Sew Mitschrift

Java Exceptions

Exception = Ausnahme

(Meistens) ein schwerwiegender Fehler, der ohne Behandlung zum Absturz oder unerwünschten Programmzuständen führt

Es gibt aber auch hier Ausnahmen wo es sich keinen Fehler, sondern eine Ausnahme

Treten immer zur Laufzeit (bei Ausführung) auf

An der Fehlerstelle wird eine Exeption „geworfen“

Bsp:

* Division durch 0
* Verletzung von Arraygrenzen
* Fehlende Ressourcen -> IOExeption
* …..

Arten von Exceptions

Geprüfte Exceptions

* Vom Compiler erkannt, muss behandelt werden
* Z.B. IOException bei File
* Grund liegt in Umgebung, die nicht oder schwer kontrollierbar ist

Ungeprüfte Exeptions

* Können behandelt werden müssen aber nicht
* Z.B. ArithmeticException bei division 0
* idR auf Grund eines Programmerfehler sind also vermeidbar

Exception – Handlung

try………………………………..Ausprobieren

catch…………………………….Auffangen

throw……………………………Auslösen

finally……………………………immer ausführen

|  |
| --- |
| **try**{  *//Hier können Excepions auftretten* }**catch** (Exception e){  *//Hier werden sie abgearbeitet*  *//Es können mehrere catch-Blöcke hintereinaner stehen, um unterschiedliche Exeptions abzufangen* }**finally**{  *//Wird immer durchlaufen, ist allerdings optional* } |

Throw <> try catch

Ausnahmenbehandlung

Files Schreiben und Lesen

<files.pdf>

Methoden

* Methodenname
* Parameter
* Rückgabewerte
* Rumpf/Körper
* Deklaration
* Aufruf

Deklaration = Ankündigung, Vereinbarung

Methodendeklaration

Was braucht die Methode, um ihre Aufgabe zu können ?

Was ist das Ergebnis der Methode ?

Methoden dürfen in Java NICHT geschaltet werden (keine Methoden innerhalb von Methoden deklarieren)

Konvention : Methodenname beginnen mit Kleinbuchstaben und werden in CamelCase geschrieben

Methodenaufruf

Was muss ich in der Methode übergeben?

Was bekomme ich von der Methode zurück?

Wo gehört der Methodenaufruf hin?

Parameter werden OHNE Datentyp angegeben !!!

Methoden

Zusammenfassung

* Deklaration <> Aufruf
* Parameter: Datentyp, Anzahl und Reihenfolge
* Rückgabetyp (Deklaration, return, Aufruf)
* Methodenaufruf an der richtigen Stelle

Bisher: Strukturierte Programmierung:

Beschreibt die Programmsteuerung

Besteht aus 3 Grundbausteinen

* Anweisungen
* Verweigungen
* Schleifen

Jetzt: Objektorientierung

Entstand aus dem Verlangen, die „echte“ Welt besser in Softwareentwicklung abbilden zu können

Eigenschaften (Daten) und Operationen (Methoden) werden zusammengefasst

Vorteile:

Bessere Strukturierung des Codes

z.B. durch Einteilung von Teilaufgaben in Klassen

Kapselung von Daten

z.B. was darf von außen geändert werden?

Bessere Wiederverwertbarkait

z.B. können Klassen, die Teilaufgaben behandeln, wiederverwendet werden

Konzepte der OOP

* Datenkapselung
* Vererbung
* Polymorphie

Datenkapselung:

Encapsulation, information hiding

Auf die Eigenschaften eines Objekts kann nur über Methoden zugegriffen werden, der direkte Zugriff ist nicht möglich

Vererbung

Inheritance

Klassen können von anderen Klassen abgeleitet werden und erben damit alle Eigenschaften und Methoden der Oberklasse (Superklasse)

Polymorphie

„Mehrgestalt“, Vielgestaltigkeit

Objekte verschiedener Klasse reagieren auf den gleichen Methodenaufruf unterschiedlich

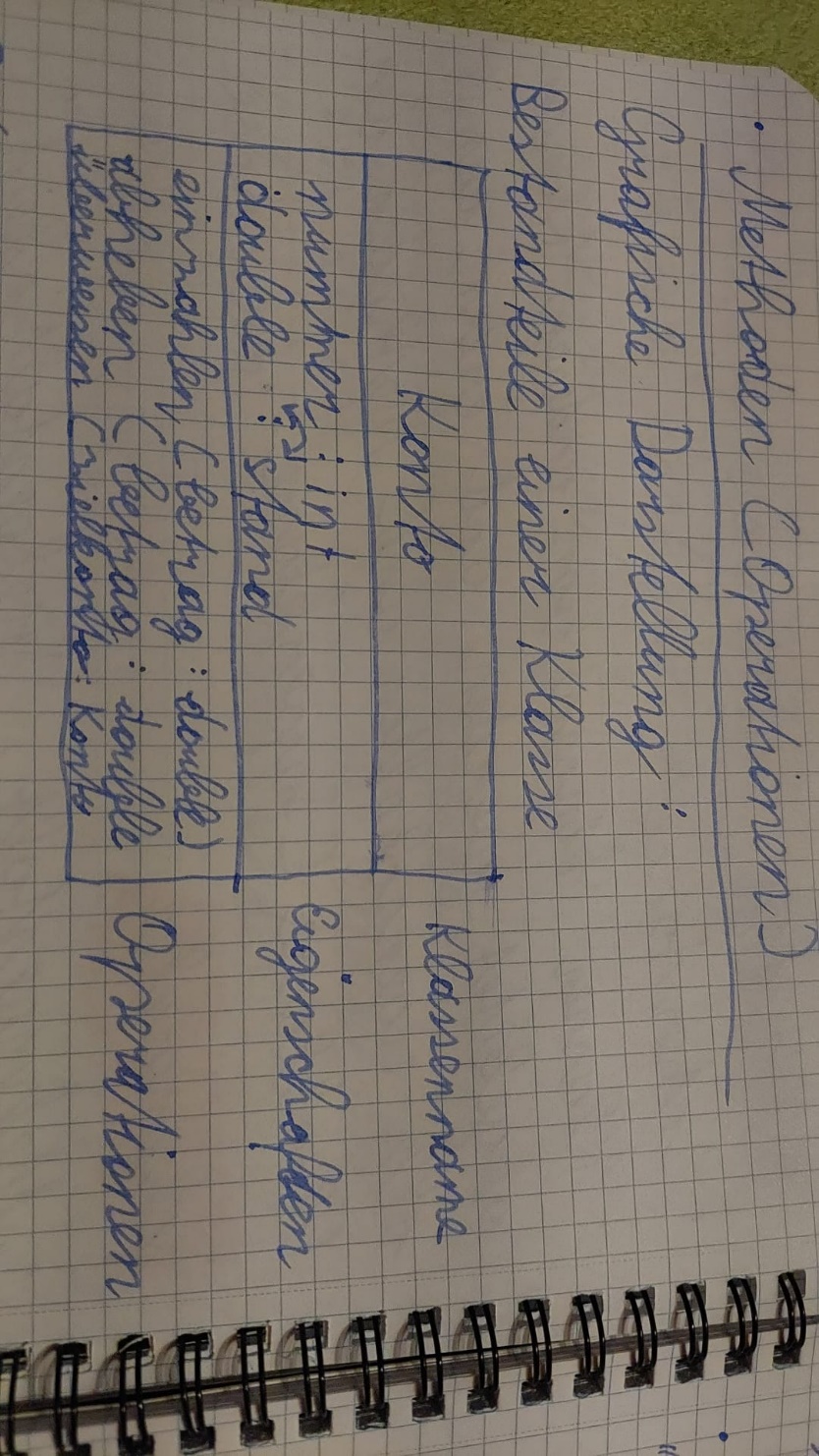
Klassen

Bauplan für gleichartige Objekte

Beschreib die Strukturen und Verhalten

Besteht aus:

* Eigenschaften (Daten, Attribute) in Variablen abgelegt



Methoden (Operationen)

Grafische Darstellung:

Code:

Class BankAccount{  
 **int** number; *//Kontonummer*  
 **double** balance; *//Kontostand*  
**void** (**double** amount){  
 balance = balance + amount;  
 }  
}

Klassen <> Objekt

Klasse ~ “Bauplan”, Schablone zur “Konstruktion” von Objektion

Objekte = Instanz einer Klasse, welches zur Laufzeit erzeugt wird

Klasse ist etwas abstraktes

Objekt ist etwas konkretes

Ein Bild, das Text enthält.

Automatisch generierte Beschreibung

Herbert-Konto  
nummer=1234  
stand= 1000,00

Maja-Konto  
nummer=2711  
stand = 20700,00

Konvention: Klassennamen sind Hauptwörter und beginnen mit einem Großbuchstaben (z.B. Account, Circle, User, ….)

Variablen

Die Daten jedes Objektes sind in den Variablen der Klasse abgelegt

Variablen können in jedem Objekt (=jeder Instanz) anders belegt sein und werden daher als Instanzvariablen bezeichnet

Methoden

Beschreiben das Verhalten der Klasse

Können den Zustand des Objektes verändern oder Informationen über den Zustand des Objekts zurückzuliefern

Konvention: Methoden sind Verben und beginnen mit einem Kleinbuchstaben (z.B. getBalance(), deposit(), setColor();….)

Objekt erzeugen

instantiieren

Class BankAccount {…}  
*// …*  
BankAcount myAccount = **new** BankAccount( );  
*//..*

Konstruktor

Spezielle „Methode“ zur Initialisierung eines neu erzeugten Objekts (Anfangszustand herstellen)

Name des Konstruktors = Name der Klasse

Beginnt mit einem Großbuchstaben

Liefert nie ein Ergebnis zurück, es wird auch kein Ergebnis angegeben (auch nicht void)

Parameterliste möglich

Solange kein Konstruktor implementiert (programmiert) ist, verwendet Java einen Default-Konstruktor

**class** BankAcount {  
**private** **int** number;  
**public** BankAccount (**int** accountNumber) {  
number = accountNumber;  
}  
}  
*//…*  
BankAccount myAccount = **new** BankAccount(1000);  
*//…*

Zugriffsrechte

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Zugriffsmodifier | In gleicher Klasse | In gleichem Package | In Unterklasse (Vererbung) | Global |
| privat | J | N | N | N |
| default | J | J | N | N |
| protected | J | J | J | N |
| public | J | J | J | J |

Kommunikation zwischen Objekten

Attribute und Methoden müssen adressierbar sein

Punktnotation zum Trennen von Objekt und Attribut/Methode

myAccount.depoist(100);

getter:

Bsp.:

public static getOwner(){

return owner;

}

Setter:

Bsp.:

public void setOwner(String owner){

this.owner=owner;

}

this

Problemstellung

Class BankAccount {  
**private** **int** number;  
 **public** BankAccount(**int** number){  
 **this**.number = number;  
 }  
}

This bedeutet in etwa „diese Instanz“

Die this-Referenz löst das Problem, wenn Parameter oder lokale Variablen Objektvariablen (Instanzvariablen) verdecken.

Häufig in Konstruktoren und in Settern („Setter“-Methoden) zu finden.

**public** **void** setOwner(String owner){  
 **this**.owner = owner;  
}

static

Die Modifier **static** dient dazu, Methoden und Variablen der Klasse als Ganzes zuzuordnen, anstatt den einzelnen Instanzen

**static**-Komponenten sind nur einmal pro Klasse vorhanden sind

**static**-Methoden können aufgerufen werden, ohne dass (mit **new**) eine Instanz einer Klasse erzeugt wird

**static**-Methoden einer Klasse haben nur Zugriff auf **static**-Komponenten einer Klasse

Zugriff über **Klassenname.Komponentenname**, z.B. **BankAccount.nextAccountNumber**, oder **BankAccount.getCounrtyCode()**, weil statische Komponenten der Klasse (und nicht der Instanz) gehören.

Man spricht auch von Klassenvariablen, bzw. Klassenmethoden.

**Unified Modeling Language**

UML Klassendiagramm

Klassen (Klassenname)

-attribut0: type (statisches Attribut)

-attribut1: type (Attribute)

+attribut2: type (Datentyp)

+operation0 (type): type

#operation1 (type, type): type

(Zugriffsmodifier) -operation2 (): type

~operation3 (): type (Rückgabe-Datentyp)

Sichtbarkeit

Zugriffsmodifier

+ public

- private

# protected

~ package

Bei static wird es unterstrichen

Klassendiagramm

BankAccount

BankAccount (Klassennamen)

-number: int

-balance: double

+BankAccount(int)

+getBalance (): double

+ deposit(double): void

+withdraw(double): void

+transfer(BankAccount, double): void

Überladen von Methoden/Konstruktoren

Mindestens 2 Methoden (oder Konstruktoren) einer Klasse haben denselben Methodennamen

->müssen unterschiedliche Parameter haben, damit sie von Java unterschieden werden können

Parameterliste ist unterschiedlich, wenn

* Anzahl der Parameter verschieden
* Datentypen der Parameter verschieden
  + Nur Datentypen der Parameter sind relevant, nicht ihre Namen
  + Rückgabe der Methoden ist hier irrelevant

int add(int a, int b) 🡪 In derselben Klasse erlaubt?

flout add(int a, int b)

int add (int x, int y)

flout add(int a, flout b)

int add (int a, int b, int c)

Erzeugen eines Objekts aus einem String

* objekt.toString() 🡪 String
* Klasse.valueOf(String) 🡪 Objekt

Erzeugen eines Objektes aus einem String

* new – Aufruf ist in der Methode valueOf() gekapselt

Aufruf von valueOf() auf der Klasse

* ist eine statische Methode (static Keyword)

z.B.:

objekt[i]= Klasse.**valueOf(….)**;

String Methoden

Bereits Bekannte Methoden

Methoden, die auf Instanzen der Klasse String aufgerufen werden können:

* chartAt()
* length()
* equels()
* toLowerCase()
* toUpperCase()

In valueOf() Implementierung verwendet

* indexOf()
* lastIndexOf()
* substring()

IndexOf()

Liefert den Index des ersten Auftretens des angegebenen Zeichens oder Strings ab einer bestimmten Position.

* int indexOf(int ch)
* int indexOf(int ch, int fromIndex)
* int indexOf(String substring)
* int indexOf(String substring, int fromIndex)

(csv-File : comma seperated value)

int a =5;

int b = a;

a=3;

Date date ….

Date dz = date

Referenztyp <> elementarer Typ

|  |  |
| --- | --- |
| Elementarer Datentyp | Referenztyp |
| Ist direkt in Java definiert | Ist nicht direkt in Java definiert |
| Datentypen beginnen mit Kleinbuschstaben | Datentypen beginnen mit Großbuchstaben |
| Die Größe von elementaren Datentypen ist Typabhängig | Alle Referenztypen haben die gleiche (Speicher) Größe |
| Auf einem Datentypen kann keine Methode aufgerufen werden | Kann zum Aufruf von Methoden verwendet werden |
| Haben immer einen Wert (ist nie **null**) | Kann **null** sein |
| Bsp. Int boolean, char, ….. | Bsp. String, Arrays, Klassen, …. |
| Wird eine Variable kopiert, wird ihr **Wert** kopiert  Eine Änderung der Original-Variablen hat keine Auswirkung auf ihre Kopie (und umgekehrt) | Wird eine Variable kopiert, wird die **Referenz** auf das eigentliche Objekt kopiert, das heißt, dass beide Variablen auf das selbe Objekt verweisen  Eine Änderung bei einer der beiden Variablen hat Auswirkungen auf die andere Referenz |

Vergleich von Objekten

* == ist bei Referenztypen problematisch
* == vergleicht die Referenzen (Speicheradressen) von Objekten und nicht deren Eigenschaften
* == liefert also false, selbst wenn verglichenen Objekte völlig ident belegte Attribute besitzen

Referenz

Wird eine Variable kopiert, wird die Referenz auf das eigentliche Objekt kopiert, dass beide Variablen auf das selbe Objekt verweisen.

Eine Änderung bei einer der beiden Variablen hat Auswirkung auf die andere Referenz

BankAccount ba1 = **new** BankAccount(„John“, ”Oliver”);  
BankAccount ba2 = ba1; *// Referenz auf das selbe Objekt*  
ba1.deposit(5000.0);  
ba1.getBalance(); *// 5000*  
ba2.getBalance(); *//5000*

equals()

* Wird implementiert, um Objekte zu vergleichen
* Vergleicht alle Attribute

**public** **boolean** equals(Object o) {  
 **return** **this**.attr1 == o.attr1 && **this**.attr2 == o.attr2;  
}

**public** **boolean** equals(Object o) {  
 **if** (**this** ==o) { **return** **true**; }  
 **return** **this**.attr1 == o.attr1 && **this**.attr2 == o.attr2;  
}

String s1 = “abc”;

String s2 = “def”;

s1.equals(s2);

**instanceof** : Schaut ob es eine Instanze von der Klasse ist

Überprüft die Instanzen ob sie gleich sind.

Task o = (Task) other; *// Type cast*

**compareTo()**

Zwei Objekte **x** und **y** müssen vergleichbar sein

Ein Aufruf von **x.compareTo(y)** muss einen negativen Wert, den Wert 0 oder einen positiven Wert liefern, je nachdem, ob **x** kleiner als **y**, **x** gleich **y** oder **x** größer als **y** ist.

**compareTo()** sollte konsistent zu **equals()** implementiert werden:

**x.compareTo(y)** == 0 liefert den gleichen Wert wie **x.equals(y)**

* .Nur im Zusammenhang mit: **implements Comparable <KLASSE>**

**public** **class** Book **implements** Comparable<Book> {  
@override  
Public **int** compareTo(Book other) {…}  
}

**Vergleichsmethoden**  
**am Beispiel String**

* **equals( )** überprüft die Inhaltliche Gleichheit
* **eqaulsIgnoreCase( )** wie equals ( ) allerdings wird die Groß-/Kleinschreibung ignoriert
* **compareTo( )** lexikalischer Vergleich. Prüft, ob die Inhalte der Zeichenketten kleiner oder größer als eine andere ist.  
  Das erste Zeichen von s1 wird mit dem ersten Zeichen von s2. Ist es kleiner wird mit dem ersten Zeichen von s2. Ist es kleiner , wird eine negative Zahl zurückgegeben, ist es größer, eine positive zahl. Sind die beiden ersten Zeichen gleich, wird das nächste Zeichen untersucht.

Lexikalisch: die Untersuchung von isolierten Wörtern ‚  
ohne Berücksichtigung des Textzusammenhangs betreffend

**clone( )**

* liefert eine „Kopie“ (einen „Klon“) des Objekts
* zur Erinnerung: **Date d2 =d1**; erzeugt nur eine neue Referenz, keine Kopie!
* Für Kopie: **d2 = d1.clone( )**

**public** Object clone( ) {  
**return** **new** Object (attr1, attr2, attr3);  
}

**hashCode( )**

* Liefert einen ganzzahligen Wert (“Hashcode“), der z.B.: für die Speicherung in „Hash-Tabellen“ benötigt wird
* Sollte immer gemeinsam mit equals( ) implementiert werden
* Sind zwei objekte gleich müssen sie auch den gleichen Hashcode liefern (vgl. equals muss true liefern, wenn Objekte gleich sind)

**public** **int** hashCode( ) {  
 **return** Objects.hash(attr1, attr2, attr3);  
}

**Basisdatentypen:**

**int, long, float, char**

**Referenzen Arrays, String, Klaseen (groß-geschrieben)**

Gefährlich

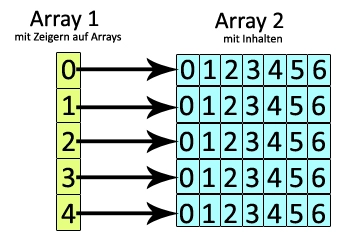
**Mehrdimensionale Arrays**

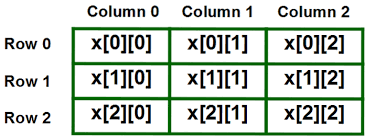
Mehrdimensionale Arrays werden als geschachtelte Arrays angelegt:  
Arrays von Arrays usw.

int [] [] m = new int [2] [3] ;

// erzeugt eine 2x2 Matrix: 2 Zeilen und 3 Spalten

m[0] und m[1] verweisen jeweils auf ein Array mit 3 Elementen

Zugriff auf einzelne Elemente z.B.: m[1] [0]



**final** **int** ROWS = 10;  
**final** **int** COLS = 20;  
**int** [] [] table = **new** **int** [ROWS] [COLS];  
**for** (**int** row = 0; row < table.length; row++){  
 **for** (**int** col = 0; col < table[row].length; col++){  
*// ….*  
}  
}

Schriftarten

Diktengleiche Schriften = Monospace Schriften

|i|m|o| Courier, Consdas

Wrapper-Klassen🥙

Zu jedem elementaren Datentyp gibt eine sogenannte Wrapper-Klassen, deren Objekte Werte dieses Datentyps speichern

🡪„einfache“ Werte können somit als Objekte angesprochen werden

Objekte von Wrapper-Klassen sind nach ihrer Erzeugung nicht mehr veränderbar

Elementare Datentypen sind effizienter als Wrapper-Klassen

„Wrapper“ ~ „Hülle“🥙

|  |  |
| --- | --- |
| Elementarer Datentyp | Wrapper-Klassen |
| Byte | Byte |
| short | Short |
| Int | Integer |
| Long | Long |
| Float | Float |
| Double | Double |
| Boolean | Boolean |
| Char | Charakter |

Elementarer Datentype 🡪 Wrapper-Objekt

Über den Konstruktor

z.B. Integer intObj = new Integer(244);

oder

Zu jeder Wrapper-Klassen gibt es die Klassenmethode valueOf(), mit der eine Instanz auf Basis eines Wertes von einfachen Datentyp erzeugt werden kannn:

z.B. Integer intObj = Integer.valueOf(244);

Wrapper-Objekt 🡪 elementarer Datentyp

Über Methode …value():

* z.B. int number = intObj.intValue()
* z.B. float balance = budget.floatValue()
* etc.

In elementaren Datentyp: Über parse…() Methode, z.B.:

int myInt = Integer.parseInt(„234“);

float myfloat = Float.parseFloat(„12.24“);

In Wrapper-Objekt: Über valueOf() Methoden z.B.:

Integer myIntObj = Integer

Auto-Boxing

Compiler kann das Umwandeln auch automatisch erledigen („Auto-Boxing“/“Unboxing“):

integer intObj = 244; // autoboxing Compiler macht daraus: Integer intObj = Integer.valueOf(244)

int i = someINtegerObj; // unboxing

Nummerische Konstaten (Auswahl)

static final typ MIN\_VALUE

static final typ MAX\_VALUE

z.B. Integer.MIN\_VALUE

import Statement

Vieles von der Funktionalität von Java ist nicht direkt verfügbar, weil man sie nicht in jedem Programm braucht, sondern in externen Paketen zu finden

Pakete sind zumeist nach Themen zusammengefasst und in einer hierarchischen Struktur (wie Verzeichnisse) abgelegt.

import

Um auf die Funktionalität eines Paketes zugreifen zu können, muss man Java mitteilen, in welchem Paket zu suchen ist.

z.B.: **java.nio.file.Paths.get() /**

**java.nio.file.Files.newBufferedWriter( )**

Oder: Ein gesamtes Paket importieren

z.B.: **import java.nio.file.Files;**

Wenn ein Paket importiert wird, steht sein gesamter Inhalt (die Funktionalität) zur Verfügung

**import java.util.Scanner;**

**import java.nio.file.Paths;**

**import java.nio.file.Files;**

ODER mit Wildcard (~ Platzhalter)

**import java.nio.files.\*;**

Aber Vorsicht: \* gilt nur für eine Ebene

import java.nio.\* macht nicht den Inhalt von **java.nio.files** verfügbar

**Plain Old Java Object: POJO**

(wörtlich: “ganz normales Java Objekt“)

„Datenhaltung“

Mehrere logisch zusammenhängende Daten können unter einen gemeinsamen Namen (Typ) behandelt werden

Eine Datenhaltungsklasse besteht nur aus:

**Konstruktor**

**Atributen** und **Accessor-Methoden** (Getter- & Setter-Methoden)

Werden z.B.: auch benutzt, wenn mehrere Daten von einer Methode zurück geliefert werden sollen

* als Return-Typ oder als Call-by-reference

es gelten keine speziellen Konventionen

sogar Atribute dürfen public sein

dienen dazu, die Lesbarkeit und Wiederverwendbarkeit von Code zu erhören

Beispiele:

Date (int year, int month, int day)

Person (String firstname, String lastName, Date birthday)

….

**StringBuilder**

Objekte der Klasse String sind nicht veränderbar (auch, wenn es so aussieht)

**String s = „abc“;**

**s = s + „def“;**

ein neues Objekt wird angelegt, das alte freigeben … sehr aufwendig

* Alternative: StringBuilder – Objekte sind veränderbar („mutable“)

Es gibt eigene Methoden zum Verändern des Strings

**String s = „abc“;**

**s = s + „def“;**

**StringBuilder sb = new StringBuilder(“abc”);**

**sb.append(“def”);**

**.append(), .insert(), .replace(), .delete(), .deleteCharAt(), …**

**enum**

(~aufzählung)

Alle möglichen Werte sind bereits zur Compile-Timer bekannt

Wochentage

Größen ( Small, Medium, Large, Extra Large)

Schulnoten

Zustände

…

Aufzählung mit Integer-Konstanten ist nicht typsicher

* Stattdessen ENUMs

// Syntax: enum Bezeichnung

// Konstanten wie üblich in all-caps (GROSSBUCHSTABEN)

**enum** Size {  
 SMALL, MEDIUM, LARGE, EXTRA\_LARGE  
}

// Variablen werden wie mit anderen Datentypen angelegt:

Size pizzaSize;

// Zugriff auf Werte über Punktnotation:

Size pizzasize = Size.LARGE;

Achtung: Reihenfolge der Werteliste ist relevant

Methode ordinal() liefert Positionsnummer einer Konstante: 0,1,…  
pizzaSize.ordinal()

toString() liefert Namen einer Konstante: SMALL, MEDIUM, ….  
pizzaSize.toString()

Klassenmethode values() liefert ein Array mit allen Aufzählungskonstanten  
Size.values()

enum-Konstanten können als case-Konstanten in switch-Anweisungen verwendet werden

**switch** (pizzaSize) {  
**case** SMALL:  
*//…..*  
**case** MEDIUM:  
*// …..*  
**case** LARGE:  
*// …*  
*// etc.*

Jede enum-Aufzählung ist automatisch static

Jeder Wert (SMALL, MEDIUM, …) einer enum-Aufzählung ist vom Typ der Aufzählung selbst: Size.LARGE ist vom Typ Size

Jeder enum Wert kommt zur Laufzeit nur ein einziges mal vor. Die Gleichheit von enum-Werten kann mit == überprüft werden

Enum-Aufzählungen können Teil einer Klasse oder global

Test Driven Develoment

TDD

**Unit Test**

~Komponententests

Kleine Einheiten der Software werden getestet (Methoden/Klassen)

Unit Test sollten Teil des gesamten Entwicklungsprozesses sein und immer wieder ausgeführt werden

* Auch bei späteren Änderungen an der Software muss sichergestellt werden, dass die Tests noch erfolgreich durchlaufen und die Software wie erwartet funktioniert!

Unit Tests werden daher in der Regel automatisiert, um ihre Ausführung schnell und einfach zu machen.

Unit ~ Einheit oder Komponente

**Code Coverage**

Möglichst viel der geschriebenen Software sollte von Tests abgedeckt werde

z.B. alle Methoden, alle „Verzweigungen“ (if/else, etc.)

🡪 „Code coverage“ („Code-Abdeckung“)

2 Verschiedene Teststrategien

White Box Testing

* Ersteller der Tests kennt den Source Code

Black Box Testing

* Tester kennt den Source Code nicht und hat keinen Zugriff darauf
* Testen auf Basis der Anforderungen

**Javadoc📖**

*// Zeilenkommentar*  
  
*/\*  
\*Blockkommentar  
\*/*  
  
*/\*\*  
\* Javadoc  
\*/*

Javadoc sind spezielle Blockkommentare, die mit /\*\* beginnen

Aus Javadoc kann HTML erzeugt werden, bzw. kann HTML-Elemente enthalten

Javadoc für die Standardklassen sind z.B im Internet abrufbar

Javadoc wird direkt über dem zu dokumentierenden Element (Klasse, Methode oder Attribut einer Klasse) geschrieben

Üblicherweise wird Javadoc bei Komponenten mit private Zugriffsmodifier nicht verwendet

Klasse/ Allgemein Methoden

@author @param

@version @return

@see @throws