Programmkonstruktion - Zusammenfassung für den Nachtragstest WS2015

Barbara Elias, 1028094

February 13, 2016

Contents

1 Datentypen und wie sie sich darstellen können

short
byte
char
int
long
float
double
String

Deklaration: int x;
Definition: Deklaration + Initialisierung
Initialisierung: x = 3;

Die Definition einer Variablen ist also die Deklaration und die Initialisierung.

2 shift in Java

Bei einem "»" so oft mit 2 multiplizieren, wie rechts vom Shiftoperator steht.
Bei "«" so oft mit 2 multiplizieren, wie rechts vom Shiftoperator steht.

3 Ascii-Tabelle

Hex	Dec	Char		Hex	Dec	Char	Hex	Dec	Char	Hex	Dec	Char
0x00	0	NULL	null	0x20	32	Space	0x40	64	@	0x60	96	`
0x01	1	SOH	Start of heading	0x21	33	!	0x41	65	A	0x61	97	a
0x02	2	STX	Start of text	0x22	34	"	0x42	66	B	0x62	98	b
0x03	3	ETX	End of text	0x23	35	#	0x43	67	C	0x63	99	c
0x04	4	EOT	End of transmission	0x24	36	$	0x44	68	D	0x64	100	d
0x05	5	ENQ	Enquiry	0x25	37	%	0x45	69	E	0x65	101	e
0x06	6	ACK	Acknowledge	0x26	38	&	0x46	70	F	0x66	102	f
0x07	7	BELL	Bell	0x27	39	'	0x47	71	G	0x67	103	g
0x08	8	BS	Backspace	0x28	40	(	0x48	72	H	0x68	104	h
0x09	9	TAB	Horizontal tab	0x29	41	)	0x49	73	I	0x69	105	i
0x0A	10	LF	New line	0x2A	42	*	0x4A	74	J	0x6A	106	j
0x0B	11	VT	Vertical tab	0x2B	43	+	0x4B	75	K	0x6B	107	k
0x0C	12	FF	Form Feed	0x2C	44	,	0x4C	76	L	0x6C	108	l
0x0D	13	CR	Carriage return	0x2D	45	-	0x4D	77	M	0x6D	109	m
0x0E	14	SO	Shift out	0x2E	46	.	0x4E	78	N	0x6E	110	n
0x0F	15	SI	Shift in	0x2F	47	/	0x4F	79	O	0x6F	111	o
0x10	16	DLE	Data link escape	0x30	48	0	0x50	80	P	0x70	112	p
0x11	17	DC1	Device control 1	0x31	49	1	0x51	81	Q	0x71	113	q
0x12	18	DC2	Device control 2	0x32	50	2	0x52	82	R	0x72	114	r
0x13	19	DC3	Device control 3	0x33	51	3	0x53	83	S	0x73	115	s
0x14	20	DC4	Device control 4	0x34	52	4	0x54	84	T	0x74	116	t
0x15	21	NAK	Negative ack	0x35	53	5	0x55	85	U	0x75	117	u
0x16	22	SYN	Synchronous idle	0x36	54	6	0x56	86	V	0x76	118	v
0x17	23	ETB	End transmission block	0x37	55	7	0x57	87	W	0x77	119	w
0x18	24	CAN	Cancel	0x38	56	8	0x58	88	X	0x78	120	x
0x19	25	EM	End of medium	0x39	57	9	0x59	89	Y	0x79	121	y
0x1A	26	SUB	Substitute	0x3A	58	:	0x5A	90	Z	0x7A	122	z
0x1B	27	FSC	Escape	0x3B	59	;	0x5B	91	[	0x7B	123	{
0x1C	28	FS	File separator	0x3C	60	<	0x5C	92	\	0x7C	124	|
0x1D	29	GS	Group separator	0x3D	61	=	0x5D	93	]	0x7D	125	}
0x1E	30	RS	Record separator	0x3E	62	>	0x5E	94	^	0x7E	126	~
0x1F	31	US	Unit separator	0x3F	63	?	0x5F	95	_	0x7F	127	DEL

4 Unsigned

Es gibt in Java keine nicht vorzeichenbehafteten Zahlen, Java ist immer vorzeichenbehaftet.

5 Casten

5.1 Implizites Casten

Daten des kleineren Datentyps werden durch den Compiler automatisch dem größeren angepasst, aber nur dann, wenn sich dadurch kein Datenverlust ergibt. Impliziter cast passiert bei:
short -> int, long, float, double
int -> long, float, double
long -> float, double
float -> double

5.2 Explizites Casten

Der User kann einen Typ erzwingen, teilt Java also mit, dass er weiß was er tut.

6 Float vs. double

float floatVar = 3.14E5F;
// F am Ende erforderlich,
// sonst Compilezeitfehler.
// Ohne F oder f wird die
// Zahl als double interpretiert.

d double doubleVar = 3.14E5;
// D ist am Ende nicht
// erforderlich.

7 Prä- dekrement/inkrement

sum += –v; ein prädekrement, hier wird v um 1 vermindert und dann zu sum addiert.

8 Post-dekrement/inkrement

sum += v–; ist ein postdekrement, es wird also zuerst zu sum addiert, danach -1 von v abgezogen.

9 Kennzeichnen von Float und Double

10 Ausdruck und Anweisung

Ein Ausdruck ist immer die Repräsentation eines Wertes. Eine Anweisung ist beispielsweise, im einfachsten Fall eine Zuweisung, die aber wiederum einen Ausdruck enthalten kann.

11 Seiteneffekte

Unter einem Seiteneffekt versteht man eine Änderung des Programmzustands. Die Auswertung eines Ausdrucks hat oft keine Seiteneffekte, da nur ein Wert berechnet wird. Eine Zuweisung dagegen hat einen Seiteneffekt, da sie den Programmzustand ändert.
-> Bei Seiteneffekten werden Inhalte des Speichers verändert, meist der Wert einer statischen Variablen.
Beispiel für einen Seiteneffekt:
Es gibt Operatoren, die eine schnelle und kurze Programmierschreibweise erlauben und es ermöglichen, während der Auswertung eines Ausdrucks Programmvariablen nebenbei zu verändern. Ein Beispiel:
int u = 1;
int v;
v = u++;

Der Rückgabewert des Ausdrucks u++ ist hier der Wert 1. Mit dem Zuweisungsoperator wird der Variablen v der Rückgabewert von u++, d.h. der Wert 1 zugewiesen.
Die Zuweisung v = u++ ist ebenfalls ein Ausdruck und v = u++; stell eine **Ausdrucksanweisung** dar. Als Seiteneffekt des Operators ++ wird die Variable u inkrementiert und hat nach der Inkrementierung den Wert 2.
Man sollte mit Seiteneffekten sparsam umgehen, da sie leicht zu unleserlichen und fehlerträchtigen Programmen führen.

11.0.1 Verschiedene Arten von Seiteneffekten:

-> Seiteneffekte bei Zuweisungen
-> Seiteneffekte bei Operatoren
-> Seiteneffekte bei Methoden, die nicht nur lesend, sondern auch schreibend auf Variablen zugreifen.
-> Seiteneffekte bei Ausdrücken, die mit new ein Objekt erzeugen.

11.1 Seiteneffekte bei unären und binären Operatoren

In Java wird jeder Operand eines Operators vollständig ausgewertet, bevor irgendein Teil der Operation begonnen wird. Damit haben vor einer Operation die Seiteneffekte der Operanden stattgefunden.
-> AUSNAHME: **die Operatoren &&, || und ? :**
Wird der Operator & bzw. | zwischen zwei Operanden verwendet, so wird der rechte Operand immer ausgewertet, egal ob der linke Operand true oder false ist. Wird dagegen der Operator && bzw. || verwendet, so wird der rechte Ausdruck nur dann ausgewertet, wenn der linke Ausdruck true ist. Das ist zu beachten, wenn die Operanden Seiteneffekte beinhalten.

12 Division in Java - wie wird gerundet?

13 Assoziativität, Priorität, Stelligkeit

Precedence order. When two operators share an operand the operator with the higher precedence goes first. For example, 1 + 2 * 3 is treated as 1 + (2 * 3), whereas 1 * 2 + 3 is treated as (1 * 2) + 3 since multiplication has a higher precedence than addition.

Associativity. When an expression has two operators with the same precedence, the expression is evaluated according to its associativity. For example x = y = z = 17 is treated as x = (y = (z = 17)), leaving all three variables with the value 17, since the = operator has right-to-left associativity (and an assignment statement evaluates to the value on the right hand side). On the other hand, 72 / 2 / 3 is treated as (72 / 2) / 3 since the / operator has left-to-right associativity.

Precedence and associativity of Java operators. The table below shows all Java operators from highest to lowest precedence, along with their associativity. Most programmers do not memorize them all, and even those that do still use parentheses for clarity.

Typen der Operanden bestimmen auszuführende Operation wenn überladen,

beeinflussen Stelligkeit, Priorität und Assoziativität aber nicht

14 Call by Reference - Call by Value

Sowohl einfache Datentypen (int, char, ...) als auch Referenzen werden "by value" übergeben. Da Referenzen aber auf Objekte zeigen, wird quasi ein " call by reference" simuliert.

15 Operatoren

16 Operatorpriorität

Priorität	Operatoren	Bedeutung	Assoziativität
Priorität 1	`[]`	Array-Index	links
	`()`	Methodenaufruf	links
	`.`	Komponentenzugriff	links
	`++`	Postinkrement	links
	`--`	Postdekrement	links
Priorität 2	`++`	Präinkrement	rechts
	`--`	Prädekrement	rechts
	`+ -`	Vorzeichen (unär)	rechts
	`~`	bitweises Komplement	rechts
	`!`	logischer Negationsoperator	rechts
Priorität 3	`(type)`	Typ-Umwandlung	rechts
	`new`	Erzeugung	rechts
Priorität 4	`* / %`	Multiplikation, Division, Rest	links
Priorität 5	`+ -`	Addition, Subtraktion	links
	`+`	Stringverkettung	links
Priorität 6	`<<`	Linksshift	links
	`>>`	Vorzeichenbehafteter Rechtsshift	links
	`>>>`	Vorzeichenloser Rechtsshift	links
Priorität 7	`< <=`	Vergleich kleiner, kleiner gleich	links
	`> >=`	Vergleich größer, größer gleich	links
	`instanceof`	Typüberprüfung eines Objektes	links
Priorität 8	`==`	Gleichheit	links
	`!=`	Ungleichheit	links
Priorität 9	`&`	bitweises/logisches UND	links
Priorität 10	`^`	bitweises/logisches Exclusiv-ODER	links
Priorität 11	`|`	bitweises/logisches ODER	links
Priorität 12	`&&`	logisches UND	links
Priorität 13	`||`	logisches ODER	links
Priorität 14	`? :`	Bedingungsoperator	rechts
Priorität 15	`=`	Wertzuweisung	rechts
	`*= /= %= += -= <<= >>= >>>= &= ^= |=`	kombinierter Zuweisungsoperator	rechts

Tabelle 7-9 Priorität und Assoziativität der Operatoren von Java

Operator	Description	Level	Associativity

`[]`
`.`
`()`
`++`
`--`
	access array element
	access object member
	invoke a method
	post-increment
	post-decrement
		1	left to right

`++`
`--`
`+`
`-`
`!`
`~`
	pre-increment
	pre-decrement
	unary plus
	unary minus
	logical NOT
	bitwise NOT
		2	right to left

`()` `new`	cast object creation	3	right to left

`*`
`/`
`%`
	multiplicative	4	left to right

`+ -` `+`	additive string concatenation	5	left to right
`<< >>` `>>>`	shift	6	left to right

`< <=`
`> >=`
`instanceof`
	relational
	type comparison
		7	left to right

`==` `!=`	equality	8	left to right
`&`	bitwise AND	9	left to right
`^`	bitwise XOR	10	left to right
`|`	bitwise OR	11	left to right
`&&`	conditional AND	12	left to right
`||`	conditional OR	13	left to right
`?:`	conditional	14	right to left

`= += -=`
`*= /= %=`
`&= ^= |=`
`<<= >>= >>>=`
	assignment	15	right to left

Strichrechnung
Bitshift
arithmetischer Vergleich (größer/kleiner) und instanceof
Vergleich (gleich/ungleich)
Bitoperatoren (außer Shift und)
logische Operatoren (außer !)
ternärer Operator (? :)
Zuweisung.

17 Scanner - Befehle und deren Wirkung

Mit Scanner myScanner = new Scanner(System.in); wird ein Objekt vom Typ Scanner erzeugt.
Mittels scanner.hasNext(), scanner.hasNextInt(), kann man dann abfragen, ob überhaupt Eingaben oder Eingaben eines gewünschten Typs (in unserem Fall hier int), vorhanden sind. In diesem Fall kann man die Eingaben mittels scanner.next(); bzw. scanner.nextInt(); abrufen.

18 Formale und aktuelle Parameter

Mit den formalen Parametern wird festgelegt, wieviel Übergabeparameter existieren, von welchem Typ diese sind und welche Reihenfolge sie haben.
Beim Aufruf einer Methode mit Parametern finden Zuweisungen statt. Der Wert eines aktuellen Parameters wird dem entsprechenden formalen Parameter zugewiesen. Eine solche Aufruf- schnittstelle wird als call by value-Schnittstelle bezeichnet.

19 Rekursion

Ein Algorithmus heißt rekursiv, wenn er Abschnitte enthält, die sich selbst direkt oder indirekt aufrufen.

Beispiel: Eine kleine Wunsch-Funktion

Auf dem Weg durch den Wald begegnet uns eine Fee. Sie spricht zu uns:
»Du hast drei Wünsche frei«....

```
static void fee() {
    wunsch();
    wunsch();
    fee();
}
```

Durch den dauernden Aufruf der fee()-Funktion haben wir unendlich viele Wünsche frei.
Rekursion ist also das Aufrufen der eigenen Methode, in der wir uns befinden.
→ direkte Rekursion

Dies kann auch über einen Umweg funktionieren.
→ indirekte Rekursion.

19.1 Fundiertheit und Fortschritt

Fundiertheit: Abbruchbedingung
Fortschritt: Rekursionsschritt.

20 Arrays

Ein Array ist ein Objekt, das aus Komponenten (Elementen) zusammengesetzt ist, wobei jedes Element eines Arrays vom selben Datentyp sein muss.
Der Zugriff auf ein Element eines Arrays erfolgt über den Array-Index. Hat man ein Array mit n Elementen, so ist darauf zu achten, dass in Java die Indizierung der Arrayelemente mit 0 beginnt und bei n-1 endet. Arrays sind stets Objekte, auch wenn man Arrays aus einfachen Datentypen als Komponenten anlegt. Arrays werden zur Laufzeit auf dem Heap angelegt. Die Definition einer Array-Variablen bedeutet in Java nicht das Anlegen eines Arrays, sondern die Definition einer Referenzvariablen, die auf ein Array-Objekt zeigen kann. Dieses Array-Objekt muss im Heap angelegt werden.
Bsp:
int[] myArray = new int[4];
Arrays werden in 3 Schritten angelegt:
1.) **Definition** einer Referenzvariablen, die auf das Arrayobjekt zeigt.
2.) **Erzeugen des Arrays**, d.h. eines Array-Objekts, welches aus Komponenten (Elementen) besteht.
3.) Belegung der Array-Elemente mit Werten, d.h. **Initialisiierung des Arrays**

20.1 Referenz

Bei einer Objektvariablen steht an der Speicherstelle nicht das Objekt, sondern nur eine weitere Speicheradresse, an der sich das Objekt befindet. Man spricht von einer Referenz oder einem Zeiger auf das Objekt.

20.2 null

Die Funktionsweise von Objektvariablen als Referenz auf ein Objekt erklärt auch den Sinn des speziellen Wertes null. Er bedeutet, dass eine Variable auf kein Objekt zeigt.

20.3 Lokale Variablen

Eine lokale Variable lebt nur in der Methode, die die Variable deklariert hat.

20.4 Instanzvariablen

Eine Instanzvariable lebt so lange wie das Objekt. Solange das Objekt lebt, leben seine Instanzvariablen.

21 Klassen

21.1 Zugriffsmodifier

21.2 final Variablen, Klassen und Methoden

-> final auf Variable: Variable wird einmal intialisiert und kann danach nicht mehr verändert werden.
-> final auf Methode
-> final auf Klasse: Klasse kann nicht abgeleitet werden.

21.2.1 Zugriffsmodifier auf Variablen

final, private, protected, public, static
Konstanten werden meist in Großbuchstaben geschrieben und als public static final deklariert und müssen somit bei der Deklaration initialisiert werden.

final Variablen werden einmal initialisiert und können dann nicht mehr verändert werden.

21.2.2 Zugriffsmodifier auf Methoden

abstract, final, private, protected, public, static

21.2.3 Zugriffsmodifier auf Klassen

abstract, final, protected, public, static
In jeder *.java Datei darf nur eine public Klasse enthalten sein.
Nicht public Klassen sind nur im selben Paket verwendbar.

Von als final deklarierten Klassen kann man nicht ableiten.

21.2.4 Zugriffsmodifier auf Schnittstellen

private, protected, public, static

21.2.5 Sichtbarkeit

public, private, protected
public wird verwendet, wenn auf die Daten von überall aus der Klasse und auch von Außen zugegriffen werden können soll.
private, wenn von Außen niemand auf die Methoden zugriff haben soll.
protected

Wenn du der Variablen oder der Methode den folgenden Wert gibst, ist sie in ...	... der gleichen Klasse ...	... einer anderen Klasse im gleichen Package ...	... den Unterklassen im gleichen Package ...	... Unterklassen in einem anderen Package ...	... Nicht-Unterklassen in einem anderen Package ...
`public`	... sichtbar.	... sichtbar.	... sichtbar.	... sichtbar.	... sichtbar.
`protected`	... sichtbar.	... sichtbar.	... sichtbar.	... sichtbar.	... **nicht** sichtbar.
kein Modifikator	... sichtbar.	... sichtbar.	... sichtbar.	... **nicht** sichtbar.	... **nicht** sichtbar.
`private`	... sichtbar.	... **nicht** sichtbar.	... **nicht** sichtbar.	... **nicht** sichtbar.	... **nicht** sichtbar.

21.3 Statische und nicht-statische Methoden

A static method belongs to the class and a non-static method belongs to an object of a class. That is, a non-static method can only be called on an object of a class that it belongs to. A static method can however be called both on the class as well as an object of the class. A static method can access only static members.

In Java muss alles zu einer Klasse gehören, aber nicht unbedingt zu einem Objekt.
Klassen dienen zwar hauptsächlich als Vorlagen für Objekte, aber sie haben auch eine unabhängige Existenz. Klassen können Felder und Methoden haben, die nicht zu einem Objekt gehören. Alle diese Elemente werden mit dem Schlüsselwort static gekennzeichnet. Diese Elemente gibt es genau einmal, unabhängig davon, wie viele Instanzen der Klasse erzeugt wurden.
Da statische Methoden nicht zu einem Objekt gehören, haben sie auch keinen Zugriff auf nichtstatische Elemente der Klasse.
Wenn statische Elemente von außen sichtbar sind, so kann auf sie mit dem Punktoperator direkt am Klassennamen zugegriffen werden: Song.getGesamtLaenge(). Es ist dazu nicht notwendig, ein Objekt zu instantiieren.

Statische Felder (Klassenvariablen die statisch defniert sind) zeichnen sich dadurch aus, dass sie nur einen Wert auf Ebene der Klasse haben, unabhängig von deren Instanzen und sogar, wenn niemals eine Instanz erzeugt wird. Sie sind deshalb gut geeignet, Daten zu halten, die nicht zu einem Objekt gehören oder die von allen Instanzen der Klasse gemeinsam genutzt werden. Genau wie auf statische Methoden greift man auch auf statische Felder durch den Punktoperator direkt in der Klasse zu.

21.4 final

final String titel = “ ”;
im Konstruktor kann dann titel EINMAL gesetzt werden (pro Objekt) und hat dann ein Objektlebenlang diesen Wert.

21.5 Konstruktor

Ein Konstruktor heißt genau gleich wie die Klasse, kann Argumente übergeben bekommen. Ein Konstruktor sieht auch einer Methode sehr ähnlich, hat aber KEINEN RÜCKGABEWERT. Es kann auch mehrere Konstruktor(en) geben, das heißt dann’ “überladen”, allerdings müssen diese dann unterschiedlich viele Parameter bekommen bzw. Parameter unterschiedlicher Typen bekommen.

21.6 Klassenvariablen / Objektvariablen

Klassenvariablen, die für alle Objekte einer Klasse als globale Daten zur Verfügung stehen, werden mit Hilfe des Schlüsselwortes static deklariert. Klassenvariablen sind nicht Teil von Objekten und werden deshalb nicht auf dem Heap geführt. Klassenvariablen werden bei der Klasse in der sogenannten Method Area geführt und sind deshalb für alle Objekte einer Klasse nur ein einziges Mal vorhanden.

21.7 Wrapper-Klassen

Für alle primitiven Datentypen gibt es zusätzlich sogenannte Wrapperklassen, das sind quasi die Objektversionen für die primitiven Datentypen. Dies hat vor allem den Hintergrund, dass viele Klassen und Datenstrukturen in Java nur mit Objekten funktionieren, nicht aber mit primitiven Datentypen.

22 Objekte

22.1 Erzeugen von Objekten

Ein Objekt wird immer folgendermaßen erzeugt:
Klassenname meinNeuesObjekt = new Klassenname();

22.2 Die this - Referenz

Eine Methode muss das Objekt finden, für welches sie aufgerufen wird. Man muss das betreffende Datenfeld des Objekts bekanntgeben.

22.3 this() - im Unterschied zu this

Man nutzt this(), um einen Konstruktor aus einem anderen überladenen Konstruktor aufzurufen. this() kann nur in einem Konstruktor verwendet werden und muss die ERSTE Anweisung in einem Konstruktor sein. Ein Konstruktor kann entweder einen Aufruf von super() ODER einen Aufruf von this() haben, aber nie beides.

22.4 Datenkapselung / Information Hiding

Ein Ziel der objektorientierten Programmierung ist es, die Daten und die Implementierung der Methoden eines Objekts zu verbergen. Das bedeutet, dass das Prinzip des Information Hidings angewandt werden soll. Es soll kein Unbefugter die Daten eines Objekts verändern können. Nur die eigenen Methoden eines Objektes sollen auf die Daten entsprechenden Objektes Zugriff haben. Man erlaubt klassenfremden Methoden in der Regel nicht den Zugriff auf die Daten einer Klasse. Dies ist ausschließlich Aufgabe der klasseneigenen Methoden.
ACHTUNG in Programmkonstruktion unterscheidet man explizit zwischen:
- Datenkapselung:
Der Zusammenschluß von Daten und Methoden.
und
- Data-Hiding:
Verstecken interner Details bzw. das Verhindern direkter Zugriffe von außen.
Datenkapselung und Data-Hiding werden als **Datenabstraktion** bezeichnet.
Durch getter - bzw. setter- Methoden geht der Vorteil des Data-Hiding weitgehend verloren.
Identität:

Wenn Objekte den gleichen Speicherplatz haben.
Zustand:
equal zweier Objekte, wenn sie den selben Zustand (und das selbe Verhalten) haben.
Verhalten:
Das Verhalten beschreibt, was das Objekt beim Empfang einer Nachricht macht.

22.5 toString

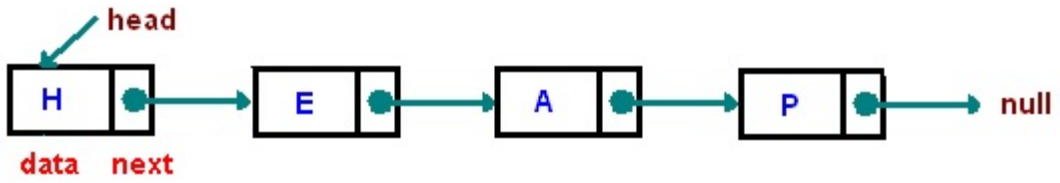
die toString- Methode erbt von Object und bekommt keine Parameter übergeben. toString gibt einen String aus.
Ein Beispiel für eine toString Methode:

```
public String toString() {

 return this.adress + "␣" + this.district + "␣" + this.longitude;
   }
```

23 Datenstrukturen

23.1 Listen



Listen sind geordnete Collections, die mittels Indizes einen beliebigen Zugriff (wahlfreier Zugriff [random access]) auf ein Element innerhalb der Collection ermöglichen - analog zu den Arrays. Im Unterschied zu Arrays können Elemente an beliebiger Stelle zusätzlich eingefügt werden.

23.1.1 Einfach verkettete Listen - Linked Lists und ArrayLists

Methoden der Liste:
add();
add() fügt die übergebene Referenz immer an das Ende der bestehenden Liste an.

23.1.2 Verkettete LIste

Eine verkettete Liste verbindet Objekte über Referenzen. Ein Listenelement besteht also aus den (sichtbaren) Nutzinformationen und aus einer (unsichtbaren) Referenz, die auf das nächste (im Falle einer doppelt verketteten Liste) oder das nächste und das vorherige Element zeigen kann, um die Verkettung zu realisieren. Eine doppelte Verkettung macht zwar mehr Aufwand, gestattet es aber dafür, die Liste in beiden Richtungen zu durchlaufen (traversieren). Eine einfach verkettete Liste besteht also aus:
- einer Referenz, die auf den Listenanfang zeigt.
- sowie Referenzen zur Verkettung der einzelnen Elemente.
- aus Nutzdaten.

23.2 Queue - Warteschlange

Eine Warteschlange ermöglicht ebenfalls die Aufbewahrung von Elementen. Die Besonderheit einer Queue ist die Regelung des Zugriffs. Typischerweise verwenden Queues das Zugriffsprinzip: First in First out. FIFO.
Anlegen einer Queue
Queue<String> meineQueue = LinkedList<>()
Methoden von Queue add()
offer()
remove()
poll()
element()
peek()

23.3 Deque

Anlegen einer Deque
Deque<String> myDeque = new LinkedList();
Methoden von Deque peek()
push()
pop()
remove()
contains()
size()
iterator()
addFirst()
addLast()
offerFirst()
offerLast()
removeFirst()
removeLast()
pollFirst()
pollLast()
getFirst()
getLast()
peekFirst()
peekLast()
add()
offer()
remove()
poll()
element()

23.4 Stack

Funktioniert wie eine Ablage, das Schriftstück, das man zuletzt in die Ablage hineinlegt, wird auch wieder als erstes herausgeholt. Auf einen Stack kann man Objekte nur "oben" drauflegen und sie auch nur von "oben" wieder wegnehmen. (LIFO) Es besteht also NICHT die Möglichkeit wie bei einer Liste, ein Element aus der Mitte zu entfernen oder ein Element

in der Mitte einzufügen.

Anlegen eines Stacks
Stack<String> myStack = new Stack<>();

Methoden von Stack:
push()
Legt ein Element auf den Stack.
pop()
Entfernt das oberste Element vom Stack.
empty()
gibt true zurück, wenn der Stack leer ist.
peek()
Gibt eine Referenz auf das Objekt zurück, das ganz oben auf dem Stack liegt, entfernt die Referenz aber NICHT.
equals()
Vergleicht

23.5 Map

Bei Maps wird statt über einen numerischen Index über ein eindeutiges Schlüsselobjekt auf Werte zugegriffen. In einer Map werden also Schlüssel-Werte-Paare repräsentiert.

Anlegen einer Map
Map<String, Objekt> map = new TreeMap<String, Objekt>();

Methoden von Map
put(key, value)
erzeugt eine Verknüpfung zwischen dem angegebenen Schlüssel und Wert.
get()
isEmpty()
containsKey()
containsValue()
put()
remove()
putAll()
clear()
keySet()
values()
entrySet()
equals()
hashCode()
liefert den Wert zum angegeben Schlüssel zurück. Ist der Schlüssel mit keinem Wert verknüpft, wird null zurückgeliefert.
containsKey()
prüft, ob die Map den referenzierten Schlüssel enthält.
containsValue()
prüft, bo die Map einen oder mehrer Schlüssel enthält, die mit dem durch die Referenzvariable value referenziert Objekt verknüpft sind.

23.6 Bäume

Bei einem Baum kann ähnlich wie bei einer verketteten Liste in jedem Knoten eine Referenz auf ein Objekt gespeichert werden. Im Unterschied zu einer verketteten Liste haben die Knoten eines Baumes jedoch mehrere Nachfolger. Eine einfache Variante ist ein binärer Baum, bei dem jeder Knoten zwei Nachfolger hat.
Sind n Elemente in einem Objekt der Klasse TreeSet<E> enthalten, so beträgt die Anzahl der Schritte, um ein Element einzufügen, zu löschen oder zu suchen maixmal log(n).
Für einen binären Baum mit n Elementen braucht man nämlich maximal log(n)/log(2) Ebenen mit je $1, 2, 4, .., 2^{n-1}$ Elementen. Die Zugriffszeit ist also nicht (quasi) konstant wie beider Klasse HashSet<E>, aber immer noch ziemlich schnell. Der oberste Knoten heißt Wurzel, die restlichen Knoten heissen Blätter.
Durch die Implementierung der Schnittstelle SortedSet<E> garantiert die Klasse TreeSet<E>, dass sie die Elemente in sortierter Reihenfolge abspeichert.

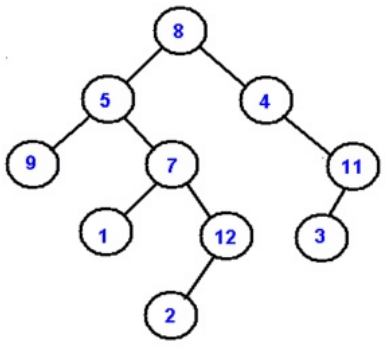
There are three different types of depth-first traversals, :

- PreOrder traversal - visit the parent first and then left and right children;
- InOrder traversal - visit the left child, then the parent and the right child;
- PostOrder traversal - visit left child, then the right child and then the parent;

There is only one kind of breadth-first traversal--the level order traversal. This traversal visits nodes by levels from top to bottom and from left to right.

As an example consider the following tree and its four traversals:

PreOrder - 8, 5, 9, 7, 1, 12, 2, 4, 11, 3
InOrder - 9, 5, 1, 7, 2, 12, 8, 4, 3, 11
PostOrder - 9, 1, 2, 12, 7, 5, 3, 11, 4, 8
LevelOrder - 8, 5, 4, 9, 7, 11, 1, 12, 3, 2



23.7 Sortieralgorithmen

23.7.1 Bubblesort

Der einfachste Sortieralgorithmus, der allerdings auch am längsten dauert. Er sortiert ein Array von Datensätzen so lange durch wiederholtes Vertauschen von Nachbarfeldern, die in der falschen Reihenfolge stehen, bis das Array vollständig sortiert ist.
Die durchschnittliche Laufzeit liegt bei n^2
Die Laufzeit hängt stark von der Vorsortierung ab.

23.7.2 Tree-Sort

23.7.3 Quicksort

Der Quicksort ist ein sogenannter “Teile - und herrsche” Algorithmus. Dieser Ausdruck besagt, dass das Problem so lange in Teilprobleme aufgeteilt wird, bis die Teilprobleme gelöst werden können. Danch werden die Teillösungen wiedr zu einer Gesamtlösung zusammengefügt. Beim Quick-Sort wird aus einem zu sortierenden Array das mittlere Element als so genanntens Pivot-Element herausgesucht. Dann wird im Wesentlichen verglichen, ob jedes Element aus der linken Hälfte des Arrays kleiner ist als das Pivot-Element. Ebenso wird verglichen, ob in der rechten Hälfte des Arrays jedes Element größer ist als das Pivot-Element. Falls man zwei Elemente gefunden hat, die jeweils auf der falschen Seite stehen, werden sie miteinander getauscht.

23.8 Vererbung

Unterklassen können von Oberklassen durch das Schlüsselwort "extends" erben, dadurch ergibt sich Folgendes:
- Alle in der Oberklasse definierten Methoden und Variablen sind auch in der Unterklasse vorhanden. (So sie in der Oberklasse nicht static deklariert wurden).
- Alle Als private deklarierte Methoden und Variablen in der Oberklasse sind in der Unterklasse nicht sichtbar, allerdings können vererbte Methoden von der Oberklasse auf diese Variablen/Methoden zugreifen.
- Von der Oberklasse geerbte Methoden können in der Unterklasse überschrieben werden. (ausgenommen final)
- Unterklassen können die Oberklasse erweitern, also neue eigenständige Methoden und Variablen implementieren.
Hat eine neu definierte Variable den gleichen Namen, wie eine Variable in der Oberklasse, so wird die Variable in der Oberklasse verdeckt. In der Unterklasse ist nur die Variable der Unterklasse zu sehen, obwohl beide Variablen existieren.
- Beim Erzeugen einer Unterklasse wird zuerst immer der Konstruktor der Oberklasse ausgeführt.
In Java kann eine Klasse immer nur von EINER anderen Klasse abgeleitet werden - Keine Mehrfachvererbung in Java möglich.
- Jedes Interface kann ein oder mehrere andere Interfaces erweitern und jede Klasse kann mehrere Interfaces implementieren (damit wird die nicht vorhandene Mehrfachvererbung weitgehend umgangen).

23.9 Interface

- In Interfaces können keine Variablen deklariert werden und keine Methoden implementiert werden, sondern nur definiert werden.
- Von Interfaces können keine Instanzen erzeugt werden.

23.10 Abstrakte Klassen

- Von als abstract definierten Klassen darf keine Instanz erzeugt werden.
Diese Klassen enthalten abstrakte Methoden, die ebenso wie in Interfaces, nur definiert werden in der abstrakten Klasse. Diese können dann ähnlich wie Interfaces an eine Unterklasse vererbt werden und müssen dort implementiert werden.
"abstract" und "final" sind somit fast gegensätzlich. (?)

23.11 Polymorphismus

Ein Objekt kann durch Vererbung mehrere Typen haben.

23.12 super()

Das Schlüsselwort super() ruft den Konstruktor der Oberklasse auf.

23.13 Klassenableitung

23.14 Object

Object ist die oberste Oberklasse aller Klassen, Object selbst erbt von keiner anderen Klasse.

23.14.1 Methoden von Object

Jede Klasse erbt automatisch alle Methoden von Object:
- toString()
- equals()
- hashCode()
- clone()

23.14.2 toString

toString() bekommt keine Parameter und hat den Rückgabewert String.

23.14.3 instanceOf, class, getClass, und Cast

Mit dem instanceof-Operator kann getestet werden, ob eine Referenz auf ein Objekt eines bestimmten Typs zeigt. Dies ist beispielsweise dann wichtig, wenn eine Referenz vom Typ einer Basisklasse ist. Eine solche Referenz kann ja auf Objekte aller abgeleiteten Klassen zeigen. Mit Hilfe des instanceof-Operators lässt sich nun nachprüfen, ob das referenzierte Objekt tatsächlich vom angenommenen Typ ist.
Mit dieser Erkenntnis kann dann die Referenz in den entsprechenden Typ gecastet werden. Das heißt, man kann überprüfen, ob ein expliziter Cast zulässig ist.
Die Syntax ist:
a instanceof Klassenname

statt instanceof kann auch getClass() verwendet werden.

23.15 equals und hashCode

Für die Berechnung von Hash-Codes werden Hash-Funktionen verwendet. Ihr Ziel ist es, einem Objekt einen möglichst eindeutigen Kennwert zuzuordnen. Ein bestimmtes Objekt erhält stets denselben Hash-Code. Es ist jedoch möglich, dass zwei unterschiedliche Objekte denselben Hash-Code erhalten. Das heißt, Hash-Funktionen sind in der Regel nicht umkehrbar.
Bei Objekten mit gleichem Hash-Code muss das richtige Objekt in einem zweiten Schritt per equals()-Methode ermittelt werden.

Beim Zugriff wird also zunächst hashCode() der Hash-Code berechnet, mit dem %-Operator auf die Tabellengröße angepasst, um die richtige Liste zu finden und anschließend mit Hilfe der equals()-Methode nach dem richtigen Objekt gesucht. Dies muss nicht das ursprünglich eingefügte Objekt sein, da die equals()-Methode lediglich auf inhaltliche Gleichheit prüft (nur ihre Basisimplementierung prüft auf Identität).

Identität: gleicher Speicherplatz
Zustand: equal zweier Objekte, wenn sie den selben Zustand (das selbe Verhalten) haben.
Verhalten: beschreibt, was das Objekt beim Empfang einer Nachricht macht.

Dynamisches Binden versus static und private
Statisches Binden: Compiler weiß, welche Methode ausgeführt wird -> Laufzeiteffizienz

Wenn x!= null gilt, dann muss x.equals(null) als Ergebnis immer false liefern, auch wenn wir equals selbst implementieren.
x != null -> x.equals(null) == false;
x != null -> x.equals(x);
x != null && y != null -> x.equals(y) == y.equals(x);
x.equals(y) && y.equals(z) -> x.equals(z);

23.16 Dynamisches Binden:

Jede Lösung mit dynamischem Binden ist auch durch if- bzw. switch - Anweisungen lösbar. Programme, die durch dynamisches Binden gelöst wurden, sind einfacher wartbar als solche, die mittels switch und if-Anweisungen erstellt wurden.

23.17 Abstrakte Klassen und Methoden

23.18 Instrumentierung für Debugging

23.19 Exception-Handling

23.19.1 Checked-Exceptions

Können auch bei fehlerfreier Programmierung immer auftreten. (Beispiel: IOException).

23.19.2 Unchecked-Exceptions

Das sind solche Exceptions, die nicht in der Methodensignatur angegeben werden müssen. Oft werden sie auch RuntimeExceptions genannt, nach der Oberklasse, die all diese Fehler gemein haben.
(Beispiel: IndexOutOfBoundsException, ClassCastException...)

23.19.3 Errors

Erben direkt von Throwable - Ein Error wird geworfen, wenn bei der Ausführung des Programms ein ernsthaftes Problem aufgetreten ist und ein normales Fortsetzen des Programms nicht mehr möglich ist.

23.20 Finally-Blöcke

Um sicherzustellen, dass nach einer Exception beispielsweise eine falsch geöffnete Datei wieder geschlossen wird, kann try-catch mit einem finally-Block ergänzt werden. Der Inhalt dieses Blocks wird auf jeden Fall ausgeführt, wenn der try-Block verlassen wird, egal ob

regulär, oder mit einem Fehler. - Es gibt nur eine Möglichkeit, wie ein finally-Block nicht zur Ausführung kommt: Die gesamte JVM stürzt ab.
} finally { if(reader != null) {
reader.close();
}

23.21 Design-by-Contract

- Entwurf durch Verträge:
Eine Klasse besteht nicht nur aus Methoden und Datenfeldern - eine Klasse wird benutzt von anderen Klassen, hier Kunden (Client) genannt, und hat damit Beziehungen zu all ihren Kunden.
Das Konzept "Design by Contract" sieht diese Beziehungen als eine formale Übereinkunft zwischen den beteiligten Partnern an und definiert präzise, unter welchen Umständen ein korrekter Ablauf des Programms erfolgt.
Worum es bei Design-by-Contract vor allem geht, ist, dass sich beim Aufruf einer Methode der Aufrufer und die aufgerufene Methode gegenseitig aufeinander verlassen können.
Die Beziehung zwischen Aufrufer und aufgerufener Methode kann man formal als einen Vertrag einer Methode bezeichnen, der nicht gebrochen werden darf, da ansonsten eine Fehlersituation entsteht.
Bei einem Vertrag haben in der Regel beide Seiten Rechte und Pflichten. So wie im Alltag ein Vertrag die Beziehungen zwischen Vertragsparteien (Personen, Organisationen) regelt, beschreibt ein Vertrag die Beziehung eines Objekts einer Klasse mit seinem Kunden, dem Aufrufer.
So lange bei der Ableitung von einer Basisklasse der Vertrag der Basisklasse in einer Unterklasse nicht gebrochen wird, ist es möglich, den für die Basisklasse geschriebenen Code auch für die Unterklassen, die eventuell erst später erfunden werden, zu verwenden.

Zusicherungen
Allgemein werden Verträge durch sogenannte Zusicherungen spezifiziert. Zusicherungen bestehen aus Bedingungen über die Zustände von Objekten an einer bestimmten Programmstelle.
Der Entwurf durch Verträge verwendet drei verschiedene Arten von Zusicherungen:
* Vorbedingungen von Methoden
* Nachbedingungen von Methoden
* Invarianten von Klassen
Der Vertrag einer Methode umfasst die Vor- und Nachbedingungen einer Methode.

Vor- und Nachbedingungen einer Methode

Eine **Vorbedingung** stellt die Einschränkung dar, unter denen eine Methode korrekt funktioniert. So darf beispielsweise eine Methode push(), die ein Element auf einem Stack ablegt, nicht aufgerufen werden, wenn der Stack voll ist, genauso wenig wie eine Methode pop(), die ein Element von einem Stack abholen soll, aufgerufen werden darf, wenn kein Element mehr auf dem Stack ist.
Eine Vorbedingung stellt eine Pflicht für einen Aufrufer dar, sei es, dass der Aufruf innerhalb der eigenen Klasse erfolgt oder von einem Kunden.
Ein korrekt arbeitendes System führt nie einen Aufruf in einem Zustand durch, der nicht die Vorbedingung der gerufenen Methode erfüllen kann. Eine Vorbedingung bindet also einen Aufrufer.

Die Vorbedingung definiert die Bedingungen, unter denen ein Aufruf zulässig ist. Sie stellt die Pflicht für den Aufrufer dar und einen Nutzen für den Aufgerufenen.
Ist die Vorbedingung verletzt, so ist der Aufgerufene nicht an die Nachbedingung gebunden und kann machen, was er will. Zum Beispiel kann die Verletzung der Vorbedingung einen Programmabsturz verursachen.

Eine **Nachbedingung** stellt den korrekten Zustand nach dem Aufruf einer Methode dar. So kann nach dem Aufruf von push() der Stack nicht leer sein und die Zahl der Elemente auf dem Stack muss um 1 höher sein als vor dem Aufruf der Methode. Umgekehrt kann nach dem Aufruf von pop() der Stack leer sein, wobei die Zahl der Elemente auf dem Stack um 1 geringer sein muss als vor dem Aufruf.
Eine Nachbedingung bindet eine Methode einer Klasse. Eine Nachbedingung steltl die Bedingung dar, die von der Methode eingehalten werden muss. Die Nachbedingung ist eine Pflicht für den Aufgerufenen und ein Nutzen für den Aufrufer.
Mit der Nachbedingung wird garantiert, dass der Aufrufer nach Ausführung der Methode einen Zustand mit korrekten Eigenschaften vorfindet, natürlich immer unter der Voraussetzung, dass beim Aufruf der Methode die Vorbedingung erfüllt war.
Wichtig ist, dass kein redundanter Code geschrieben wird. Das wäre zu fehlerträchtig und außerdem nicht performant. Es gilt somit das single source Prinzip. Die Vorbedingung muss stets vom Aufrufer geprüft werden und keinesfalls vom Aufgerufenen. Umgekehrt muss die Einhaltung der Nachbedingung stets vom Aufgerufenen überwacht werden. Der Aufrufer darf die Prüfung der Nachbedingung nicht durchführen.
Wie bei einem guten Vertrag im täglichen Leben haben also Aufrufer und Aufgerufene Pflichten und Vorteile. Der Aufrufer hat die Pflicht, den Aufgerufenen korrekt aufzurufen. Damit hat der Aufgerufene den Vorteil, dass er in einer korrekten Umgebung abläuft.
Der Aufgerufene wiederum hat die Pflicht, korrekte Werte zurückzugeben. Diese Pflicht des Aufgerufenen ist der Vorteil des Aufrufers, da er korrekte Werte enthält.

Invarianten
Invarianten beziehen sich nicht auf eine einzelne Methode. Invarianten beziehen sich immer auf das gesamte Objekt. Eine Invariante muss von allen Methoden, die ein Kunde aufrufen kann, für jedes einzelne Objekt einer Klasse erfüllt werden, damit ein System korrekt arbeitet oder in einem korrekten Zustand ist.
Da die Invarianten von allen Methoden einer Klasse, die von einem Kunden aufgerufen werden können, eingehalten werden müssen, um die Korrektheit zu gewährleisten, spricht man auch von Klasseninvarianten.
Eine Invariante ist eine Zusicherung bezüglich einer Klasse. Es soll als Beispiel hierzu eine Klasse Polygon betrachtet werden. Ein Polygon besteht aus mindestens drei Eckpunkten, die mit geraden Linien verbunden sind. Somit besitzt die Klasse Polygon die Klasseninvariante, dass die Anzahl der aggregierten Punkte - die Punkte können beispielsweise durch die Klasse Punkt repräsentiert werden - mindestens drei betragen muss, damit ein Polygon vorliegt. Diese Eigenschaft gilt für die gesamte Klasse und nicht individuell nur für eine einzelne Methode. Sie ist damit eine Klassengemeinschaft im Gegensatz zu Vor- und Nachbedingungen, die jeweils einzelne Methoden charakterisieren.
Eine Invariante muss gfelten vor Aufruf einer Methode und nach dem Aufruf einer Methode durch einen Kunden. Eine Invariante kann temporär verletzt werden während der Ausführung einer Methode oder beim Aufruf von Service-Methoden, die nicht außerhalb der Klasse sichtbar sind - also nicht exportiert werden. Dies stellt kein Problem dar, da die Invariante dem Kunden erst nach Ausführung einer exportierten Methode wieder zur Verfügung steht. Nach Ausführung einer exportierten Methode muss die Klasseninvariante wieder

eingehalten sein.
So hat beispielsweise eine Klasse Quadrat - die Quadrate auf dem Bildschirm zeichnen, verschieben, drehen und skalieren kann - die Invariante, dass vor und nach dem Aufruf einer der Methoden zeichne(), verschiebe(), drehe() und skaliere() alle Seiten des Quadrats gleich lang sind und jeder Winkel ein rechter Winkel ist. Innerhalb der Methode verschiebe() kann aber temporär erst ein Teil der Eckpunkte verschoben sein, sodass temporär gar kein Quadrat vorliegt. Das sieht das "Kundenprogramm" aber nicht.
Eine Klasseninvariante muss vor und nach dem Aufruf einer nach außen sichtbaren Methode (Export-Methode) eingehalten sein.
Werden Methoden intern aufgerufen, wird eine Invariante nicht geprüft. Wenn Methoden von außen aufgerufen werden, sollte die Invariante überprüft werden, um sich der Korrektheit zu vergewissern, denn Verträge müssen eingehalten werden.
Der Vertrag einer Klasse umfasst die Verträge der Methoden und die Invarianten der Klasse. Werden verschiedenen Kunden einer Klasse jedoch verschiedene Leistungen der Klasse zur Verfügung gestellt, so ordnet man die Verträge der Methoden in verschiedene Verträge der Klasse jeweils mit dem entsprechenden Kunden ein.
Invarianten gelten bei exportierten Methoden vor und nach dem Aufruf, bei Aufruf von Konstruktoren nach dem Aufruf.

23.22 Aus dem Complang-skriptum:

Anbieter wird Server genannt und Kunde Client, in der Softwareentwicklung können diese auch Objekte sein.
-> Der Server bietet Services in Form von Methodenausführungen an.
-> Der Client nimmt Services in Form von Methodenausführungen in Anspruch.
Details sind im sogenannten Softwarevertrag geregelt: Man weiß, was man sich von dem Aufruf einer Methode erwarten kann, ohne die genaue Implementierung zu kennen.

23.22.1 Vorbedingungen und Nachbedingungen

Vorbedingungen:
Vorbedingungen in Untertypen dürfen schwächer, aber nicht stärker sein, als entsprechende Bedingungen im Obertyp.
Nachbedingungen
Nachbedingungen im Untertyp dürfen stärker, jedoch nicht schwächer als die Bedingungen im Obertyp sein.
-> **Instanzen eines Untertyps verhalten sich so, wie man es sich von der Instanz eines Untertyps erwarten würde.**

23.22.2 Rechte und Pflichten des Servers

- Der Server darf davon ausgehen, dass die Vorbedingung und die Invarianten vor dem Methodenaufruf erfüllt sind.
- Der Server muss dafür sorgen, dass nach Methodenaufruf die Nachbedingungen eingehalten sind und die Invarianten des Objekts erfüllt sind.
Vorbedingungen:
Müssen bereits vor Ausführung einer Methode erfüllt sein.
Nachbedingungen:
Müssen erst nach der Methodenausführung erfüllt sein.

Invarianten beschreiben quasi unveränderliche Eigenschaften.
Entsprechend dem Softwarevertrag kann sich der Server darauf verlassen, dass der Client für die Einhaltung der Vorbedingungen vor jeder Ausführung einer Methode sorgt. Der Client kann sich darauf verlassen, dass der Server für die Einhaltung der Nachbedingungen und Invarianten am Ende der Ausführung einer Methode sorgt. Das alles gehört zur Entwurfsphase, nicht zur Analysephase.

Richtlinien für die Beziehung von Zusicherungen im Unter- und Obertyp:
- Vorbedingungen in Untertypen dürfen schwächer, aber nicht stärker als entsprechende Bedingungen in Obertypen sein. Wenn eine Vorbedingung im Obertyp beispielsweise x>0 lautet, darf die entsprechende Bedingung im Untertyp x >=0 sein. Im Untertyp steht die Verknüpfung beider Bedingungen mit ODER.
-Nachbedingungen und Invarianten in Untertypen dürfen stärker, jedoch nicht schwächer als die Bedingungen in Obertypen sein. Wenn eine Nachbedingung oder Invariante im Obertyp z.B. x >= 0 lautet, darf die entsprechende Bedingung im Untertyp x>0 sein. Im Untertyp steht die Verknüpfung beider Bedingungen mit UND.
Diese Beziehungen stellen sicher, dass sich eine Instanz eines Untertyps so verhält, wie man es sich von einer Instanz des Obertyps erwartet.
Vorbedingungen: Einschränkungen auf formalen Parametern sowie alles, um das sich Aufrufer von Methoden kümmern müssen, sind Vorbedingungen. Invarianten: Beschreibungen unveränderlicher Eigenschaften von Objektzuständen.

23.23 Software-Dokumentation und Zusicherungen

- Kommentare sind als Information im Programm zentral
- Kommentare können als Zusicherungen verstanden werden
Zusicherungen:
* Vorbedingungen:
Bedingungen, die vor der Ausführung einer Methode oder eines Konstruktors erfüllt sein müssen.
* Nachbedingungen: Bedingungen, die nach der Ausführung von Methoden erfüllt sein müssen.
* Invarianten:
Bedingungen auf Variablen, Klassen und Interfaces, die stets eingehalten werden müssen, damit das Programm reibungslos funktioniert.

23.23.1 Schleifeninvarianten

Das sind Bedingungen, die zu Beginn und am Ende jedes Schleifendurchlaufs erfüllt sein müssen.

23.24 Testen

Testfälle decken nicht alle Möglichkeiten ab.
Gefundene Fehler lassen Rückschlüsse über vorhandene Fehler im Programm zu.
Beseitigen von Fehlern führt meist zu neuen Fehlern -> Behebung der Ursache des Fehlers

und nicht der Symptome.
Fehler die nur selten Auftreten stellen eine besondere Bedrohung dar.
Unittest
Funktionaler Test eines klar abgegrenzten Programmteils.
Integrationsstest
Überprüft korrekte Zusammenarbeit zwischen den Units.
Systemtest
Prüft Erfüllung aller funktionalen und nicht funktionalen Eigenschaften
Abnahmetest
Test unter realen Bedingungen.

23.25 Seiteneffektfreie Programme

23.26 Beispiele für unveränderliche Datenstrukturen

24 Theoriefragen von alten Tests

24.0.1 Testfragen:

- Der Client muss dafür sorgen, dass Invarianten am Anfang und Ende jeder Methodenausführung eingehalten werden.

- Der Client darf davon ausgehen, dass Nachbedingungen am Ende einer Methodenausführung eingehalten sind.

- „Design by Contract“ bietet die Grundlage für die Zusammenarbeit zwischen Softwareentwickler und Auftraggeber.

- In Design-by-Contract sind unerwartete Zustandsänderungen zu vermeiden.

- Der Server muss dafür sorgen, dass nach Rückkehr aus der Methode die Nachbedingungen eingehalten sind.

- Nachbedingungen einer Methode sind im Untertyp gleich oder schwächer als im Obertyp.
Wenn x != null gilt, dann muss x.equals(null) als Ergebnis immer false liefern, auch wenn wir equals selbst implementieren.
true
Auf null kann man keine Methoden aufrufen.

x != null -> x.equals(null) == false;
x != null -> x.equals(x);
x != null && y != null -> x.equals(y) == y.equals(x);
x.equals(y) && y.equals(z) -> x.equals(z);
wenn x und y unverändert -> dann liefern wiederholte Aufrufe von x.equals(y); gleiche Ergebnisse.

Der Ausdruck X.class ist auch anwendbar, wenn X ein Interface ist.
ja, geht

In jeder .java-Datei darf höchstens eine nicht-public Klasse vorkommen.
nur eine erlaubt.
Als static und final deklarierte Variablen dürfen in Interfaces vorkommen.
ja, dürfen.

Der Ausdruck null instanceof X liefert für jedes Interface X stets true.
nein, liefert false;

Eine als final definierte Methode muss überschrieben werden.
Eine Methode, die mit dem Modifier final versehen ist, darf nicht überschrieben werden.

private Methoden werden immer statisch gebunden.
richtig, für static, final, private.

Aus x.equals(y) folgt x.hashCode() == y.hashCode()
wahr

null.equals(null) liefert immer true als Ergebnis.
falsch
In jeder .java-Datei darf höchstens eine (nicht geschachtelte) public Klasse vorkommen. ja

Aus x.hashCode() == y.hashCode() muss stets x.equals(y) folgen.
false;

Der Ausdruck null instanceof X liefert für jede Klasse X stets false. wahr

final Methoden werden immer statisch gebunden. richtig

Eine als abstract definierte Methode muss überschrieben werden.
richtig

Als static und final deklarierte Variablen dürfen in Interfaces vorkommen.
ja, richtig

Der Ausdruck X.class ist nur anwendbar, wenn X eine Klasse ist.
falsch, weil es auch bei Interfaces geht.

Namen von als static und final deklarierte Variablen bestehen meist nur aus Großbuchstaben.
ja, korrekt.

Der Ausdruck (X)null führt zu keiner Fehlermeldung wenn X eine Klasse oder ein Interface ist.
richtig

Aus !x.equals(y) muss stets x.hashCode() != y.hashCode() folgen.
falsch

Abstrakte Methoden werden immer statisch gebunden.
falsch

Von einer als final definierten Klasse können keine Objekte erzeugt werden.
man kann final Klassen instanzieren.

Jede .java-Datei muss genau eine Klasse oder genau ein Interface enthalten.
nicht nur "GENAU" eine.

Wenn x.equals(y) && y.equals(z) true ergibt, dann muss stets auch x.equals(z) true ergeben. ja, ist korrekt

Der Ausdruck x.getClass() ist nur anwendbar wenn x ein Objekt eines Referenztyps ist. korrekt

Invarianten sind meist in der Nähe der Deklarationen entsprechender Objektvariablen zu finden. korrekt

Der Server darf davon ausgehen, dass Vorbedingungen zu Beginn der Methodenausführung eingehalten sind. ja, darf davon ausgehen.

In Design-by-Contract sind unerwartete Zustandsänderungen zu vermeiden. korrekt.

Exceptions, die vom Typ Error abgeleitet sind, müssen in throws-Klauseln angegeben werden.
falsch (??)

Nachbedingungen einer Methode sind im Untertyp gleich oder schwächer als im Obertyp.
falsch

Der Server muss dafür sorgen, dass nach Rückkehr aus der Methode die Nachbedingungen eingehalten sind.
korrekt

Auch sehr selten auftretende Fehler stellen ein Sicherheitsrisiko dar.
ja natürlich

Exceptions vom Typ IOException müssen in throws-Klauseln angegeben werden (falls sie nicht abgefangen werden).
korrekt

Nachbedingungen einer Methode sind im Untertyp gleich oder stärker als im Obertyp. wahr

Schleifeninvarianten stehen meist in den Kommentaren ganz zu Beginn einer Klassendefinition.
falsch

Der Client muss dafür sorgen, dass Invarianten am Anfang und Ende jeder Methodenausführung eingehalten sind.
Nein, das ist Aufgabe des Servers.

Der Client darf davon ausgehen, dass Nachbedingungen am Ende einer Methodenausführung eingehalten sind.
richtig

Die Anzahl der im Programm gefundenen Fehler ist oft direkt proportional zur Anzahl der vorhandenen Fehler.
korrekt

"Design-by-Contract" bietet die Grundlage für die Zusammenarbeit zwischen Softwareentwickler und Auftraggeber.
falsch

Bubblesort ist ein sehr effizientes Sortierverfahren und wird daher häufig in

Programmbibliotheken angeboten.
falsch

Eine Klasse, die ein Interface implementiert, kann von keiner anderen Klasse abgeleitet sein. falsch

Wenn ein Interface X ein Interface Y erweitert, dann ist jedes Objekt vom Typ Y auch ein Objekt vom Typ X.
falsch

Das Einfügen in einen binären Baum ist fast immer effizienter als das Einfügen am Kopf einer Liste. falsch

Das Prinzip von Quicksort beruht darauf, dass wiederholt benachbarte Elemente eines Arrays vertauscht werden.
falsch

Die binäre Suche benötigt ungefähr so viele Suchschritte wie die Suche in einem (gut ausbalancierten) Suchbaum. korrekt

Die Methode getClass() ist in jedem Objekt definiert. ja, weils von Object erbt.

Vergleiche ganzer Zahlen mittels == können unvorhersehbare Ergebnisse liefern daher sollten ganze Zahlen mittels equals verglichen werden.
falsch

Eine rekursive Methode ist fundiert wenn eine erreichbare Abbruchbedingung existiert. korrekt

Eine lokale Variable von einem Referenztyp enthält ohne explizite Initialisierung den Wert null. falsch (??)

Die Pseudovariable this darf nur in Konstruktoren verwendet werden. falsch

Statt einer Variable vom Typ char[] kann man stets eine Variable vom Typ String verwenden, ohne dadurch eingeschränkt zu werden.
ja, korrekt

Der Compiler sichert zu, dass Variablen des Typs boolean nicht null enthalten. ja, korrekt

Wenn eine Klasse X ein Interface Y implementiert, dann sind Objekte vom Typ X auch Objekte vom Typ Y. korrekt

Die Methode toString() ist in jedem Objekt definiert. ja, weil sie aus Object erbt.

Jeder Referenztyp ist in Java Untertyp von Object. ja, korrekt

Die binäre Suche setzt voraus, dass das durchsuchte Array sortiert ist. ja, korrekt

Tree-Sort kann auf vorsortierten Datenstrukturen sehr ineffizient sein. richtig.

Listen, die zu Bäumen entartet sind, stellen ein erhebliches Effizienzproblem dar. falsch

Bubblesort ist auf einem Array mit Zufallszahlen deutlich effizienter als auf einem vorsortierten Array. nein

Jede Variable eines Referenztyps kann statt einem entsprechenden Objekt auch null enthalten. korrekt.

Der Wert null wird für die Fundiertheit rekursiver Datenstrukturen benötigt. falsch
Nachbedingungen einer Methode sind im Untertyp gleich oder stärker als im Obertyp
Der Client darf davon ausgehen, dass nach Rückkehr aus der Methode, die Nachbedingungen eingehalten sind
Der Ausdruck null instanceof X liefert für jede Klasse X stets false
private Methoden werden immer statisch gebunden
Beim Grey-Box-Test werden Testfälle schon vor dem Programmieren festgelegt
Aus !x.equals(y) folgt x.hashCode() != y.hashCode()
Unter "Propagation einer Exception" versteht man das Abfangen einer Exception in einer catch-Klausel
Unter "Datenkapselung" versteht man die klare Trennung zwischen Daten und Methoden
Die Objektvariablen einer als final deklarierten Klasse sind nach der Initialisierung nicht mehr änderbar
Eine lokale Variable von einem Referenztyp enthält ohne explizite Initialisierung den Wert null
Ein Stack verfolgt die FIFO-Strategie
Wenn eine Klasse X ein Interface Y implementiert, dann sind Objekte vom Typ Y auch Objekte vom Typ X.
Eine Objektvariable vom Typ double enthält ohne explizite Initialisierung den Wert 0.0
Die Pseudovariable this ist nur in Konstruktoren verwendbar

---

public interface Foo ...
public interface Bar ...
public interface Laa extends Bar, Foo ...
public class Mu implements Foo ...
public class Ne extends Object implements Laa ...

Welche der folgenden Aussagen treffen zu ?
- Mu ist Unterklasse von Object
- Ne überschreibt Mu
- Ne ist Untertyp von Bar
- Foo ist Obertyp von Mu
- Foo ist Oberklasse von Ne

25 Kontrollfragen - Skriptum

Kontrollfragen - Kapitel 1

Was ist eine Deklaration bzw. Definition
Deklaration: private int zahl;
Definition: private int zahl = 1;

Wozu braucht man Variablen und was haben Variablen mit dem Speicher eines Computers gemeinsam? Variablen reservieren Speicherplatz im Rechner.

Wodurch unterscheidet sich ein formaler Parameter von einem aktuellen Parameter bzw. Argument? Bestehen die Unterschiede auf syntaktischer oder semantischer Ebene?
Aktueller Parameter:
Formaler Paramter:

Woran erkennt und wozu braucht man Kommentare?
Kommentare beginnen entweder mit /* Ich bin ein Kommentar */ oder nur mit: // hier steht ein Kommentar. (Einzeilige Kommentare.)

Was macht ein Konstruktor und wie unterscheidet er sich von einer Methode?
Ein Konstruktor hat keinen Rückgabewert.

Was versteht man unter dem Initialisieren eines Objekts oder einer Variablen?
Der Variablen oder dem Objekt einen Wert zu weisen.

Was ist eine Schleifenbedingung, ein Schleifenzähler, ein Schleifenrumpf und eine Iteration?
Schleifenbedingung:
Schleifenzähler: was passiert in jedem Schleifendurchlauf. Bspw. i++;
Schleifenrumpf: das was die Schleife tun soll, beispielsweise: 4*m ? 0 : 8*m
Iteration: Wiederholung

Wie hängen bedingte Anweisungen mit Programmzweigen und Fallunterscheidungen zusammen?

Wozu dienen deklarierte Typen?

Was unterscheidet einen Compiler von einem Interpreter?

Was versteht man unter Quell- Ziel und Zwischencode? Der Bytecode ist ein Zwischencode. Er stellt noch nicht den Maschinencode eines existierenden physikalischen Prozessors dar. Er stellt die Maschinensprache für eine abstrakte Maschine, die Java Virtuelle Maschine, dar. Eine Instruktion ist ein Byte lang daher auch die Bezeichnung Bytecode für den Zwischencode.

Kontrollfragen - Skriptum - Kapitel 2

Welche Ähnlichkeiten und Unterschiede bestehen zwischen den Begriffen "Objekt", "Wert" und "Daten"?

Welche Möglichkeiten zur Steuerung des Programmflusses gibt es?

Was ist eine Sequenz, Fallunterscheidung und Iteration?

Was ist ein Ausdruck? Die Repräsentation eines Wertes.
Was ist eine Variable?

Was ist eine Zuweisung?

Was ist eine Speicheradresse?
An dieser Adresse wird der Wert der Variablen gespeichert. Man bezeichnet die Adresse auch als L-Wert.

Was ist der Unterschied zwischen Lebensdauer, Gültigkeitsbereich und Sichtbarkeitsbereich einer Variablen?

Lebensdauer: Eine Variable existiert nur im Zeitraum vom Reservieren ihres Speicherbereichs (Memory Allocation) bis zu dessen Freigabe (Memory Deallocation). So lebt eine lokale Variable einer Methode nur so lange die Methode ausgeführt wird.

Gültigkeitsbereich: Variablen sind im Allgemeinen nicht im gesamten Programm gültig. Der Gültigkeitsbereich bestimmt, von wo aus im Programm der Zugriff auf die Variable möglich sein kann. So sind lokale Variablen einer Methode nur innerhalb der Methode gültig. Dadurch sind Variablen nur während ihrer Lebensdauer zugreifbar.

Sichtbarkeitsbereich: Möglicherweise ist eine Variable an einer bestimmten Programmstelle zwar gültig, aber es darf trotzdem nicht darauf zugegriffen werden. Der Sichtbarkeitsbereich gibt an, von welchen Programmstellen aus ein Zugriff auf die Variable erlaubt ist. Der Sichtbarkeitsbereich kann im Vergleich zum Gültigkeitsbereich stärker eingeschränkt sein, etwa durch Deklaration als "private".

Was ist der Unterschied zwischen Deklaration und Definition?
Deklaration: Bevor eine Variable benutzt werden kann, muss sie deklariert werden. Bei der Deklaration wird der Name der Variablen und deren Typ vereinbart. Die Ausführung der Deklarationsanweisung bewirkt auch, dass der für die Variable benötigte Speicherplatz reserviert wird.
Definition:

Was ist eine Initialisierung?
Die Zuweisung eines Wertes.

Erfolgt die Initialisierung in Java automatisch?
nein.

Was sind Typen?

Welche elementaren Typen gibt es in Java?

Was sind Referenztypen?

Was ist ein Operator, was ist ein Operand?

Was heißt infix, präfix und postfix? infix: Zwischen zwei Operanden (3 + 2);
präfix: n = ++i;
postfix: n = i++;

Was ist ein L-Wert und was ein R-Wert?
L-Wert: Der Wert links einer Zuweisung
Analog dazu der R-Wert: der zuzweisende Wert auf der rechten Seite.

Was bewirkt der Modifier final auf Variablen?
mit final deklarierte Variablen kann man später nicht mehr verändern.

Was ist eine Ausdrucksanweisung?
Anweisungen sind Konstrukte, die Zustände von Programmen dynamisch ändern oder den Programmfluss steuern. Im Gegensatz zu Ausdrücken müssen Anweisungen keine Werte zurückgeben. Beispielsweise sind while-Schleifen Anweisungen, aber keine Ausdrücke. Es gibt auch Ausdrucksanweisungen, also Ausdrücke, die zugleich als Anweisungen verwendbar sind. Das bedeutet, sie liefern Werte und ändern zusätzlich den Programmzustand durch Ändern von Variablenwerten. Man sagt, diese Ausdrücke haben Seiteneffekte - also neben dem Haupteffekt des Zurückgebens von Werten noch zusätzliche Effekte. Der Begriff "Seiteneffekt" macht klar, dass der zusätzliche Effekt nur am Rande der Programmausführung passiert und seine Auswirkungen nicht immer leicht erkennbar sind. Das Ausnutzen von Seiteneffekten in komplexeren Ausdrücken wird daher als schlechter Programmierstil angesehen. Programme könnten unübersichtlich werden. Trotzdem beruhen Programmausführungen in imperativen Sprachen zu einem großen Teil auf Seiteneffekten. Ausdrucksanweisungen ermöglichen eine kurze Schreibweise. Beispiel: int y = x++; **In welcher Reihenfolge werden Ausdrücke in Java ausgewertet?** nach Operatorpriorität, bzw. nach Assoziativität, Stelligkeit und Priorität.
Was ist eine Operatorpriorität und wozu braucht man sie?

Was ist die Assoziaitivität und wozu braucht man sie für Operatoren?

Welche einstelligen und zweistelligen Operatoren gibt es in Java?

Gibt es in Java dreistellige Operatoren?
ja, den "ternären Auswahloperator" bzw. den ?-Operator.
Wie wird bei einer Division gerundet?

Was ist der Restwertoperator?
Modulo (%)
Was ist ein Verkettungsoperator?
+=, bzw. +
Wie verhalten sich Zuweisungsoperatoren?

Welche relationalen Operatoren gibt es?

Wie unterscheiden sich logische Operatoren von Bitoperatoren?

Was macht der Bedingungsoperator?

Was ist eine Typumwandlung?
casten.
Was passiert bei einer Typumwandlung?
ein Datentyp wird in einen anderen "gesteckt".

Wie unterscheidet sich die einschränkende von der erweiternden Typumwandlung?

Wann erfolgt eine implizite Typumwandlung, wann benötige ich eine explizite Typumwandlung? eine implizite Typumwandlung erfolgt immer dann, wenn ein kleinerer Datentyp in einen größeren Datentyp hineinpasst.
Was sind konstante Ausdrücke?

Was ist ein Literal?

Welche Fehler treten bei der Verwendung von Literalen auf?

Was ist ein Block?

Was bedeutet Überladen von Methoden?
Mehrere Methoden mit dem gleichen Namen, aber unterschiedlichen formalen Parametern.
Was bedeutet Rekursion?
Eine Methode, die sich selbst immer wieder aufruft, bis ein Wert ganz abgearbeitet wurde. (vgl. Schleifen).

25.1 Kontrollfragen - Kapitel 3

Wodurch unterscheidet sich der objektorientierte vom prozeduralen Programmierstil?

Was ist ein Objekt?

Wodurch unterscheiden sich Objektvariablen von lokalen Variablen?

Was bedeutet der Begriff Faktorisierung?

Was ist und wozu dient Kapselung, Data-Hiding und Datenabstraktion?

Was sind und wozu verwendet man Klassen?

Wie kann man in Java die Sichtbarkeit beeinflussen?

Wo sollen die meisten Objektvariablen sichtbar sein?

was haben Getter- und Setter-Methoden mit Data-Hiding zu tun?

Wie vergleicht man in Java Objekte auf Identität bzw. Gleichheit?

Wozu dient ein Konstruktor und wie definiert man ihn?

Wofür verwendet man die Pseudovariable this sowie Ausdrücke der Form this(...)?

Wie setzt man statische Methoden und Klassenvariablen ein?

Was unterscheidet Konstante von Klassenvariablen?

Wozu dienen Interfaces?

Was meint man, wenn man von der Implementierung eines Interfaces spricht?

Wann spricht man von Polymorphismus? Welche Rolle spielt der Polymorphismus in der objektorientierten Programmierung?
Ein Objekt kann durch Vererbung mehrere Typen haben.
Unter welchen Bedingungen ist ein Typ U Untertyp eines Typs T?

Wozu benötigt man dynamisches Binden?

Inwiefern hängt dynamisches Binden mit Mehrfachverzweigungen zusammen?

Warum ist dynamisches Binden gegenüber switch-Anweisungen zu bevorzugen?

Welchen Zweck haben Spezialisierungen und Analogien zur realen Welt?

Was besagt das Ersetzbarkeitsprinzip

Was versteht man unter Vererbung?

Erklären Sie die Begriffe Basisklasse, abgeleitete Klasse, Unterklasse und Oberklasse

Was ist eine überschriebene Methode?

Warum deklariert man Variablen nicht generell als protected?

Wie werden Objekte abgeleiteter Klassen initialisiert?

Wozu dient super(...) und wo kann man diese Anweisung verwendet werden?

Unterscheidet sich ein Methodenaufruf von einem Variablenzugriff hinsichtlich dynamischesm Binden?

Zu welchem Zweck kann man Klassen und Methoden mit einem Modifier ab-

stract bzw. final versehen?

Wie kann man durch Interfaces zusätzliche Struktur in ein Programm bringen?

Welche Methoden sind in Object vorhanden und welchen Zweck haben sie?

Wie kann man zur Laufzeit den dynamischen Typ, also die Klasse eines Objekts feststellen?

Was unterscheidet Casts auf Referenztypen von solchen auf elementaren Typen?

Wodurch unterscheiden sich die Pseudovariablen this und super voneinander? Was macht man mit ihnen?

Warum eignet sich hashCode nicht für Vergleiche von Objekten?

Welche Informationen soll in Komponenten von Klassen, Interfaces, Methoden, Konstruktoren und Objektvariablen enthalten sein?

Welche Arten von Zusicherungen in Form von Kommentaren kann man unterscheiden?

Inwiefern können Namen und Kommentare altern? Was kann man dagegen tun?

Wie können schlecht gewählte Namen und Kommentare zu unnötigem Programmcode führen?

Was zeichnet gut faktorisierte Programme aus?

Erklären Sie die Begriffe Klassenzusammenhalt und Objektkopplung. Wie hängen sie mit der Faktorisierung zusammen?

25.2 Kontrollfragen - Kapitel 4

Unter welchen Bedingungen sind zwei Algorithmen bzw. Datenstrukturen gleich? Wann sind sie es nicht?
Wie hoch ist der Aufwand für das Einfügen bzw. das Suchen in der verketteten Liste, im binären Baum sowie in der Hashtable im Durchschnitt und im schlechtesten Fall? Was ist der jeweils schlechteste Fall und wann tritt er ein?
Wie funktionieren Bubblesort, Mergesort und Quicksort? Wie hoch ist der Aufwand dafür im Durchschnitt und im schlechtesten Fall?
Was ist eine binäre Suche?

25.3 Kontrollfragen - Kapitel 5

Was versteht man unter Design-by-Contract?
Welche Bestandteile eines Softwarevertrags sind vom Client zu erfüllen, welche vom Server?

Woran erkennt man, ob eine Bedingung eine Vorbedingung, Nachbedingung oder Invariante darstellt?
Wie müssen sich Vor- und Nachbedingungen bzw. Invarianten in Unter- und Obertypen zueinander verhalten?
Warum stellen Invarianten einen Grund dafür dar, dass man keine public Variablen verwenden sollte?
Inwiefern lassen sich Bedingugen statt als Zusicherungen auch in Form von Typen ausdrücken?
Welche Rolle spielen Namen in Programmen (im Zusammenhang mit Zusicherungen?
Was versteht man unter der Namensgleichheit bzw. Strukturgleichheit?
Wozu verwendet man assert-Anweisungen?
Sind assert-Anweisungen auch sinnvoll, wenn deren Überprüfung ausgeschaltet ist, warum?
Was ist eine Schleifeninvariante und wozu verwendet man sie?
Wann muss eine Schleifeninvariante gelten?
Wie geht man vor, um Schleifeninvarianten zu finden?
Welche Formen von Zusicherungen ersetzen Schleifeninvarianten bei Verwendung von Rekursion statt Iteration?
Ist es besser beim Testen viele Fehler zu finden, als wenige? warum?
Was charakterisiert White- Grey- und Black-Box-Tests?
Was versteht man unter einer Ausnahmebehandlung?
Wie und warum fängt man Ausnahmen ab?