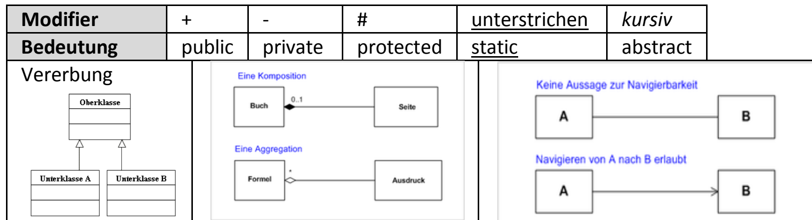
abstract	continue	for	new	switch	assert	default	goto†	package	synchronized	boolean
do	if	private	this	break	double	implements	protected	throw	byte	else
import	public	throws	case	enum	instanceof	return	transient	catch	extends	int
short	try	char	final	interface	static	void	class	finally	long	strictfp
volatile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

Initialisierung von Variablen

	Lokale Variablen	Klassen- / Instanzdatenfelder
Wert nacht Definition	undefiniert	Default-Wert
Initialisierung	Muss manuell geschehen	automatisch Default-Wert

UML

Modifier	+	-	#	unterstrichen	*kursiv*
Bedeutung	public	private	protected	static	abstract



Speicherbereiche für Variablen
Stack (statischer Speicher): lokale Variablen, lokale Referenzen; Heap (dynamischer Speicher): dynamische Objekte (mit new angelegt); Method Area (Code der Methoden): Klassenvariablen, Code der Methoden einer Klasse

Initalisierung von Variablen
Empfohlen: Immer alle Variablen von Hand initialisieren. Zeitpunkt der Initialisierung: Beim Laden der Klasse, die Klassendatenfelder. Beim Anlegen eines Objektes die Instanzdatenfelder. Primitive Datentypen werden bei der Initialisierung mit dem Default-Wert gefüllt. int zb. mit 0

Interface: Pfeil wie Vererbung, aber GESTRICHELT.

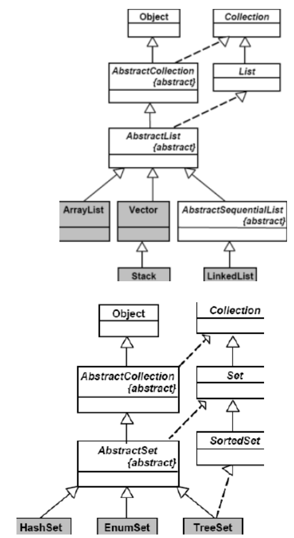
Datentypen

Typ	Byte	Wertebereich	Default-Wert
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 $2^{63}-1$	0L
float	4	$-3.4 * 10^{38}$ bis $3.4 * 10^{38}$	0.0f
double	8	$-1.7 * 10^{308}$ bis $1.7 * 10^{308}$	0.0d
char	2	alle Unicode Zeichen max. 65535	\u0000

Operatoren nach Priorität (höchste zuoberst, links nach rechts)

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

		Organisation	Freiheit beim Zugriff	Mechanismus beim Zugriff	Duplikate	Besonderheiten
Listen	ArrayList	geordnet	wahlfrei	über Index	ja	schneller Zugriff
	LinkedList	geordnet	wahlfrei	über Index	ja	schnelles Einfügen
	Vector	geordnet	wahlfrei	über Index	ja	synchronisiert
	Stack	geordnet	sequenziell	letztes Element	ja	LIFO
Queues	LinkdedList	geordnet	sequenziell	nächtes Element	ja	-
	PriorityQueue	sortiert	sequenziell	nächstes Element	ja	-
Sets	HashSet	ungeordnet	wahlfrei	einfacher Test	nein	schneller Zugriff
	TreeSet	sortiert	wahlfrei	einfacher Test	nein	-
Maps	HashMap	ungeordnet	wahlfrei	über Schlüssel	Schlüssel nein Werte ja	schneller Zugriff
	TreeMap	sortiert	wahlfrei	über Schlüssel	Schlüssel nein Werte ja	-



Sequenz (feste Ordnung): Liste
Schneller Zugriff über Index: ArrayList
Viele Elemente am Anfang und Ende einfügen: LinkedList
Reihenfolge uninteressant, aber schnell Entscheidung ob Element in der Liste ist: HashSet
Elemente nur einmal und immer sortiert: TreeSet
Assoziation zwischen Schlüssel und Elemente: Map von Vorteil.

Hashtable ist auch synchronisiert
Wichtigste Methode vom Interface List: add/get
<T extends A> T muss vom Typ A sein oder abgeleitet davon.

Überladene Methoden: Unter einer "überladenen" Methode versteht man zwei Methoden gleichen Namens in derselben Klasse, mit gleichen Rückgabetypen und mit unterschiedlichen Parametern.

Modulo: Bei Modulo gibt das Vorzeichen des Ersten Wertes das Vorzeichen des Resultats an, unabhängig ob der zweite Wert >0 oder <0 ist. b = 2; b = b++; ← b ist nach der Zuweisung 2. 2 wird intern vermerkt, b erhöht und der vermerkte Wert wird zugewiesen.

Operatoren: ^: XOR (3^4=7), &: AND (3&5=1), |: OR (3|5=7), >>: vorzeichenbehaftet Rechtsshift (VZ= -1: mit 1 auffüllen, sonst 0), >>>: vorzeichenloser Rechtsshift (immer mit 0 auffüllen), <<: linksshift (mit 0en auffüllen). Bei den Bitshift OPs achten auf Anz. Bits (zb. 32 bei int)

Call-By-Value: Bei primitiven Datentypen gibt es nur Call-by-Value, call-by-reference gibt es nicht! Wird eine Methode mit einem Referenztyp als Parameter aufgerufen, wird dessen Referenz übergeben und einer lokalen Variable zugewiesen. Das Objekt ist aber nur einmal vorhanden, es gibt aber zwei Referenzen!

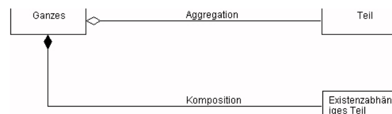
Float

Vorzeichen: 1 Bit	Exponent: 8 Bit	Mantisse: 32 Bit

Gleitkommazahl zu Dezimalzahl:

$$Z = (-1)^{VZ} * \left(1 + \frac{M}{2^{23}}\right) * 2^{E-127}$$

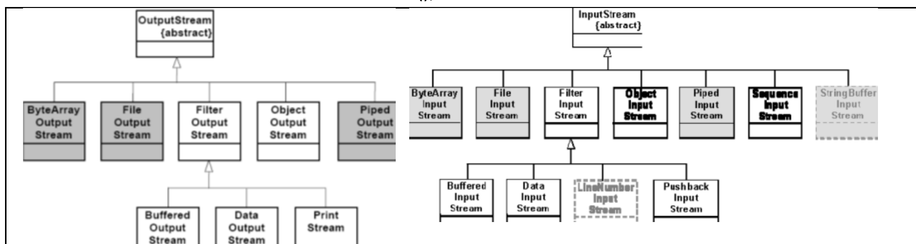
Zu Gleitkommazahl:
Zahl = 11.25
Vorzeichen: 0 (da die Zahl positiv ist, bei negativen Zahlen ist VZ 1)
Exponent: $ld(11.25) = 3.49 = 3 \rightarrow 3 + 127 = 130 \rightarrow 1000'0010$
Mantisse: $\left(\frac{11.25}{2^3} - 1\right) * 2^{23} = 3407872 \rightarrow 0110'1000'0000'0000'0000'000$
Resultat: VZ + Exp. + Mant. = $0'1000'0010'0110'1000'0000'0000'0000'000$



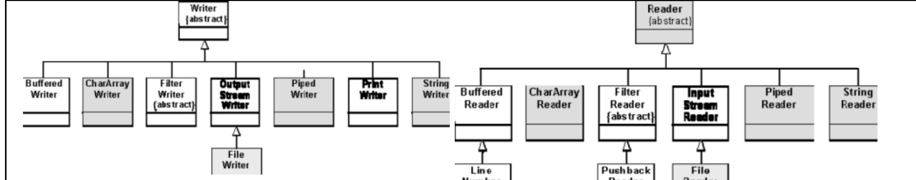
Double			
Vorzeichen: 1 Bit	Exponent: 11 Bit	Mantisse: 52 Bit	Formel: 2^{52} statt 2^{23} und 2^{E-1023} statt 2^{E-127}
Spezialfälle: Exponent 11111111: Mantisse nur 1en: Wert: -Infinity, wenn Mantisse nur 0en: +Infinity, Mantisse != 0: NaN (not a number)			

2er Komplement: Zahl binär aufschreiben und mit 0en auffüllen (siehe Anz. Bytes in Wertetabelle), alle Bytes umkehren (0->1, 1->0) und 1 addieren (0+1=1, 1+1=0 mit Übertrag 1 auf nächste Stelle). Bsp: short 118: **00000000‘01110110**, invertieren: **11111111‘10001001**, +1= **11111111‘10001010** <- (-118), Vorderstes Bit ist Vorzeichen.

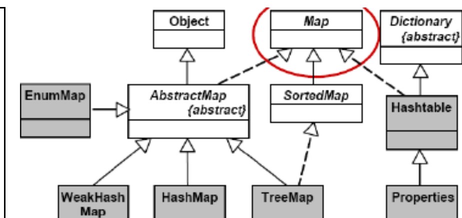
Klassenvariablen: static, **Konstanen**: final.
Späte / dynamische Bindung: Zur Laufzeit wird erkannt, dass die Referenz auf eine abgeleitete Klasse zeigt. Fehler werden also erst zur Laufzeit erkannt. Java verwendet fast immer späte Bindung. Ausnahmen: Klassenmethoden, private und finale Methoden.
Konstruktor / Vererbung: super() ist nur als erstse Instruktion des Konstruktors erlaubt.
Compiler macht immer einen default-Konstruktor, wenn gar kein Konstruktor vorhanden ist.
In einem Konstruktor kann ein überladener Konstruktor mit „this(…)“ aufgerufen werden.
Finalize: In der finalize()-Methode ist als letztes super.finalize() aufzurufen!
Casting: Einer Referenz vom Typ Kind, kann keine Referenz der Eltern zugewiesen werden. Ohne Casting: Compilerfehler, mit casting: Laufzeitfehler. Umgekehrt nicht: Eltern refEltern = refKind;**Innere Klassen instanzieren**: Normale Klasse: Test, innere Klasse InnerTest. Um Test x = new Test(); InnerTest bla = x.new InnerTest();
Finale Klassen: Von ihnen kann nicht geerbt werden.
Methoden können nur abstract sein, wenn die Klasse auch abstract ist.
Abstract Methoden MÜSSEN public oder protected sein. Final, r static oder private geht NICHT.
static Methoden können nur static Methoden aufrufen, ausser die nicht static Methode wird per Instanzvariable aufgerufen.
Methoden in einem Interface können nur public und/oder abstract sein. Finale Methoden können nicht überschrieben werden, überladen allerdings schon.
Klonen: Klasse: implements Cloneable, public Typ clone() { try { super.clone() } catch {CloneNotSupportedException e {}}
Exceptions: Wenn die Sohnklasse von der Vaterklasse erbt und Methoden überschreibt, die eine Exception werfen, so darf der Exceptiontyp in der Sohnmethode nicht allgemeiner sein. zB. in Vater FileNotFoundException und in Sohn IOException. Geht nicht! Umgekerht schon! Beide IOException auch gut!



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

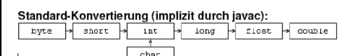


Klassen, die von Reader (Springstream) oder Writer (Sinkstream) abgeleitet sind, nennt man Characterstream-Klassen – sie sind hauptsächlich für die Verarbeitung von Zeichen verantwortlich. Die Input/OutputStreamReader & -Writer Klassen können mit den Input/Output Streams verknüpft werden (Konstruktor-Argument).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




```
int x = -13;
double d = 7.0;
int y = 2;
System.out.println( ++x + " und " + y++ ); //-12 und 2
System.out.println( - x % 7 ); //5
System.out.println( y^7 ); //4
System.out.println( d / 0.0 ); //Infinity
System.out.println( x / y ); //-4
System.out.println( x / (y - 2) ); //-12
public class Initialisierung {
    String str = 1+2; //-
    short[][][] s_3d = new short[3][3][3]; //+
    float zahl = 2f; //+
    final int = 3.23; //-
    Kreis[] refArray = new Kreis[]; //-
    final long VARIABLE; //+
    char[] ch = {12, 13, 14}; //+
    Object objekt = new Object(); //+
}
```

Standard-Konvertierung (implizit durch javac):



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

Signatur: Methodenname + Parameterliste. Rückgabetyp nicht!
Überladen: Verschiedene Methoden mit gleichen Namen aber anderen Parametern. (Rückgabetyp darf ändern). Ein Überladen mit gleicher Signatur, aber anderem Rückgabetyp ist nicht möglich.
Überschreiben: Signatur und Rückgabetyp müssen identisch, mit der überschriebenen Methode sein. Nur bei Instanzmethoden möglich.
Geschachtelte Klassen: Wenn in der äusseren Klasse, eine Instanz der geschachtelten Klasse gemacht werden soll, so muss dies durch eine Instanz der äusseren Klasse gemacht werden: InnereKlasse inner = new AussereKlasse().new InnereKlasse();
Parameterübergabe: Double d1 = 5.0; static void setToTen(Double d) { d+= 10.0;} setToTen(d1); ← d1 ist nachher immernoch 5, da in setToTen bei der Übergabe die Referenz kopiert und einem neuen Objekt zugewiesen wurde, dann wird: d = new Double(d.getValue()); gemacht. D.h. dem lokalen Objekt wird eine neue Referenz zugewiesen, d1 ist davon aber nicht betroffen. Double ist unveränderbar.
CallByValue: Alle Wrapperklassen sind unveränderbar. Float f = 5; f += 2; ← f = new Float(f.getValue()+2);
Anonyme Klassen:lokale Klassen ohne Namen. Sofort bei deren Definition wird ein Objekt erzeugt. Konstruktor definieren: Nicht möglich
Switch: case 1+2: ← funktioniert, wird als case 3 ausgewertet.
Stringvergleich: String s1 = „xyz“; String s2 = „xyz“; s1 == s2; ← true, s2 = new String(„xyz“); s1 == s2; ← false
Final: Final Variablen können einmal instanziert werden, danach darf (und kann) die Referenz nicht mehr verändert werden. Nurnoch der Wert.

Initialisierungsblock: Initialisierungsblöcke werden vor den Konstruktoren ausgeführt. Zuerst von der Oberklasse, dann von der erbenden Klasse, dann Konstruktor der Oberklasse, dann Konstruktor der erbenden Klasse.

Was nicht verebrt wird: Konstruktor, private Sachen, Initialisierungsblöcke
Objekt-Orientierung: Wichtig: Kapselung, Information hiding, Vererbung, Polymorphie, Identität
Float: float f = 2.0; ← geht nicht, müsst 2.0f sein, oder (float)2.0;
Variablennamen: Wenn vor ein reserviertes Wort ein $ kommt: erlaubt!