Zusammenfassung Java [Prog1]

sgehrig

Basis-Umwandlung

Hex -> Dezimal

"5F0A"$_{16}$ -> ?$_{10}$ = 24330$_{10}$

'5'	'F'	'0'	'A'
5*163	+ 15*162	+ 0*161	+ 10*160
((5*16	+ 15) *16	+ 0)*16	+ 10

Dezimal -> Hexadezimal

24330$_{10}$ -> ?$_{16}$ = "5F0A"

24330 =	1520*16	, Rest 10	letzte Ziffer: A
1520 =	95*16	, Rest 0	Ziffer davor: 0
95 =	5*16	, Rest 15	Ziffer davor: F
5 =	*0*16	, Rest 5	Ziffer davor: 5

Zweierkomplement

- Grundgesetz: a + (-a) = 0 / -(-a) = 0
- Java Operator ~ -> Einer-Komplement
  - jedes Bit wird invertiert (1110$_b$ -> 0001$_b$)
- a + (~a) = 11111111$_b$ = -1$_{10}$
- -1$_{10}$ + 1$_{10}$ = 11111111$_b$ + 1$_b$ = 0
- a + (~a) + 1$_b$ = 0
- -a = (~a) + 1$_b$ (Zweierkomplement)

Negative Zahlen von SIGNED-Datentypen

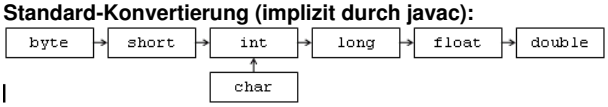
+a = 118$_{10}$ = 01110110$_b$
~a = ~01110110$_b$ = 10001001$_b$
-a = -118 = (~a + 1$_b$) = 10001001$_b$ + 1$_b$ = 10001010$_b$

00000000$_b$ = 0$_{10}$	10000000$_b$ = -128$_{10}$
0xxxxxxx$_b$ = x$_{10}$	1xxxxxxx$_b$ = -128$_{10}$ + x$_{10}$
01111111$_b$ = 127$_{10}$	11111111$_b$ = -1$_{10}$

Datentypen

Typ	Byte	Wertebereich	Init.
boolean	1	true, false	false
byte	1	- 2^7 bis 2^7 -1	0
short	2	- 2^{15} bis 2^{15} -1	0
int	4	- 2^{31} bis 2^{31} -1	0
long	8	- 2^{63} bis 2^{63} -1	0
float (~7 Stellen)	4	- 3.40282347 *10^{38} bis 3.40282347 *10^{38}	0.0
double (~16 Stellen)	8	- 1.79769313486231570 *10^{308} bis 1.79769313486231570 *10^{308}	0.0
char	2	alle Unicode - Zeichen	'\u0000'

Standard-Konvertierung (implizit durch javac):



Autoboxing zu Referenztyp *Object*, unboxing benötigt expliziten Cast.

Zahlenbasis (Prefix):

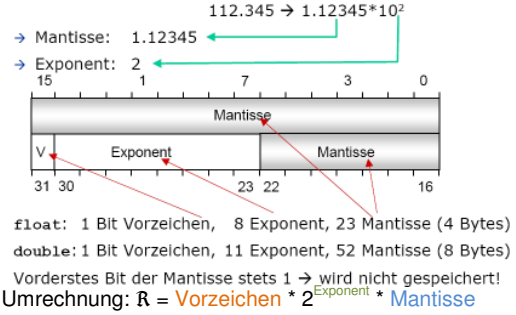
dezimal: int i = 30; // auch für byte / short / int
oktal: int i = **0**36;
hexa: int i = **0x**1e;

Zahlenbasis (Postfix)

long i = 30**L**;
double i = 30**D**;
float i = 30**F**;
Exponent float i = 6.1**e-3F**;

Exponentialdarstellung der Mantisse (float / double)

Interne Darstellung von Fliesskommazahlen (z.B. 112.345)



Umrechnung: ℝ = Vorzeichen * $2^{Exponent}$ * Mantisse

Berechnen von -6.5 (float):

-6.5 = 1 10000001 10100000000000000000000$_b$

Vorzeichen (1 bit) = 1$_b$	Vorzeichen von 6.5 ist ein Minus, daher V=1$_b$.
Exponent (8 bit) = 10000001$_b$	6.5 = 6 + 0.5 **6 = 110$_b$:** 6 / 2 = 3 Rest 0 (LSB) 3 / 2 = 1 Rest 1 1 / 2 = 0 Rest 1 (MSB) **0.5 = 0.1$_b$:** 0.5 * 2 = 1 [- 1] (MSB) 0.0 * 2 = 0 [- 0] 0.0 * 2 = 0 [- 0] (LSB) (…) 6.5 = 110,10…$_b$ 6.5 = 1,**1010…$_b$** * $2^{\mathbf{2}}$ Exponent = **2** + Exzess = **2** + 127 Exzess = (2^{n-1})-1, dabei ist n die Anzahl der Bits im Exponenten der Gleitkommazahl.
Mantisse (23 bit) = 1010…$_b$ [mit 19 Nullen auffüllen]	Die Mantisse entspricht dem Faktor vor dem Exponent ohne die führende Eins. Berechnung siehe oben.

Spezialfall (Exponent = 11111111)	Beschreibung
Mantisse = 11111111	Wert = -∞ (z.B. 2.0 / 0.0)
Mantisse = 00000000	Wert = ∞
Mantisse != 0	Wert = Not A Number (z.B. 0.0 / 0.0, oder ∞ - ∞)

Spezialfall (Exponent = 00000000)	Beschreibung
Vorzeichen = 0	Wert = 0.0
Vorzeichen = 1	Wert = -0.0

Operatoren nach Priorität (höchste zuoberst, links nach rechts)

[.] (z.B. *Klasse.Eigenschaft*)
++ / -- / + / - / ~ / ! / (*Cast*)
* / [/] / %
<< / >> / >>>
< / <= / > / >= / instanceof
== / !=
&
^
|
&&
||
?:
= / += / -= / *= / [/=] / %= / &= / |= / ^= / <<= / >>= / >>>=

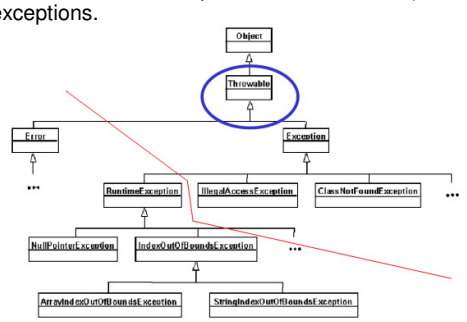
Zugriffs-Modifikatoren

private	kein Überschreiben möglich, aber neue Def. In Unterklasse
default	Package Scoped, nur im definierten Paket sichtbar.
protected	In der aktuellen und allen abgeleiteten Klassen sichtbar.
public	Von überall her sichtbar.

Achtung: Eine überschreibende Methode (in einer Unterklasse) darf die Zugriffsmodifikatoren nur erweitern, nicht einschränken.

Exceptions

Links unchecked exception, rechts checked (throws-pflichtige) exceptions.



Klassen-Definition

[Modifikatoren] Klassenname[<[TypParameter] [, TypParameter]*>]
[extends Abstrakte-Klassenname
| implements [Interfacename] [, Interfacename]*]

Methoden-Definition

[Modifikatoren] [<[TypParameter] [, TypParameter]*>] Rückgabetyp
Methodenname**(**[Parameterliste]**)**

enum-Definition

[Modifikatoren] enum Enumname

```
public enum AmpelFarbe { ROT, GELB, GRUEN }
if (farbe == Ampelfarbe.ROT) { /* … */ }
Ampelfarbe f = AmpelFarbe.value("GELB");
```

Generics

Annahme: T ist Formaler Typ-Paramter:

- T kann nicht in Klassenmethoden/-feldern verwendet werden.
- ref instanceof T ist nicht zulässig
- new T(…) nicht zulässig
- T[] arrayOfT = new T[5]; nicht zulässig
- C<T>[] arr = new C<T>[5]; nicht zulässig.

SubTyping ist nicht erlaubt; mit der „unbounded Wildcard“ <?> kann eine Referenz auf irgendeine Instanz definiert werden.

Das „Liskov Substitution Prinziple“

Methoden, die Referenzen auf Klassen benutzen, müssen in der Lage sein, Objekte von davon abgeleiteten Klassen zu benutzen, ohne es zu bemerken.

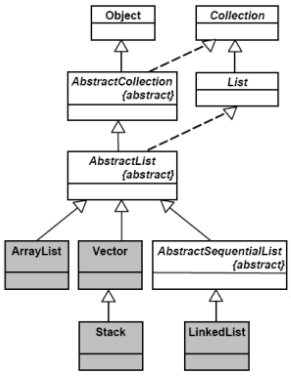
Polymorphie / OOP

Code für Basisklassen kann später von Objekten beliebiger, abgeleiteter Klassen benutzt werden. (mehrdeutig)

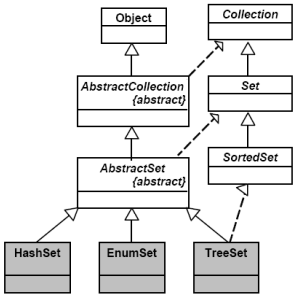
Objektorientierung ist…

- Alles ist ein Objekt; Programm ist ein Haufen Objekte
- Objekte können Objekte enthalten
- Objekt hat Typ
- Objekte eines bestimmten Typs können dieselbe Menge von Meldungen (Meldungen=Methd.) empfangen (-> Protokoll).

Lists / Collections & Sets



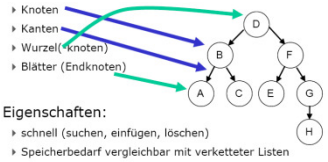
Alle Collections geordnet.
- LinkedList (schnelles einfügen) / ArrayList (schneller Zugriff) / Vector (synchronisiert): Zugriff über Index
- Stack: Sequenzielle Liste nach dem FILO-Prinzip



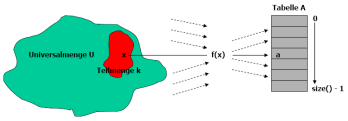
Sets beinhaltet eine ungeordnete Menge von Objekten, wobei jedes nur einmal vorkommen darf.
- TreeSet: Sortiertes Set
- HashSet: Ungeordnet, schneller Zugriff

Spezielle Container:

TreeSet:

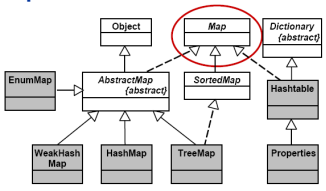


HashSet:



Schnell (wenn gute Streuung); muss deutlich mehr Einträge aufweisen als erwartet werden (ansonsten Kollisionsgefahr)

Maps / Dictionaries / Assoziative Arrays



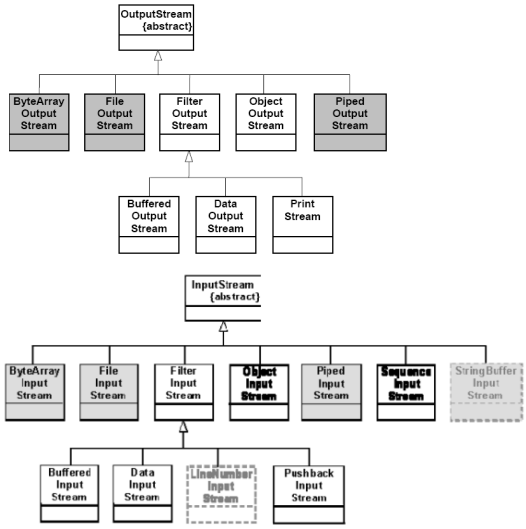
Maps beinhalten Schlüssel-Werte-Paare; Ablage/Zugriff mit (eindeutigem) Key; Values können Duplikate enthalten.
- TreeMap: Sortiert
- HashMap: Ungeordnet, schneller Zugriff
- Hashtable: Sync., Key-Valuelist

Queues:

Sequenzielle Listen nach dem FIFO-Prinzip (LinkedList geordnet / PriorityQueue sortiert); Zugriff sequenziell auf nächstes Element.

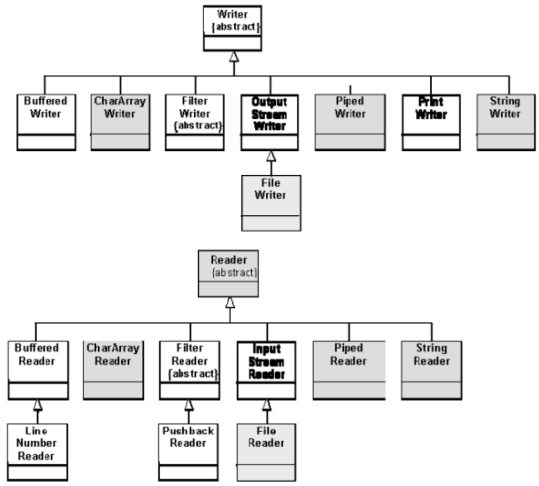
Input/Output Streams (Sink,Spring=grau / Processing=weiss)

Klassen, die von InputStream (Springstream) oder OutputStream (Sinkstream) abgeleitet sind, nennt man Bytestream-Klassen – sie sind hauptsächlich für die Verarbeitung einzelner Bytes verantwortlich.



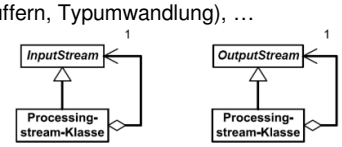
Reader / Writer (Sink,Spring=grau / Processing=weiss)

Klassen, die von Reader (Springstream) oder Writer (Sinkstream) abgeleitet sind, nennt man Characterstream-Klassen – sie sind hauptsächlich für die Verarbeitung von Zeichen verantwortlich.
Die Input/OutputStreamReader & -Writer Klassen können mit den Input/Output Streams verknüpft werden (Konstruktor-Argument).



Processingstream-Klassen

Ein Objekt einer Processingstream-Klasse benutzt intern ein Objekt einer Sinkstream-/Springstream- Klasse (Decorator-Pattern) und erweitert deren Funktionalität, zum Beispiel für Optimierungen, Erweiterungen (Puffern, Typumwandlung), …



- Statt `InputStream` könnte auch `Reader`, statt `OutputStream` könnte auch `Writer` stehen.
- Es gibt auch „gemischte“ Fälle: `InputStreamReader` erbt von `Reader` und verwendet `InputStream`

Inner- & Nested- Klassen

Elementklasse (In Klasse / mit Zugriffsmodifikator)
Hat den Charakter eines Instanzfeldes. Elementklassen können nur existieren, wenn ein Objekt der umschliessenden Klasse existiert. Zugriff auf äussere Klasse mit *this* oder *[Elternklassenname].this*. Instanzierung mit *ref.new Inner();*

Anonyme Klasse / Lokale Klasse (In Block / ohne Zugriffsmodifikator)
Objekte von lokalen und anonymen Klassen haben den Charakter von lokalen Variablen. Es kann nur auf die lokalen *final*-Variablen des umschliessenden Blocks Zugegriffen werden. Die Werte dieser Variablen werden bei der Instanziierung der inneren Klasse in die innere Kl. kopiert.
Static Klasse (In Klasse / mit Zugriffsmodifikator + static)

Gleiche Eigenschaften wie normale Klasse, die Elternklasse gilt als übergeordneten Namensraum.