

FOP Reference Sheet

Frederick Wichert

Last Edited: 31. Mai 2020

Inhaltsverzeichnis

1	Definitionen	1
2	Computerspeicher	2
3	Datenstrukturen	2
4	Klassen und Objekte	3
5	Referenzen	4
6	Enumerationen	4
7	Methoden	5
8	Interfaces	6
9	Vererbung	6
10	Zugriffsrechte und Packages	7
11	Errors und Exceptions	7
12	Javadoc	9
13	JUnit-Tests	9
14	Wrapper	10
15	Generics, Wildcards und Comperator	11
16	Collections, Iterator	13
17	Eigene LinkedList Klasse	15
18	Optional	16
19	Streams und Files	17
20	Racket: Streams	19
21	Exkurs: Methodennamen als Lambda	19
22	Zahlendarstellung	20
23	Korrektheit von Software	21
24	Effizienz von Software	22

1 Definitionen

Scope	<ul style="list-style-type: none"> ▷ Sichtbarkeitsbereich von Definitionen und Identifiern - Variablen müssen erst geschrieben werden, bevor sie gelesen werden
Literale	▷ Literal (wörtlich) value einer Variable (5, "Hello", true, null, etc.)
Identifier	▷ Identifizieren Entitäten, alles selbsbenannte (Klassen, Variablen)
Konversion (Casting)	▷ Schreibt man einen Typen in Klammern vor einen Wert oder eine Variabel, ((int) new short(5)) wird der Typ konvertiert
Ternärer Operator	▷ BoolescherAusdruck ? IfValue : ElseValue
rvalue u. lvalue	<ul style="list-style-type: none"> ▷ Rechtsaudrücke (rvalue) haben Typ und Wert - z.B. Zuweisungen, Initialisierungen etc. ▷ Linksausdrücke (lvalue) verweisen auf Speicherstelle - z.B. Linke seite einer Zuweisung
Zuweisungen	<ul style="list-style-type: none"> ▷ Werden von rechts nach links abgearbeitet - z.B. <code>a = b += d *= f;</code> ▷ Durch das Semikolon wird aus dem Ausdruck eine Anweisung ▷ Zuweisungen haben Seiteneffekt ▷ <code>var1 operator= var2</code> ist gleich <code>var1 = var1 operator var2</code> - z.B. <code>a *= b;</code> ist gleich <code>a = a * b;</code>
Switch	<ul style="list-style-type: none"> ▷ Unterscheideds zwischen möglichen werten einer Variable <pre> switch (number) { case 3: System.out.println("It is prime"); break; case 5: System.out.println("It is prime as well"); break; default: System.out.println("This happens when neither case is true"); } </pre> <ul style="list-style-type: none"> ▷ Hinter case muss immer ein Literal stehen, und keine Variable ▷ Um Endlosschleifen zu vermeiden ist es sinnvoll <code>default:</code> zu implementieren ▷ Jeder case sollte mit einem <code>break;</code> beendet werden

```

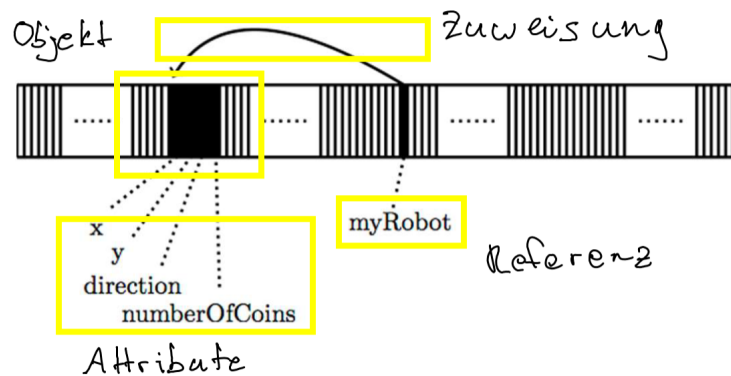
{ ... { ... double x; ... { ... { ... } ... } ... }

```

Abbildung 1: Scope von Variablen

2 Computerspeicher

Unsere Vorstellung	▷ großes Feld aus Maschinenwörtern mit eindeutiger Adresse
Primitive Datentypen	▷ Name verweist tatsächlich auf die Speicherstelle
Objekte im Computerspeicher	▷ Beim erzeugen wird ausreichend großer Speicherplatz reserviert
Attribute	▷ Werden relativ zur Anfangsadresse des Objektes gespeichert
Programm zu Prozess	▷ (Java-) Quellcode zu (Java-) Bytecode, dieser wird ausgeführt ▷ Das Ausführen nennt man Prozess ▷ Mehrere Prozesse laufen parallel ▷ Durch teilen der Prozessorkern Zeit laufen Prozesse quasi parallel
Abarbeitung der Anweisungen	▷ Programm Counter, speziell reserviert, enthält Speicheradresse der nächsten Anweisung ▷ Wird nach Ausführung erhöht, und verweist auf nächst Anweisung ▷ Bei Sprunganweisungen (Schleifen) wird er einfach überschrieben
Garbage Collector	▷ Der Garbage Collector ist teil des Laufzeitsystems ▷ Wird hin und wieder aufgerufen werden, kann aber optimiert werden ▷ Gibt alle nicht erreichbaren Speicherplatz frei



Robot myRobot = new Robot (3, 1, UP, 0);

3 Datenstrukturen

Zugriff	▷ Verschiedene Arten, z.B.: <code>int myNumber;</code> - Schreibend: <code>myNumber = 5;</code> - Lesend: <code>otherNumber = myNumber;</code> ▷ Referenz muss erst beschrieben werden um gelesen zu werden
Arrays	▷ Zum speichern Variablen selben Types ▷ Statische größe, nicht mehr veränderbar ▷ <code>Typ[] array = new Typ[n];</code> ▷ n gibt die Menge der Variablen an ▷ kann von Objekten wie von primitiven Datentypen erzeugt werden
X	▷ X

4 Klassen und Objekte

Klassen

Attribute	<ul style="list-style-type: none">▷ Eigenschaften einer Klasse, bzw. des Objektes▷ z.B. <code>privat int weight;</code>▷ z.B. <code>public String name;</code>▷ static Deklarierte Attribute heißen Klassenattribute<ul style="list-style-type: none">- Bei Objekterzeugung wird nicht immer eine Kopie erzeugt- Nur eine einzige Speicherstelle für die ganze Klasse▷ Zugriff über <code>myObject.attribute</code> oder über <code>MyClass.attribute</code>
Sichtbarkeit	<ul style="list-style-type: none">▷ Entweder public oder gar nichts▷ Nur eine public Klasse oder Interface pro Datei▷ Diese muss den Namen der Datei haben▷ Nicht public entspricht private
Konstruktor	<ul style="list-style-type: none">▷ Selber Name wie die Klasse, kein Rückgabewert▷ Hat keinen Rückgabewert, z.B. <code>public MyClass { ... }</code>▷ Falls kein Konstruktor definiert ist, wird Standardkonstruktor genutzt▷ kann natürlich auch überladen werden
Abstraktion	<ul style="list-style-type: none">▷ Klasse kann abstrakt sein, muss es aber sobald eine Methode abstrakt ist▷ Hat Klasse eine abstrakte Methode muss sie auch abstrakt definiert sein▷ z.B. <code>abstract public MyClass</code>▷ Keine Objekterzeugung möglich▷ Siehe <u>Methoden</u>▷ Bei Vererbung müssen nicht alle abstrakten Methoden implementiert werden▷ Siehe Vererbung
<code>java.lang.Object</code>	<ul style="list-style-type: none">▷ Erbt eine Klasse nicht, so erbt sie von <code>java.lang.Object</code>▷ <code>lang.Object</code> hat diverse Klassenmethoden<ul style="list-style-type: none">- z.B. <code>public boolean equals (Object obj) {...}</code>- z.B. <code>public String toString () {...}</code>
static initializer	<ul style="list-style-type: none">▷ Die besondere Static Methode: Kopf besteht nur aus dem Wort static<ul style="list-style-type: none">- <code>static { ... }</code>▷ Wird zu Beginn der Abarbeitung des Programms ausgeführt▷ z.B. zur Initialisierung von Konstanten etc.

Objekte

Erzeugung	<ul style="list-style-type: none">▷ Kann nur von <u>Klassen</u> erzeugt werden▷ Beim erstellen wird der Konstruktor (Siehe: <u>Klassen</u> mittels "new" aufgerufen▷ z.B. <code>MyClass myObject = new MyClass();</code>
Kopieren	<ul style="list-style-type: none">▷ Shallow vs Deep Copy<ul style="list-style-type: none">- Shallow: Kopieren der Referenzadresse- Deep: Kopieren der Werte in ein neues Objekt▷ Beim erstellen wird der Konstruktor (Siehe: <u>Klassen</u> mittels "new" aufgerufen▷ z.B. <code>MyClass myObject = new MyClass();</code>

5 Referenzen

Funktion	<ul style="list-style-type: none">▷ Verweist mittels einer Speicheradresse auf den Stack▷ Verwendet bei Objekten etc. Nicht bei primitiven Datentypen▷ Vergleich (Gleichheit: <code>==</code>) nur <code>true</code> bei gleicher Referenz, trotz "Wertgleichheit"
Verwendung	<ul style="list-style-type: none">▷ Referenztypen können quasi alles außer primitiven Datentypen sein▷ D.h. Klassen, Interfaces und Enumerationen▷ z.B. <code>MyClassIntfEnum reference = ... ;</code>▷ "Gefäße" um direkt oder indirekt abgeleitete Entitäten zu speichern▷ Wo der Referenztyp steht, können Subtypen verwendet werden
Statisch vs. Dynamisch	<ul style="list-style-type: none">▷ z.B. <code>MyStaticType sample = MyDynamicType();</code>▷ Zugriff nur auf Funktionen und Werte die in beiden Typen definiert sind, also nicht auf eine Funktion die nur in <code>MyDynamicType</code> zu finden ist, umgekehrt ist dies gar nicht möglich

6 Enumerationen

Funktion	<ul style="list-style-type: none">▷ Wird von <code>java.lang.Enum</code> abgeleitet▷ Kann benutzt werden um Spezielle Wertbezeichner zu erstellen▷ z.B. <code>public enum MyEnum {UP, DOWN, LEFT, RIGHT}</code>▷ Davon können Variablen mit entsprechendem Wert erstellt werden▷ Um nicht immer <code>MyEnum.VALUE</code> zu schreiben, <code>import static MyEnum</code>
Klassenmethoden	<ul style="list-style-type: none">▷ <code>MyEnumReference.values()</code><ul style="list-style-type: none">- liefert ein Array mit allen <code>MyEnum</code> Komponenten zurückin Definitionsreihenfolge▷ <code>MyEnumReference.name()</code><ul style="list-style-type: none">- liefert den Namen des Objektes zurück

7 Methoden

Aufbau	<ul style="list-style-type: none"> ▷ Bestehend aus Kopf (Head) und Rumpf (Body) ▷ <code>public static void main (String[] args) throws Exception</code> <ul style="list-style-type: none"> - bestehend aus dem Namen, einem Identifier (nur eine 'main' Methode) - dem Rückgabetyt, optional auch void - den Modifiern, public, static, private etc. (Verändern 'nur' die Nutzung) - der Parameterliste, spezifiziert durch den Typen gefolgt vom Namen - und die optionalen throws deklaration (Exception) ▷ Die Signatur muss bei jeder Methode unterschiedlich sein
Ausführung	<ul style="list-style-type: none"> ▷ Stackpointer verweist auf die nächste Anweisung ▷ Bei Aufruf einer Methode, Frame auf dem Callstack wird erzeugt ▷ Dieser bietet allen eventuell benötigten Speicherplatz bereit ▷ Sowie Verweis auf den Methoden beginn, und Rücksprungadresse ▷ Der Stackpointer verweist nun auf den Frame ▷ Nach Ausführung wird Stackpointer auf Rücksprungadresse gesetzt
Overloading	<ul style="list-style-type: none"> ▷ Methoden gleichen Namens, unterschiedlicher Parameterliste werden differenziert ▷ Siehe <u>Vererbung</u>
Abstraktion	<ul style="list-style-type: none"> ▷ Eine Methode kann abstrakt definiert sein ▷ Diese hat keinen Methodenrumpf ▷ z.B. <code>abstract public void myMethod</code> ▷ Siehe <u>Klassen</u>
Formal vs Aktualparameter	<ul style="list-style-type: none"> ▷ Die in dem Methodenkopf deklarierten Parameter nennt man Formalparameter ▷ Diese entsprechen quasi lokalen Variablen in der Funktion ▷ z.B. <code>public int sumFunction (int a, int b) {...}</code> ▷ a, b sind in diesem Fall Formalparameter ▷ Beim Aufruf werden sie durch Aktualparameter ersetzt ▷ z.B. <code>sumFunction(13, 37)</code>
Klassenmethoden	<ul style="list-style-type: none"> ▷ Mit <code>static</code> deklarierte Methoden heißen Klassenmethoden ▷ Klassenmethoden dürfen nicht: <ul style="list-style-type: none"> - Objektattribute lesen oder schreiben - Objektmethoden aufrufen ▷ Klassenmethoden dürfen: <ul style="list-style-type: none"> - Klassenattribute lesen oder schreiben - Klassenmethoden aufrufen ▷ Können ohne Objektinitialisierung aufgerufen werden
Variable Parameteranzahl	<ul style="list-style-type: none"> ▷ Wird durch drei Punkte nach dem Variablentyp deklariert ▷ z.B. <code>public static void m (char c, double... args)</code> <ul style="list-style-type: none"> <code>{ for (double x : args) { . code . } }</code> ▷ Aufruf kann auf verschieden Arten erfolgen <ul style="list-style-type: none"> - <code>X.m('c', new Array[])</code> - <code>X.m('a', 23.1, 321, 57.3)</code> - <code>X.m('b', 1337)</code>
Signatur	<ul style="list-style-type: none"> ▷ Die Signatur einer Methode besteht aus Name, und Parameterliste

	1. Richte int-Variable i ein und setze i auf 1
	2. Erhöhe den Stackpointer um den geeigneten Wert
int i = 1;	3. Berechne die Adresse der nächsten Anweisung (also von i++)
robot.move();	4. Schreibe diese an die Position R der Rücksprungadresse im neuen Frame
i++;	5. Kopiere die Adresse in robot an die dafür vorgesehene Stelle im neuen Frame
robot2.move();	6. Springe zum Code von Robot.move
i --;	7. Führe diesen Code aus
	8. Vermindere den Stackpointer um den geeigneten Wert
	9. Springe nach R
	10. Führe die Anweisung an R aus (i++)

Abbildung 2: FoP-Bot Methodenaufruf

8 Interfaces

Struktur	<ul style="list-style-type: none"> ▷ Sind ähnlich aufgebaut wie <u>Klassen</u> aber komplett abstrakt ▷ Implementierung wie folgt: <code>public MyClass implements MyInterface</code> ▷ Abstrakte Methoden werden nicht implementiert sondern nur definiert ▷ Alle Methoden müssen <code>public</code> sein
Funktion	<ul style="list-style-type: none"> ▷ z.B. <code>public MyClass implements MyInterface, Interface2</code> ▷ Alle Methoden müssen in <code>MyClass</code> implementiert werden ▷ Dient der Modularität und der Fehlervermeidung ▷ Wird bei Implementierung eine Methode nicht implementiert, ist diese, und somit die ganze Klasse abstrakt ▷ Eine Klasse kann mehrer Interfaces implementieren

9 Vererbung

Funktion	<ul style="list-style-type: none"> ▷ Reicht alle Methode und Funktionen weiter ▷ z.B. <code>public class MyHeritageClass extends MyClass</code> ▷ Eine Klasse kann maximal von einer anderen Klasse erben <ul style="list-style-type: none"> - also keine Mehrfachvererbung (Ausnahme Interfaces) ▷ Wo ein Referenztyp (Supertyp) erwartet wird kann ein Subtyp genutzt werden <ul style="list-style-type: none"> - aka. direkt oder indirekt abgeleitete Entitäten ▷ Exceptionklasse durch abgeleitete Exceptionklasse ersetzbar
Suberkonstruktor	<ul style="list-style-type: none"> ▷ Mit <code>super(x, y)</code> wird der Konstruktor der Superklasse gerufen ▷ Muss im Konstruktor vor allen anderen Anweisungen stehen ▷ Siehe <u>Klassen</u>
Overloading	<ul style="list-style-type: none"> ▷ Vererbte Methoden können mit anderer Parameterliste überschrieben werden ▷ Siehe <u>Methoden</u>
Overwrite	<ul style="list-style-type: none"> ▷ Vererbte Methoden können mit gleicher Paramterliste überschrieben werden ▷ Dies bedeutet die Signatur muss gleich bleiben
Interfaces	<ul style="list-style-type: none"> ▷ z.B. <code>public interface MyIntf extends Intf2, Intf3</code> - Beispiel von Mehrfachvererbung"

10 Zugriffsrechte und Packages

Packages	<ul style="list-style-type: none">▷ Ist eine Zusammenfassung von Klassen, Interfaces und/oder Enumerationen<ul style="list-style-type: none">- Normale Verzeichnisstruktur- In Subpackages unterteilt▷ Dient unter anderem der Vermeidung von Namenskonflikten▷ Deklariert mit <code>package mypackage</code>▷ Importiert mit <code>import package.subpackage.class</code> oder mit <code>import package.subpackage.*;</code> um alle Klassen in <code>subpackage</code> zu importieren weitere Subpackages werden nicht importiert▷ <code>java.lang.*</code> wird automatisch vom Compiler importiert
Zugriffsrechte	<ul style="list-style-type: none">▷ <code>private</code>, Nur in der Klasse selbst▷ <code>keine Angabe</code>, Zusätzlich zu (Package) <code>Private</code> - im ganzen Package sichtbar▷ <code>protected</code>, Zus. zu keine Angabe - in allen Abgeleiteten Klassen▷ <code>public</code>, Zus. zu <code>protected</code> überall wo die Klasse Importiert wird ▷ Bei Klassen o.ä. nur eine <code>public</code> Definition pro Quelldatei (Name der Datei)▷ Nur <code>public</code> oder gar nichts

11 Errors und Exceptions

Errors

Arten	<ul style="list-style-type: none">▷ Kompeilerzeitfehler (compile-time error)<ul style="list-style-type: none">- z.B. falsche Verwendung von Keywords oder Variablentyp- Klammerung falsch gesetzt▷ z.B. Laufzeitfehler (run-time errors)<ul style="list-style-type: none">- z.B. Division durch Null- Zugriff auf nicht existierende Objekte (null, Array out of bounds)
Laufzeitfehler	<ul style="list-style-type: none">▷ Kompeilerzeitfehler (compile-time error)<ul style="list-style-type: none">- z.B. falsche Verwendung von Keywords oder Variablentyp- Klammerung falsch gesetzt▷ aka. run-time errors▷ z.B. Division durch Null▷ Zugriff auf nicht existierende Objekte (null, Array out of bounds)▷ Kann nicht vom compiler entdeckt werden▷ Manche Fehler werden von der Sprache ausgeschlossen<ul style="list-style-type: none">- Nur Methoden aufruf des statischen Typs- Kein Objekte von abstrakten Klassen oder Interfaces
Throwable	<ul style="list-style-type: none">▷ Alle Exceptions und Errors sind von <code>Throwable</code> abgeleitet▷ Diese ermöglicht die Nutzung des <code>try / catch</code> Blocks▷ Errors sollten nicht abgefangen werden▷ Errors sind meist so schwerwiegend, dass recovern davon schwer Möglich ist▷ Kann aber in manchen Situationen trotzdem sinnvoll sein
AssertionError	<ul style="list-style-type: none">▷ Dienen zur Fehlerbehebung▷ Werden nicht abgefangen, da sie Error extenden▷ Geben Aussagekräftige Fehlermeldungen▷ z.B.:<pre>if (n < 0 n % 2 == 1) throw new AssertionError("Bad n!");</pre>▷ Ist äquivalent zu:<pre>assert n >= 0 && n % 2 == 0 : "Bad n!"</pre>▷ So eine Anweisung sicher zu (asserts), dass eine Bedingung erfüllt ist

Exceptions

Struktur	<ul style="list-style-type: none"> ▷ Exceptions dienen der Fehlerbehandlung während das Programm läuft ▷ Abgeleitet von <code>java.lang.Exception</code> ▷ Und abgeleitet von <code>Throwable</code> ▷ z.B.: <pre> 1 public int divide (int a, int b) throws Eception, Exception2 { 2 if (b = 0) { 3 throw new Exception("Divide by zero"); 4 return a / b; 5 } </pre> ▷ throws kündigt eine mögliche Exception an ▷ Exception ist hierbei der Typ der geworfenen Exception ▷ throw deklariert, dass hier eine Exception geworfen wird ▷ Danach wird <code>new Exception()</code> ein Objekt vom Typ <code>Exception</code> erstellt
Arten	<ul style="list-style-type: none"> ▷ <code>IndexOutOfBoundsException</code> <ul style="list-style-type: none"> - z.B. ungültiger Array Zugriff ▷ <code>IOException</code> <ul style="list-style-type: none"> - Input/Output Stream Exception ▷ <code>java.lang.RuntimeException</code> <ul style="list-style-type: none"> Muss nicht gefangen werden Alle Programmabbrüche kommen von nicht abgefangenen <code>RuntimeExceptions</code> Exisitiert damit nicht jede triviale Anweisung in <code>try / catch</code> stehen muss
Exception fangen	<ul style="list-style-type: none"> ▷ Während der Laufzeit können Exceptions gefangen werden ▷ z.B.: <pre> 1 int a = 4, b = 0; 2 try { 3 divide(a, b); 4 System.out.println(Reached this?"); 5 } 6 catch (Exception exc) { 7 System.out.println(exc.getMessage()); 8 } </pre> ▷ Wenn eine Methode eine Exception werfen kann, muss diese gefangen werden ▷ Entweder wird Methode in der Sie gerufen wird mit throws versehen ▷ Oder der aufruf geschiet in einem <code>try</code> Block ▷ Können mehrere Exception geworfen werden, können auch mehrere <code>catch</code> Blöcke hintereinander geschrieben werden
try / catch	<ul style="list-style-type: none"> ▷ Sobald eine Anweisung im <code>try</code> Block eine Exception wirft, wird zum <code>catch</code> Block gesprungen. ▷ Wird keine Exception geworfen wird <code>catch</code> Block übersprungen
try-with-ressources	<pre>try (Printer printer = ... ; Ress ress = ...) { ... }</pre> <ul style="list-style-type: none"> ▷ Der <code>try</code> Block kann auch mit Ressourcen ausgeführt werde ▷ Es muss darauf geachtet werden alle ressourcen
Laufzeitfehler	<ul style="list-style-type: none"> ▷

12 Javadoc

Methoden	<div>▷ Javadoc für Methoden sehen z.B. so aus:</div> <pre>/** * @param x the dividend * @param y the divisor, must not be zero * @throws ... * @return x integer-divided by y */ public int quotient (int x, int y) { return x / y; }</pre> <div>▷ xxx</div>
----------	---

13 JUnit-Tests

Struktur	<div>▷ Dienen der Fehlerbehandlung</div> <div>▷ Notwendige Import:</div> <pre>import static org.junit.Assert.assertEquals; import static org.junit.Assert.assertTrue; import org.junit.jupiter.api.Test; import org.junit.jupiter.api.BeforeEach; import static org.junit.Assert.assertThrows</pre> <div>▷ Steht in einer eigenen Quelldatei, dabei wird die zu testnde Datei importiert</div>
Beispiel	<div>▷ Beispiel Test:</div> <pre>1 @Test 2 public void testSomething() { 3 someVar = value; 4 assertEquals(anyValue, myFunction(someVar)); 5 assertTrue(anyCondition); 5 assertSame(object1, object2); //asserts same Objectaddress 6 }</pre> <div>▷ Beispiel BeforeEach:</div> <pre>1 @BeforeEach 2 public void testWheatherReady() { 3 assertTrue(ressource.isReady()); 4 assertThrows(myOtherFunction(ressource)); 4 }</pre> <div>assertNotEquals(x,y)</div>
Funktion	<div>▷ Wird eine JUnit Datei aufgerufen, werden alle @Test Methoden aufgerufen</div> <div>▷ Alle mit @BeforeEach versehenen Methoden werden, vor jeder Testmethode einmal ausgeführt.</div>
Notation	<div>▷ @Test - Deklariert eine Testmethode</div> <div>▷ @BeforeEach - Wird vor jedem Test ausgeführt</div> <div>▷ @BeforeAll - Wird zu beginn einmal ausgeführt</div>

14 Wrapper

Wrapper Klassen	<ul style="list-style-type: none">▷ Wrapper Klassen ümwickeln primitive Datentypen mit einer Klasse▷ Alle primitiven Datentypen können, mittels Konstruktors aufgerufen werden▷ Im Konstruktor steht immer der Wert des primitiven Datentyps<ul style="list-style-type: none">→ z.B. <code>Boolean bo = new Boolean(true);</code>→ z.B. <code>Integer in = new Integer(1337);</code>▷ Um auf den umhüllten Wert zuzugreifen gibt es eine Methode<ul style="list-style-type: none">→ z.B. <code>System.out.println(c.charValue());</code>
Boxing / Unboxing	<ul style="list-style-type: none">▷ Boxing/Unboxing bieten Annehmlichkeiten bezüglich primitiven Datentypen<ul style="list-style-type: none">→ <code>Integer i = 123;</code>▷ Die Zahl impliziert in diesem Fall die Erstellung eines Integer Objekts<ul style="list-style-type: none">→ <code>System.out.println(i);</code>▷ Hier wird ein <code>int</code> erwartet, <code>i.intValue()</code> wird impliziert

15 Generics, Wildcards und Comperator

Generics

Beispiel: Klasse	<p>▷ Beispiel einer generischen Klasse</p> <pre> 1 public class Pair <T1, T2> { 2 private T1 attribute1; 3 private T2 attribute2; 4 public T1 getAttribute() { 5 return attribute1; 6 } 7 public void setAttribute (T1 a1) { 8 attribute1 = a1; 9 } 10 }</pre>
Beispiel: Aufruf	<p>▷ Der Konstruktor der die beiden Attribute setzt ist impliziert</p> <p>▷ Ein Beispiel für ein Generisches Objekt:</p> <pre> 1 Integer i = new Integer(123); 2 Double d = new Double(3.14); 3 Pair<Integer, Double> pair = new Pair<Integer, Double>(i, d);</pre> <p>▷ Beim Konstruktor Aufruf müssen zusätzlich die Typen übergeben werden</p>
Funktion	<p>▷ Generics dienen als Platzhalter für Typen</p> <p>▷ In diesem Beispiel ist <code>Pair</code> generisch und mit <code>T1</code> und <code>T2</code> <i>parametisiert</i></p> <p>▷ <code>T1</code> und <code>T2</code> sind <i>Typparameter</i> der Klasse <code>Pair</code></p> <p>▷ Erst bei der Einrichtung von Variablen und Objekten der Klasse <code>Pair</code> werden <code>T1</code> und <code>T2</code> festgelegt</p> <p>▷ Man sagt, <code>Pair</code> ist mit <code>Integer</code> und <code>Double</code> <i>instanziiert</i></p> <p>▷ Ansonsten kann in der Klasse <code>T1</code> und <code>T2</code> wie ein Typ benutzt werden</p> <p>▷ Wie üblich sind überall auch Sybtypen möglich außer:</p> <p>→ Wenn eine Typinstanziierung festgelegt ist!!!</p> <p>▷ Eine Methode könnte zum Beispiel ein <code>Pair<X, Y></code> erwarten</p> <p>→ Hier ist keine Subklasse möglich</p> <p>\$\rightarrow\$ aka. nichtübertragbarkeit von Vererbung auf Typparameter</p> <p>▷ Klassenmethoden (<code>static</code>) können keine Generischen Typen verwenden</p>
Beispiel: Generische Methoden	<p>▷ Generische Methoden können ohne spezielle Parametrisierung funktionieren</p> <pre> 1 public class X { 2 public <T1, T2> Pair<T1, T2> makePair(T1 t1, T2 t2) { 3 return new Pair<T1, T2>(t1, t2); 4 } 5 }</pre> <p>▷ Aufruf:</p> <pre> 1 X x = new X(); 2 Pair<A, B> pair = x.makePair(new A(), new B());</pre>
Generische Methoden	<p>▷ Bei generischen Methoden wird der Kopf angepasst:</p> <p>▷ der Scope wird definiert <code>public</code></p> <p>▷ die Typparametrisierung <code><T1, T2></code></p> <p>▷ der Rückgabety <code>Pair<T1, T2></code></p> <p>▷ Funktionsname und Parameterlist <code>make Pair(T1 t1, T2 t2)</code></p> <p>▷ Die generischen Parameter müssen nicht spezifiziert werden</p> <p>▷ Der compiler erkennt selbst, welche das hier sein müssen, <code>A</code> und <code>B</code></p>
Typparamter	<p>▷ Typparameter können alle Klassen oder Interfaces sein</p> <p>▷ Also auch generische Klassen oder Interfaces</p> <p>▷ Was nicht geht sind primitive Datentypen</p> <p>→ Dafür muss die Wrapperklasse genutzt werden</p> <p>▷ Ausnahme: Arrays von primitiven Datentypen, diese sind wieder eine Klasse</p>

Eingeschränkte Typparameter	<ul style="list-style-type: none"> ▷ Anstatt jeden Typ zuzulassen kann dieser eingeschränkt werden ▷ <code>public class A <T extends X> { ... }</code> ▷ Dies geht auch mit Interfaces etc. ▷ <code>public class B <T extends Y & Intf1 & Intf2></code> ▷ Bemerkung: Interfaces können beliebig viele implementiert werden ▷ Aber, ist eine Klasse dabei, muss diese als erstes stehen
Einschränkungen bei Generics	<ul style="list-style-type: none"> ▷ Keine primitiven Datentypen als Instanziierung von Typparametern ▷ Keine Erzeugung von Objekten / Arrays von Typparametern mit <code>new</code> ▷ Keine Klassenattribute / Konstanten von Typparametern ▷ Kein Downcast oder instanceof mit Typparametern ▷ Kein throw-catch mit Typparametern ▷ Keine Methodenüberladung auf Typparametern

Wildcards

Beispiel: Wildcards	<ul style="list-style-type: none"> ▷ Beispiel einer Methode mit Wildcards <pre> 1 public class X <T> { 2 public T attribute; 3 } 4 5 public class A { 6 public double m (X<? extends Number> n) { 7 return n.attribute.doubleValue(); 8 } 9 }</pre>
Funktion	<ul style="list-style-type: none"> ▷ Eine Form der Typbeschränkung bei der <i>Instanziierung</i> ▷ Eine Möglichkeit wäre eine Methode generisch zu machen und den Typ mit extends zu beschränken ▷ Eine andere Möglichkeit sind Wildcards
Erklärung	<ul style="list-style-type: none"> ▷ Mit <code>X<? extends Number></code> wird sichergestellt, dass der Typ von X von Number erbt ▷ Sehr ähnlich zu generischen Methoden mit eingeschränktem Typparameter ▷ Aber: essentieller Unterschied → nicht generische ▷ Wird ein Fragezeichen alleine verwendet, ist es mit <code><? extends Object></code> gleichzusetzen ▷ → Gleiche Funktionalität mit <code><X super X></code> möglich
Empfehlung	<ul style="list-style-type: none"> ▷ In-Parameter → extends <ul style="list-style-type: none"> - Parameter deren Werte nur gelesen werden ▷ Out-Parameter → super <ul style="list-style-type: none"> - Parameter die beschrieben werden ▷ In/Out-Parameter → weder noch ▷ Rückgabe → weder noch

Comparator

Nutzen	<ul style="list-style-type: none"> ▷ Aus java.lang.util ▷ Ein Interface, dient zum Vergleichen ▷ Häufig bei Collections verwendet
Funktion	<ul style="list-style-type: none"> ▷ Das Interface hat eine Methode compare z.B.: <code>public int compare(T t1, T t2) { ... }</code> ▷ Diese liefert: <ul style="list-style-type: none"> - 0, wenn die Objekte äquivalent sind - -x, wenn $t1 < t2$, wenn der Wert von t1 dem von t2 vorangeht - +x, wenn $t1 > t2$, wenn der Wert von t1 dem von t2 nachfolgt

16 Collections, Iterator

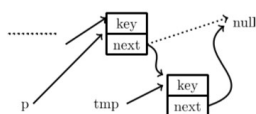
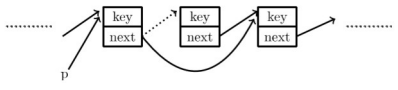
Collections

Nutzen	<ul style="list-style-type: none">▷ Collections sind Sammlungen von Elementen▷ Diese sind generische Klassen, und somit von jedem Objekt erzeugbar▷ Zentral ist das Interface <code>Collection</code> - ohne s▷ Dieses wird von der Klasse <code>Collections</code> - mit s - implementiert▷ Diese bietet noch weitere Funktionalitäten, z.B. sortieren▷ Alles aus <code>java.util</code>
Elemente	<ul style="list-style-type: none">▷ <code>List</code> - Ein Interface, dass <code>Collection</code> erweitert▷ <code>Iterator</code> - Ein Interface, dass zum iterieren dient▷ Beispiele für <code>Collections</code>:<ul style="list-style-type: none">- <code>Vector</code>- <code>LinkedList</code>- <code>ArrayList</code>- <code>TreeSet</code>- <code>HashSet</code>
Beispiel: <code>Collection</code>	<pre>1 Collection<Number> c1 = new ArrayList<Number>(); 2 Collection<Number> cc = new HashSet<Number>(); 3 Collection<Number> c1 = new Vector<Number>();</pre> <ul style="list-style-type: none">▷ Diese können wir nun mit beliebigen Numbers füllen - <code>Double</code>, <code>Integer</code>, <code>Float</code>
Methoden von <code>Collections</code>	<ul style="list-style-type: none">▷ <code>get</code> - gibt das Element an Index i zurück▷ <code>add</code> - fügt ein Element an's Ende der <code>Collection</code>▷ <code>addAll</code> - fügt alle Elemente der übergebenen <code>Collection</code> ein▷ <code>size</code> - gibt die aktuelle Anzahl an Elementen zurück▷ <code>isEmpty</code> - gibt <code>true</code> zurück wenn die <code>Collection</code> leer ist▷ <code>contains</code> - gibt <code>true</code> zurück wenn das übergeben Element in der <code>Collection</code> ist▷ <code>containsAll</code> - gibt <code>true</code> zurück, wenn <code>contains</code> <code>true</code> für alle übergeben Elemente <code>true</code> ist▷ <code>clear</code> - entfernt alle Elemente der <code>Collection</code>, <code>isEmpty</code> → <code>true</code>▷ <code>remove</code> - liefert <code>true</code> wie <code>contains</code> zurück und entfernt das Objekt
List: Methoden	<ul style="list-style-type: none">▷ <code>indexOf</code> - liefert den ersten Index, an dem das Objekt zu finden ist▷ <code>set</code> - setzt an index i ein objekt ein und gibt das vorherige Elemente zurück▷ <code>add</code> - überladene Methode, fügt ein Element an Index i ein Alle Elemente werden um 1 nach rechts verschoben
Sortieren mit <code>Comparator</code>	<ul style="list-style-type: none">▷ <code>Collections</code> können mit dem, bekannten Interface <code>Comparator</code> sortiert werden▷ Dafür wird im Idealfall eine Eigene <code>Comparator</code> Klasse erstellt▷ <code>Collections.sort(studentList, new EnrollmentNumberComparator());</code>

Iterator

Nutzen	<ul style="list-style-type: none">▷ Zu jeder Collection gibt es eine Iterator Klasse▷ Können einzeln über eine Collection iterieren▷ Dabei ist alles Iterator spezifisch
Beispiel: Iterator	<ul style="list-style-type: none">▷<pre>1 Iterator<Number> it1 = c1.iterator();</pre>▷<pre>2 while (it1.hasNext())</pre>▷<pre>3 System.out.println(it1.next().doubleValue());</pre>▷ Hiermit wird eine Iterator einer Collection erstellt und über diesen iteriert
Kurzform for-Schleife	<ul style="list-style-type: none">▷ Um Über Objekte die Iterierbar sind zu itereieren gibt es eien Kurzform▷ Dies ist nur Möglich wenn die Klasse vin <code>Iterable</code> erbt▷<pre>1 for String str : collec</pre>▷<pre>2 System.out.println(str);</pre>▷ Erste wird der Typ, dann der Name der lokalen Variable und dann das zu iterierende Objekt angegeben
Maps	<ul style="list-style-type: none">▷ Maps - wie funktionen - weisen einen Wert immer einem Key zu▷ Somit lassen sich gut Fukntionen modellieren▷ Dabei muss der Key von jedem Element einzigartig sein▷ z.B.:<pre>1 Map<String, X> map = new HashMap<String, X>();</pre>2 <pre>for (int i = 0; i < 100; i++) {</pre>3 <pre>String key = Nr. "+ i;</pre>4 <pre>X value = new X (2 * i + 3);</pre>5 <pre>map.put(key, value);</pre>6 <pre>}</pre>

17 Eigene LinkedList Klasse

Beispiel: Prinzip	<p>▷ Beispiel der LinkedList Klasse:</p> <pre> 1 public class ListItem <T> { 2 public T key; 3 public ListItem<T> next; 4 } </pre> <p>▷ Beispiel der Anwendung:</p> <pre> 1 ListItem<T> head = new ListItem<T>(); 2 head.next = new ListItem<T>(); </pre>
Prinzip	<p>▷ LinkedLists sind Objekte die einen Wert haben</p> <p>▷ Und einen Verweis auf das nächste Element der Liste</p> <p>▷ So wird einfach ein Objekt head erzeugt das wiederum auf das nächste, usw., Objekt verweist</p>
Iteration	<p>▷ Über eine LinkedList zu iterieren ist etwas anders</p> <p>▷ Dadurch, dass es keinen Index gibt, wird anders Iteriert</p> <p>▷ Dies kann folgendermaßen aussehen:</p> <pre> 1 for (ListItem<T> p = head; p != null; p = p.next) { 2 doSomething(p.key); 3 } </pre>
Entfernen und Einfügen	<p>▷ Ganz Wichtig: Am Anfang muss immer der Fall betrachtet werden, dass am ersten Element, head, etwas verändert werden soll</p> <p>▷ z.B. dass head = null ist</p>
Einfügen an Index	<p>◇</p> <p>Neues Element einfügen:</p>  <pre> tmp.next = p.next; p.next = tmp; </pre> <pre> if (pos == 0) { if (head == null) head = tmp; else { tmp.next = head; head = tmp; } return; } for (ListItem<T> p = head; p != null; p = p.next) { pos--; if (pos == 0) { tmp.next = p.next; p.next = tmp; return; } } throw new IndexOutOfBoundsException ("Wrong position: " + pos); </pre> <p>◇</p>
Entfernen nach Wert	<p>◇</p> <pre> public boolean remove (Object obj) { if (head == null) return false; if (obj.equals(head.key)) { head = head.next; return true; } for (ListItem<T> p = head; p.next != null; p = p.next) if (obj.equals(p.next.key)) { p.next = p.next.next; return true; } return false; } </pre> <p>Element entfernen:</p>  <pre> p.next = p.next.next; </pre> <p>◇</p>

18 Optional

Beispiel: Prinzip	<pre>1 Optional<Number> opt1 = Optional.ofNullable(new Integer(123)); 2 Optional<Number> opt2 = Optional.ofNullable(null); 3 Number n1 = opt1.get(); // n1 == 123 4 Number n2 = opt2.get(); // NoSuchElementException 5 Number n3 = opt1.orElseGet(() -> 0); 6 Number n4 = opt2.orElseGet(() -> 0);</pre>
Funktion	<ul style="list-style-type: none">▷ Aus dem Package <code>java.lang</code>▷ Kapselt ein Objekt ein, auch <code>null</code>▷ Relativ einfache Methode mit dem Wert <code>null</code> umzugehen▷ Die Methode <code>orElseGet</code> entspricht dem Sinn von <code>Optional</code>▷ Existiert das Objekt nicht, wird einfach etwas anderes zurückgegeben▷ Dies wird durch den Supplier, in dem Fall die Lambda Methode definiert▷ Diese gibt bei uns Konstant den <code>int 0</code> zurück▷ Der Rückgabebetyp muss von dem selben Typparametern sein wie das <code>Optional</code>▷ <code>Optional</code> kann auch ein anderer Lambda Ausdruck benutzt werden für Funktionen <code>ifPresent</code>, <code>map</code> und <code>filter</code>
Maps	as

19 Streams und Files

Streams

Beispiel: Streams	<pre> 1 List<Number> list = new LinkedList<Number>(); 2 for(;;i<100;) { list.add(new Integer(4 + 2*i)) } 3 Stream<Number> stream1 = list.stream(); 4 Stream<Number> stream2 = stream1.filter(myPred); 5 Stream<Number> stream3 = stream2.map(myFuc) 6 Optional<Number> opt = stream3.max(new MyNumberComparator); </pre>	
Funktion	<ul style="list-style-type: none"> ▷ Generische Klasse Stream ▷ Einfache Schnittstelle von Listen, Arrays, Dateien als Sequenzen ▷ <code>list.stream()</code> wandelt eine Liste in einen Stream um ▷ Genau so kann aus einem Array ein Stream geamcht werden <pre>Stream<Number> str1 = Arrays.stream(arr);</pre> ▷ Oder direkt der Inhalt spezifiziert werden <pre>Stream<Number> str2 = Stream<Number>.of(new Integer(12), ...);</pre> ▷ Natürlich auch iterieren mit Iterator möglich 	
Stream zu...	<ul style="list-style-type: none"> ▷ Stream zu List: <pre>List<String> list = stream.collect(Collectors.toList());</pre> ▷ Stream zu Array: <pre>Number[] a = stream.toArray(Number[]::new);</pre> ▷ Daher heben sich diese Methoden auf: <pre>Arrays.stream(arr).stream().toArray(Number[]::new)</pre> 	
IntStream LongStream DoubleStream	<ul style="list-style-type: none"> ▷ Da primitive Datentypen und Generizität nicht kompatibel sind gibt es bei Streams einen Workaround ▷ So gibt es Streams extra für Primitve Datentypen 	
Random Zahlen und Streams	<ul style="list-style-type: none"> ▷ So kann man eine Zufällige Zahl erzeugen <pre> 1 Random random = new Random(); 2 Double x = random.nextDouble(); </pre> ▷ Oder man kann einen Stream von Zufälligen Zahlen erzeugen <pre>1 IntStream str1 = random.ints();</pre> 	
System Streams	<ul style="list-style-type: none"> ▷ Die Klasse <code>System</code> besitzt einige Streams: <ul style="list-style-type: none"> - <code>System.in</code> // Standardmäßig die Tastatur - <code>System.out</code> // Die Konsole - <code>System.err</code> // Fehlerdatie mit allen Errors ▷ Diese können gesetzt werden: <ul style="list-style-type: none"> - <code>System.setIn()</code> - <code>System.setOut()</code> - <code>System.setErr()</code> 	
Arten von Streams	<ul style="list-style-type: none"> ▷ Im Allgemeinen Bauen alle aufeinander auf: ▷ <code>BufferedReader(FileReader(Path))</code> ▷ <code>PrintStream(BufferedOutputStream(FileOutputStream(Path)))</code> ◇ ▷ Für Bytedaten oder ähnliches: <ul style="list-style-type: none"> - <code>BufferedInputStream</code> - <code>BufferedOutputStream</code> - <code>FileOutputStream</code> - <code>PrintStream</code> ▷ Für andere Dateien: <ul style="list-style-type: none"> - <code>Reader</code> // Als Superklasse - <code>FileReader</code> - <code>BufferedReader</code> // Bekommt einen FileReade im Konstruktor - <code>InputStreamReader</code> - <code>BufferedWriter</code> // Bekommt einen FileWriter im Konstruktor - <code>FileWriter</code> 	

Files

Systemproperties	<p>▷ Die Methode <code>System.getProperty()</code> bekommt einen Key und gibt einen String zurück</p> <p>▷ Beispiele:</p> <pre>System.getProperty(über.home") → Gibt den Heimatordner an, bei uns oft der Benutzer Ordner</pre> <pre>System.getProperty(über.dir") → Gibt den Ordner des Programmes an</pre> <pre>System.getProperty(über.name") → Gibt den Benutzer an, der das Programm ausführt</pre> <pre>System.getProperty(file.separator") → Gibt den Systemspezifischen Programmpfad Teiler an Bei Windows oft "\", bei UNIX eher "/"</pre> <pre>System.getProperty(line.separator") → Gibt den Systemspezifischen Zeilenumbruch an</pre>
Path und Paths	<p>▷ Path und Paths dienen dazu Dateipfade zu speichern und zu benutzen</p> <pre>1 String homeDir = System.getProperty(über.home"); 2 Path path = Paths.get(homeDir, fop", streams.txt");</pre> <p>▷ Die Methode <code>get</code> der Klasse <code>Paths</code> bekommt Strings übergeben und setzt daraus einen Dateipfad abhängig vom System, zusammen</p> <p>→ <code>C:\Users\Home\fop\streams.txt</code></p>
Lesen von Dateien	<p>▷ Es gibt verschiedene Arten Dateien zu lesen</p> <p>▷ z.B.:</p> <pre>3 try (Stream<String> stream = Files.lines(path)) { 4 String fileContent = stream.reduce(String::concat); }</pre> <p>▷ Die Methode <code>reduce</code> erstellt aus allen Teilen eines Streams eine Datei</p> <p>▷ Somit kann der Text einer Datei eingelesen werden</p>
Klasse Files Beispiele	<p>▷ Dies sind ein paar selbsterklärende Methoden:</p> <pre>Path paths = Paths.get(...); if (Files.exist(path)) if (Files.isReadable(path)) if (Files.isWritable(path)) if (Files.isRegularFile(path)) if (Files.isDirectory(path)) long size = File.size(path)</pre>
Klasse Files Methoden	<p>▷ Erstellt eine Datei an dem Pfad:</p> <pre>Files.createFile(path);</pre> <p>▷ Kopiert eine Datei:</p> <pre>Files.copy(path);</pre> <p>▷ Verschiebt oder benennt eine Datei um:</p> <pre>Files.move(path1, path2);</pre> <p>▷ Löscht eine Datei:</p> <pre>Files.delete(path);</pre> <p>▷ Löscht eine Datei, wenn eine vorhanden ist:</p> <pre>Files.deleteIfExists(path);</pre>

20 Racket: Streams

Funktion	<ul style="list-style-type: none"> ▷ Streams in Racket funktionieren ähnlich wie in Java ▷ Leider nicht in DrRacket enthalten
Beispiel Metehoden	<ul style="list-style-type: none"> ▷ Viele Methoden funktionieren wie gewohnt: <ul style="list-style-type: none"> (stream-cons x str) (stream-first str) (stream-rest str) (stream-empty? str) (stream-map fct str) (stream-filter pred str) (stream-fold init fct str)

21 Exkurs: Methodennamen als Lambda

Funktion	<ul style="list-style-type: none"> ▷ Etwas omplizierte Angelegenheit ▷ Kurz gefasst, können Methoden als ersatz für einen Lambda ausdruck stehen ▷ Oder auch eine Interface Methode ersetzen ▷ Siehe Scope Identifiert aus C++
Beispiel: Gleiche Ausdrücke	<pre> ◇ public interface DoubleConsumer { void accept (double x); } public class IntPrinter implements DoubleConsumer { public void accept (double x) { System.out.print(x); } } DoubleConsumer cons1 = new IntPrinter(); DoubleConsumer cons2 = x -> { System.out.print(x); } DoubleConsumer cons3 = System.out::print; </pre> <p>◇</p> <ul style="list-style-type: none"> ▷ <code>System.out</code> gibt hier den scope der Methode an ▷ <code>print</code> ist in dem Fall die Methode
Erklärung	<pre> ◇ public interface StringToStringFunction { String applyAsString (String str); } public interface StringSupplier { String get (); } StringToStringFunction fct1 = String::new; StringToStringFunction fct2 = X::allUpperCase; StringSupplier sup1 = String::new; StringSupplier sup2 = Y::createHelloWorld; public class X { public String allUpperCase (String str) { return str.toUpperCase(); } } StringToStringFunction fct2 = X::allUpperCase; public class Y { public String createHelloWorld () { return new String ("Hello World"); } } StringSupplier sup2 = Y::createHelloWorld; </pre> <p>◇</p> <ul style="list-style-type: none"> ▷ Der Compiler erkennt, dass die Methode auf die des Interfaces passt und können so Interfaces ersetzt ▷ So kann auch das Interface <code>Function</code> ersetzt werden

22 Zahlendarstellung

Bitanzahl	<ul style="list-style-type: none"> ▷ byte 8 -128 +127 ▷ short 16 -32.768 +32.767 ▷ int 32 -2.147.483.648 +2.147.483.647 ▷ long 32 -9.223.372.036.854.775.808 +9.223.372.036.854.775.807
Informationen	<ul style="list-style-type: none"> ▷ Die Binär Kodierung einer Zahl nennt man "Bitmuster" ▷ Jede Ziffer ist als char ist behält ihre Bitsignatur, zusätzlich werden noch 48 dazu addiert. ▷ Bei Berechnungen muss dies natürlich wieder abgezogen werden
2-Komplement	<ul style="list-style-type: none"> ▷ Um eine Zahl negativ darzustellen: ▷ Positive Bitsignatur invertieren, und 1 addieren ◇ 5: 0101 // negieren ◇ x: 1010 // und eins addieren ◇ -5: 1011
Gebrochene Zahlen	<ul style="list-style-type: none"> ▷ float-Genauigkeit: 1 zu 8.388.608 ▷ double-Genauigkeit: ca. 1 bis 4,5 Milliarden → Trotzdem Probleme mit der Genauigkeit Umkehrrechnung liefert möglicherweise nicht das richtige Ergebnis Bei Addition und Subtraktion ungenauigkeit: z.B.: Subtrahieren zweier sehr großen Zahlen, kann ungenau werden

23 Korrektheit von Software

Korrektheit auf einzelnen Abstraktionsebenen

Lexikalische Ebene	<ul style="list-style-type: none">▷ Rechtschreibung Nicht korrekte Schreibweise von Keywords▷ Regeln für Identifier sind festgelegt:<ul style="list-style-type: none"><code>identifier ::= «letter» «ident-char-list»</code><code>ident-char-list ::= ε «ident-char» «ident-char-list»</code><code>ident-char ::= «letter» «digit» _ \$</code><code>letter ::= a...z A...Z</code><code>digit ::= 0...9</code>
Syntaktische Ebene	<ul style="list-style-type: none">▷ Ähnlich zur Grammatik▷ Normalerweise werden diese vom Compiler angefangen▷ die Regeln nach denen zu Entscheiden ist ob ein Quelltext ein korrektes Programm in der Sprache ist<ul style="list-style-type: none">- Verstöße gegen kontextfreie Regeln▷ Typische Fehler enthalten z.B. Klammersetzung:<ul style="list-style-type: none">◊ Zu jeder öffnenden Klammer gehört eine folgende schließende Klammer◊ Niemals überlappende Klammern▷ Syntaktische Konstrukte:<ul style="list-style-type: none">◊ Beinhaltet z.B. Köpfe bekannter Schleifen:◊ <code>for (...;...;...), while (...), do .. while (...)</code>▷ Auch hier ist eine Formalisierung der Regeln möglich:<ul style="list-style-type: none">◊ <code>statement ::= «compund-statement» «if-statement» ...</code>◊ <code>compund-statement ::= { «statement-sequence» }</code>◊ <code>statement-sequence ::= ε «statement» «statement-sequence»</code>
Semantische Ebene	<ul style="list-style-type: none">▷ Werden in der Regel nicht vom Compiler gefunden▷ was ein gegebenes Programm tatsächlich macht▷ z.B. Runtime Exceptions▷ Typische Fehler:<ul style="list-style-type: none">◊ Teilen durch Null◊ Falsche nutzung des Array Index◊ Zugriff auf ein nicht existierendes Objekt
Logische Ebene	<ul style="list-style-type: none">▷ Logische Fehler sind umsetzungs Fehler▷ Man weis was das Programm tun soll, setzt es aber nicht richtig um▷ Typische Fehler:<ul style="list-style-type: none">◊ off-by-one-error - Mathematische um 1 verrechnet
Spezifikatorische Ebene	<ul style="list-style-type: none">▷ Bereits der umzusetzende Gedanke war Falsch▷ Beispiel 2000-Fehler:<ul style="list-style-type: none">◊ Nicht erwartet, dass Programma bis und nach 2000 im Betrieb sind◊ Daher Jahreszaahl codierung nur mit 2 Ziffer statt 4

Korrektheit von Software

Überblick	<ul style="list-style-type: none"> ▷ Kein Programmabbruch durch Fehler ▷ Termination <ul style="list-style-type: none"> ◊ wenn Aufgabe erledigt oder ◊ wenn Befehl zur Termination von außen ▷ Korrekte Ausgaben und Effekte
Korrektheit von Klasse	<ul style="list-style-type: none"> ▷ Darstellungsinvariante (representation invariant) von Klassen u. Interfaces <ul style="list-style-type: none"> ◊ Beschreibt Darstellung der Objekte gegenüber Nutzer und Klasse ◊ Die Sicht, die Attribute und Methoden vermitteln, die public sind ▷ Implementationsinvariante (implementation invariant) von Klassen <ul style="list-style-type: none"> ◊ Analog zur Darstellungsinvariante ◊ Behandelt den Teil der Klassendefinition, der nicht public ist ▷ Liskov Substitution Principle (LSP): <ul style="list-style-type: none"> ◊ Jede Aussage über das logische Verhalten der Basisklasse, muss auch für die abgeleitete Klasse gelten. ▷ Zusammenfassend: <ul style="list-style-type: none"> ◊ Die Darstellungsinvariante und, soweit es die protected Attribute betrifft, auch die Implementationsinvariante müssen immer eingehalten werden
Korrektheit von Subroutinen	<ul style="list-style-type: none"> ▷ Subroutine als Oberbegriffe für Methoden und Funktionen ▷ Zweiter Teil des LSP: <ul style="list-style-type: none"> ◊ Vorbedingung darf nur abgeschwächt, nicht verschärft oder ersetzt ◊ Nachbedingung darf nur verschärft, nicht abgeschwächt oder ersetzt
Korrektheit von rekursiven Subroutinen	<ul style="list-style-type: none"> ▷ Rekursionsabbruch <ul style="list-style-type: none"> ◊ Muss vorhanden sein, damit Rekursion ordentlich terminiert ▷ Rekursionsschritt <ul style="list-style-type: none"> ◊ Schritt näher an den Rekursionsabbruch ▷ Beweises der Korrektheit mittels Induktion <ul style="list-style-type: none"> ◊ Induktionsbehauptung: Aufstellung für Problemgröße ◊ Induktionsanfang: z.B. Problemgröße = 1 ◊ Induktionsvoraussetzung: Der Vertrag gelte für ... ◊ Induktionsschritt: z.B. Verringerung der Listenlänge
Korrektheit in Subroutine: Schleifen	<ul style="list-style-type: none"> ▷ Schleifeninvariante: Aussagen darüber, was sich während Schleife nicht ändert <ul style="list-style-type: none"> ◊ Formulierung Nach $h \geq 0$ Schritten ist ..." ◊ Verwendung einer Variable h die nicht im Code vorkommt ▷ Schleifenvariante: Aussagen darüber, was sich während der Schleife ändert <ul style="list-style-type: none"> ◊ Formulierung: for: "h steigt um 1" → Zusammenfassung: ◊ Formulierung: Nach Schleifenende ..." ▷ Induktion bei Schleifen: <ul style="list-style-type: none"> ◊ Invariante = Induktionsbehauptung: Nach $h \geq 0$ Schleifendurchläufen gilt.." ◊ Induktionsanfang, also $h=0$: "Die Initialisierung vor der Schleife sorgt dafür, dass die Invariante unmittelbar vor dem ersten Durchlauf erfüllt ist." ◊ Induktionsvoraussetzung für $h>0$: "Die Invariante gelte für $h-1$." ◊ Induktionsschritt: Unter Voraussetzung, dass..., nach h Durchläufen gilt."

24 Effizienz von Software

Nebenaspekte der Effizienz

Lexikalische Ebene	▷ Rec
Syntaktische Ebene	▷ Ähnlich
Semantische Ebene	▷ Werden
Logische Ebene	▷ Logische
Spezifikatorische Ebene	▷ Bereits der um

Hauptaspekte der Effizienz

Lexikalische Ebene	▷ Rec
Syntaktische Ebene	▷ Ähnlich
Semantische Ebene	▷ Werden
Logische Ebene	▷ Logische
Spezifikatorische Ebene	▷ Bereits der um