Zusammenfassung - Programmieren

10 October 2014 11:44

Version: 1.2.0
Studium: 1. Semester, Bachelor in Wirtschaftsinformatik
Schule: Hochschule Luzern - Wirtschaft
Author: Janik von Rotz (http://janikvonrotz.ch)

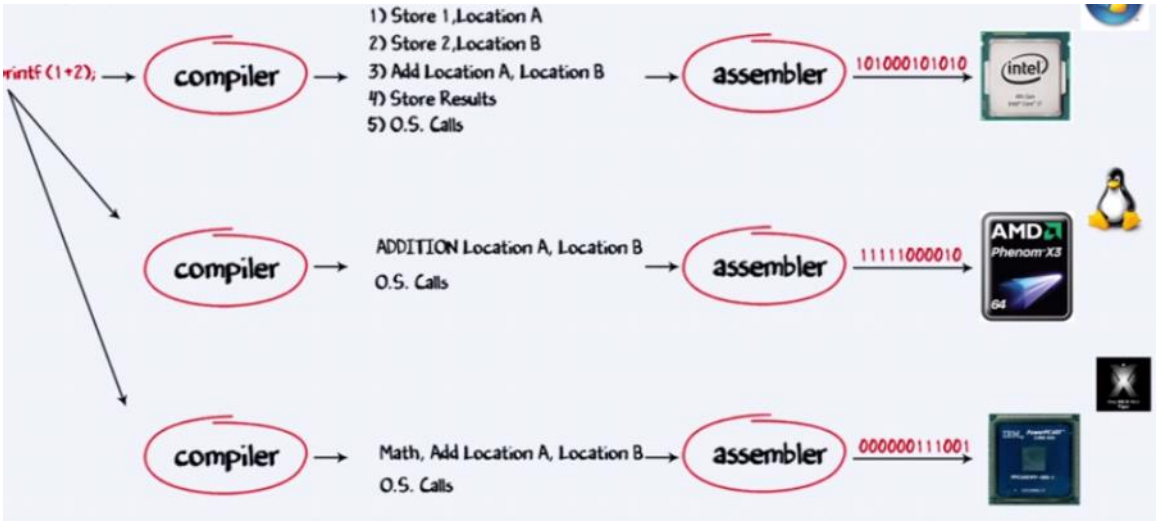
Java Language

05 November 2014 19:10

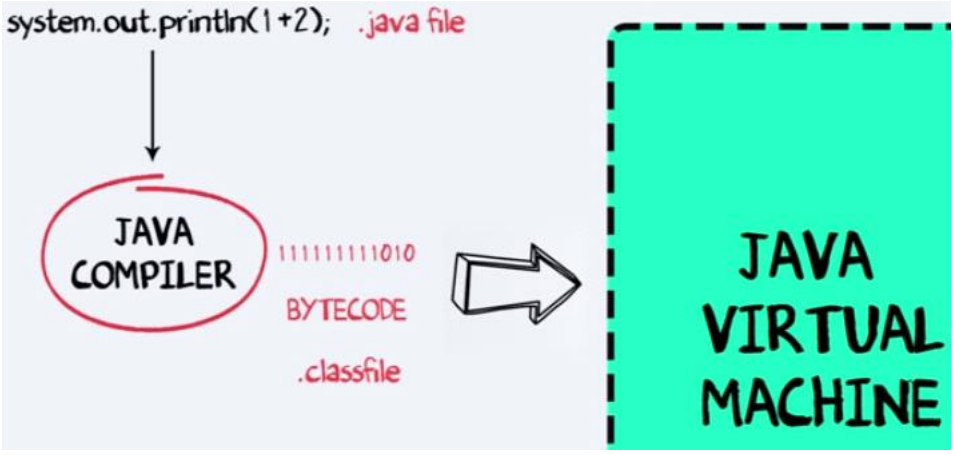
Platform

Operating System + Processor = Platform, bsp. Windows + Intel = WinTel

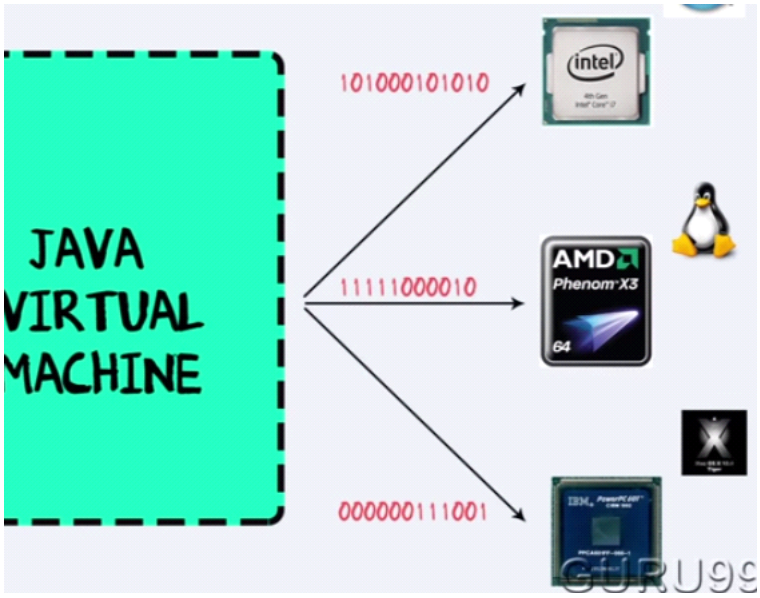
Um zu verhindern, das für jede Plattform einen separaten Compiler gekauft werden muss…



..gibt es die Java Virtual Machine, welche den Bytecode vom Java Compiler aufnimmt…



.. identifiziert mit welcher Plattform es läuft und dann den Bytecode in die richtige Maschinensprache umwandelt:



Heisst: Sobald Java einmal kompiliert ist, läuft es auf jeder Hardware, welche über die JVM verfügt!

Deshalb der Satz: „Java ist nicht nur eine Sprache, sondern auch eine Plattform!“

Geschichte

Stammbaum Java: Algol (1960) -> C (1970)

Direkte Vorfahren: Smaltalk-80 -> Objective C -> C++ -> Java

Java Basic

25 October 2014 18:35

Programmiersprachen

Eine Sprache wird definiert durch Semantik und Syntax.

Bezeichner (=Identifier)

Das sind Dinge für die im Programm Code ein Namen gewählt werden muss. Z.B. ein Variablennamen.

Ein Bezeichner startet mit einem UnicodeBuchstaben ('A'-'Z','a'-'z','_' und '$'), gefolgt von keinem oder mehreren Unicode-Zeichen.

- Bezeichner dürfen keine Schlüsselwörter sein.
- Der Unicode-Zeichensatz kann auch Buchstaben aus anderen Landessprachen verwenden (z.B. ¥€$)
- Es wird streng zwischen Gross und Kleinbuchstaben unterschieden (Product ≠ product)

Die Java Schlüsselwörter:

abstract	continue	for	new	switch
assert	default	if	package	synchronized
boolean	do	goto	private	this
break	double	implements	protected	throw
byte	else	import	public	throws
case	enum	instanceof	return	transient
catch	extends	int	short	try
char	final	interface	static	void
class	finally	long	strictfp	volatile
const	float	native	super	while

Escape Sequenzen

Nur rot Markierte.

Symbol	Unicode-Escape	Bedeutung
\b	\u0008	BS (backspace)
\t	\u0009	HT (horizontal tab)
\n	\u000a	LF (line feed)
\f	\u000c	FF (form feed)
\r	\u000d	CR (carriage return
\"	\u0022	" (double quote)

\"	\u0022	" (double quote)
\'	\u0027	' (single quote)
\\	\u005c	\ (backslah)

Literale

Ein Literal ist die Sourcecode-Repräsentation eines Wertes eines Primitiven Typs (“Zahl”, Boolean, Zeichen…), eines String Typs oder des Null-Typs

- Ein Literal ist ein Wert der sich nicht ändert.
- Ein Bezeichner „stellt einen Wert dar“ - ein Literal ist der Wert selber.

Beispiel:

```
age = 35
```

35 ist ein Literal
age ist ein Bezeichner (hier eine Variable), die z.B. den Wert 35 haben kann, wenn dieser Wert dem Bezeichner zugewiesen wurde!

- **Ganzzahl-Literal** `int z = 3428;` `//3428`
- **Fliesskomma-Literal** `double f = 123,45;` `//123,45`
- **Boolsche-Literale** `true, false`
- **Zeichen-Literale** `´a´, ´7´`
  - **Escape-Sequenzen** `\n, \", \\`
- **String-Literal** `String str = "Test";`

Seperators & Operators

Separators
Nine ASCII characters are the *separators* (punctuators).
Separator: one of
( ) { } [ ] ; , .

Operators
37 tokens are the *operators*, formed from ASCII characters.
Operator: one of
= > < ! ~ ? :
== <= >= != && || ++ --
+ - * / & | ^ % << >> >>>
+= -= *= /= &= |= ^= %= <<= >>= >>>=

Statement

Elementare Operationen, die ein Programm ausführen muss, werden Anweisungen genannt.

Beispiel:

```
Product aProduct = WarehouseFactory.newProduct();
```

Expression (Ausdruck)

Elementare Operationen, die ein Programm ausführen muss, werden Anweisungen genannt.

Beispiele:

```
    aRack.showNews(product.getWeight());
```

z.B. «Wert»: 15.5

Ebenfalls übliche mathematische Ausdrücke:

```
a + b
14 * 2
```

Code-Blöcke

Bei einem Code-Block handelt es sich um eine Gruppe von Anweisungen, die als Einheit funktionieren. Java umschliesst CodeBlöcke mit geschweiften Klammern ({ und }).
Code-Blöcke sind beispielsweise

- Klassendefinitionen
- loop-Anweisungen
- bedingte Anweisungen
- Exception-Anweisungen
- Methoden

```
public getValue() {
try {jbInit();}
catch (Exception e) {e.printAccountValue();} }
```

Compiler

05 November 2014 19:16

Auch „Übersetzer“ genannt, wandeln menschenverständliche Programmiersprachen (Java) in Maschinen ausführbaren Programmcode um.
In Java gibt es noch einen weiteren Zwischenschritt: Erst wird der Programmcode in Bytecode umgewandelt und dann von einem Just-in-time Compiler (JIT Compiler) während der Programmausführung in Maschinencode übersetzt.

Compiler – Frontend

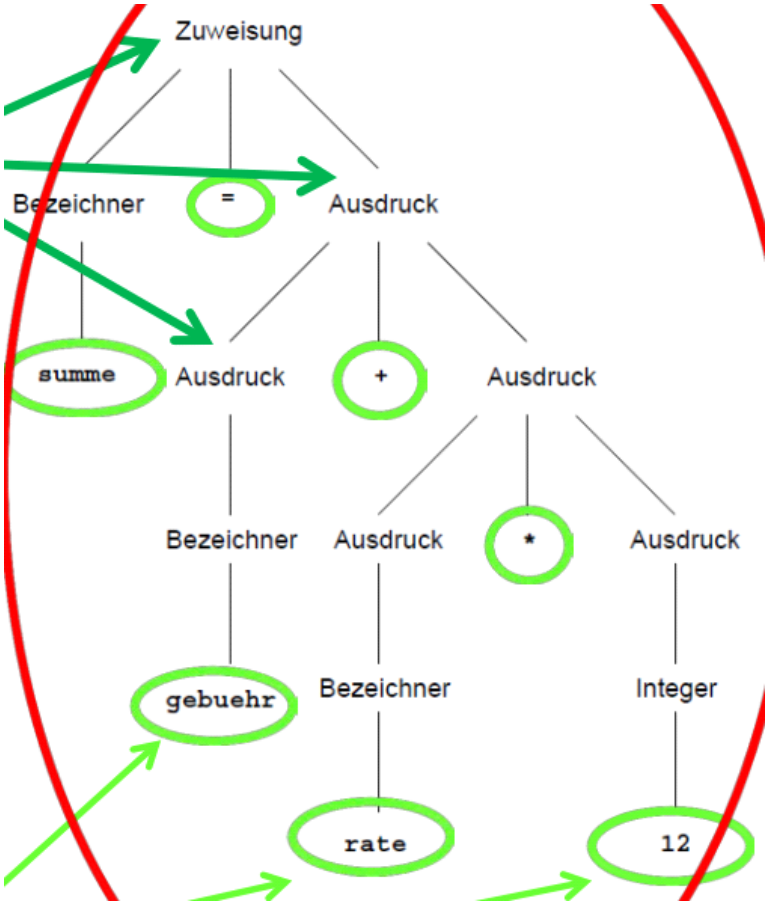
Auch Analysephase genannt, besteht aus folgenden drei Schritten.

Lexikalische Analyse (Scanner, Lexer)
Zerteilt die Anweisungen in Tokens.
summe = gebuehr + rate * 12;

wie folgt auf:
Bezeichner: **summe**, Operator: **=,** Bezeichner: **gebuehr**,
Operator: **+**, Bezeichner: **rate**, Operator: *****,
Ganzzahl-Literal: **12**, Trenner: **;**

Syntaktische Analyse (Parsing)
Programmiersprachenabhängige Regeln werden überprüft.

- Symbole des Textes werden zu grammatikalischen Einheiten zusammengefasst.
- Diese Zusammenfassung wird durch Syntaxregeln bestimmt.
- Dies kann durch einen Syntaxbaum veranschaulicht werden.



Semantische Analyse

Der Text wird auf semantische Fehler hin untersucht.
Beispiele:

- Verwendung von nicht vorher definierten Bezeichnern
- Typüberprüfung bei Zuweisungen

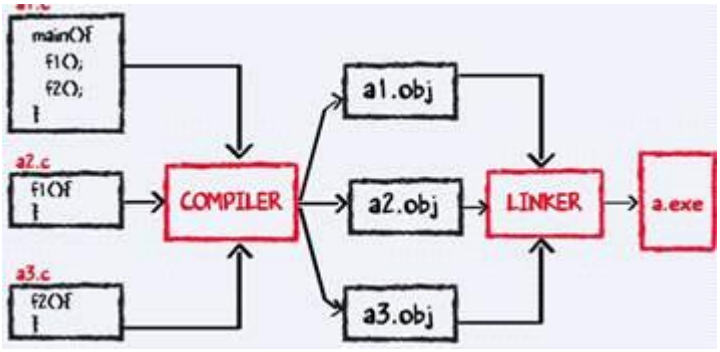
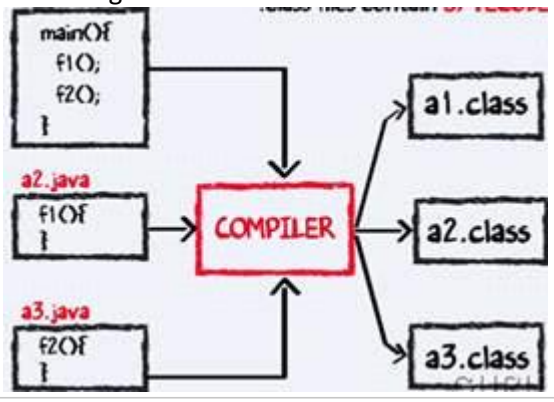
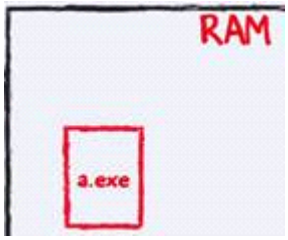
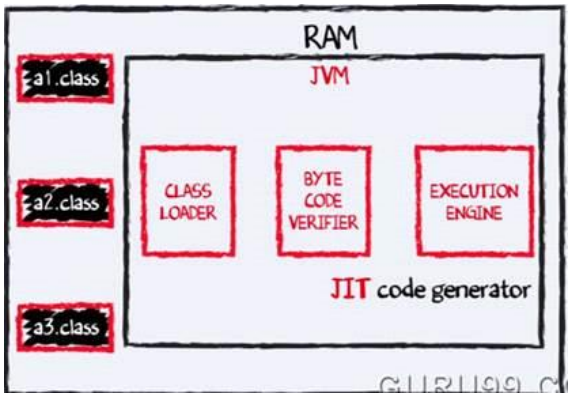
Erzeugung eines mit weiteren Attributen versehenen Syntaxbaum (d.h. attributierten Syntaxbaum).

Compiler – Backend

Hauptaufgaben:

- Auswertung des attributierten Syntaxbaumes
- Optimierung am Code (Syntaxbaum)
- Transformation der Syntaxbaumes in die Enddarstellung (oft Maschinencode eines OS)

Kompilierung

Sprache C	**Sprache Java**
Davon ausgegangen, ich habe der Dateien (a1.c, a2.c, a3.c). Diese werden vom Compiler in .obj-Dateien umgewandelt (welche den Maschinencode enthalten) und der „Linker“ produziert daraus eine ausführbare exe.-Datei	In Java werden die drei Dateien ebenfalls kompiliert, jedoch enthalten die .class-Dateien nicht den Maschinencode, sondern erst den Bytecode
	No linking is Done! 
Während der Programmausführung wird die a.exe ins RAM geladen und ausgeführt: 	Will man das Java-Programm ausführen, kommt die Java Virtual Maschine zum Zug! 1. Die .class-Dateien werden vom **Class Loader** ins RAM gezogen. 2. Die .class-Dateien werden durch den **Byte Code Verifier** auf Sicherheitslücken überprüft 3. Am Schluss wird der Bytecode der .class-Dateien durch die **Execution Engine** in den entsprechenden Maschinencode umgewandelt
	
	Dieser Vorgang nennt sich **JIT – JUST-IN-TIME**

	Kompilierung. Dieser Prozess wird bei jeder Ausführung des Programms durchgeführt, deshalb ist Java im Vergleich zu anderen Programmiersprachen langsam.

Kompilierung auf der Kommandozeile

Folgendermassen wir eine Klasse mit einer abhängigen library kompiliert:

```
javac -classpath lib/Jar.jar ch\hslu\wi\Test.java
java -classpath lib/Jar.jar;. ch.hslu.wi.Test
```

Der "." befiehlt Java alle Unterordner vom aktuellen Standort nach Klassen zu durchsuchen.

Objekte und Klassen

26 September 2014 11:32

Objekte haben immer eine …

- Identiät
- Zustand
  - Daten, Instanzvariablen
  - Daten sind meistens Privat
- Verhalten
  - Methoden
  - Methoden immmer public

Objekte existieren zur Laufzeit eine Programmes.
Klassen stehen dem Programm zur Verfügung um Objekte zu erzeugen.

Klasse = word.dotx
Objekt = word.docx

Klasse ist eine Vorlage, die aber auch andere Objekte erzeugen kann.

Methoden

Methoden verändern den Zustand eines Objekte.

Es gibt Setter und Getter Methoden
- Get um Werte abzufragen
- Set um Werte zu setzen

Methoden die mit void deklariert wurden geben keinen Wert zurück.

Eine Methode, die rechts von einem "="-Zeichen ist immer eine Get-Methode.

Methoden mit Argumenten

```
        Objekt.Execute(int a,string b,double c)
Objekt.Execute(string a, string b, doublce c)
```

Die Kombination der Anzahl der Argumente und deren Typen (int, double, float, …) und der Reihenfolge der Argumente wird Parameterliste genannt.

Die Methodensignatur also die Identifikation wird wie folgt zusammengesetzt.

Parameterliste + Name der Methoden = Methodensignatur

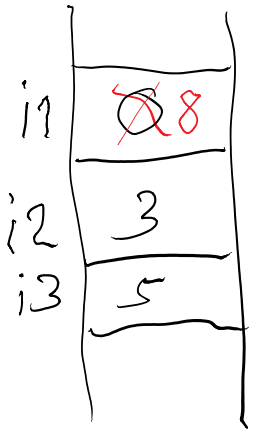
- Mehrere Parameter werden durch ein Kommata getrennt.
- Methodennamen und Parameterliste bezeichnen die Methodensignatur (der Rückgabetyp gehört nicht dazu).
- Die Parameterliste wird durch Anzahl Parameter, deren Reihenfolge und Typ spezifiziert.
- Die Methodensignatur ist innerhalb einer Klasse einmalig. Es kann aber mehrere Methoden mit gleichem Namen geben.

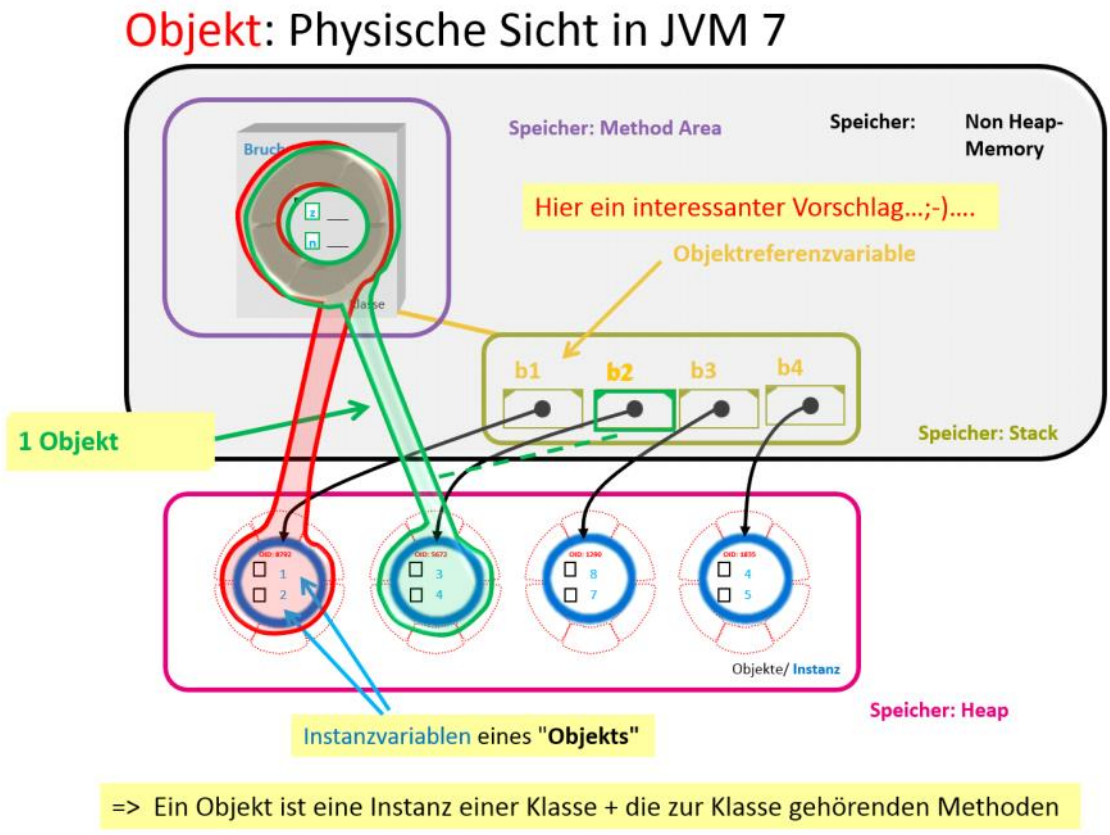
Klasseninitialisierung

```
Bruch b1 = new Bruch(1,2);
```

new = Fordere vom OS Speicher für einen "Bruch" an und der Konstruktor initialisiert diesen reservierten Speicher.

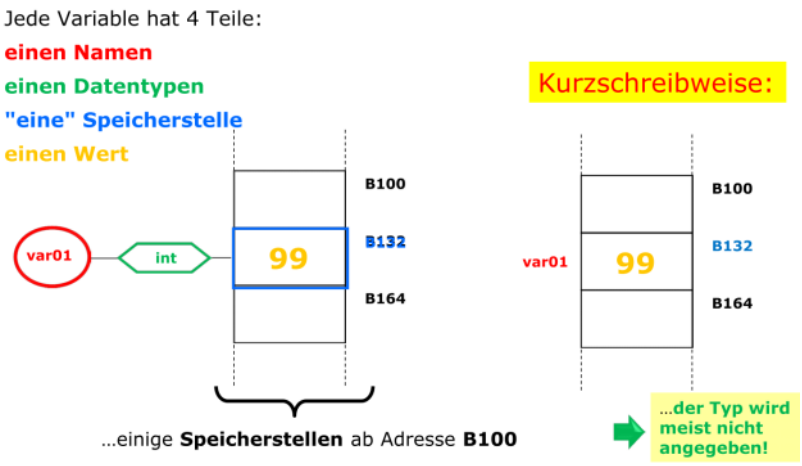
```
int i1;

int i2 = 3;

int i3 = 5;

i1 = i2 + i3;
```



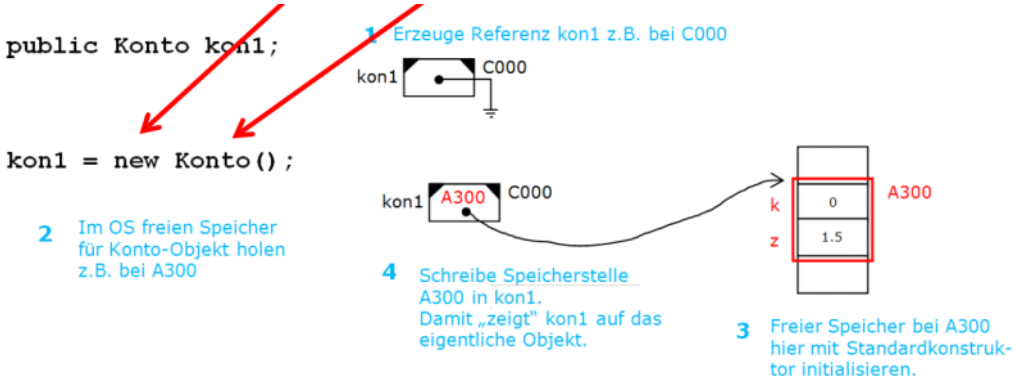


Variablen Zeichnen HSLU

Normale Variablen

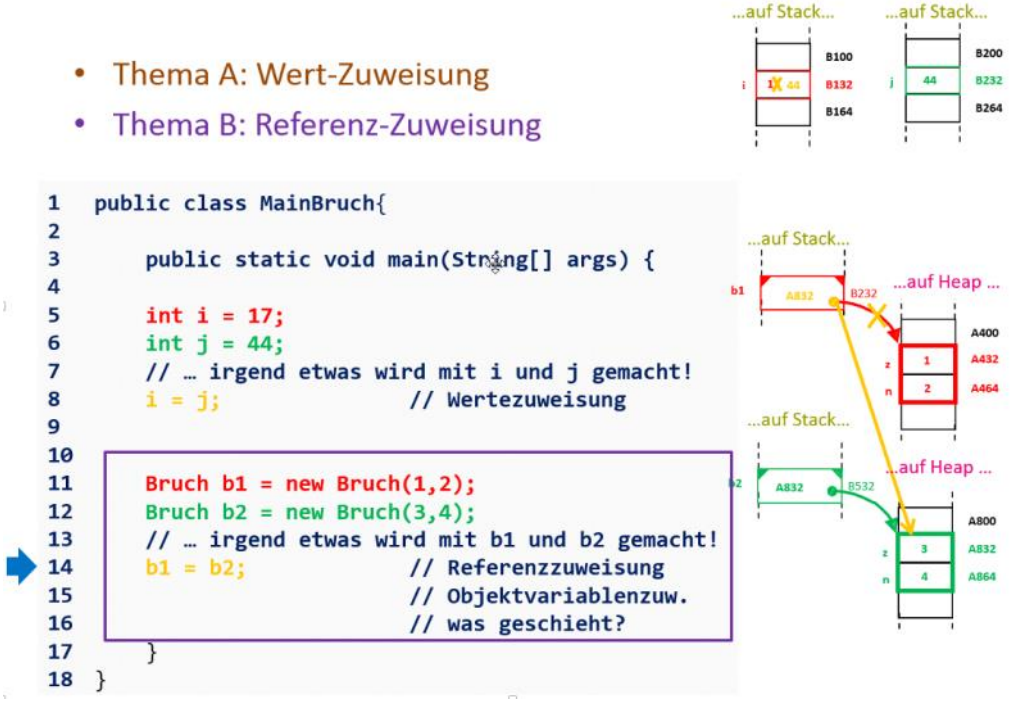


Referenzvariablen

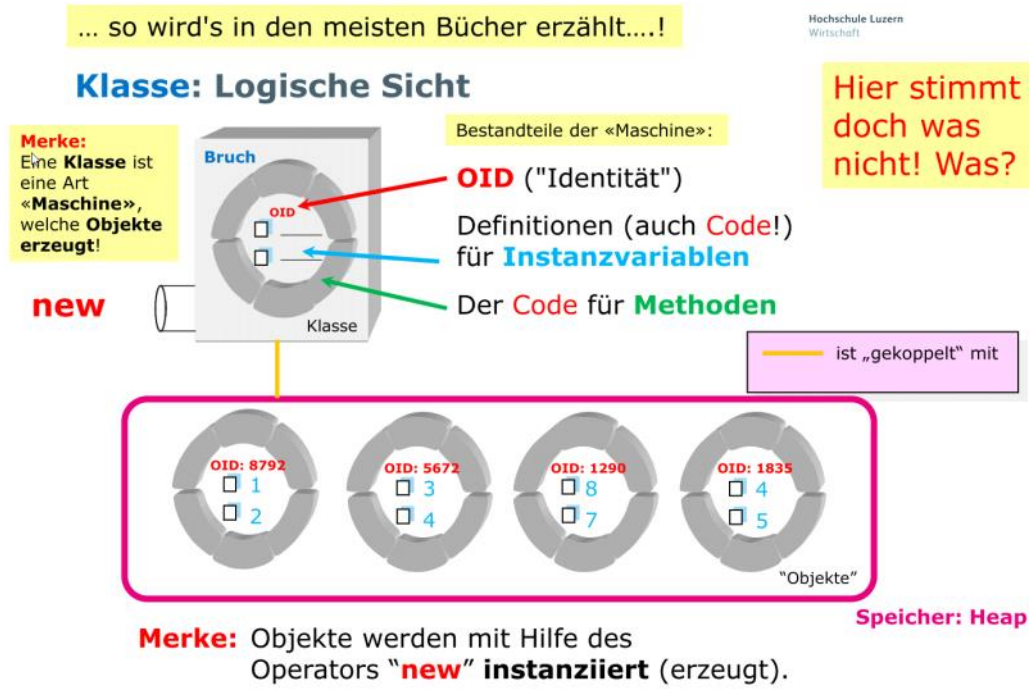


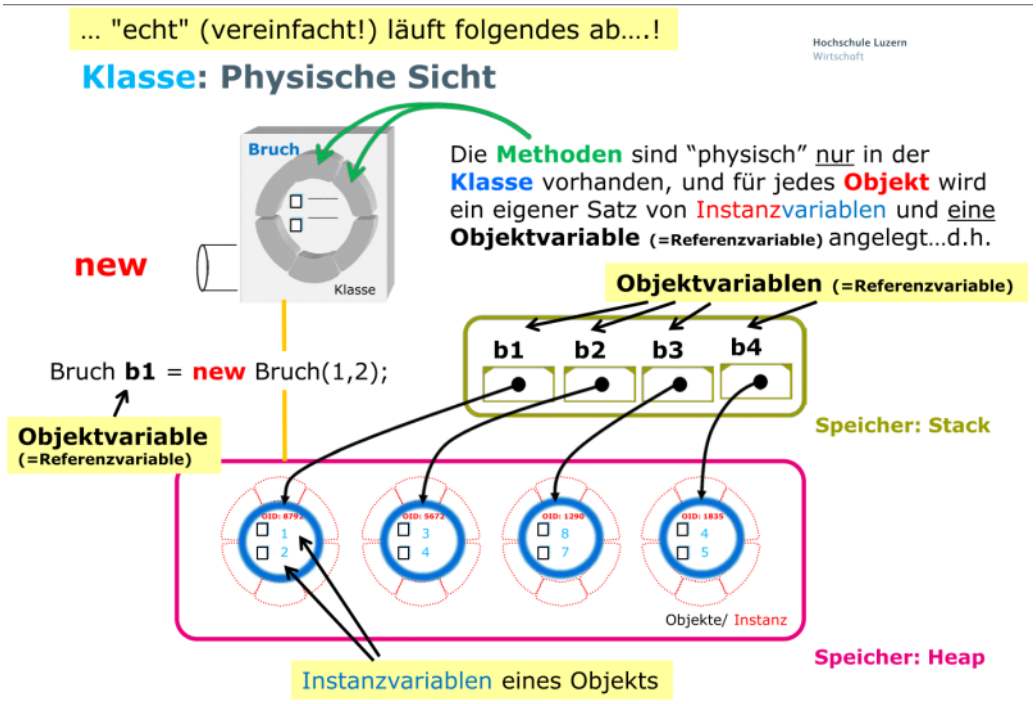
Achtung!: Auch ein String muss als Objekt dargestellt werden. Jedoch kann der Wert anstatt die Adresse eingetragen werden.

Wertezuweisung Variablen vs. Referenzvariablen



Logische- und physische Sicht auf Klassen





Differenz:

- Methoden sind in der Klasse vorhanden.

Wichtig!: Ein Objekt ist eine Instanz einer Klasse + die dazugehörigen Methoden dieser Klasse.

Objektvariablen enthalten die Adresse des Speicherblocks der ersten Instanzvariable.

Sichtbarkeit

Public - Methoden
Private - Variablen

new - Operator

Zum Erzeugen von Objekten d.h. genauer "zum Bereitstellen von Speicher" wird der Operator **new** und die erzeugende Klasse verwendet:

```
Product produkt4711;
produkt4711 = new Product();

Box box = new Box();

Rack rack = new Rack();
```

Methoden, welche gleich heissen wie die Klasse nennt man **Konstruktoren**. Sie "initialisieren" eine Objektvariable.

Variablen

03 October 2014 11:32

Variablen vs. Referenzvariablen

Unterschied Variablen

- Variablen wird in der Regel für Datentypten verwendet.
- Variablen haben immer:
  - Namen
  - Datentyp
  - Speicherstelle
  - Wert

Unterschied Referenzvariablen

- RV wird im Zusammenhang mit einer Klasse verwendet.
- RV zeigt während der Laufzeit eine Programmes auf konkrete Instanze oder auf NULL.

Gemeinsamkeiten

- Beiden Variablen haben immer einen bestimmten Datentyp
- Bei Variablen z.B. byte, float, double, usw.
- Bei RV sind Datentypen = Klassen

Beide müssen deklariert werden.

- Aufbau: Datentyp + Variablennamen
- Deklaration auch mit Anfangswert möglich
- Meherer Variablen können auf einmal deklariert und nacheinander initialisiert werden
- Deklaration erfolgt am Anfang eines umschliessenden Blockes an

Wichtigste Arten von Variablen

- lokale Variablen innerhalb einer Methode
- Attribute eines Objektes

Variable enthält:

- Einen Wert: "Kommazahl" (float), Buchstabe (char) …

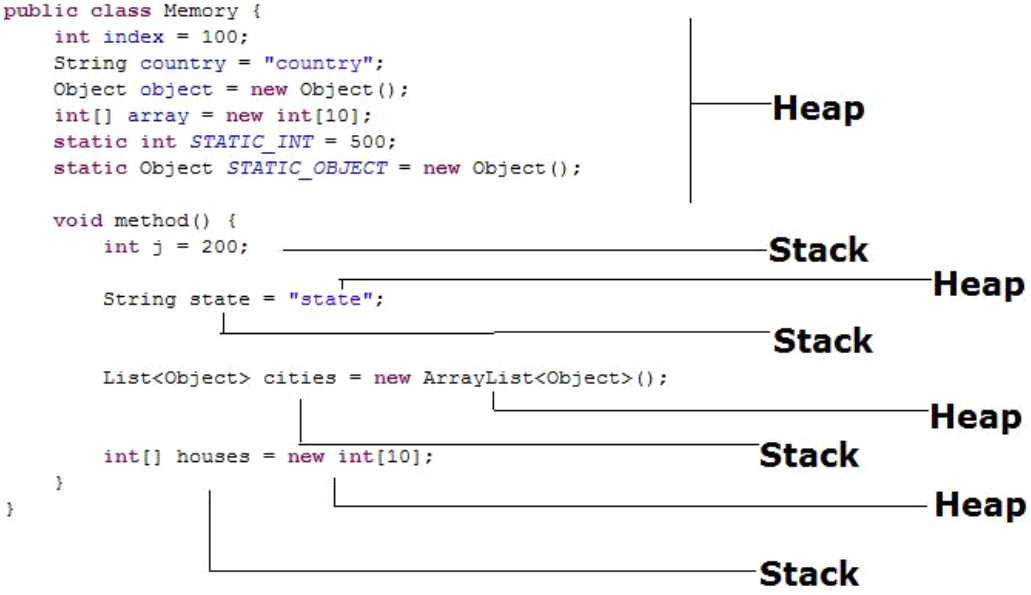
Eine Objektvariable enthält:

- Eine Adresse auf eine Instanz / "Objekt". Man bezeichnet diese Adresse auch als "Referenz"

Basistypen

Art	Bezeich-nung	Byte	Wertebereich
logischer Datentyp	**boolean**	1	**true** oder **false**
Buchstabe	**char**	2	16-Bit-Unicode-Zeichen (0x0000 … 0xFFFF)
ganze Zahlen	**byte**	1	-2^7 bis $2^7 - 1$ (−128 … 127)
	short	2	-2^{15} bis $2^{15} - 1$ (−32.768 … 32.767)
	int	4	-2^{31} bis $2^{31} - 1$ (−2.147.483.648 … 2.147.483.647)
	long	8	-2^{63} bis $2^{63} - 1$ (−9.223.372.036.854.775.808 … 9.223.372.036.854.775.807)
Gleitkomma-zahlen	**float**	4	$1{,}40239846E^{-45}$f … $3{,}40282347E^{38}$f
	double	8	$4{,}94065645841246544E^{-324}$… $1{,}79769131486231570E^{308}$

Arten von Variablen



Referenzvariablen

- Bei der Erzeugung eines Objekts wird eine Referenzvariable erstellt
- Sie enthält die Adresse auf eine Instanz eines Objektes
- werden im Heap gespeichert

Instanzvariablen

- Beschreiben die Eigenschaften eines Objektes.
- Werden in den meisten Fällen mit der Sichtabarkeit Privat deklariert
- werden im Heap gespeichert
- Lebenszyklus: gleich dem Objekt, entsteht bei der Objekt Erstellung und endet mit der Objektvernichtung (garbage collection)
- Auch Attribute gennant

Klassenvariablen

- Klassenvariablen stehen beim Laden einer Klasse zur Verfügung
- Müssen nicht zustätzlich initialisiert werden
- werden im Heap gespeichert

Lokale Variablen

Variablen, die als Zwischenspeicher von Werten innerhalb einer Methode deklariert werden, nennen wir lokale Variablen.

- Müssen immer initialisiert werden
- Die Lebensdauer beginnt mit der Ausführung und dem Ende des Blockes
- Lokale Variablen werden auf dem Stack-Speicher abgelegt
- Haben die höchste Priorität bei gleichen Namen

Konstruktoren

10 October 2014 11:44

- Heissen gleich wie die Klasse
- Werden mithilfe des Operators new aufgerufen
- Sie haben keinen Rückgabewert (auch nicht void)

Aufgaben des Konstruktors

- den Speicher für das Objekt auf dem Heap reservieren
- die Instanzvariablen mit Standardwerten initialisieren
- die Speicheradresse des erzeugten Objekts zurückgeben

Standardkonstruktor

- hat eine leere Parameterliste
- wird standardmässig von der Laufzeitumgebung zur Verfügung gestellt, muss also nicht explizit erstellt werden.

```
public class Haus{
     private int length = 0, width = 0, height = 0;

     public Haus(){
          // Code …
     }
}
```

Ein Standardkonstruktor hat in der Regel einen leeren Rumpf (leere Implementierung)

Das muss nicht immer so sein: im Rumpf des Standardkonstruktors können in bestimmten Situationen sinnvolle Anweisungen platziert werden

- Erzeugen von Container-Objekten (Arrays, Collections usw.)
- Aufruf von Methoden, die diverse Initialisierungsarbeiten durchführen usw.
- Der Konstruktor sollte möglichst einfach gehalten werden
    - kein Aufbau der Verbindung zur Datenbank
    - kein Zugriff auf Dateien
    - usw.

Benutzerdefinierte Konstruktoren

- hat eine nicht leere Parameterliste
- muss explizit erstellt werden.
- Falls ein benutzerdefinierter Konstruktor vorhanden ist und der Standard Konstruktor trotzdem genutzt werden möchte, muss der Standard Konstruktor im Code definiert sein!

```
public class Haus{
     private int length = 0, width = 0, height = 0;

     public Haus (int length, int width, int height){
          // Code …
     }
}
```

Verwendung:

- Objekt bei Erzeugung nur teilweise oder vollständig initialisieren
- Aufrufen von Setter-Methoden vermeiden
- Attribute ohne Schnittstelle mit Initialwert versehen

Klassenelemnte

25 October 2014 17:57

Eine Klassenmethode / Klassenvariable

- wird mit Modifikator static versehen
- kann als globale Methode / Variable auf Klassenebene angesehen werden
- Ist nicht an Instanz einer Klasse gebunden
- Steht beim Laden der Klasse zur Verfügung
- benötigen keinen Zugriff auf Instanzvariablen und Instanzmethoden
- erhalten alles für ihre Arbeit über die Parameterliste

Beispiel:

```
public class Util{

    private satic long initValue = 0;

    public static boolean isPrime(long value){

    }
}
```

Zugriff auf Klassenvariable

Direkter Zugriff:

```
initValue = value;
```

Zugriff über den Klassenamen:

```
Util.initValue = value;
```

Zugriff über die Referenz auf eine konkrete Instanz der Klasse:

```
Util objRef = new Util();
objRef.initValue = value;
```

Oder auch:

```
(new Util()).initValue = value;
```

Zugriff auf Klassenmethoden

Direkter Zugriff (nur innerhalb der Klasse möglich):

```
boolean prime = isPrime(number);
```

Zugriff über den Klassennamen:

```
boolean prime = Util.isPrime(number);
```

Zugriff über die Referenz auf eine konkrete Instanz der Klasse:

```
boolean prime = (new Util()).isPrime(number);
```

Operatoren

20 October 2014 14:58

Operatoren Prioritäten

Priorität	Operatoren	Typ	Assoziativität	Beschreibung
1	[ ]	A	L	Arrayindex
	( )	A	L	Methodenaufruf
	.	A	L	Komponentenzugriff
2	++	N	R	Prä- und Postfixinkrement
	--	N	R	Prä- und Postdekrement
	+	N	R	Vorzeichen
	-	N	R	Vorzeichen
	~	I	R	Einerkomplement
	!	L	R	Logisches NICHT
	(type)	A	R	Typ-Konvertierung
3	*	N, N	L	Multiplikation
	/	N, N	L	Division
	%	N, N	L	Restwert (Modulo)
4	+	N, N	L	Adition
	-	N, N	L	Subtraktion
	+	S, A	L	String-Konkatenation
5	<<	I, I	L	Linksschieben
	>>	I, I	L	Rechtsschieben
	>>>	I, I	L	Rechtsschieben mit Null-Expansion
6	<	N, N	L	Kleiner
	<=	N, N	L	Kleiner gleich
	>	N, N	L	Grösser
	>=	N, N	L	Grösser gleich
	instanceof	R, R	L	Typüberprüfung eines Objekts
7	==	P, P oder R, R	L	Gleichheit
	!=	P, P oder R, R	L	Ungleichheit
8	&	I, I	L	Bitweises UND
	&	L, L	L	Logisches UND ohne SCE
9	^	I, I	L	Bitweises Exklusiv- ODER
	^	L, L	L	Logisches Exklusiv-ODER
10	|	I, I	L	Bitweises ODER
	|	L, L	L	Logisches ODER ohne SCE
11	&&	L, L	L	Logisches UND mit SCE
12	||	L, L	L	Logisches ODER mit SCE
13	? :	L, A, A	R	Bedingte Anweisung
14	=	V, A	R	Zuweisung
	+=	V, A	R	Additionszuweisung
	-=	V, A	R	Subtraktionszuweisung
	*=	V, A	R	Multiplikationszuweisung
	/=	V, A	R	Divisionszuweisung
	%=	V, A	R	Restwertzuweisung
	&=	N, N oder L, L	R	Bitweises UND-Zuweisung und logisches UND-Zuweisung
	|=	N, N oder L, L	R	Bitweises ODER-Zuweisung und logisches UND-Zuweisung
	^=	N, N oder L, L	R	Bitweises Exklusiv-ODER-Zuweisung und logisches Exklusiv-ODER-Zuweisung
	<<=	V, I	R	Linksschiebezuweisung
	>>=	V, I	R	Rechtsschiebezuweisung
	>>>=	V, I	R	Rechtsschiebezuweisung mit Nullexpansion

Tabelle 1 - Operatoren mit deren Prioritäten und Assoziativitätsregeln

Legende:

A – alle, N – numerisch, I – ganzzahlig, R – Referenz, S – String, P – primitiver Typ, V – Variable verlangt

SCE

Wenn der Anweisung aus gleichen Operatoren besteht und der erste Ausdruck das Ergebnis bestimmen kann werden die restlichen Ausdrücke nicht mehr ausgeführt.

Beispiele

```
boolean a = false, b = true, c = true, d = false, v = false;

d = !b;                                  // d = false

// Ausdruck A
v = !a || b && d || !c && !b;            // v = true
    1    1    0    0    0
// Ausdruck B
v = !a || b && d || !c && !b;            // v = else
    1    1    0    0    0
int p = 8, q = 10;
                                         SCE nur mit gleichen Operatoren
// Ausdruck C
v = a && b && (p++ < q);                 // v = false  p = 8  q = 10
    0    1
// Ausdruck D
v = a & b & (p++ < q);                   // v = false  p = 9  q = 10
    0   1       1
```

Arithmetische Operatoren

- Inkremente und Dekremente werden Ausdruckweise verarbeitet
- Postifx für den aktuellen Ausdruck
- Präfix für den kommenden Ausdruck

Zur berechnung der Werte einer Operatoren Gleichung empfiehlt es sich diese in einer Tabelle aufzuführen:

	Ausdruck 1	Ausdruck 2
Var1	10	14
Var2	11	

Kontrollstrukturen

19 December 2014 11:13

Über Kontrollstrukturen lassen sich logische Entscheidungen abbilden.

Selektion

Für eine einfache Selektion bietet sich die IF-ELSE Struktur an.

```
// Eine Zufallszahl zwischen 0 und 99 generieren
int x = (int)(Math.random() * 100);
if (x < 50) {
     System.out.println("Die generierte Zahl ist kleiner als 50.");
}else if (x == 50) {
     System.out.println("Es wurde die Zahl 50 generiert.");
} else {
     System.out.println("Die generierte Zahl ist groesser als 50.");
}
```

Switch Case

Gibt es eine bestimmte Auswahl an Entscheidungsmöglichkeiten ist die Switch Anweisung die beste Wahl.
Für jeden Case gibt es eine Anweisung, trifft gar nichts zu wird der Default Case ausgeführt. Jeder Case muss mit break beendet werden, ansonsten wird auch der default Block ausgeführt.

```
int nummer = 4;
switch(nummer){
     case 1:
          // Code;
          break;
     case 2: {
          // Code;
          break;
     }
     case 3:
          // Code;
          break;
     case 4:
          // Dieser Code wird ausgeführt;
          break;
     default:
          // Code;
}
```

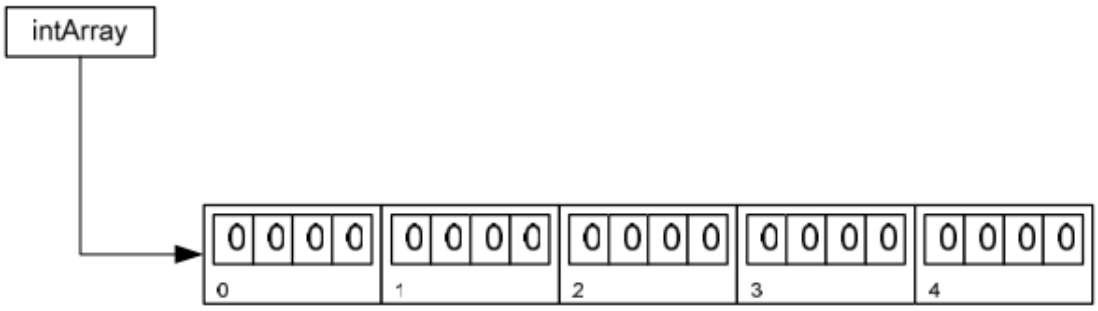
Vergleich Objekt Eigenschaften

Das ist ein Spezialfall, der Vergleichs-Operator kann nicht mehr angewendet werden.

```
Person p = new Person("Thomas");
if("Thomas".equals(p.getName())){
     // Code
}
```

Arrays

24 October 2014 11:00



- Sind strukturierte Datentype
- besteht aus Anzahl von Elementen des gleichen typs
- Im Speicher aufeinander folgend
- Zugriff auf Element erfolgt über einen ganzzahligen Index
  - 0 ist erstes Element
  - letztes Element hat Wert -1
- Speicher Daten in Stack-Speicher
- Arrays sind Objekte

Erzeugung

```
Variablennamen = new Datentyp[ Anazahl Elemente ];
```

```
    intArr = new int [10];
```

Erzeugung ohne new Operator

```
    int[] intArray = {1,2,4,6};
```

- Länge des Arrays wird abgeleitet
- Vollständige Initialisierung bei Erstellung

Erzeugung durch neue Referenz

intArray = new int[]{100,200,300};

Referentvariable wird "umgebogen"

Array kopieren

```
    int[] orgRef = {1, 3, 5, 7, 9};
    // Leeres Array (kopieRef) erzeugen
    int [] kopieRef = new int[5];
    // Elemente kopieren
    kopieRef[0] = orgRef[0];
    kopieRef[1] = orgRef[1];
    kopieRef[2] = orgRef[2];
    kopieRef[3] = orgRef[3];
    kopieRef[4] = orgRef[4];
```

Nur primitive Datentypen (int, char, …, String)
Anstatt Feld für Feld kopieren kann auch das Objekt geklont werden, der Ansatz bleibt aber gleich

Mehrdimensional

Java kennt auch mehrdimensionale Arrays

```
int[][] tabelle = new int[anzahlZeilen][anzahlSpalten]
```

Zugriff erfolgt dann so

```
tabelle [2][3] = 8
```

Iteration

24 October 2014 11:25

- Java kennt Kopf- und Fussgesteuerte Schleifen
  - Kopf: for und while
  - Fuss: do while
- Aweisungen werden solange ausgeführt bis eine Bedingungsausdruck false liefert.

Die For Schleife

```
for(Init; Bedingungsausdruck; ReInit){
    Anweisung;
}
```

Der Init-Teil wird einmal ausgeführt
Der ReInit-Teil für jeden durchgang

Beispiel das die Quadratzahlen bis 4 liefert:

```
for (int i = 0; i < 5; i++) {
    System.out.println("Quadrat: " + ( i * i ));
}
```

Die Whilfe Schleife

```
Init;
while (Bedingungsausdruck){
    Anweisung;
    ReInit;
}
```

entspricht dem Verhalten der for-Schleife

Beispiel:

```
int a = 10;
while( a > 0) {
    System.out.println("a = " + a);
    a--;
}
```

Die Do While Schleife

```
do{
    Anweisung;
}while(Bedingunsausdruck);
```

Ist sinnvoll wenn ein Ausdruck mindestens einmal ausgeführt werden muss

```
java.util.Scanner sc = new java.util.Scanner(System.in);
int n = 0; do {
    System.out.print("Bitte eine Zahl eingeben (abbrechen mit 0): ");
    n = sc.nextInt(); System.out.println("Eingegebene Zahl: " + n);
} while(n != 0);
```

break und continue

mit break kann man die Ausführung einer Schleife abbrechen.

```
break;
```

continue bricht nur die aktuelle Iteration ab und springt wieder zum Kopf der Schleife.

```
continue;
```

Bitweise Operatoren

24 October 2014 12:24

Operator	Bezeichnung	Syntax	Bedeutung
~	Einerkomplement	~a	entsteht aus a, wenn alle Bits invertiert werden
|	ODER	a | b	ergibt den Wert, der entsteht, wenn die korrespondierenden Bits von a und b miteinander ODER-verknüpft werden
&	UND	a & b	ergibt den Wert, der entsteht, wenn die korrespondierenden Bits von a und b miteinander UND-verknüpft werden
^	EXKLUSIV ODER	a ^ b	ergibt den Wert, der entsteht, wenn die korrespondierenden Bits von a und b miteinander EXKLUSIV-ODER-verknüpft werden

z.B. XOR

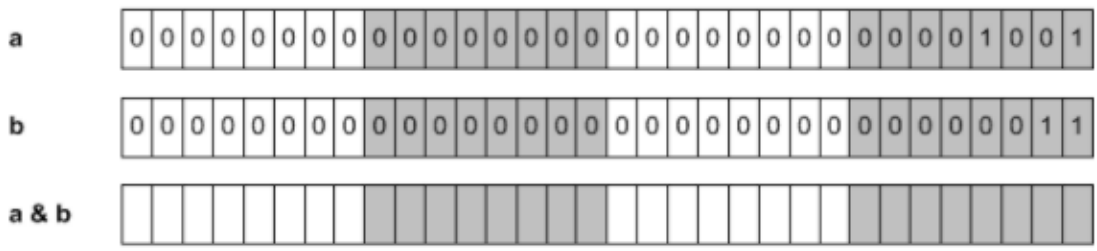
0 xor 0 > 0
0 xor 1 > 1
1 xor 0 > 1
0 xor 0 > 0

Operator	Bezeichnung	Syntax	Bedeutung
<<	Links-Schieben	a << b	ergibt den Wert, der entsteht, wenn alle Bits von a um b Positionen nach links geschoben werden. Das höchstwertige Bit wird mit von links kommenden Bits überschrieben, die "leer gewordenen" Bits auf der rechten Seite mit Nullen aufgefüllt.
>>	Rechts-Schieben	a >> b	ergibt den Wert, der entsteht, wenn alle Bits von a um b Positionen nach rechts geschoben werden. Falls das höchstwertige Bit gesetzt ist (a ist eine negative Zahl), wird auch das höchstwertige Bit des Resultats gesetzt. Die "leer gewordenen" Bits auf der linken Seite werden dadurch bei negativen Zahlen mit Einsen, bei positiven Zahlen mit Nullen nachgefüllt
>>>	Rechts-Schieben ohne Vorzeichen	a >>> b	ergibt den Wert, der entsteht, wenn alle Bits von a um b Positionen nach rechts geschoben werden. Dabei wird das höchstwertige Bit des Resultats immer auf 0 (positiver Wert) gesetzt und die "leer gewordenen" Bits auf der linken Seite mit Nullen nachgefüllt.

- Operiert wird mit einzelnen bits

a = 9, b = 3

1001



Lösung: 1

Typumwandlung

24 October 2014 12:29

Beispiel einer Umwandlung:

```
byte byteNr = 25; int intNr = 100; long longNr= 20000000000L;
long summe = byteNr + intNr + longNr;
```

Funktoniert trotz verschieden grossen Datentypen.

Implizite Typumwandlung

Bei einer erweiternden (impliziten) Typumwandlung wird ein kleiner (enger) Datentyp in einen grösseren (breiteren) Datentyp konvertiert.

Ist möglich von einem weniger breiten Typ in einen breiteren Typ

Umwandlung mit Verlust

```
byte byteNr = 25; int intNr = 100; long longNr= 20000000000L;
int summe = byteNr + intNr + longNr;
```

Den Versuch, die obigen zwei Codezeilen zu kompilieren, quittiert der Compiler mit einer Meldung, in der er von einem möglichen Verlust redet: possible loss of precision

Explizite Typumwandlung

Bei einer expliziten Typumwandlung trägt der Programmier die Verantwortung für einen möglichen Genauigkeitsverlust. Dabei wird ein grösserer Datentyp in einen kleineren Typ konvertiert. Dabei gehen die Werte verloren, die nicht im Wertebereich des kleineren Typen enthalten sind.

```
long longNr = 50;
byte byteNr = 20;
int intNr = 0;

intNr = (int) longNr;
intNr = (int)(2 * longNr + byteNr) ;
byteNr = (byte)longNr;
```

Linked Lists

03 November 2014 14:13

Ein Verfahren damit die statische Grösse von Arrays umgehen kann.

Man verbindet verschiedene Objekte zu einer List.
+ Dynamisch
- Sequentieller Zugriff

Beispiel einer Implementation:

Nodes verbinden die einzelnen Objekte zu einer Liste.

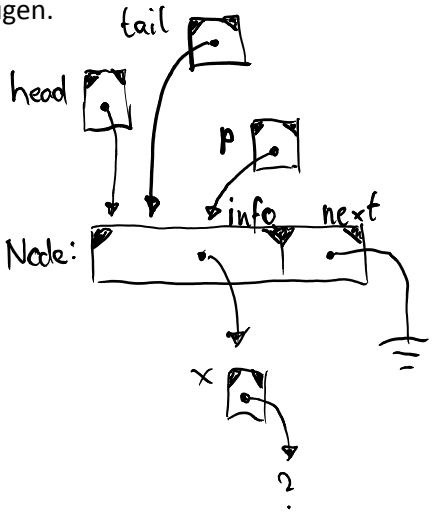
```
class Node {
     Object info;
     Node next;
     Node(Object x){
          info = x;
     }
}
```

LinkedList ist der Zugriffspunkt auf die gesamte Liste.

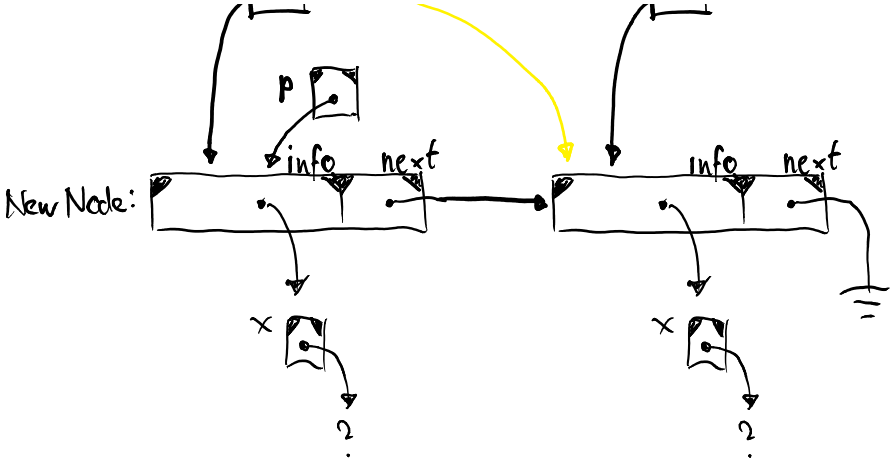
```
public class LinkedList {
     private Node head, tail;
     public LinkedList(){
          head = null;
          tail = null;
     }
     boolean isEmpty(){
          return head == null;
     }
     ...
}
```

Und folgendermassen fügt man den ersten Node einfügen.

```
void insertFirst(Object x){
     Node p = new Node(x);

     if(isEmpty()){
          tail = p;

     }else{
          p.next = head;
     }

     head = p;
}
```

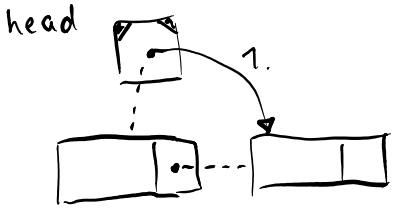


Bei einem zweiten Durchgang sieht das dann folgendermassen aus:



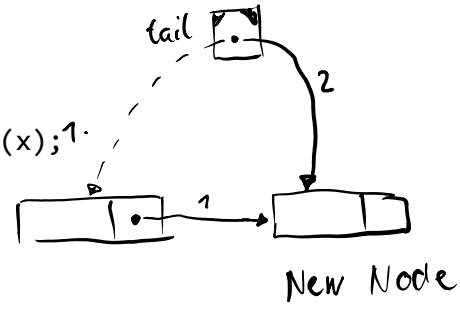
Möchte man das erste Element entfernen muss man der head auf das nächste Objekt zeigen.

```
void removeFirst(){
     if(isEmpty()) return null;
     head = head.next;
     if(isEmtpy()) tail = null;
}
```



Elemente kann man auch am Ende einfügen:

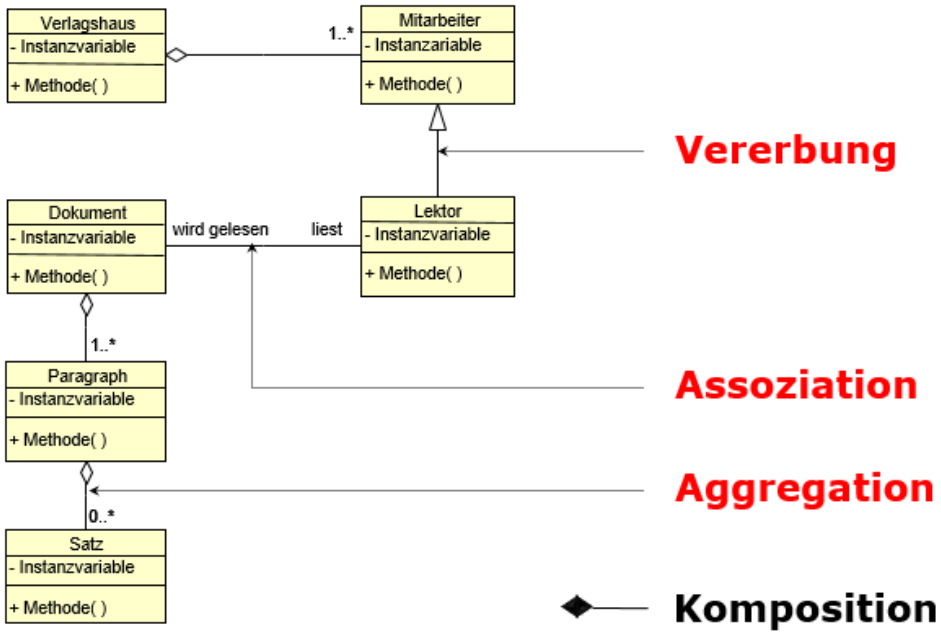
```
void insertLast(Object x){
     if(isEmtpy())
          insertFirst(x);
     else{
          tail.next = new Node(x);
          tail = tail.next;
     }
}
```



Vererbung

03 November 2014 15:54

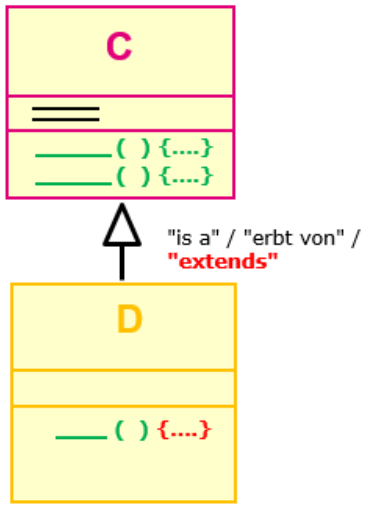
Klassen können in Beziehung stehen. Zur Darstellung von solchen Beziehungen wird UML benutzt.



Legende:

- heisst private, sind nur innerhalb der Klasse verfügbar.
+ heisst public, wird natürlich vererbt.
# heisst protected, nur öffentlich für die Unterklassen.

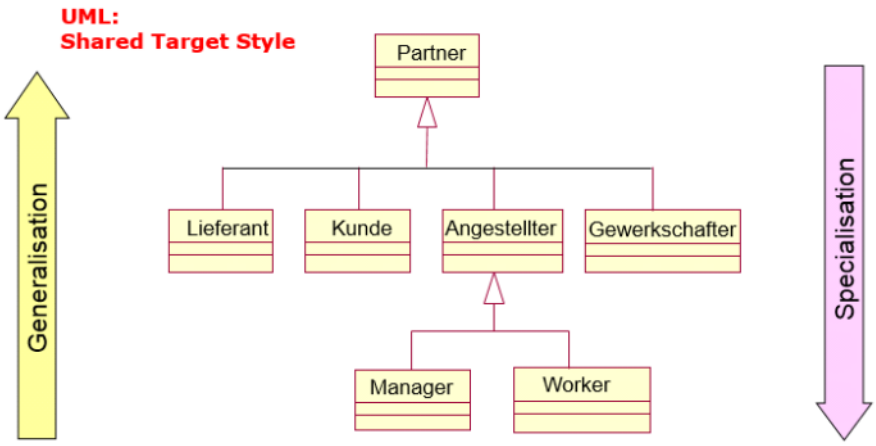
Instanzvariablen und Methoden können vererbt werden. Im folgenden Beispiel erbt D die Methoden und Variablen von der Klasse C.



Dabei kann D mit Methoden ergänzt werden oder sogar Methoden aus C überschrieben werden (Gültigkeitsbereicht ist natürlich nur die Klasse D).

Klasse D wird als abgeleitete Klasse bezeichnet (Oberklasse).
C ist eine Basisklasse (Unterklasse).

Komplexere Vererbung mit "echten" Klassen.



Alle Klassen erben von der Mutterklasse OBJECT.

Erweitern einer Klasse

Das keyword um eine Klasse zu erweitern ist "extends".

```
public class ToxicProduct extends Product {

    /** * Standardkonstruktor. */
    public ToxicProduct() {
        super();
        setView(new ProductView(this, "/bilder/toxicProduct.png"));
    }

     /* (non-Javadoc) * @see
    ch.hslu.wi.stock.business.goods.Product#description() */
    @Override public String description() {
        return "giftiges Produkt";
    }
}
```

Mit @Override kann eine bestehende Methode überschrieben werden.

Super-Methode

Super() ist der Konstruktor aufruf der unmittelbaren Oberklasse.
In einem Konstruktor muss super() vor allem anderen kommen (auch wenn es in diesem Beispiel nichts weiter gibt.) Erinnern Sie sich, dass auch wenn Sie es nicht explizit tun, der Compiler automatisch super() aufrufen wird, als das erste was ein Konstruktor tut.

```
class KinderGeburtstag extends Geburtstag{
  public KinderGeburtstag ( String e, int jahre )  {
    super( e, jahre );
  }

  public void gruss()  {
    super.gruss();
    System.out.println("Was bist du gross geworden!!\n");
  }
}
```

Abstrakte Klassen und Methoden

07 November 2014 13:47

Abstrakte Klassen

Eine abstrakte Klasse ist in Java eine Klasse, die niemals instanziiert wird. Ihr Zweck besteht darin die Superklasse verwandter Klassen zu sein. Die Subklassen erben von der abstrakten Superklasse.

Das keyword zur deklarierung einer abstrakten Klasse lautet abstract.

```
abstract class KlassenName
{

   . . . . .  // Definition von Methoden und Variablen

}
```

Abstrakte Methoden

Eine abstrakte Methode hat keinen Körper. (Sie besitzt keine Anweisungen.) Sie deklariert Zugriffsmodifizierer, Rückgabetyp und Methodensignatur gefolgt von einem Semikolon. Eine nicht-abstrakte Subklasse erbt die abstrakte Methode und muss eine nicht-abstrakte Methode definieren, die mit der abstrakten Methode übereinstimmt.

```
abstract class Karte
{
  String empfaenger;             // Name des Empfängers
  public abstract void gruss();  // abstrakte gruss() Methode
}
```

Anwendung von abstrakten Klassen und Methoden

Hier ist eine Klassendefinition für die Klasse Feiertag. Es ist eine nicht-abstrakte Subklasse einer abstrakten Superklasse:

```
class Feiertag extends Karte
{
  public Feiertag( String e )
  {
    empfaenger = e;
  }

  public void gruss()
  {
    System.out.println("Dear " + empfaenger + ",\n");
    System.out.println("frohe Feiertage!\n\n");
  }
}
```

Polymorphismus

07 November 2014 13:01

Polymorphie bedeutet "Vielgestaltigkeit." In Java bedeutet es, dass eine einzelne Variable für verschiedene Objekte verwandter Klassen (zu verschiedenen Zeitpunkten) in einem Programm verwendet werden kann. Wenn die Variable mit der Punktnotation variable.methode() verwendet wird, um eine Methode aufzurufen, hängt es vom Objekt ab, auf das die Variable gegenwärtig verweist, welche Methode tatsächlich ausgeführt wird.

```
public class KarteTester
{
  public static void main ( String[] args )
  {

    Karte postkarte = new Feiertag( "Amy" );
    postkarte.gruss();                    //Feiertag gruss()
aufrufen

    postkarte = new Valentin( "Bob", 3 );
    postkarte.gruss();                    //Valentin gruss()
aufrufen

    postkarte = new Geburtstag( "Cindy", 17 );
    postkarte.gruss();                    //Geburtstag gruss()
aufrufen

  }
}
```

Eine Variable kann eine Referenz auf ein Objekt halten, dessen Klasse ein Nachkomme der Klasse der Variablen ist.

Interface

14 November 2014 11:57

In Java gibt es nur Einfachvererbung. Das bedeutet, dass eine Subklasse nur von einer Superklasse erbt. Meistens ist das alles, was Sie brauchen. Aber manchmal wäre Mehrfachvererbung zweckmäßiger.
Interfaces geben Java die Vorteile der Mehrfachvererbung ohne deren Nachteile.

- Eine Klasse, die ein Interface implementiert, muss jede Methode des Interface implementieren.
- Jede Methode muss public sein (das passiert standardmäßig).
- Konstanten des Interface können verwendet werden, als wenn sie in der Klasse selbst definiert worden wären.
- Konstanten sollten in der Klasse nicht neu definiert werden.
- Konstanten müssen immer deklariert werden

Beispiel eines Interface:

```
interface MeineSchnittstelle
{
  public final int    EINEKONSTANTE = 32;       // eine Konstante
  public final double PI            = 3.14159;  // eine Konstante

  public void methodeA( int x );          // eine Methodendeklaration
  double methodeB();                      // eine Methodendeklaration
}
```

Eine Klassendefinition muss immer eine Superklasse erweitern, aber sie kann 0 oder mehr Schnittstellen implementieren:

```
public class GrosseKlasse extends Superklasse
    implements  SchnittstelleA, SchnittstelleB, SchnittstelleC
{

   Körper der üblichen Klassendefinition

}
```

Gesamte Implementation

Ein Interface

```
interface Besteuerbar
{
  final double STEUERSATZ = 0.06 ;
  double berechneSteuer() ;
}
```

Eine Oberklasse

```
class Waren
{
  String beschreibung;
  double preis;
```

```
  Waren( String beschreibung, double preis )
  {
    this.beschreibung = beschreibung;
    this.preis        = preis;
  }

  void anzeigen()
  {
    System.out.println( "Artikel: " + beschreibung +
        " Preis: " + preis );
  }
}
```

Eine dazu implementierte Unterklasse:

```
class Spielwaren extends Waren implements Besteuerbar
{
  int mindestalter;

  Spielwaren( String beschreibung, double preis, int mindestalter)
  {
    super( beschreibung, preis );
    this.mindestalter = mindestalter;
  }

  void anzeigen()
  {
    super.anzeigen();
    System.out.println( "Mindestalter: " + mindestalter );
  }

  public double berechneSteuer()
  {
    return preis * STEUERSATZ ;
  }
}
```

UML

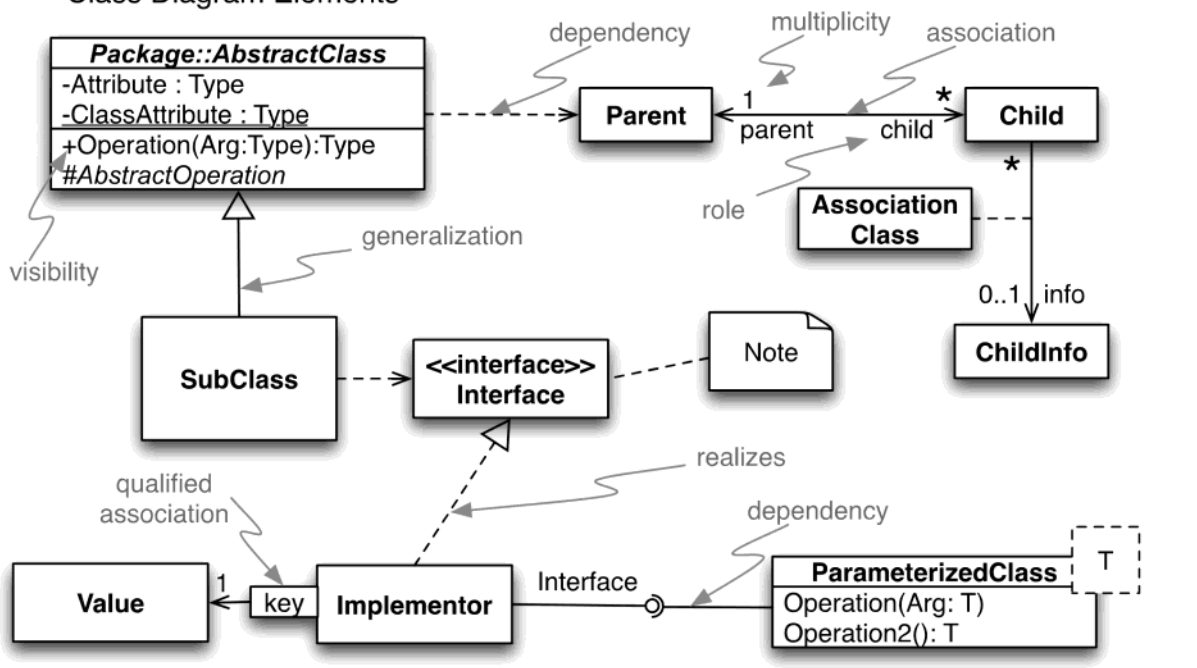
17 November 2014 13:54

Mit UML beschreiben wir die Konzeption eines Programmes. UML umfasst mehrer Methoden und Darstellungen zur Visualisierung verschiedener Aspekte (Beziehungen, Vererbung) eines Software Projekts.

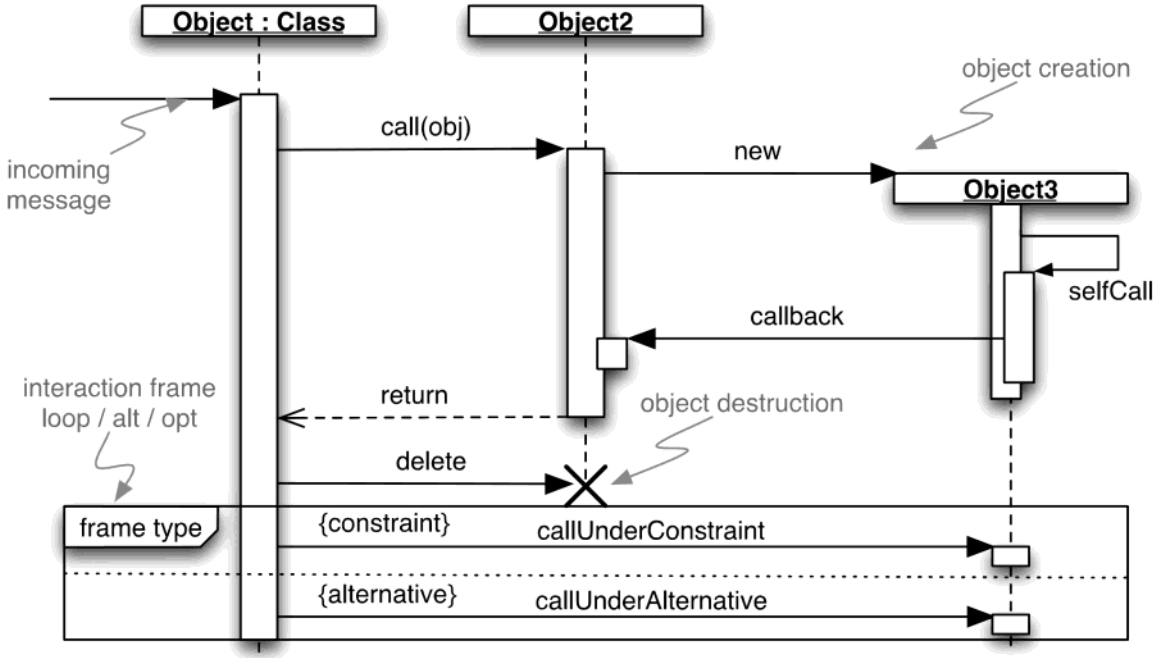
cheatsheet

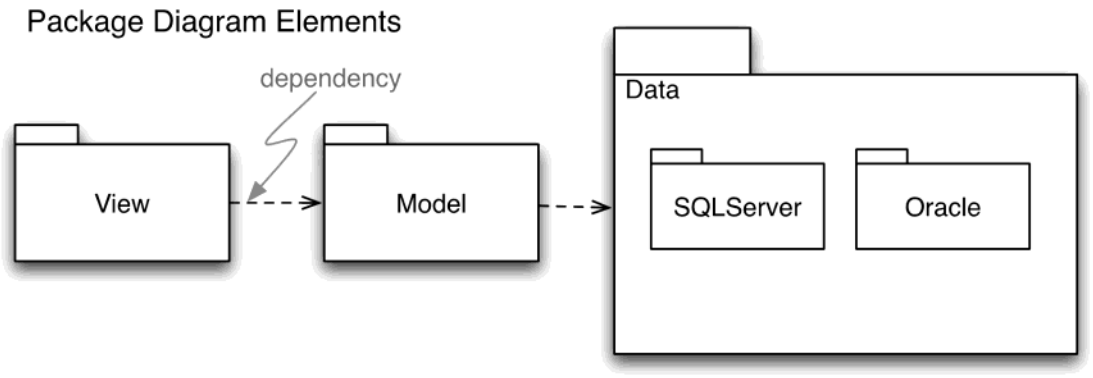
UML Cheatsheet

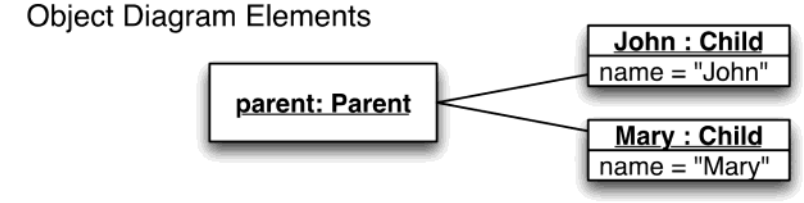
Class Diagram Elements

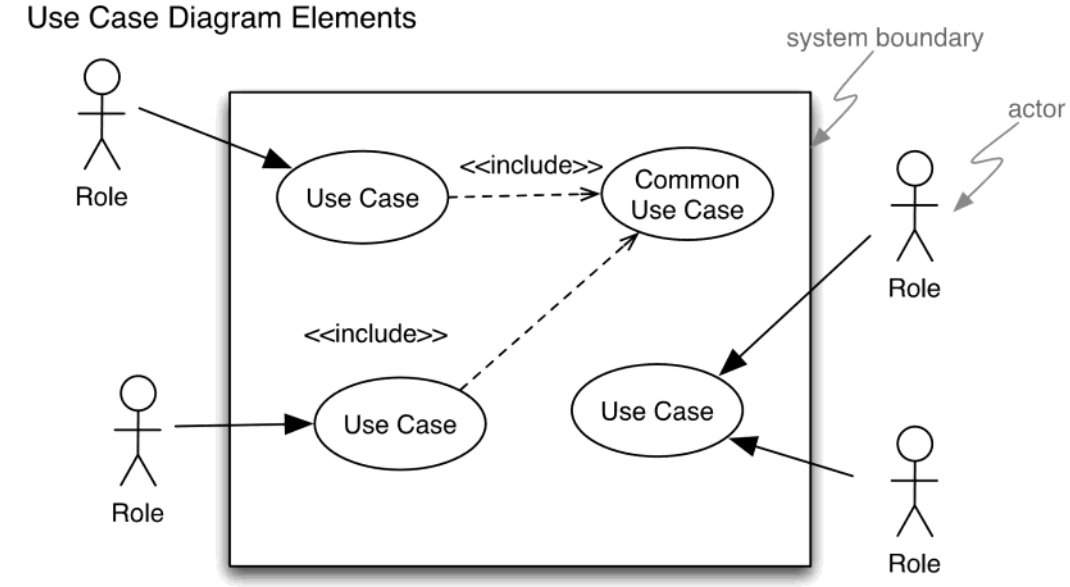


Sequence Diagram Elements

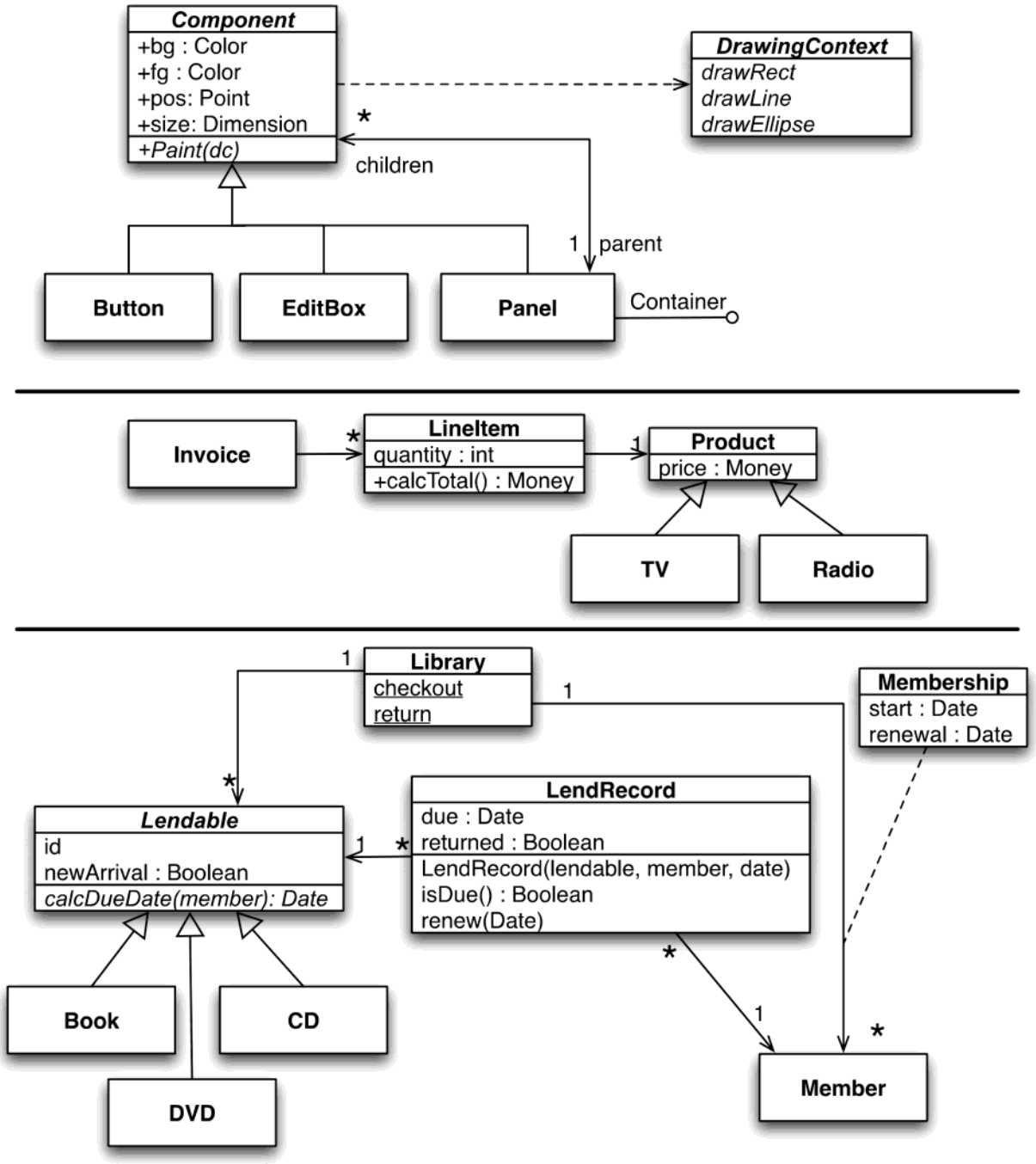




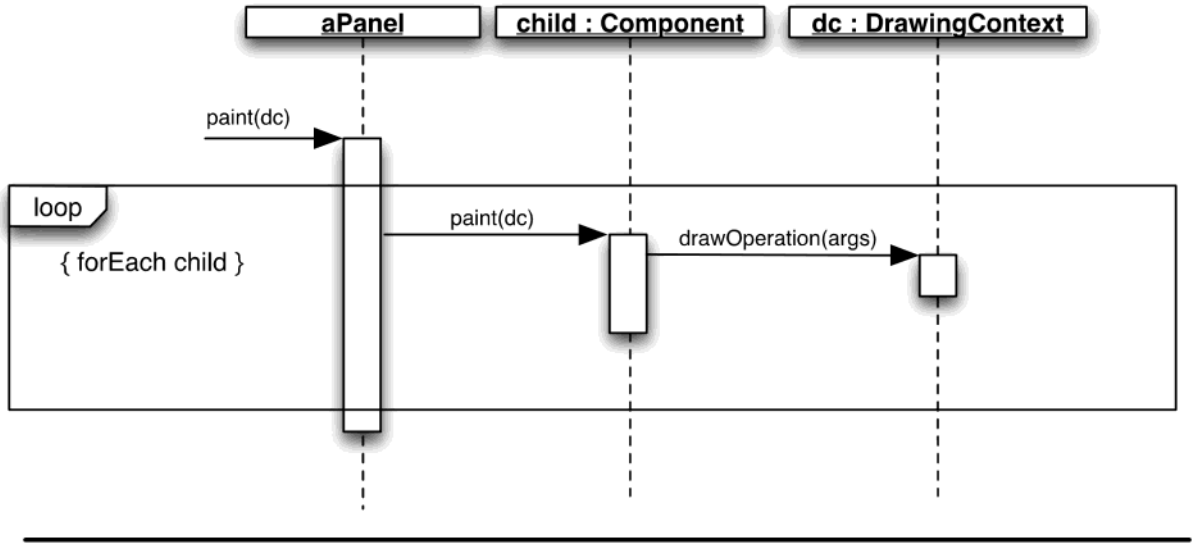


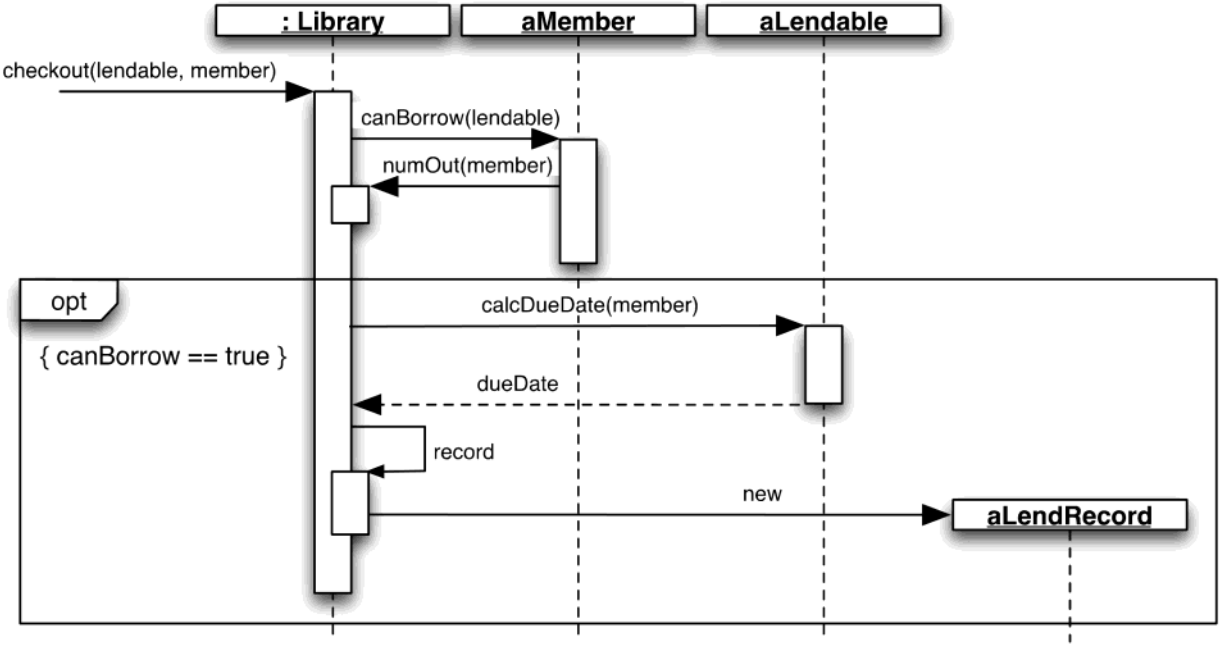


Sample Class Diagrams



Sample Sequence Diagrams





foreach Schleife

21 November 2014 10:52

Die erweiterte For-Schleife (foreach) hat folgende Syntax

```
for (formalerparameter : ausdruck){
    anweisung;
}
```

formalparameter besteht aus Datentyp und Variablennamen.
ausdruck ist eine Instanz oder ein Ausdruck des Typs java.lang.Iterable oder ein Array

Beipsiel Array

```
for (int I : args){
    list.add(i);
}
```

Beispiel java.lang.Iterable

```
for (Integer j : l ){
    System.out.print((j*10) + " ");
}
```

Iterator, Iterable und ListIterator

21 November 2014 10:59

Ein Iterator ist eine Referenz, welches eine Menge (Array) von anderen objekten durchläuft (iteriert).

Die Iterator-Refernz erlaubt es die Elemente einer Collection oder Map einmal abzufragen.

Iterator Methoden

```
hasNext()
```

Gibt true zurück wenn ein weiteres Element exisitert.

```
next()
```

Liefert das nächste Element.

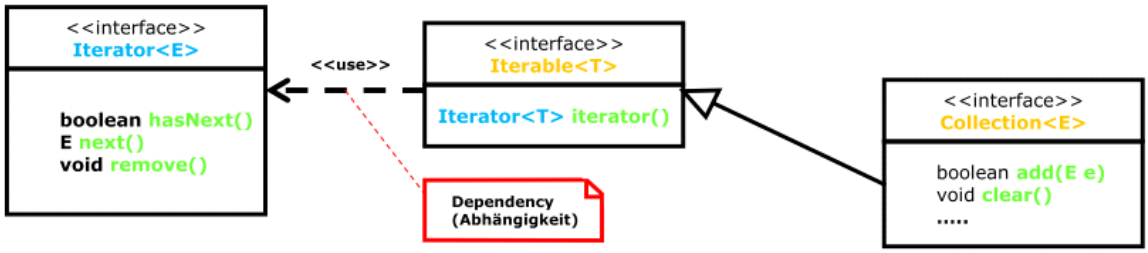
```
remove()
```

Löscht das Element in der Collection

Iterable

Jede Collectin die das Interface Collection implementiert gibt über die Methode iterator() einen Iterator zum Durchlaufen aller Elemente zurück.

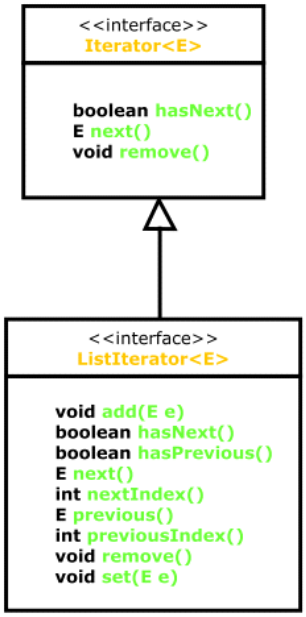
```
public interface Iterable <T> { Iterator <T> iterator(); }
```



Beispeil:

```
ArrayList<String> list = new ArrayList<String>();
Iterator<String> it = list.iterator();
while(it.hasNext()){
     String s = (String) it.next()
     System.out.println(s);
     it.remove();
}
```

ListIterator



Neben dem Iterator-Interface gibt es auch ein abgeleitetes Inteface ListIterator.
Im Untschied zum Iterator durchläuft der List-Iterator die Elemente nach einer bestimmten Reihenfolge.

Steht nur bei Collection's des Typs List zur Verfügung

```
hasPrevious()
previous()
hasNext()
next()
```

Liste kann in beide Richtungen durchlauft werden

```
add()
remove()
set()
```

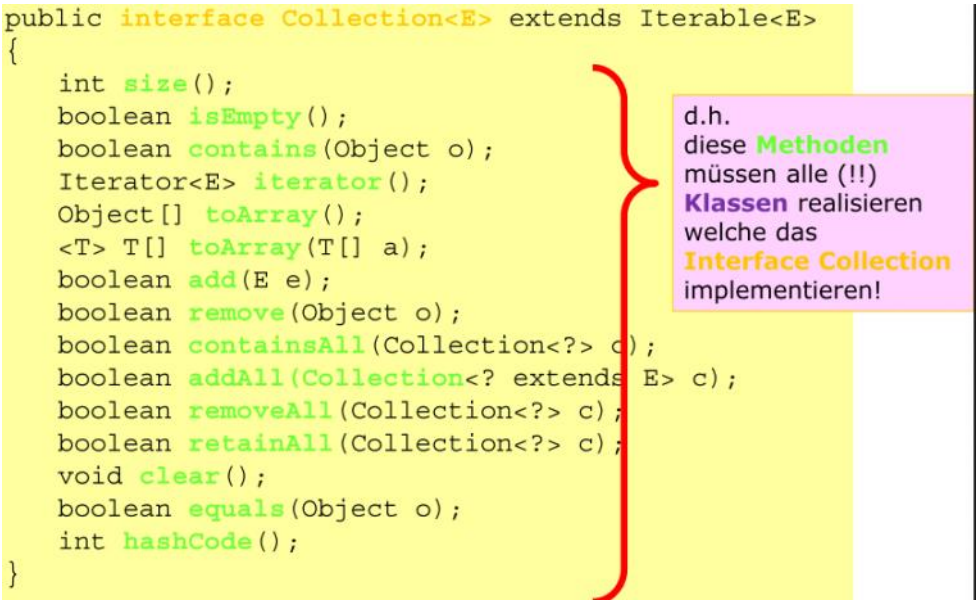
ListIterator kann aktuelle Elemente verändern

Collections

21 November 2014 11:23

Ist eine Sammlung einer Datenstruktur
Kann andere Objekte aufnehmen und verwalten
Grösse der Collection ist dynamisch
Der Zugriff auf die Elemente erfolgt über Methoden

Methoden von Interface Collection



Lambda und anonyme Klassen

21 November 2014 10:56

Mit Java 8 wurden Lambda - Ausdrück eingeführt.

Ziele:

- Umgang mit anonymen Klassen einfacher
- Erweiterung von Java mit funktionalen Aspekten
- Parallelverarbeitung

Eine Lambda-Ausdruck sieht wie folgt aus:

```
(Parameterliste) -> { Methodenrumpf}
(double v) -> {return Math.sort(v);}
```

Die Integration von Lambas war nur durch Ergäznung von Default-Methoden in Interfaces möglich.

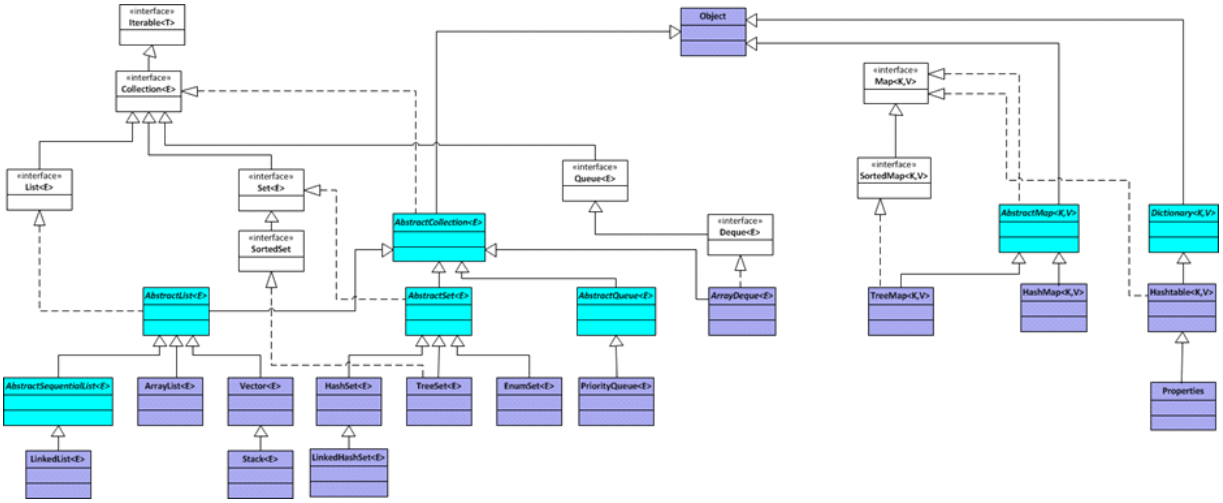
Anonyme Klassen

- Haben keinen Namen
- Werden ad hoc definiert und können nicht mehr angesprochen werden
- Ist nur dann angebracht wenn nur eine einzige Instanz dieser Klasse benötigt wird
- Kann erben (extends) und Schnittstellen implementieren (implements)

```
new KlasseOderSchnittstelle(){

     // Methoden und Attribute der lokalen Klasse
}
```

Collection Framework

21 November 2014 11:34



Für diesen Teil sind die folgenden parameter typen relevant:

- E - Element (used extensively by the Java Collections Framework)
- K - Key
- N - Number
- T - Type
- V - Value

Grundtypen

Das Framework verfügt über folgende Interfaces:

```
List<E>
```

Ist eine beliebig grosse Liste von Elemente beliebigen Typs.

```
Set<E>
```

Ist eine Menge von Elementen im mathematischen Sinn.
Der Zugriff erfolgt mit typischen Mengenoperationen.

```
Map<K,V>
```

In einer Map wird ein Value über einen anderen Typ abgebildete.
Es handelt sich also um eine Menge von Objektpaaren (key, value).

Konstruktoren

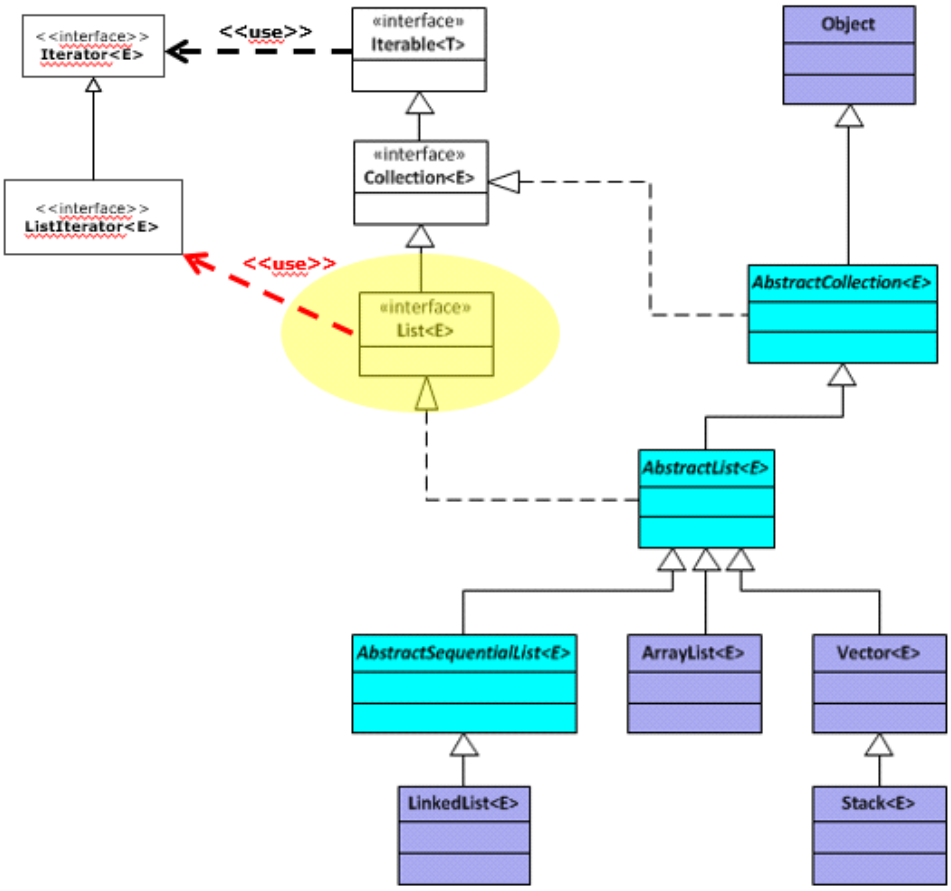
Jede Collection-Klasse verfügt über zwei Konstruktoren:

```
public Vector()
public Vector(Collection c)
```

Der zweite Konstruktor erlaubt es eine Collection mitzugeben und somit die neue Collection zu befüllen.

Interface List

21 November 2014 11:43



Das List Interface erbt von der Collection Klasse und wurde um zusätzliche Methoden ergänzt.
Die AbstractList dienst als Basisklasse.
Im Endeffekt nutzen wir die LinkedList und ArrayList.

```
        // Zusätzliche Methoden
        boolean addAll(int index, Collection<? extends E> c);
        E get(int index);
        E set(int index, E element);
        void add(int index, E element);
        E remove(int index);
        int indexOf(Object o);
        int lastIndexOf(Object o);
        ListIterator<E> listIterator();
        ListIterator<E> listIterator(int index);
        List<E> subList(int fromIndex, int toIndex);
    }
```


Dabei wird nicht der Iterator verwendet, sondern der ListIterator.

Korrekte Verwendung

ArrayListe

- Kleine Liste
- wahlfreier Zugriff -> Zugriff erfolgt direkt, da eine Umwandlung in ein Array erfolgt. Ist also Performanter
- Vorwiegend Lesezugriff

LinkedList

- Grosse Liste
- Verfügt über erweitere Manipulationsmöglichkeiten (delete, insert, upate)

Vector

- Zugriff für mehrer Threads
- Verfügt über Methode mit Deklaration synchronized

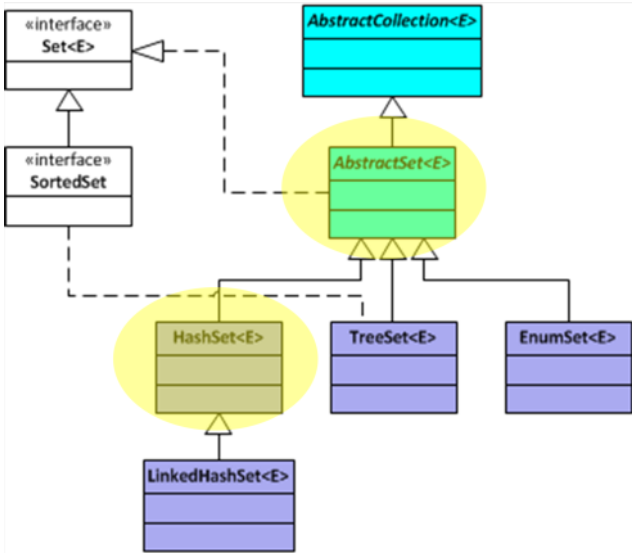
Beispiel:

```
static void fillList(List<String> list){
    for (int i = 0; i < 10; ++i) {
        list.add("" + i);
    }
    System.out.println(list.toString());
    list.remove(3); // remove 4. Element
    list.remove("5"); // remove Objekt mit Inhalt "5"
  }

static void printList(List<String> list){
    for (int i=0;i<list.size();++i){
        System.out.println((String)list.get(i));
    }
    System.out.println("---");
 }

//Create LinkedList
LinkedList<String> list1 = new LinkedList<String>();
fillList(list1);
System.out.println("Ausgabe als LinkedList: ");
printList(list1);

//CreateArrayList
ArrayList<String> list2 = new ArrayList<String>();
fillList(list2);
System.out.println("Ausgabe als ArrayList: ");
printList(list2);
```

Interface Set

21 November 2014 12:28

- Im Gegensatz zur Liste kommen keine doppelte Elemente vor
- Das Set erbt vom Basis-Interface Collection
- Ist das Elemente bereits vorhaden geben Methoden false zurück
  - add()
  - addAll(collection c)
- Das Set benutzt den einfachen Iterator
  - Der Durchlauf erfolgt nicht nach einer definierten Reihenfolge



Die überprüfung erfolgt über eine HasMap implementierung. Vor dem Einfügen wird überprüft ob das Element in der HasMap enthalten ist.

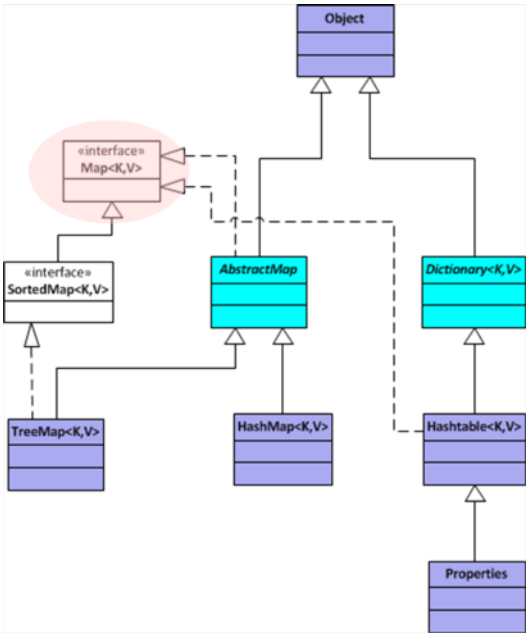
```
HashSet<Integer> set = new HashSet<Integer>(10);
int doubletten = 0;

//Generate Lotto-numbers
while (set.size() < 6) {
     int num = (int) (Math.random() * 49) + 1;
     if (!set.add(new Integer(num))) {
          ++doubletten;
     }
}
```

```
//Print Lotto-numbers
Iterator<Integer> it = set.iterator();
while (it.hasNext()) {
     System.out.println(((Integer) it.next()).toString());
}
```

Jedesmal wenn ein Element nicht eingefügt werden kann wir die doublette inkrementiert.

Interface Map

21 November 2014 12:39



Eine Collection des Typs Map realisiert einen assoziativen Speicher, der Schlüssel auf werte abbildet.
Der Schlüssel muss Eindeutig sein.
Schlüssel und Werte sind Objekte bliebigen Typs
Jeder Schüssel in einer Map zeigt auf null oder einen Wert
Bei Einfügung von existierenden Schlüsseln wird dessen Wert ersetzt.

Spezielle Methoden

```
Set keySet(…)
```

Liefert die Menge der Schlüssel zurück.
Hinweis: Set erlaubt keine doppelten werte.

```
Collection<V< values(…)
```

Liefert die Menge aller Werte zuürck.

```
Set<Map.Entry<k,V>> entrySet(…)
```

Diese Methode liefert die Menge von Schlüssel-Wert-Paaren.
Jedes Element vom Typ Map.Entry[1] implementiert das Interface Entry.
Entry verfügt über die Methoden:

```
getkey()
getValue()
```

Damit erfolgt der Zugriff auf die Komponente des Paares.

Konstruktoren

Eine Map-Interface Implementierung vefügt über zwei Konstrukturen

```
HashMap()
```

```
HashMap(Map<? Extends K, ? Etends V< m)
```

Der zweite Konstrukutor erzeugt eine neue Map mit denselben Schlüssel-Wert-Paaren.

Beispiel:

```
HashMap<String, String> h = new HashMap<String, String>();

//Some Aliases
h.put("john","john.losinger@sb.de");
h.put("ogi","adolf.ogi@trivadis.com");
h.put("pascal","pascal@mathis.ch");
h.put("pascal","pascal@edulu.ch");

//Output
Iterator<Map.Entry<String, String>> it = h.entrySet().iterator();
while (it.hasNext()) {
     Map.Entry<String,String> entry = (Entry<String, String>)
     it.next();
     System.out.println((String)entry.getKey()+
     "-->"+(String)entry.getValue());
}
```

Output Beispiel:

```
ogi-->adolf.ogi@trivadis.com
john-->john.losinger@sb.de
pascal-->pascal@mathis.ch
```

Comparable und Comparator

21 November 2014 13:03

Anhand eines Comparator können Elemente sortiert werden, der Comparator dient dabei als explizites Vergleichsobjekt.
Die Sortierung kann aber auch nach einer natürlichen Ordnung erstellt werden. Dabei benutzt man die copareTo Methode aus dem Comparable Interface

Comparable Interface

- Elemente der Collection müssen eine compareTo() Methode besizen.
- Diese stammt aus dem Interface Comparable.

Dazu ein Beispiel mit einem String der Comparable implementiert:

```
String a = "Halla";
String b = "Halli";
res = a.compareTo(b)
```

Das Resultat ist -8.
Beim resultat gilt:

- Wert < 0, Element liegt vor dem zu vergleichenden Wert
- Wert > 0, Element liegt nach dem zu vergliechenden Wert
- 0, Elemente sind gleich

Comparator Interface

Das Comparator Interface hat folgende Methoden definiert:

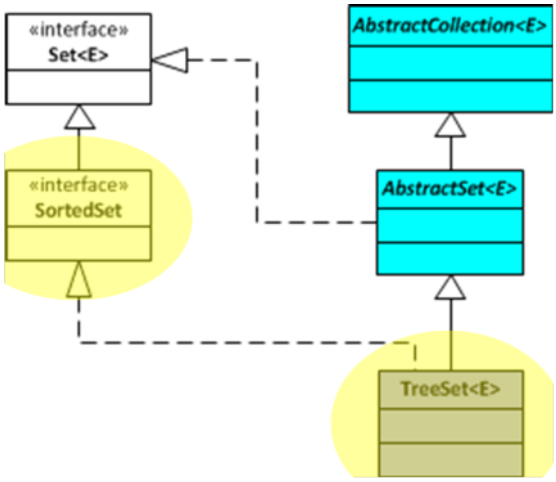
```
public interface Comparator {
    int compare(Object o1, Object o2);
    boolean equals(Object obj);
}
```

Das übergebene Comparator-Objekt übernimmt die Aufgabe einer Vergleichsfunktion.
Die compare() Methode hat die gleiche Semantik wie compareTo()

SortedSet und TreeSet

24 November 2014 14:21

Diese Interfaces sind Anwendungsbeispiele zu Comparable und Comparator



Das SortedSet ist eine Menge mit einer Ordnung.
Die Ordnung basiert auf natürlichen Ordnung oder kann mit einem mitgegebenen Comparator zur Erstellungszeit überschrieben werden.
Das TreeSet ist eine konrkete Implementierung von SortedSet.

Das Interface implementiert folgende Methoden:

```
public interface SortedSet<E> extends Set<E>{
     Comparator<? super E> comparator();
     SortedSet<E> subSet(E fromElement, E toElement);
     SortedSet<E> headSet(E toElement);
     SortedSet<E> tailSet(E fromElement);
     E first();
     E last();
}
```

TreeSet Klasse

Das TreeSet implementiert noch weitere Konstruktoren:

- Parameterlos: natürliche Ordnung
- Comparator: Zukünftige Ordnung mit Comparator
- Collection: Set mit Menge befüllen
- SortedSet: Kopie erstellen

Das folgende Beispiel gibt die Elemente alphabetisch sortiert aus:

```
//Set-construction
TreeSet<String> s = new TreeSet<String>();
s.add("Hans");
s.add("Adolf");
s.add("Eva");
s.add("Anna");
s.add("Lydia");
s.add("Rudolf");
```

```
//Sorted output
Iterator<String> it = s.iterator();
while (it.hasNext()) {
    System.out.println((String) it.next());
}
```

Eine Rückwärtssortierung müsste dann wie folgt implementiert werden:

```
// Comparator Objekt als Parameter
TreeSet s = new TreeSet(new ReverseStringComparator());
```

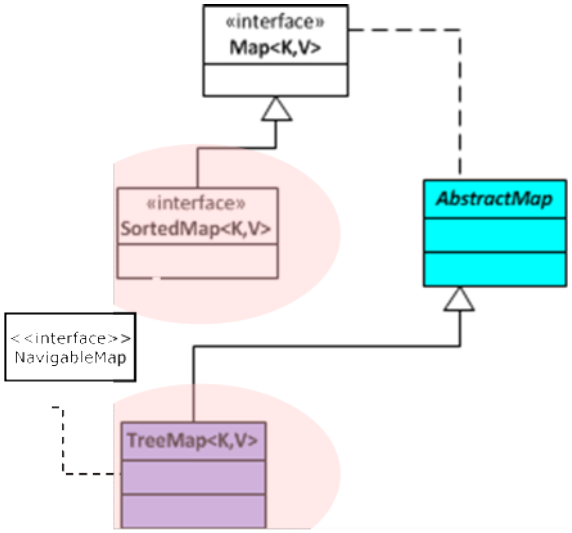
Das Comparator Objekt sieht dann wie folgt aus:

```
class ReverseStringComparator implements Comparator<String>{
    public int compare(String o1, String o2){

        /* "Neue" compare(…) Methode wird hier mit hier mit der
        Methode compareTo(…) der
        Klasse String abgewickelt. Das ist in Ordnung, geht aber
        natürlich) nur, wenn
        String Objekte miteinander verglichen werden!*/

        return o2.compareTo(o1);
    }
}
```

SortedMap und TreeMap

24 November 2014 14:36

Standardmässig erfolgt auch hier die Ordnung nach natürlicher Ordnung.



Wie im SortedSet kann mit der Übergabe eines Comparators die Sortierung übersteuert werden.

Beispiel einer natürlich sortierten TreeMap:

```
TreeMap<String, Double> tm = new TreeMap<String, Double>();

// Put elements to the map
tm.put("John Doe", new Double(3434.34));
tm.put("Tom Smith", new Double(123.22));
tm.put("Jane Baker", new Double(1378.00));
tm.put("Todd Hall", new Double(99.22));
tm.put("Ralph Smith", new Double(-19.08));

// Get an iterator
Iterator<Entry<String, Double>> itr = set.iterator();

// Display elements
while(itr.hasNext()) {
     Map.Entry<String, Double> me = (Map.Entry<String, Double>)itr.next();
     System.out.print(me.getKey() + ": ");
     System.out.println(me.getValue());
}
```

Die Ausgabe wird dann nach dem double value sortiert.

Ausnahmen

28 November 2014 10:48

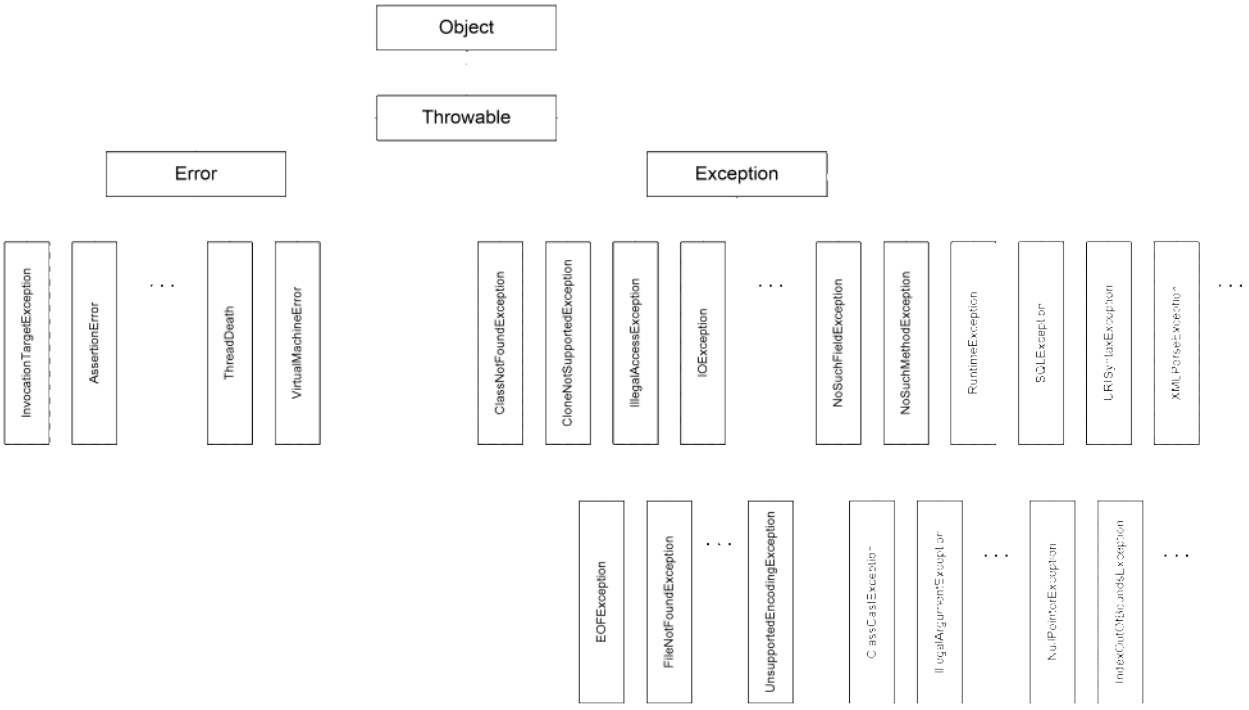
Eine Abweichung von gleichgearteten Fällen.

Ziele:
- Benutzer benachrichtigen
- Applikation auf sanfte Weise herunterfahren
- Schaden minimieren

Ausnahmen müssen behandelt werden

Ursachen
- Fehler in der Implementierung
- Umstände auf die die Applikation keinen Einfluss hat

Exception Types



Java stellt einen mächtigen Mechanismus zur Behandlung von Ausnahmen zur Verfügung, bei dem Ausnahmen als Objekte gehandhabt werden.
Besteht aus zahlreichen vordefinierten Exception-Klassen, kann aber auch selbst ergänzt werden

Es existieren folgende drei Ausnahme-Kategorien:

Ausnahmen vom Typ Error (Error-Ausnahmen, errors)

- Gravierender Fehler, auslöser meist JVM

nicht geprüfte Ausnahmen (unchecked exceptions)

- Entscheid ob abgefangen werden soll wird dem Entwicker überlassen
- Compiler kann diese Fehler nicht überprüfen (Laufzeitfehler und semantische Fehler)
- java.lang.RuntimeException (Typ)

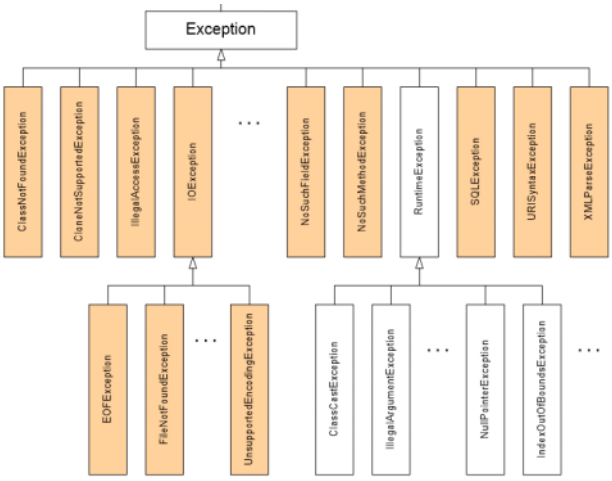
Beispiel:

```
public class Main {
    public static void main(String[] args) {
        int[] arr = new int[5];
        for (int i = 1; i <= arr.length; i++) {
            arr[i] = i + 1;
        }
    }
}
```

Dieser Code ist syntatkisch korrekt auber enthält semantische Fehler.
Resultat der Ausführung:

```
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 5
        at Main.main(Main.java:5)
```

geprüfte Ausnahmen (checked exceptions)



- Müssen explizit behandelt werden
- Compiler verweigert ansonsten kompilierung
- java.lang.Exception (Typ)

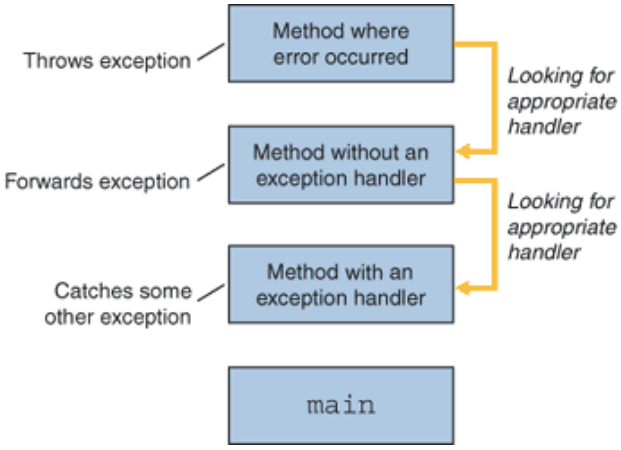
Exception Klassen

- Wenn eine geworfene Ausnahme von keinem Programmteil behandelt wird, landet sie bei der JVM, was in der Regel zu einem Programmabsturz führt.
- Nächst höhere Aufrufmehtode ist veranwortlich für Ausnahmebehandlung

Mit folgenden Code lässt sich eine Exception abfangen und behandeln:

```
try {
        // Programmcode, der Ausnahmen werfen könnte.
} catch (ExceptionTyp e) {
        // Rettungsmassnahmen (die Behandlung der Ausnahme)
} finally {
        // Programmcode, der in jedem Fall ausgeführt werden muss.
}
```

Die Behandlung kann an verschiedenen Punkten durchgeführt werden:



An Ort und Stelle

Sofern möglich und sinnvoll

```
try {
     reader = new FileReader(fileName);

     while ((c = reader.read()) != -1) {
          System.out.println((char) c);
     }
} catch (FileNotFoundException fnfe) {
     /* TODO - Passende Behandlung der Ausnahme */
} catch (IOException ioe) {
     /* TODO - Passende Behandlung der Ausnahme */
}
```

Im catch-Block können auch mehrere Exceptions behandlet werden:

```
catch (ExceptionA | ExceptionB | … | ExceptionN e){
     // Behandlung …
}
```

Exceptions dürfen aber nicht in Beziehung stehen.

An nächsthöhere Instanz weiterleiten

Im Kopf der Methode kann die Ausnahme angekündigt werden und muss dann im Methodenaufruf verarbeitet werden.

```
public static void main(String[] args) {
     String filename = "C:/temp/file01.txt";
     try {
          Main.writewithFileWriter(filename);
     } catch (IOException e) {
          System.out.println(e.getMessage());
     }
}

private static void writewithFileWriter(String filename) throws
IOException{
     FileWriter fwriter = new FileWriter(filename);
     String msg = "Guten Morgen allerseits!!!";
     fwriter.write(msg);
     fwriter.close();
}
```

Eigene Exception Klasse

- Ausnahme-Klassen vom Typ geprüfte Ausname können selber definiert werden.
- Wird von der Klasse Exception oder einer Subklasse abgeleitet.

Beispiel:

```
public class NotSupportedVersionException extends Exception {
     public NotSupportedVersionException(){
     }
     public NotSupportedVersionException(String message) {
          super(message);
     }
}
```

Diese beiden Konstruktoren müssen mindestens implementiert werden.

Die dazugehörige Implementierung kann so aussehen:

```
public void sendData(byte[] buffer, String protocol, String version) throws
NotSupportedVersionException {
    if (!isSupported(protocol, version)) {
        String msg = "Die Version " + version + " wird nicht unterstützt!";
        NotSupportedVersionException e = new
NotSupportedVersionException(msg);
        throw e;
    } else {
        // TODO - Daten senden ...
    }
}
```

Character Streams

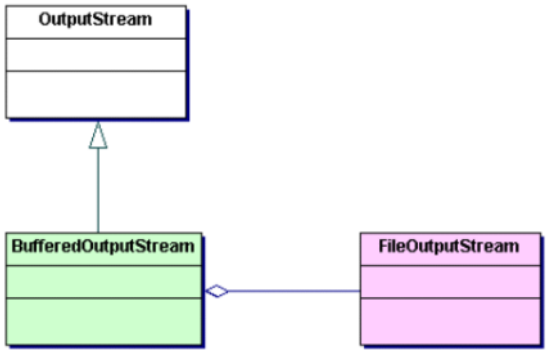
28 November 2014 11:40

Unter einem Stream ist eine geordnete Folge von Bytes zu verstehen, deren Länge nicht bekannt bzw. "nicht relevant" ist.

Es gibt zweit Typen von Streams in Java:
- Byte-Stream (einzelne Bytes)
- Character-Streams (ganze Zeichen)

Alle Streamklassen sind Teil von java.io bzw. java.nio

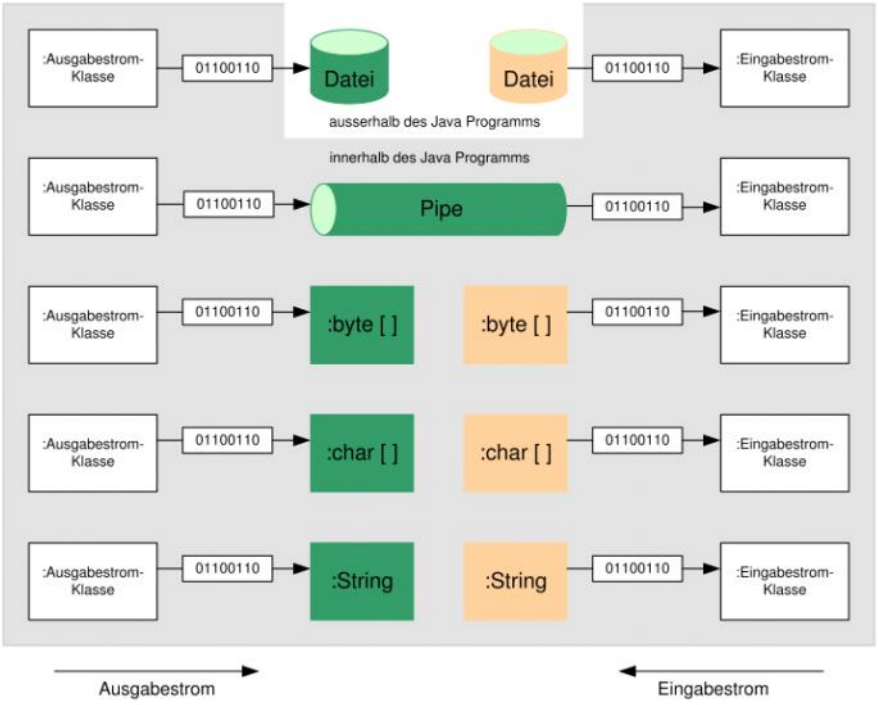
Diese Streams werden in weitere Kategorien unterteilt:
- FirstHand
  - direkt schreibend und lesend
  - Kommuniziert mit konkreten Geräten
  - Erbt von Basisklasse
- Processing
  - weiterverarbietend und veredelnd
  - brauchen Hilfe von FirstHand



FileOutPutStream -> Processing
BufferedOutputStream -> FirstHand

```
BufferedOutputStream bufferedOutput = new BufferedOutputStream(new
FileOutputStream("C:/Temp/TestFile.txt"));
```

Global unterscheidet zwischen zwei Subtypen dieser Streams:
- InputStream (Datenquelle -> Applikation)
- OutputStream (Applikation -> Datensenke)



Output-Character-Streams

Writer ist die abstrakte Basisklasse für alle Ausgabe-Character-Streams

FileWriter

Schreibt Character in eine Datei.

Beispiel

```
public static void main(String[] args) {
     String filename = "C:/temp/file01.txt";
     try {
          Main.writewithFileWriter(filename);
     } catch (IOException e) {
          System.out.println(e.getMessage());
     }
}

private static void writewithFileWriter(String filename) throws IOException{
     FileWriter fwriter = new FileWriter(filename);
     String msg = "Guten Morgen allerseits!!!";
     fwriter.write(msg);
     fwriter.close();
}
```

BufferedWriter

Ist eine Proessing Klasse, die das Schreiben in eine Datensenk gepuffert realisert

```
try (BufferedWriter bufWriter = new BufferedWriter(new FileWriter(fileName))){
    // In die Datei schreiben
    bufWriter.write("Guten Tag");
    bufWriter.newLine();
```

PrintWriter

Ist eine Writer-Variante der Klasse PrintStream.

Bietet Ausgabemthoden für primitive Datentypen (ohne byte), Object und String an.

```
String str = new String("Das ist ein String!");
double d = 5.987654321;
char c = 'A';

try (PrintWriter pOut = new PrintWriter(new FileWriter(fileName), true)) {
      // Schreiben in die Datei
      pOut.println(str);
      pOut.println(d);
      pOut.println(c);
}
```

Exception Handling Writer

Eine sauber ausgeführtes Exception-Handling sieht wie folgt aus:

```
try {
      writer = new FileWriter(fileName);
      writer.write(text);
} catch (IOException e) {
      // Ausnahmen (Konstruktor oder write)
      // behandeln ...
} finally {
      // Stream schliessen
      if (writer != null) {
            try {
                  writer.close();
            } catch (IOException e) {
                  throw new RuntimeException(e);
            }
      }
}
```

Ist jedoch sehr aufwendig!
Lösung "try-with-resources" mit JDK 1.7

```
try(FileWriter fwriter = new FileWriter(filename)){
     fwriter.write("Guten Morgen allerseits!!!");
}catch(IOException e){
     // Code
}
```

Vorteil: Die Resource (FileWriter) wird automatisch geschlossen.

Input-Chrachter-Streams

Reader ist die abstrakte Basisklasse für alle Eingabe-Character-Steams

FileReader

Ist eine FirstHand-Stream-Klasse die von der Bride-Klasse InputStreamReader abgeleitet wird.

```
try(FileReader fReader = new FileReader(filename)){
     int c = -1;
     while( (c= fReader.read()) != -1){
          System.out.print((char)c);
     }
```

```
}
```

BufferedReader

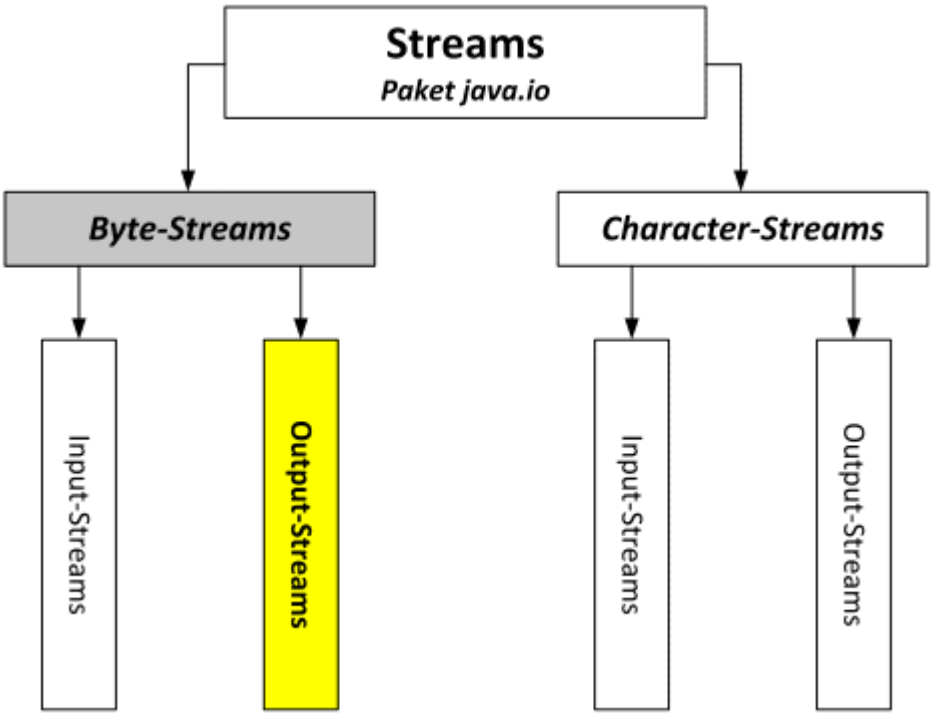
Ist eine Processing-Stream-Klasse, die Daten gepuffet ausliest.

```
try (FileReader fileReader = new FileReader(fileName);
        BufferedReader bufReader = new BufferedReader(fileReader)) {
    // Datei-Inhalt zeilenweise lesen und auf dem Bildschirm ausgeben
    String input = "";
    while ((input = bufReader.readLine()) != null) {
        System.out.println(input);
    }
}
```

Byte Streams

28 November 2014 12:45

Byte-Streams sind in der Lage direkt mit Bytes zu arbeiten



OutPut-Byte-Steams

Klasse OutputSteam ist die abstrakte Basisklasse für alle Ausgabe-Bytestreams.

FileOutputSteam

Ist eine FirstHand-Steam-Klasse.
Ist in der Lage Daten direkt in eine datei zu schreiben.

```
try (FileOutputStream fos = new FileOutputStream(fileName, true)) {

    // in die Datei schreiben
    for (int i = 65; i < (65 + 26); i++) {
        fos.write((byte) i);
    }
}
```

BufferedOutputStream

Ist eine Processing-Stream-Klasse, die das Schreben gepuffert realisiert.
Überlässt das direkte Schreiben wie immer dem Aggregat-Objekt, welches dem Konstruktor als Argument übergeben wird.

```
try (FileOutputStream fos = new FileOutputStream(fileName);
        BufferedOutputStream bos = new BufferedOutputStream(fos)) {

    // Schreiben in die Datei
    for (int i = 65; i < (65 + 26); i++) {
        bos.write((byte) i);
    }
}
```

DataOutputStream

Ist eine Processing-Stream-Klasse, welche die Schnittstelle DataOutput realsiiert.
Kann alle primitiven Datentypen in Java in eine Senke schreiben.

```
try (DataOutputStream dos = new DataOutputStream(new FileOutputStream(fileName))) {

    byte b = 66;
    boolean richtig = false;
    double dbl = 3.141591415914159;
    String str = "Und hier noch ein String!!!";

    // schreiben in die Datei (byte, boolean, double und String)
    dos.writeByte(b);
    dos.writeBoolean(richtig);
    dos.writeDouble(dbl);
    dos.writeUTF(str);
}
```

PrintStream

Kann alle primitiven Datentypen auf dem Bildschirm ausgeben.
Erzeugt keine IOException.

Input-Byte-Streams

InputStream ist die abstrakte Basisklasse für alle Eingabe-Byte-Streams.

FileInputStream

Ist eine FirstHand-Stream-Klasse
Kann Inhalt einer Datei direkt einlesen.

```
try (FileInputStream fis = new FileInputStream(fileName)) {

    int c = 0;

    // aus der Datei lesen und auf dem Bildschirm ausgeben
    while ((c = fis.read()) != -1) {
        System.out.print((char) c);
    }
}
```

BufferedInputStream

Ist eine Processing-Stream-Klasse die Daten gepuffert ausliest.

```
try (FileInputStream fis = new FileInputStream(fileName);
        BufferedInputStream bis = new BufferedInputStream(fis)) {
```

DataInputStream

Ist eine Processing-Stream-Klasse, die die Schnittstelle DataInput realisiert.
Kann alle primitiven Datentypen auslesen.

```
try (DataInputStream dis = new DataInputStream(new FileInputStream(
        fileName))) {

    // Diverse primitive Datentypen aus der Datei nacheinander lesen
    byte b = dis.readByte();
    double dbl = dis.readDouble();
    String str = dis.readUTF();
```

Serialisierung

28 November 2014 12:58

- Ist die Überführung eines Objekts in einen Bytestrom, aus dem eine Rekonstruierung des Objekts möglich ist.
- Es können Objekte und elementare Datentypen serialisiert werden.
- Dasselbe gilt für die Deserialsierung.

Serializable Interface

Damit eine Instanz serialisierbar ist muss die Schnittstelle Serializable realisiert werden.
Dieses Interface deklariert keine Methoden

ObjectOutputStream Klasse

Dienst zur Serialisierung (Object -> Bytestrom).

ObjectInputStream Klasse

Dient zur Deserialisierung

Beispiel Lampe

Die Klasse implementiert Serializable:

```
public class Lampe implements java.io.Serializable {
    private int spannung = 0, leistung = 0;
    private String farbe;
```

Und folgende Methode führt die Serialization durch:

```
public static void writeToFile(Object obj, String fileName)
        throws IOException {

    try (ObjectOutputStream oos = new ObjectOutputStream(
            new FileOutputStream(fileName))) {
        // Objekt in die Datei schreiben
        oos.writeObject(obj);
    }
}
```

Die Deserialiserung wird so ausgeführt:

```
public static Collection<Lampe> readObjects(String fileName) throws FileNotFoundException,
    IOException, ClassNotFoundException {

    ArrayList<Lampe> liste = null;

    try (ObjectInputStream ois = new ObjectInputStream(new FileInputStream(
            fileName))) {
        // Das ArrayList-Objekt aus der Datei auslesen
        liste = (ArrayList<Lampe>) ois.readObject();
    }

    return liste;
}
```

Schlussendlichesieht eine Anwendung so aus:

```
String fileName = "C:/Temp/lampen.obj";

ArrayList<Lampe> listeA = new ArrayList<Lampe>();

// Zwei Instanzen der Klasse 'Lampe' erzeugen
listeA.add(new Lampe(220, 40, "yellow"));
listeA.add(new Lampe(220, 60, "blue"));

// Beide Objekte in die Datei schreiben
writeToFile(listeA, fileName);

// listeA auf null setzen
listeA = null;

// Objekte auslesen
Collection<Lampe> listeB = readObjects(fileName);

// TODO - Objekte aus der listeB verarbeiten ...
```

GUI

05 December 2014 10:51

Im vergelich zu Befehlszeilen orientierten Anwendungen bieten GUI (Graphical User Interface) orientierte Anwendungen mehr komfort bei der Bedienung. Heute sind GUIs selbstverständlich.

GUI Bibliotheken

AWT
- Stellt schwegewichtige (heavyweight) Komponenten zur Verfügung.
- Greift auf native GUI Komponenten der darunterliegenden Plattform (Windows, Linux, …)
- Problem:
  - plattformunabhängige Darstellung nicht möglich
  - Grosse Ressourcenbindung der unterliegenden Plattform

swing

Seit JDK 1.2
- Erste ernstzunehmende GUI Bibltiothek
- Stellt zahlriche Komponenten zur Verfügung
- GUI Komponenten sind plattformunabhängig (auch plattformspezifisch möglich)
- javax.swing
- Baut zum Teil auf AWT auf (Event Handling)

JavaFX
- State of the art Lösung
- Teil von JDK
- Swing interoperabilität
- Layout / Gestaltung mit CSS
- Enthält WebView

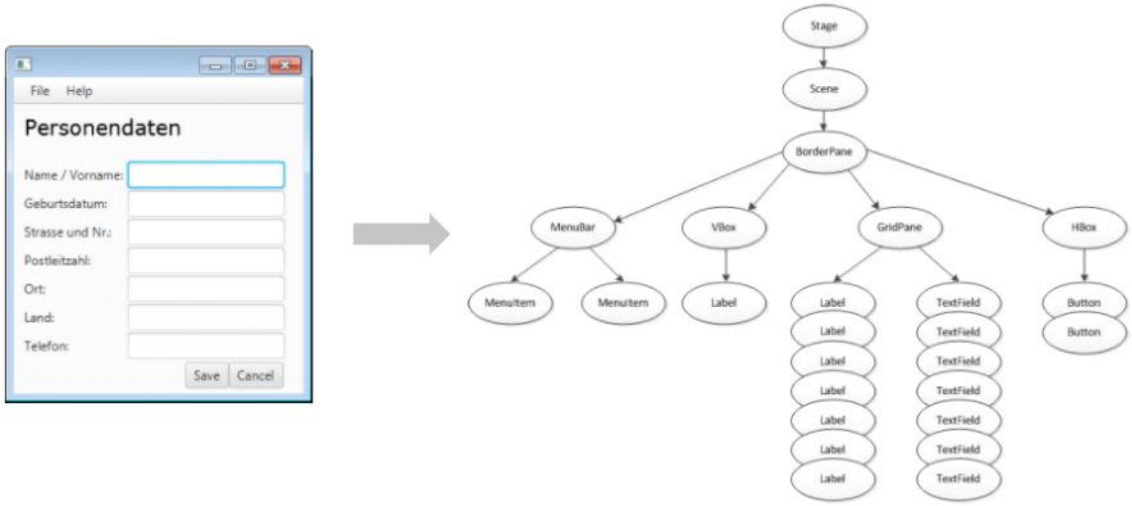
JavaFX

JavaFX bedient sich der Analogie eines Theaters

Stage (Bühne)
- Das Fenster

Scene (Szene)
- GUI Komponenten

Szenengraph
- Komponentenbaum
- Hierarchisch geordnete Komponenten (Nodes)



Die Hauptklasse wird als Subklasse von javafx.application.Application implementiert

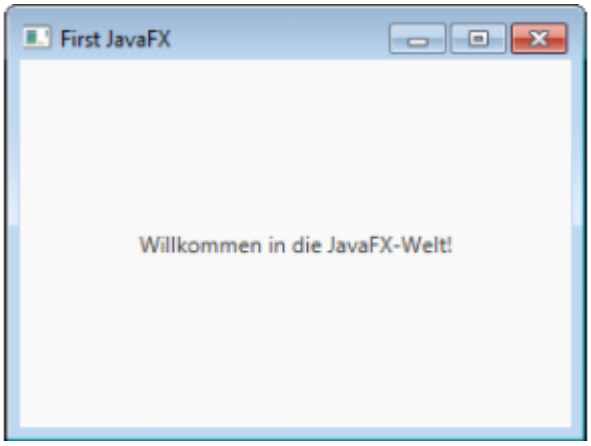
Abstrakte Methode start -> GUI anzeigen

Methoden init, stop, launch

Die Mthode Application#launch(String[]) ruf die Methode Application#start(Stage) automatisch auf.

```
public class Main extends Application{
    public void start(Stage stage){

        // Fenster Titel
        stage.setTitle("First JavaFX");

        //Container für Komponenten
        BorderPane root = new BorderPane();

        //Szene erstellen
        Scene scene = new Scene(root, 300, 200);

        // Labe hinzufügen
        Label label = new Label("Hoi");
        root.setCenter(label);

        // Scene und stage verbinden
        stage.setScene(scene);

        // Stage sichtbar machen
        stage.show();
    }
    public static void main(String[] args){
        launch(args);
    }
}
```

Ergebnis:



Komponenten

Buttons
- Schaltfläche für Klick-Event

```
Button btnSave = new Button("Save");
```

Label
- Beschriftung, nutzung in Kombination mit anderen Komponenten

```
Label lblName = new Label("Name");
```

TextField
- Erlaubt Eingabe

```
TextField txtName = new TextField();
```

TextArea
- Mehrzeilige Eingabe

```
TextArea txtBemerkung = new TextArea();
```

CheckBox
- Ja/Nein Auswahl

```
CheckBox cbBox = new CheckBox("Habs gelsen");
cbBox.setIndeterminate(false);
```

RadioButton
- Erlaubt mehrere Optionen

ComboBox
- Stellt eine Auswahl-Liste zu Verfügung

TableView
- Anzeige in tabellarischer Form
- Daten sind vom typ: ObservableListe
- Erlaubt eine laufende Aktualisierung von Änderungen
- Bietet Sortierung, Festlegung Spaltengrössen

MenuBar
- Stellt typische Menu Komponente zur Verfügung
- Besteht aus MenuItem

JavaFX Layout

Das Anordnen von Komponenten in einem Container kann sehr aufwendig sein.
Zur Abhilfe stellt JavaFX diverse Layouts zur Verfügung.
Diese lassen sich miteinander kombinieren

AnchorPane
- Bindung von Komponenten an einen Container-Rand
  - Top
  - Left
  - Right
  - Bottom

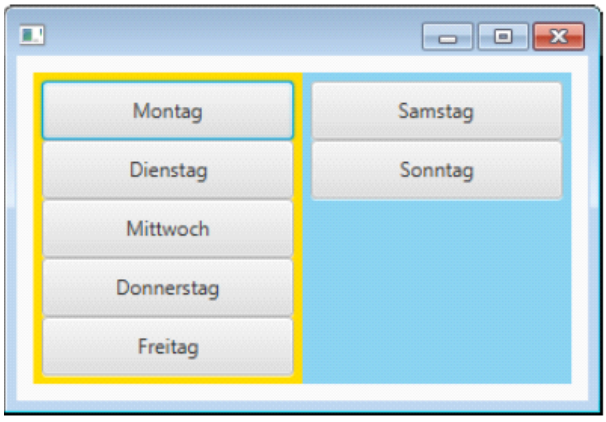
BorderPane
- Anordnung der Komponenten in 5 Bereiche
  - Top
  - Right
  - Left
  - Bottom
  - Center
- Reagiert entsprechend bei Vergrösserung oder verkleinerung des Fensters

FlowPane
- Ordnet Komponenten nacheinander an, solange Platz vorhanden ist.
- Reicht Platz nicht, verschieben sich die Komponenten in die nächste Zeile

GridPane
- Anordnung in einem Netz aus Spalten und Zeilen
- Eignung für Anordnung gleicher Komponenenten

Hbox und Vbox
- einfache Layouts um Komponenten horizontal bzw. vertikal anzuordnen
- Können beliebig geschachtelt werden

```
HBox hBox = new HBox();
hBox.setPadding(new Insets(10));
VBox vBoxLeft = new VBox();
vBoxLeft.setStyle("-fx-padding:5px;-fx-background-color:gold");
hBox.getChildren().addAll(vBoxLeft);
vBoxLeft.getChildren().addAll(components);
```



Events

05 December 2014 12:11

Ist ein Ereignis das für die Appliation von Interesse ist.

- Mausklick
- Buttonklick
- Focus

Hauptklasse ist javafx.event.Event

Jedes Event-Object enhält:

- Event typ
- Source
- Target

Manche Subklassen stellen weitere Information zur Verfügung, z.B. MouseEvent mit koordinaten des Zeigers.

Event Typ

Ist eine Instanz der Klasse javafx.event.EventTyp
Ist eine hiearchische Struktur von unterschiedlichen Ereignistypen

Event Target

- Wird von Schnittstelle javafx.eventEventTarget realisiert
- Mit der Methode buildEventDispatchChain wird definiert wie das Ereignis ans Ziel kommt
- JavaFX-Klassen Windows, Scene und Node implementieren dieses Interface
- Dadurch haben Komponenten als Subklassen dieses Ereignis-Objekt vererbt

Event delivery Prozess

umfasst 4 Phasen:

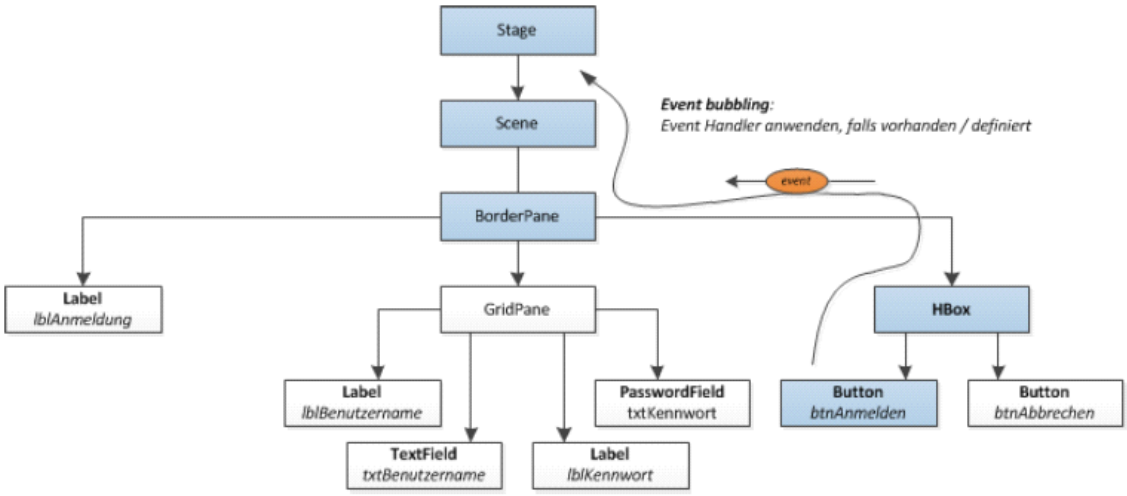
1. *Target selection*:

   Nach vordefinierten Regeln bestimmen, welche Komponente die Zielkomponente ist, wie z.B. auf welcher Komponente wurde ein Klick gemacht
2. *Route construction*:

   Den Weg von der Root-Komponente zur Ziel-Komponente bestimmen
3. *Event capturing*:

   Das Ereignis-Objekt von der Root-Komponente der Ziel-Komponente zustellen
4. *Event bubbling*:

   Das Eregnis-Objekt von der Ziel-Komponente zu der Root-Komponente zustellen



Event Hanler

Ist Objekt, das die Schnittstelle javafx.event.EventHandler implementiert
Die Methode public void handle (T event) führt eine bestimmte Aktion aus

Beispiel - Button Variante 1:

```
public class MyEventHandler implements EventHandler<ActionEvent> {
    private Button btn = null;
    private Label lbl = null;
    private String strHide;
    private String strShow;

    public MyEventHandler(Button btn, Label lbl, String strHide, String strShow) {  // …  }

    @Override
    public void handle(ActionEvent event) {
           if (lbl.isVisible()) {
               lbl.setVisible(false);
               btn.setText(strShow);
           } else {
               lbl.setVisible(true);
               btn.setText(strHide);
           }
    }
}
```

```
String strHide = "Hide Text";
String strShow = "Show Text";
String lblString = "Willkommen in die JavaFX-Welt!";

/* Label erstellen und dem 'root' hinzufügen */
Label lbl = new Label(lblString);
root.setCenter(lbl);

/* Button erstellen und dem 'root' hinzufügen */
Button btn = new Button(strHide);

/* EventHandler-Instanz erstellen */
EventHandler<ActionEvent> eHandler = new MyEventHandler(btn, lbl, strHide, strShow);

/* EventHandler dem Button hinzufügen */
btn.addEventHandler(ActionEvent.ACTION, eHandler);
```

- Das Event wurde dem Button als event target hinzugefügt.
- Die Klasse MyEventHanlder agiert als TopKlasse.
- Diese Variante eignet sich nur wenn man den Event Code auslagern möchte.

Beispiel - Button Variante 2:

```
btn.addEventHandler(ActionEvent.ACTION, new EventHandler<ActionEvent>() {
          @Override
          public void handle(ActionEvent event) {
                    if (lbl.isVisible()) {
                        lbl.setVisible(false);
                        btn.setText(strShow);
                    } else {
                        lbl.setVisible(true);
                        btn.setText(strHide);
                    }
          }
});
```

- Der EventHandler wird als lokale anonyme Klasse realisiert.
- Wie man sieht braucht es dazu weniger Code.
- Man kann direkt auf lokalen Komponenten zugreifen.

Beispiel - Button Variante 3:

```
btn.setOnAction(new EventHandler<ActionEvent>() {
    @Override
    public void handle(ActionEvent event) {
          if (lbl.isVisible()) {
              lbl.setVisible(false);
              btn.setText(strShow);
          } else {
              lbl.setVisible(true);
              btn.setText(strHide);
          }
    }
});
```

- convnienceMethode erlauben es die Arbeit noch einfacher zu ereldigen.
- Über die Methode setOnAction kann man relative einfach einen EventHandler für die Komponente festlegen.

Ohne Lambda-Ausdruck:

```
Button btnExit = new Button("Beenden");
btnExit.setOnAction(new EventHandler<ActionEvent>() {

    @Override
    public void handle(ActionEvent e) {
      Platform.exit();
    }
});
```

Mit Lambda-Ausdruck:

```
btnExit.setOnAction(e -> {
    Platfrom.exit();
});
```

Deklarative GUI Erstellung

12 December 2014 10:50

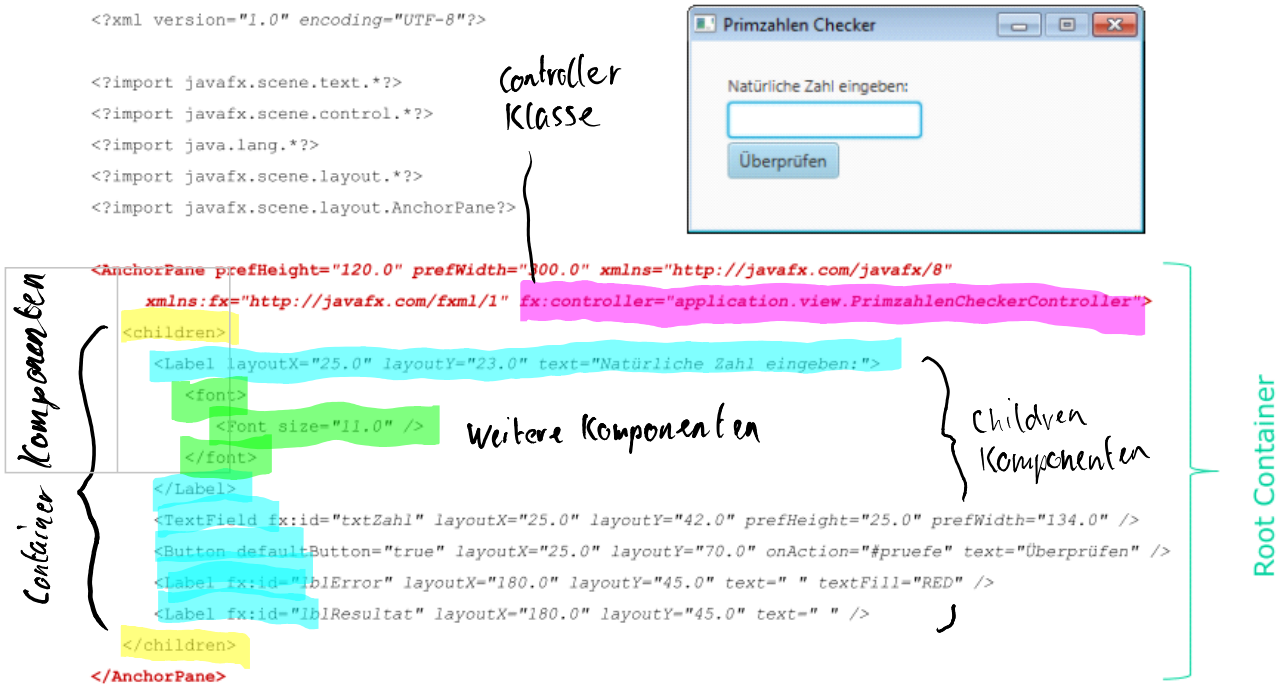
JavaFX Gui kann mit verschiedenen Komponenten erstellt und verknüpft werden.
Es ist aber auch möglich den Aufbau des gUIS mit XML, hier FXML, zu beschreiben.
Das MVC-Model sieht dann so aus:

- View: FXML
- Controller: Java
- Modell: Java

MVC bietet folgende Vorteile:

- Code-Wartbarkeit
- Getrennte GUI-Erstellung
- Einfachere Anpassung für verschiedene Ausgabegeräte (Desktop, Tablet, Phone)

Beispiel - Primzahlen Cheker:




```
<?xml version="1.0" encoding="UTF-8"?>

<?import javafx.scene.text.*?>
<?import javafx.scene.control.*?>
<?import java.lang.*?>
<?import javafx.scene.layout.*?>
<?import javafx.scene.layout.AnchorPane?>

<AnchorPane prefHeight="120.0" prefWidth="300.0" xmlns="http://javafx.com/javafx/8"
    xmlns:fx="http://javafx.com/fxml/1" fx:controller="application.view.PrimzahlenCheckerController">
  <children>
    <Label layoutX="25.0" layoutY="23.0" text="Natürliche Zahl eingeben:">
      <font>
        <Font size="11.0" />
      </font>
    </Label>
    <TextField fx:id="txtZahl" layoutX="25.0" layoutY="42.0" prefHeight="25.0" prefWidth="134.0" />
    <Button defaultButton="true" layoutX="25.0" layoutY="70.0" onAction="#pruefe" text="Überprüfen" />
    <Label fx:id="lblError" layoutX="180.0" layoutY="45.0" text=" " textFill="RED" />
    <Label fx:id="lblResultat" layoutX="180.0" layoutY="45.0" text=" " />
  </children>
</AnchorPane>
```

Das GUI wird mit der FXMLLoader-Klasse erstellt.

```
Parent root =
FXMLLoader.load(getclass().getResource("view/PrimzahlenChecke
rView.fxml"));
```

Die Root-Komponente ist als Typ javafx.scene.Parent zurückgeliefert.
Die Vererbungshierarchie erlaubt es auch die Root-Komponente explizit als Container-Typ AnchorPane umzuwandeln.