

super

switch

synchronized

this

throw

throws

transient

try

void

volatile

while

// Löschen des ersten Elementes

public void loescheErstenWert () {

if ( erstes == null ) {

return ;

}

erstes = erstes . naechstes ;

}

public int findeGroesstenWert () {

// leere Liste ?

if ( erstes == null ) {

return Integer . MIN\_VALUE ;

}

ListenElement sucher ;

sucher = erstes ;

// erster Wert als Maximum

int max = erstes . wert ;

// durch Liste laufen

while ( sucher != null ) {

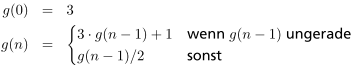
if ( sucher . wert > max ) {

max = sucher . wert ;

}

sucher = sucher . naechste ;

} return max ; }



zugriffsmodifier rückgabetyp name(Parameter) { Methodenrumpf }

**Code in Mainmethode ->Statische Methoden = Statische Variablen**

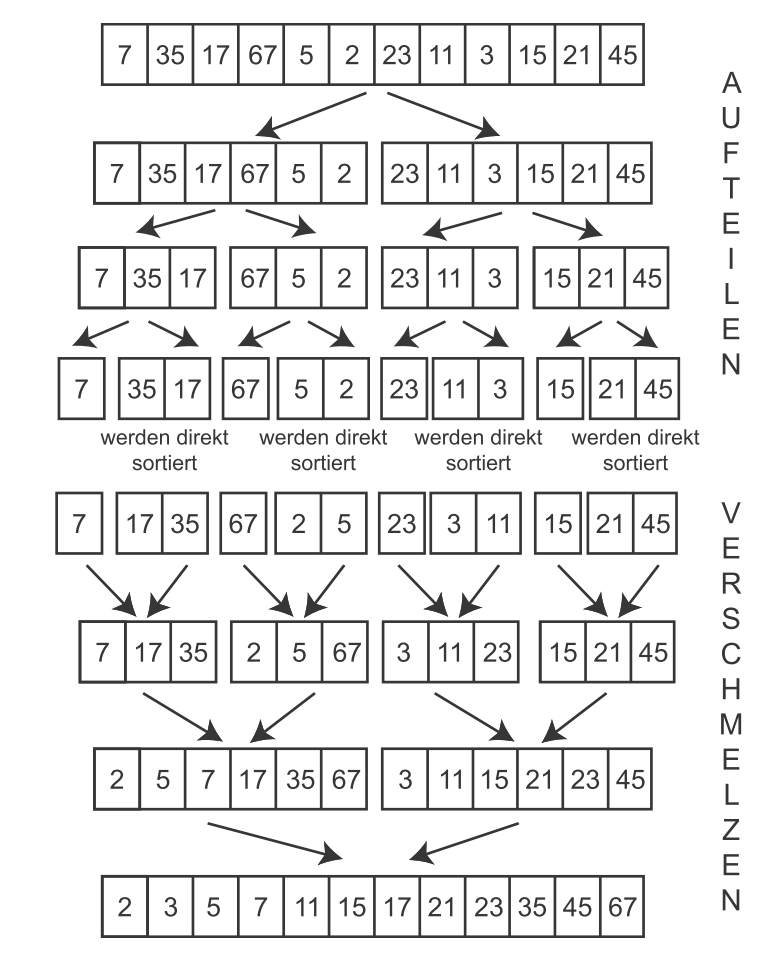
public class Z extends Y implements X implements W {}

**LOKALE Variablen initialisieren!**

**Summenzeichen = for Schleife**

**public static void main(String[] args) {System.out.println("Herzlich willkommen!");}**

**Klammern zu machen!! Strichpunkt nicht vergessen!! PUNKT statt Komma!**



Int[]array = new int [100]

Int[array.length-1] =1;

Int[] array = {1, 2};

Int array = w int {1, 2};

public int [] berechneDreieckszahlen ( int n) {

if (n < 0) { return null ; } // Fehlerabfrage

int [] d = new int [n]; // Feld anlegen

// Feld befuellen

for ( int i = 0; i < n; i ++) {

if (i == 0) { d [0] = 0;

} else {

// di mit rekursiver Formel berechnen

d[i] = d[i - 1] + i; } } }

**Artikel k = new Pflanze();**

**Pflanze m = (Pflanze) k;**

Rekursion Quadratwurzel

public double berechneWurzel ( double a, int n) {

// Basisfall

if (n == 0) { return a; }

// rekursiver Aufruf

double x = berechneWurzel (a, n - 1);

return (x + a / x) / 2; }

Strinbuilder

**StringBuilder sb = new StringBuilder("Ein Anfang");**

**sb.append(", aber jetzt geht es weiter");**

**sb.insert(30, " ein wenig");**

**sb.delete(12, 17);**

Schnittstellen

**Interface Schnittstellenname {**

**Attributdeklarationen**

**Methodendeklarationen }**

**//Implementierung**

**class Klassenname implements I [, I2, …, In] {}**

**void gebeInformationenAus() {**

**super.gebeInformationenAus(); … }**

**if (artikel instanceof BegrenztHaltbar) {}**

**istGleich = s.equals(t); //ist s gleich t? -> false**

/\*\*

 \* Berechnet n-Fakultät.

 \* Programmversion für Rekursion.

 \* @param n Zahl zur Berech von n!

 \* @return n!

 \*/

public long fakr(int n)

  {if (n<0) return -1;

    else

       if (n==0) return 1;

       else return n\*fakr(n-1); }

Object o = z (z „is a“ Object)

Object o1 = (Y) o

(o “is a” Z, o “is not a” Y)



JUnit Test-Klasse

**public class MeinTest extends junit.framework.TestCase {**

**MeinRechner ma;**

**public void setUp() { ma = new MeinRechner();**

**// wird vor jeder Testmethode ausgeführt . }**

**public void testSomething() {**

**assertEquals (“text”, 8 , ma. liefereXKoordinate ()); }**

**public void tearDown() {**

**// wird nach jeder Testmethode ausgeführt} }**

**• Testmethode muss public und void sein**

**• darf keine Parameter erwarten und**

**• der Methodenname muss mit test beginnen.**

- eine Klasse kann nur von einer anderen Klasse erben

- Eine Schnittstelle kann nur von einer anderen Schnittstelle

erben, niemals von einer anderen Klasse

**public int fiboi(int n) //Fibonacci-Zahl**

**{ int fi =1;**

**int h\_1=1; int h\_2=1;**

**if(n<=0) return fi=-1;**

**else**

**if (n==1 || n==2) return fi=1;**

**else { for (int i=3; i<n+1; i++)**

**{ fi=h\_1 + h\_2;**

**h\_2=h\_1; h\_1=fi;}**

**return fi; } } }**

Merge Sort

**public LinkedList mergeSort() {**

**int length = this.size();**

**if (length <= 1) {return this;}**

**LinkedList leftSorted = this.subList(0, length / 2).mergeSort();**

**LinkedList rightSorted = this.subList(length / 2, length).mergeSort();**

**leftSorted.mergeWith(rightSorted);**

**return leftSorted; }**

**private void mergeWith(LinkedList otherList) {**

**if (otherList.isEmpty()) { return; }**

**if (this.isEmpty()) {**

**this.head = otherList.head;**

**return; }**

**if (otherList.head.getEntry() <= this.head.getEntry()) {**

**int first = otherList.removeFirst();**

**this.mergeWith(otherList);**

**this.add(first);**

**return; }**

**int first = this.removeFirst();**

**otherList.mergeWith(this);**

**otherList.add(first);**

**this.head = otherList.head; }**

**private int removeFirst() {**

**if (this.head == null) {**

**throw new IndexOutOfBoundsException("leere Liste"); }**

**int first = this.head.getEntry();**

**this.head = this.head.getNext();**

**return first; }**

Casting

X x = new X();

Y y = new Y();

Z Z = new Z();

X Xy = new Y(); (Y “is a” X)

X xz = new Z(); (Z “is a” X)

Y yz = new Z(); (incompatible type)

Y y1 = new X(); (X “is not a” Y)

Z z1 = new X(); (X “is not a” Z)

X x1 = y (y “is a subclass of” X)

X x2 = z (z “is a subclass of” X)

Y y1 = (Y) x (x “is not a” Y)

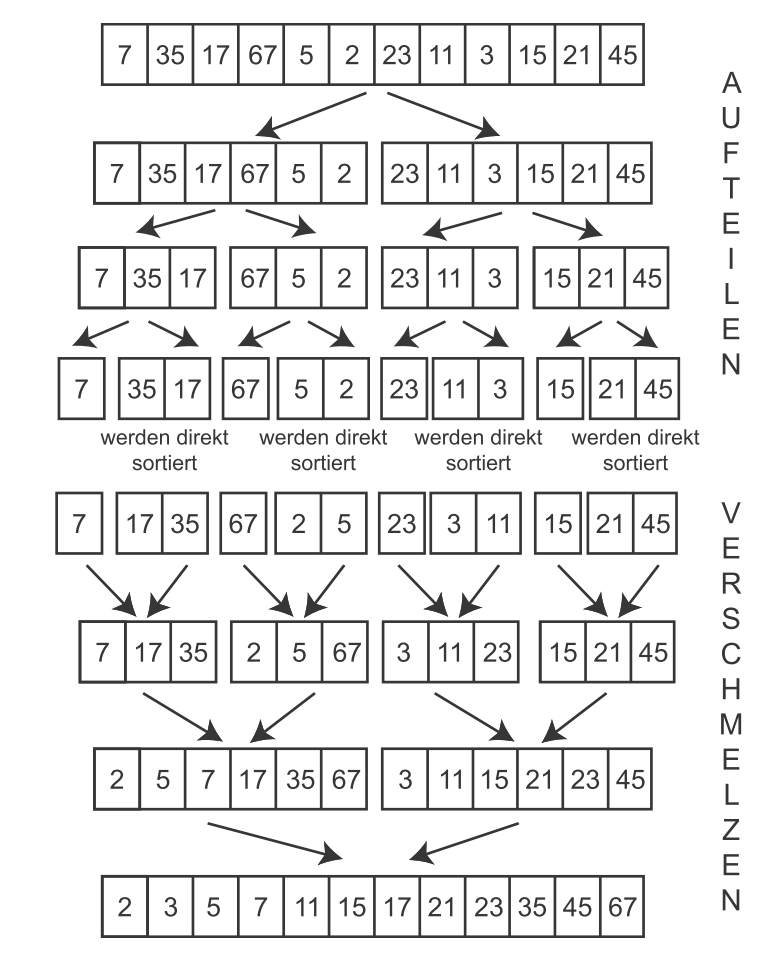
Z z1 = (Z) x (x “is not a” Z)

Y y2 = (Y) x1 (x1 “is a” Y)

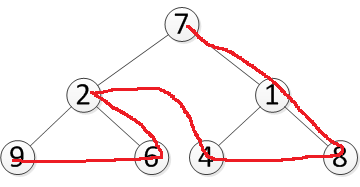
Z z2 = (Z) x2 (x2 “is a” Z)

Y y3 = (Y) z (inconvertible types)

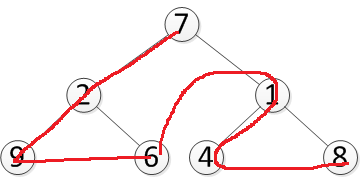
Z z3 = (Z)y = (inconvertible types)

Mergesort  


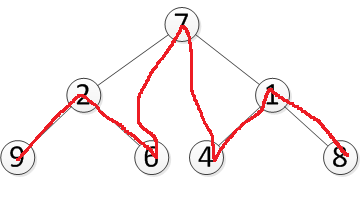
Post-order



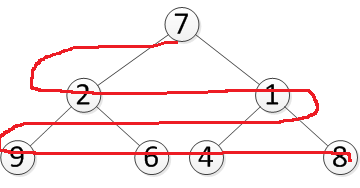
Pre-order



In-order



Level-order





Klassen-

name

Extends=

Oberklasse

Exemplarvariablen

Konstruktor (ohne return)

Methoden (mit return)

Foo(int,

double)

Huhu

11

11 7

4

dort

public class Ober{

public static int berechne(int x){

return x \* X + 2;}

public String sagHallo() {

return "Hallo!";}

public class Unter extends Ober{

return "Huhu!";}

}

Ober x = new Unter();

System.out.println(x.sagHallo());

int z = Ober.berechne(3);

System.out.println(z);

System.out.println(z + " 7");

int y = 13 / (int) 3.7;

System.out.println(y);

int a = 3 + 5 \*2;

switch(a){

case 16:

System.out.println("hier");

break;

case 12:

System.out.println("da");

break;

default:

System.out.println("dort");

}

public class A{

private int x;

public String y;

public A(int z){

x = z - 3;}

private int b(){

return 2 \* x;}

public int c() {

return b();}

}

public class Analyse {

private int z;

private void k() {

A a = new A(1.2);

A b = new A(12 + z);

int m = b.b() - 7;

b.y = "C";

b.x = m;

System.out.println("Hallo");}

public static void f(){

Analyse x = new Analyse(3);

k();

x.k();}

}

1. Wegen double, A erwartet int

2. b ist private in A

3. x ist private in A

4. Es gibt einen constructor Analyse, aber parameterlos

5. nicht statische Methode K kann nicht von statischen Kontext referenziert werden.

private static long tRekursiv(int n){

if (n == 0){

return 3;

}

if (n == 1 || n == 2 || n == 3){

return 3;

}

if (n % 3 == 0){

return tRekursiv(n/3) + 2;

}

if (n % 3 != 0) {

return 2 \* tRekursiv(n-1);

}

return 0;

}

42

public long multipliziere (long a, long b)

throws IllegalArgumentException {  
if (a < 0 | b < 0) {  
throw new IllegalArgumentException();  
}  
if (b == 0) {  
return 0;  
}  
return a + multipliziere (a, b-1);  
}

// Bei (1,2,3,4) = (1 + (1 / 2 + (1 / 3 +(1 / 4)

// In dieser Aufgabe ist die Näherungsformel für die

// Wurzel gesucht: Xn + 1 = Xn + ((a / Xn) / 2)

// Gehen wir von Wurzel a = 17 und 2 Schritten n = 2 aus

//X0 = 17 = 17

//X1 = 17 + ((17/17) / 2) = 9

//X2 = 9 + ((17/9) / 2) = 5,4444

private static double berechneWurzel(double a, int n){

// double a = Wert der ermittelt werden soll

// int n = Schritte um Nährungswert zu ermitteln

// result = Endergebnis

if (n == 0) return a;

// Wenn n = 0 ist, dann ist kein weiterer Schritt

// erforderlich und wir können a zurückgeben

// Dies wird durch die Rekursion und das "n--"

// irgendwann automatisch erreicht, an diesem

// Punkt werden wir die Rekursion dann verlassen

double result = a;

n--;

result = berechneWurzel(result, n);

//Hier ist der rekursive Aufruf der Methode

return (result + (a / result)) / 2;

//Hierbei handelt es sich um die oben angegebene

// Rechnung zur Ermittlung des Näherungswertes

}

super: Aufruf von Attributen, Methoden und Konstruktoren der Oberklasse

final: Nach einmaliger Zuweisung einer mit "final" deklarierten Variable ist keine zweite Zuweisung mehr möglich

this: Ansprechen der aktuellen Instanz innerhalb der Klassendefinition

System.out.println("TEST");

switch(w){   
 case 0:   
 mach irgendwas….;

break;   
 case 1:   
 mach irgendwas….;

break;   
 default:   
 mach irgendwas….;  
}

Literale

|  |  |  |  |
| --- | --- | --- | --- |
| *Literal* | *Bezeichnung* | *Berechnung* | *Dez.wert* |
| 2476 | Dezimalliteral | 2\*10³+4\*10²+7\*+6\* | 2476 |
| 04654 | Oktalliteral | 4\*8³+6\*8²+5\*+4\* | 2476 |
| 0x9AC | Hexadez.literal | 9\*16²+10\*+12\* | 2476 |

public interface X {}

public abstract class Y {}

public class Z extends Y implements X {}

Graph-Breitensuche



Sortierung

Insertion Sort

**void sortiereAufsteigend(int[] feld) {**

**// beginne beim zweiten Element und betrachte**

**// die Liste bis zum Index i - 1 als sortiert**

**for (int i = 1; i < feld.length; i++) {**

**// gehe solange von i nach links**

**// bis das Element an die richtige**

**// Stelle gerutscht ist**

**for (int j = i; j > 0; j--) {**

**if (feld[j - 1] > feld[j]) {**

**// wenn linkes groesser,**

**// vertausche die Elemente**

**int temp = feld[j];**

**feld[j] = feld[j - 1];**

**feld[j - 1] = temp;**

**} else {**

**// das Element ist**

**// an der richtigen Stelle**

**break;**

**}**

**}**

**}**

**}**

Bubble Sort

**void bubblesort(int[] feld) {**

**//es werden maximal feld.length - 1 Durchläufe benötigt**

**for (int i = 0; i < feld.length - 1; i++) {**

**// solange keine Vertauschungen vorgenommen werden**

**// ist das Feld sortiert**

**boolean sorted = true;**

**// Durchlaufe das Feld, in jedem Durchlauf muss**

**// ein Element weniger berücksichtigt werden**

**for (int j = 0; j < feld.length - 1 - i; j++) {**

**if (feld[j] > feld[j + 1]) {**

**// wenn linkes größer**

**// dann vertausche**

**int temp = feld[j];**

**feld[j] = feld[j + 1];**

**feld[j + 1] = temp;**

**// Feld ist nicht sortiert**

**sorted = false;**

**}**

**}**

**if (sorted) {  
 // keine Vertauschungen, Feld ist**

**// folglich vollständig sortiert**

**break;**

**}**

**}**

**}**

Quick Sort

**void quicksort(int[] feld, int start, int ende) {**

**// Basisfall: leeres Feld**

**if (ende < start) return;**

**// Basisfall: maximal 2 Elemente**

**if (ende - start <= 1) {**

**// wenn nötig die beiden Werte vertauschen**

**if (feld[start] > feld[ende]) {**

**int temp = feld[start];**

**feld[start] = feld[ende];**

**feld[ende] = temp;**

**}**

**return;**

**}**

**// Feld aufteilen**

**int grenze = aufteilen(feld, start, ende);**

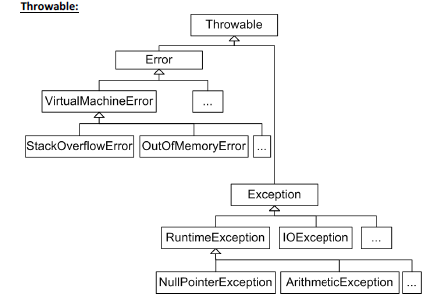
**// linken Teil (ohne Pivot) Sortieren**

**quicksort(feld, start, grenze - 1);**

**// rechten Teil (ohne Pivot) Sortieren**

**quicksort(feld, grenze + 1, ende);**

**}**



Try-Catch:

try {

// Kritische Anweisungen

} catch (Ausnahmetyp1 name1) {

// Diese Anweisungen werden

ausgeführt, wenn

// eine Ausnahme vom Typ

Ausnahmetyp1 aufgetreten ist.

// Die konkrete Ausnahme wird von der

Variablen

// name1 referenziert.

} catch (FileNotFoundException e){

System.err.println(„….“);

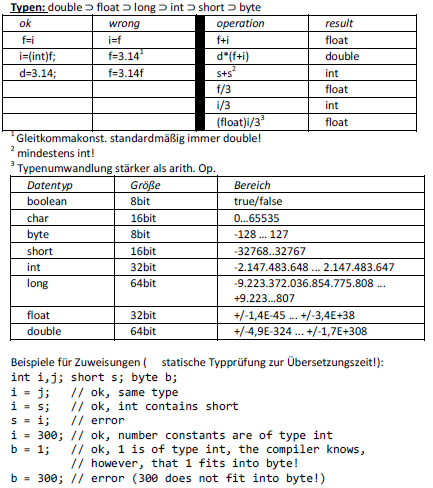
Return

} finally {

// Diese Anweisungen werden immer

ausgeführt

}



Bäume

|  |  |
| --- | --- |
| **pre-order (root,left,right)** | **7296148** |
| **post-order (left,right,root)** | **9624817** |
| **in-order (left,root,right)** | **9267418** |
| **level-order** | **7219648** |

Beispiele Literale

3.14F //float

3.14L //long

3.14D //double

24.E-13 //double

.4e0 //double

.E-45 //error

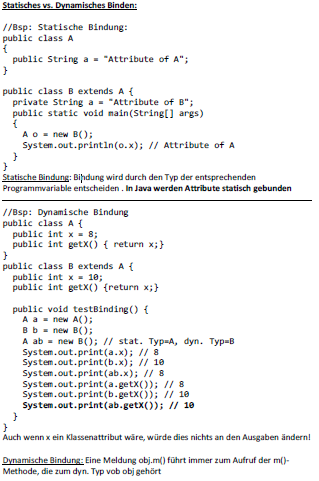
45.45 //double

.3 //double

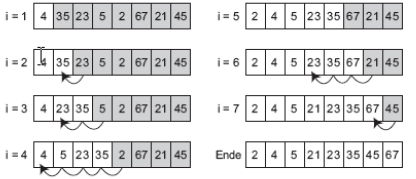
2124L //long

0x88 //int

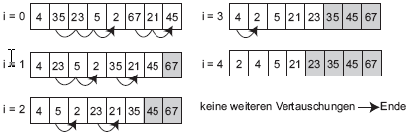
3E10 //double



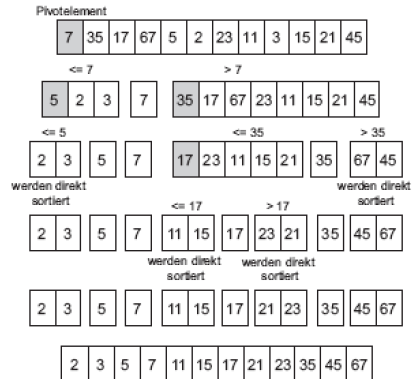
Insertion Sort:



Bubble Sort:



Quick Sort:



Rangfolge von Operatoren

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Rangfolge | | | Kategorie | Bedeutung |
| op++  op-- | | | Arithmetisch | Postinkrement  Postdekrement |
| ++op  --op  +op | | -op  ~op  !op | Arithmetisch/  Boolesch | Präinkrement, Prädekrement. Positives Vorzeichen, negatives Vorzeichen, bitweises Komplement, logische Negation |
| \* / % | | | Arithmetisch | Multiplikation, Division, Divisionsrest |
| + - | | | Arithemtisch | Addition, Subtraktion |
| << >> >>> | | | Shift | Bitweises Verschieben |
| <  > | <=  >= | | Relational | Kleiner größer, kleiner oder, gleich größer, oder gleich |
| == != | | | Gleichwertig-keit | gleich, ungleich |
| & | | | Boolesch | Boolesches/Bitweises Und |
| ^ | | | Boolesch | Logisches Exklusives Oder |
| | | | | Boolesch | Boolesches/Bitweises Oder |
| && | | | Boolesch | Logisches Und |
| || | | | Boolesch | Logisches Oder |
| ?: | | | Bedingung | Bedingte Anweisung |
| =  +=  -= | /=  %=  … | | Zuweisung | Zuweisung, Zuweisung nach Inkrement, Zuweisung nach Dekrement, usw.… |

String:

String x = “Müller”;

String y = new String(“Meier”);

char[] letters = {‘H’,’e’,’l’,’l’,’o’};

String hello = new String(letters);

String k = “Java-“ + “Kaffee” // k = “Java-Kaffee”

int i=42;

String s = “Antwort: ”+42; // s = “Antwort: 42”

int p1 = s.indexOf(‘A’); // p1=0

int p2 = s.indexOf(“wo”); // p2=3

char c = s.charAt(2); // c = ‘t’

boolean b = s.contains(“wo”); // b = true

String s = “neben”; String t = “einander”;

string u = „nebeneinander“; String w=s+t;

boolean e;

e = s.equals(t); // false

e = u.equals(s+t); // true

e = u.equals(w); // true

e = u == w; // false weil u und w sind untersch. Obj.!

int j;

j = u.compareTo(w); // 0 weil idenisch

j = t.compareTo(s); // -9 weil ‘e’ vor ‘n’

j = s.compareTo(t); // 9 weil ‘n‘ nach ‘e‘

String sub = s.substring(3,4); // sub = „wort“

StringBuilder sb1 = new StringBuilder();

StringBuilder sb2 = new StringBuilder(1000);

StringBuilder sb3 = new StringBuilder(“Hello“);

sb3.append(“ World!”);

sb3.insert(0,”X”); // insert at pos 0

sb3.delete(3,5); // delete char at pos 3 and 4

StringBuilder vn = new StringBuilder(“Heinz”);

StringBuilder nn = vorname.append(“ Müller”);

// vn und nn verweisen auf dasselbe Objekt!

Felder

**//Vereinbarung von Feldern**

**ElementTyp[] Feldname;**

**//Vereinbarung und Erzeugung von Feldern**

**ElementTyp[] Feldname = new ElementType[IntAusdruck];**

**ElementTyp[] Feldname = {v1, v2, ..., vn};**

**//Länge eines Feldes**

**Feldname.length**

**//Feldzugriff - Feldindizes beginnen immer mit 0**

**Feldname[IntAusdruck]**



JUnit assert-Methoden

|  |  |
| --- | --- |
| **assertTrue(boolescherAusdruck)** | sichert zu, dass der Ausdruck true ist |
| **assertFalse(boolescherAusdruck)** | sichert zu, dass der Ausdruck false ist |
| **assertEquals(erwarteter, tatsaechlicherWert)** | sichert zu, dass zwei Werte oder Objekte gleich sind |
| **assertEquals(erwartet, tatsaechlich, toleranz)** | sichert zu, dass zwei Gleitkommazahlen im Rahmen der angegebenen  Toleranz gleich sind: |erwartet - tatsaechlich| < toleranz |
| **assertNull(objektVerweis)** | sichert zu, dass objektVerweis null ist |
| **assertNotNull(objektVerweis)** | sichert zu, dass objektVerweis nicht null ist |
| **assertSame(objekt1, objekt2)** | sichert zu, dass zwei Objekte identisch sind |
| **assertNotSame(objekt1, objekt2)** | sichert zu, dass die beiden Objektverweise auf verschiedene Objekte  verweisen |

Baum-Begriffe



BinaryTree

**public class BinaryTreeNode {**

**private int entry;**

**private BinaryTreeNode leftChild;**

**private BinaryTreeNode rightChild;**

**public BinaryTreeNode(int e) { this.entry = e; }**

**public BinaryTreeNode(int e, BinaryTreeNode left, BinaryTreeNode right) {**

**this(e);**

**this.leftChild = left;**

**this.rightChild = right;**

**}**

**}**

**public class BinaryTree {**

**private BinaryTreeNode root;**

**public BinaryTree() {}**

**public BinaryTree(BinaryTreeNode r) {this.root = r;}**

**public void printPreorder() {printPreorder(root);}**

**private void printPreorder(BinaryTreeNode tn) {**

**// base case: empty subtree**

**if (tn == null) {return;}**

**// visit current node**

**System.out.print(tn.getEntry() + " ");**

**// visit left child**

**printPreorder(tn.getLeftChild());**

**// visit right child**

**printPreorder(tn.getRightChild());**

**}**

**public void printInorder() {printInorder(root);}**

**private void printInorder(BinaryTreeNode tn) {**

**// base case: empty subtree**

**if (tn == null) {return;}**

**printInorder(tn.getLeftChild());**

**System.out.print(tn.getEntry() + " ");**

**printInorder(tn.getRightChild());**

**}**

**public void add(int x) {**

**if (root == null) { root = new BinaryTreeNode(x);}**

**else { add(root, x);}**

**}**

**private void add(BinaryTreeNode tn, int x) {**

**// compare current entry**

**if (x < tn.getEntry()) {**

**// add in left subtree**

**if (tn.getLeftChild() == null) {**

**// base case 1: insert as left child**

**tn.setLeftChild(new BinaryTreeNode(x));**

**} else {**

**// insert recursive**

**add(tn.getLeftChild(), x);**

**}**

**} else {**

**// add in right subtree**

**if (tn.getRightChild() == null) {**

**// base case 2: insert as right child**

**tn.setRightChild(new BinaryTreeNode(x));**

**} else {**

**// insert recusive**

**add(tn.getRightChild(), x);}**

**}**

**}**

**}**

**public boolean contains(int x){return contains(root,x);}**

**private boolean contains(BinaryTreeNode tn, int x) {**

**if (tn == null) {return false; // bc1: no more nodes }**

**if (tn.getEntry() == x) { return true; // bc2: found!}**

**if (x < tn.getEntry()) {**

**return contains(tn.getLeftChild(),x); //search rec.**

**} else {**

**return contains(tn.getRightChild(),x);} //search rec.**

**}**

**}**

**}**

Java-Doc

|  |  |
| --- | --- |
| **@author** | **Autor der Klasse** |
| **@deprecated** | **Nicht mehr verwendeter Code** |
| **@param** | **Methoden-/Konstruktordefinitionen,**  **benennt und erläutert Parameter** |
| **@return** | **Art des Ergebnisses einer Methode** |
| **@see** | **Hyptertext-Verweis** |
| **@since** | **Seit welcher Versionsnummer** |
| **@throws** | **Metjode/Konstruktordefinitionen, welche Exceptions die Methode erzeugen kann** |
| **@version** | **Bei Klassen gibt Versionsnummer oder Erstellungsdatum an** |

Schleifen

**//while-Schleife**

**while (expression)**

**statement**

**//do-Schleife**

**do**

**statement**

**while (expression)**

**//for-Schleife**

**for(forInit; booleanExpr; forUpdate)**

**statement**

**//Erweiterte for-Schleife**

**for (ElementTyp varName : Feldname)**

**statement**

**//break-Anweisung**

**//Verlassen der innersten Schleife**

**do**

**if (statement)**

**break;**

**while (statement)**

**//continue-Anweisung**

**//Stoppt Ausführung der Anweisung**

**und startet nächsten ,Durchlauf**

**do**

**if (statement)**

**continue;**

**statement**

**while (statement)**

Kommentare

|  |  |
| --- | --- |
| //Kommentar | Einzeilig |
| /\* Kommentar \*/ | Block |
| /\*\* Kommentar \*/ | Block, kann für Java-Doc verwendet werden |

Datei einlesen, sortieren und ausgeben

**public StringListe einlesen(String dateiname)**

**throws IOException, DateiNichtGefundenAusnahme{**

**//File-Klasse dient zum Zugriff auf die Datei**

**File f = new File(dateiname);**

**try{**

**//StringListe für die Rückgabe erstellen**

**StringListe liste = new StringListe();**

**//Mit dem BufferedReader kann eine Datei zeilenweise eingelesen werden**

**BufferedReader br = new BufferedReader(new FileReader(f));**

**try{**

**//Erste Zeile einleisen**

**String zeile = br.readLine();**

**//Solange noch Zeilen existieren, diese einlesen**

**while(zeile != null){**

**//Aktuelle Zeile in die Liste einfügen (am Ende anhängen)**

**liste.add(zeile);**

**//Nächste Zeile einlesen**

**zeile = br.readLine();**

**}**

**}**

**finally{**

**//Reader nach dem Einlesen wieder schließen**

**br.close();**

**}**

**return liste;**

**}**

**catch(FileNotFoundException e){**

**//Abfangen der Exception und Erstellung einer eigenen**

**throw new DateiNichtGefundenAusnahme(dateiname);**

**}**

**}**

**public StringListe sortiere(StringListe unsortierteListe){**

**if (unsortierteListe == null){**

**//Wenn null übergeben wird, Ausnahme werfen**

**throw new IllegalArgumentException("null ist ein ungueltiges Argument.");**

**}**

**//Standard-Sortiermechanismus verwenden**

**Collections.sort(unsortierteListe);**

**//Die übergebene Liste ist nun sortiert und muss nur noch zurückgegeben werden**

**return unsortierteListe;**

**}**

**public void gebeAus(OutputStream out, StringListe zeilen){**

**//Die PrintWriter-Klasse dient zum Schreiben in einen Stream (Datei, Ausgabefenster,..)**

**PrintWriter pw = new PrintWriter(out);**

**//Mittels Schleife alle Einträge der Liste durchgehen und ausgeben**

**for(String s : zeilen){**

**pw.println(s);**

**}**

**pw.flush();**

**}**

**//benötigt import java.util.LinkedList;**

**public class StringListe extends LinkedList<String> {**

**public StringListe() {**

**}**

**}**

Graph-Tiefensuche



Zugriffsmodifikatoren in der UML

|  |  |
| --- | --- |
| public | + |
| private | - |
| protected | # |
| package | ~ |

Ausnahmen

**public class UngueltigerRabattAusnahme extends Exception {**

**public UngueltigerRabattAusnahme(double rabatt) {**

**// setzt die Fehlermeldung**

**super("Ein Rabatt von " + rabatt**

**+ " ist nicht zulaessig.");**

**}**

**}**

**public class Rechnung {**

**private double rabatt;**

**// ...**

**void legeRabattFest(final double neuerRabatt)**

**throws UngueltigerRabattAusnahme {**

**if(neuerRabatt < 0 || neuerRabatt > 1) {**

**throw new UngueltigerRabattAusnahme(neuerRabatt);**

**}**

**this.rabatt = neuerRabatt;**

**}**

**}**

Verkettete Listen

**public class ListNode {**

**private int entry;**

**private ListNode next;**

**public ListNode(int v) { entry = v; next = null; }**

**public ListNode(int v, ListNode nxt){entry=val;next =nxt;}**

**public void setEntry(int v) {entry = v;}**

**public void setNext(ListNode newNext) {next=newNext;}**

**public int getEntry() {return entry;}**

**public ListNode getNext() {return next;}**

**}**

**public class LinkedList {**

**private ListNode head;**

**public LinkedList() {this.head=null;}**

**public void add(int value) {**

**ListNode newNode = new ListNode(value, this.head);**

**this.head = newNode;**

**}**

**public int sizeIterative() {**

**ListNode cur = this.head; int c=0;**

**while(cur != null) { c++; cur=cur.getNext(); }**

**return c }**

**public int sizeRecursive() {return size(this.head);}**

**private int sizeRecursive (ListNode n) {**

**if (n == null) {return 0;}**

**return sizeRecursive (n.getNext())+1;n }**

**public void remove(int value) {**

**this.head = remove(this.head,value); }**

**private ListNode remove(ListNode n, int value) {**

**if (n==null) {return null;}**

**if (n.getEntry() == value) {return node.getNext();}**

**n.setNext(remove(n.getNext(),value));**

**return n; } }**

Vererbung

**class Artikel {**

**double preis**

**double lieferePreis() {**

**return this.preis; }**

**void legePreisFest(final double neuerPreis) {**

**this.preis = neuerPreis; } }**

**//Extends erstellt Unterklasse**

**class Pflanze extends Artikel {**

**void legePreisFest(final double neuerPreis) {**

**//super ruft Oberklasse auf**

**super.legePreisFest(neuerPreis);**

**//weitere Anweisungen } }**

Boolsche Operatoren

|  |  |
| --- | --- |
| Verwendung | Wird true, wenn |
| **op1 && op2** | op1 und op2 beide true sind; op2 wird  nur ausgewertet, wenn op1 zu true evaluiert |
| **op1 || op2** | entweder op1 oder op2 true ist; op2  wird nur ausgewertet, wenn op1 false ist |
| **op1 & op2** | op1 und op2 beide true sind; op1 und op2  immer ausgewertet |
| **op1 | op2** | entweder op1 oder op2 true ist; op1 und  immer ausgewertet |
| **op1 ^ op2** | op1 und op2 zu verschiedenen Wahrheitswerten  evaluieren |
| **!op** | op false ist |

Rekursion - z.B. Kettenbruch

**private double werteRekursivAus(int i) {**

**double result = 0;**

**if (werte.length == 0) return 0;**

**//Es wird erst die Methode rekursiv aufgerufen**

**//und anschließend findet die Berechnung statt**

**if (i < werte.length - 1)**

**result = werteRekursivAus(i + 1);**

**return (i == 0) ? (werte[i] + result) : (1 / (werte[i] + result));**

}

Rekursion - z.B. Quadratwurzel

**private static double berechneWurzel(double a, int n){**

**if (n == 0) return a;**

**double result = a;**

**n--;**

**result = berechneWurzel(result, n);**

**return (result + (a / result)) / 2;**

**}**