Woche 03

Basistypen, Operatoren, Datenkapselung, Call-by-Value und JUnit Testing

Organisatorisches

Erste ÜPA ist bald!

Anmeldung

- Anmeldung nicht vergessen!
- Am besten jetzt!
- Anmeldung ist Pflicht! ⇒ Keine Anmeldung = keine Teilnahme!
- Nachträgliche Anmeldung ist **nicht** möglich!

Tipps

- Zeit gut einteilen
  - Wenn ihr mit einer Aufgabe nicht weiterkommt, macht mit der nächsten weiter
- Aufgabenstellung genau lesen
  - Bestehende Methoden und Attribute nicht verändern (umbennen, Parameter hinzufügen, etc.)
- Oft pushen!
  - Nicht dass es last-minute nicht klappt, und ihr alles an Arbeit verliert
- Oft lokal testen!
- Letzter Versuch zählt!
- Aufpassen, keinen Code zu pushen, der nicht kompiliert!

Basistypen

Übersicht

Datentyp	Informations-gehalt (in Bits)	Minimaler Wert	Maximaler Wert	Beispiele für Literale
boolean	1	-	-	false, true
byte	8	-128	127	4, -3
short	16	-32 768	32 767	4, -3
int	32	-2^{31}	$2^{31}-1$	4, -3
long	64	-2^{63}	$2^{63}-1$	6L, -2L
float	32	ca. $-3.4*10^{38}$	ca. $3.4*10^{38}$	1.86f, 4.0f
double	64	ca. $-1.7*10^{308}$	ca. $1.7*10^{308}$	-3.0, 2.718
char	16	0	65535	'd', '✎', '쌤'

Basistypen

Ganzzahltypen - Overflows

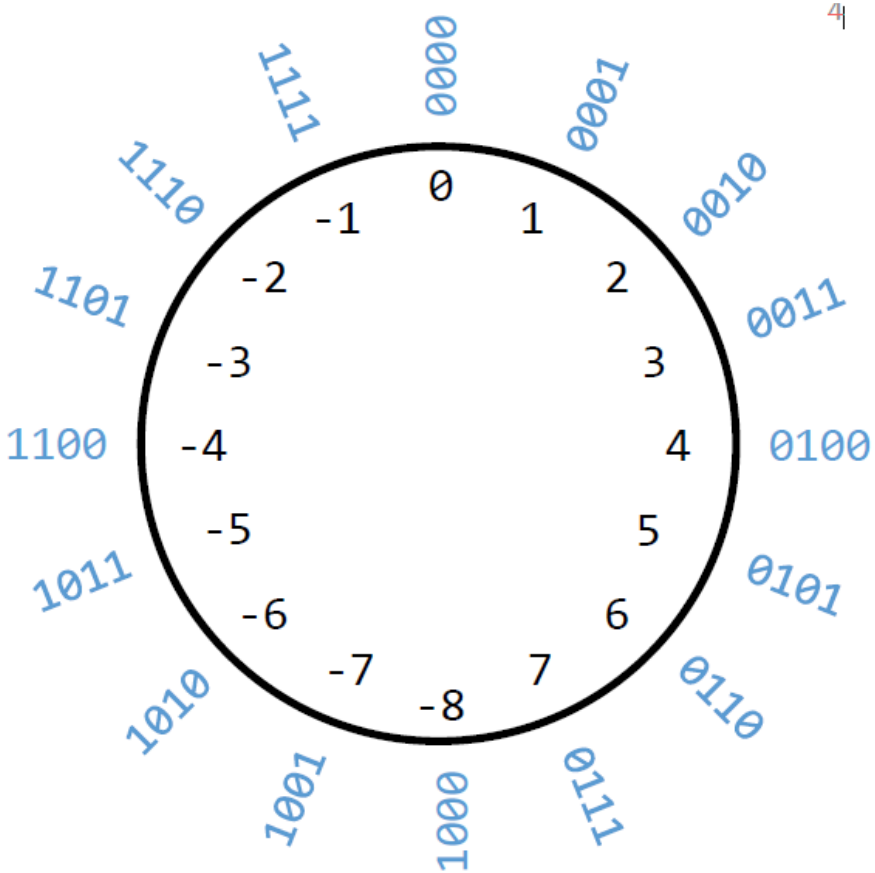
```
1  int maximalInt = 2147483647;
2  int greaterInt = maximalInt + 1;
3  System.out.println(greaterInt);
4  int evenGreaterInt = maximalInt + 5;
5  System.out.println(evenGreaterInt);
```

Output:

```
1  -2147483648
2  -2147483644
```

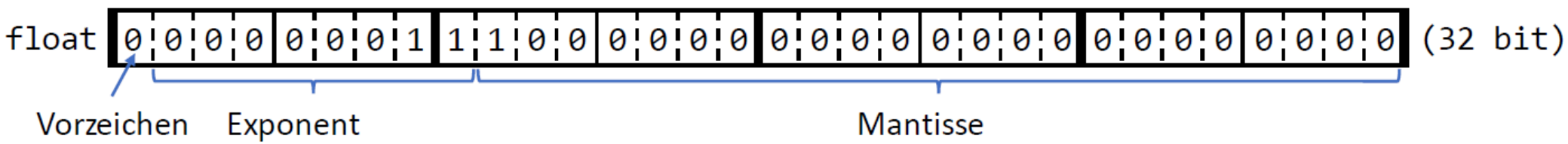
Overflows erzeugen keine Fehlermeldungen!

- sind daher häufige Fehlerquellen
- sicherstellen dass Ergebnisse in Typen passen!



Basistypen

Darstellung - Gleitkommazahlen




```
Zahl = Vorzeichen 1.Mantisse * 2^(Exponent-127)
```

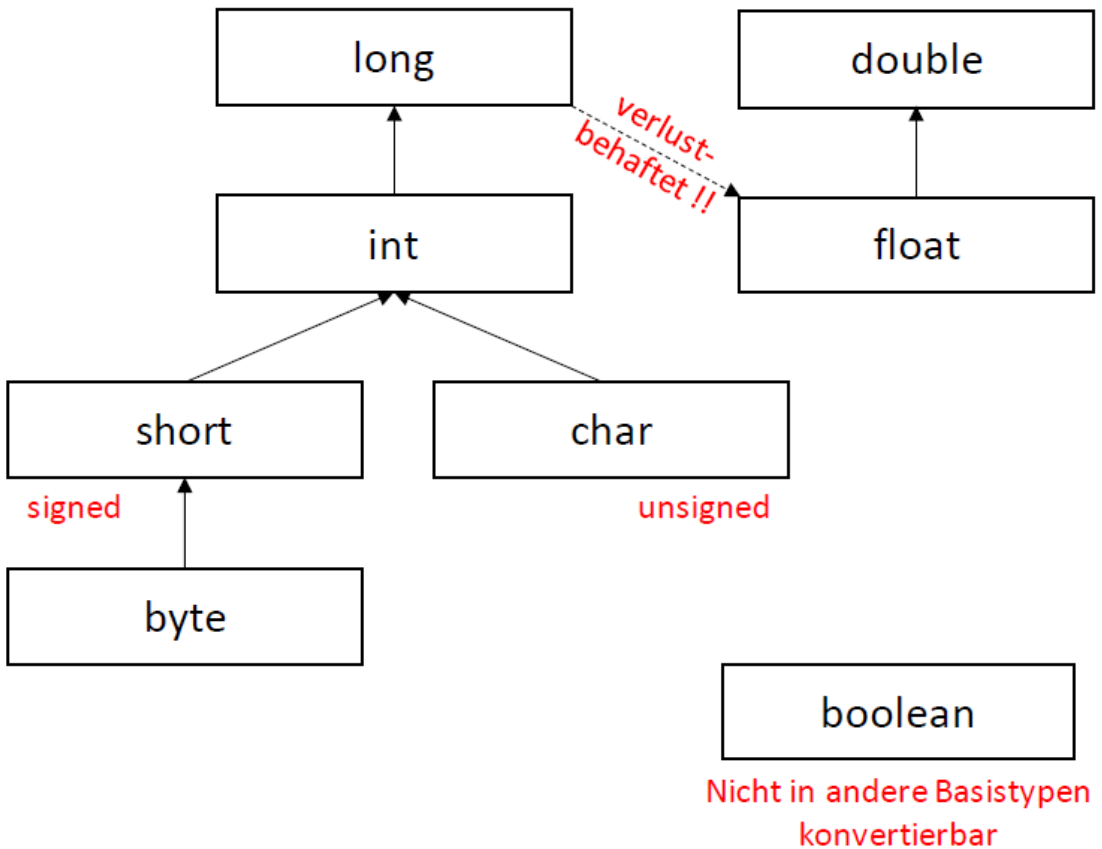
Basistypen

Implizite Casts

Entlang der Pfeile im Diagramm rechts können Werte ohne Weiteres von einer Variable eines Typs in eine Variable eines anderen übertragen werden:

```
1  byte b = 70;
2  short s = b;
3  int n = 4_000;
4  float f = n;
```

funktioniert beides ohne Probleme. Wenn ein Ganzzahltyp in einen Gleitkommatyp überführt wird, können allerdings **Datenverluste** auftreten.

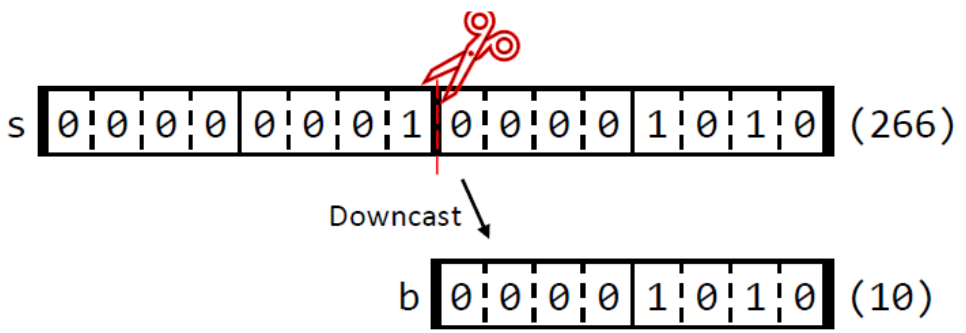


Basistypen

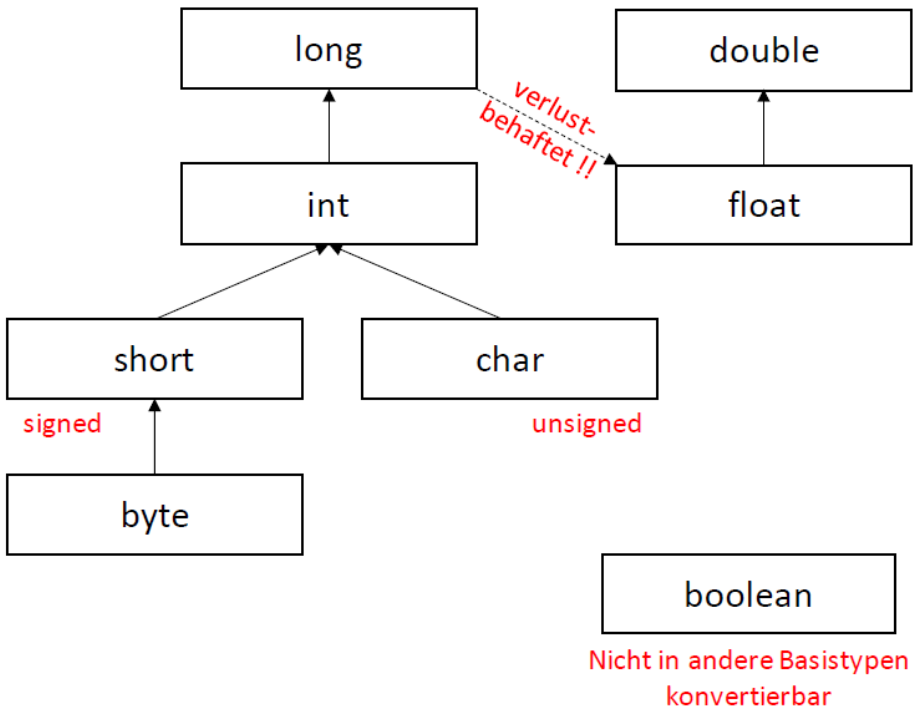
Explizite Casts

Entgegen der Pfeile muss man Java explizit mitteilen, dass man eine Umwandlung von einem Typen in den anderen vornehmen will.
Denn: Dabei geht i.d.R. Information verloren.

```
1  short s = 266;
2  byte b = (byte) s; // b == 10
```




```
1  float f = -4/5f;
2  int n = (int) f; // n == 0
```



Gleitkommatyp zu Ganzzahltyp:
Nachkommastellen werden abgeschnitten.

Operatoren

Übersicht

	Operator	Beschreibung
Unäre Operatoren	`x++, x--`	Postinkrement (-dekrement): Erhöht (erniedrigt) Variable x um eins und gibt den alten Wert zurück.
	`++x, --x`	Präinkrement (-dekrement): Erhöht (erniedrigt) Variable x um eins und gibt den neuen Wert zurück.
	`-, !, ~, +, (type)`	Weitere unäre Operatoren
Binäre Operatoren	`*, /, %`	Arithmetische Operatoren
	`+, -`	
	`>>, <<, >>>`	Bitshifts
	`<, >, <=, >=, instanceof`	Vergleiche (Ungleichheit); `instanceof` kommt später!
	`==, !=`	Vergleiche (Gleichheit)
	`&, ^, |`	Bitweise logische Operatoren
	`&&, ||`	„Normale“ logische Operatoren
	`? :`	Ternärer Operator
Binäre Op.	`=, +=, -=, *=, /=, %=, &=, ^=, |=, <<=, >>=, >>>=`	Zuweisungen

Blau: Solltest du ab dieser Woche kennen. Alle nachschlagen, die du noch nicht kennst!
Grün: Musst du nicht kennen, ist aber in einigen Situationen sehr nützlich.
Orange: Kommt später im Kurs noch dran.
Schwarz: Nicht wichtig für diesen Kurs. Bei Interesse vllt. mal anschauen.

Operatoren

Übersicht

	Operator	Beschreibung
Unäre Operatoren	`x++, x--`	Postinkrement (-dekrement): Erhöht (erniedrigt) Variable x um eins und gibt den alten Wert zurück.
	`++x, --x`	Präinkrement (-dekrement): Erhöht (erniedrigt) Variable x um eins und gibt den neuen Wert zurück.
	`-, !, ~, +, (type)`	Weitere unäre Operatoren
Binäre Operatoren	`*, /, %`	Arithmetische Operatoren
	`+, -`	
	`>>, <<, >>>`	Bitshifts
	`<, >, <=, >=, instanceof`	Vergleiche (Ungleichheit); `instanceof` kommt später!
	`==, !=`	Vergleiche (Gleichheit)
	`&, ^, |`	Bitweise logische Operatoren
	`&&, ||`	„Normale“ logische Operatoren
	`? :`	Ternärer Operator
Binäre Op.	`=, +=, -=, *=, /=, %=, &=, ^=, |=, <<=, >>=, >>>=`	Zuweisungen

Unär: Arbeitet auf einer Variablen. z.B. !isEqual
Binär: Arbeitet auf zwei Variablen. z.B. num1 + num2
Ternär: Arbeitet auf 3 Variablen.

Operatoren

Präzedenz

	Operator	Beschreibung
Unäre Operatoren	`x++, x--`	Postinkrement (-dekrement): Erhöht (erniedrigt) Variable x um eins und gibt den alten Wert zurück.
	`++x, --x`	Präinkrement (-dekrement): Erhöht (erniedrigt) Variable x um eins und gibt den neuen Wert zurück.
	`-, !, ~, +, (type)`	Weitere unäre Operatoren
Binäre Operatoren	`*, /, %`	Arithmetische Operatoren
	`+, -`	
	`>>, <<, >>>`	Bitshifts
	`<, >, <=, >=, instanceof`	Vergleiche (Ungleichheit); `instanceof` kommt später!
	`==, !=`	Vergleiche (Gleichheit)
	`&, ^, |`	Bitweise logische Operatoren
	`&&, ||`	„Normale“ logische Operatoren
	`? :`	Ternärer Operator
Binäre Op.	`=, +=, -=, *=, /=, %=, &=, ^=, |=, <<=, >>=, >>>=`	Zuweisungen

Präzedenz:
Die Operatoren sind von oben nach unten nach Präzedenz sortiert. D.h. in einem Ausdruck mit mehreren Operatoren werden die weiter oben stehenden zuerst ausgeführt.

```
-15 == 6 + 7 * -3
```

Der unäre Operator – wird zuerst auf die 15 und die 3 angewandt, dann wird * ausgeführt, dann +, dann ==. Das Gesamtergebnis ist true.

Call-by-Value

```
1  public static void main(String[] args) {
2  int myNumber = 2;
3  addOne(myNumber);
4  System.out.println(myNumber);
5  }
6  public static void addOne(int number) {
7  number = number + 1;
8  }
```

Output: 2

Call-by-Value

Anders bei Klassenattributen

```
1  public class MyNumber {
2    public int value = 2;
3    // default constructor
4    public static void inc(MyNumber n) {
5      n.value += 1;
6    }
7  }
8
9  public static void main(String[] args) {
10   MyNumber myN = new MyNumber();
11   inc(myN);
12   System.out.println(myN.value);
13 }
```

Output: 3

Datenkapselung

Beispiel: Einkaufen

Der Kunde an einem Kiosk will/kann:

- Fragen über Sortiment stellen
- Waren erhalten
- bezahlen

Er will/darf nicht:

- Artikel umsortieren, oder wissen wie sie hinter der Theke gelagert werden
- Waren hinter der Theke selber abholen
- An die Kasse greifen

Datenkapselung

Erklärung des Konzepts

Der Benutzer einer Klasse will:

- wissen welche Teile für ihn relevant sind
- sich nicht im Detail mit der Klasse auseinander setzen müssen
- nicht aus Versehen Sachen kaputtmachen

⇒ **nur mit Teil der Klasse interagieren**

Der Entwickler einer Klasse will:

- sich auf möglichst wenig festlegen
- möglichst viel später ändern können
  - geht nur, wenn der zu ändernde Teil nicht extern benutzt wird!

⇒ **nur ein Teil der Klasse sichtbar machen**

public: für alle (inkl. Benutzer) relevant
private: nur für den Entwickler der Klasse relevant

Zugriffsmodifier

Übersicht

Zugriffsmodifier	Zugriff
public	überall
protected	kommt erst nach der Vererbung
default (ohne Schlüsselwort)	vom Code aus dem gleichen Package
private	nur vom Code der selben Klasse

JUnit Testing

Setup mit Gradle

Wir verwenden JUnit 5. Das muss evt in der build.gradle spezifiziert werden.

```
...
dependencies {
  implementation 'org.junit.jupiter:junit jupiter:5.8.1'
}
...
```

Sobald du **@Test** über eine Methode tippst, sollte dir deine IDE allerdings beim darüberhovern bereits vorschlagen, JUnit in die Dependencies mit aufzunehmen, dann musst du dies nicht per Hand eintippen.
Achte darauf, dass du die richtige Version verwendest! JUnit 5 und nicht JUnit 4!

JUnit 5 Testing

Assertions

Um das Verhalten von Funktionen zu testen, vergleichen wir häufig den erwarteten Output mit dem tatsächlichen Ergebnis für einen bestimmten Input. Hierfür nutzen wir Assertions.

```
1  assertEquals(9 , square(3));
2  assertEquals("Hello World", concatenate("Hello", " ", "World");
```

JUnit bietet noch viele weitere Assertions an, wie…

- **assertArrayEquals()** für Arrays
- **assertEquals(expected, actual, delta)** für float s, bei der auf Gleichheit minus ein delta getestet wird
- **assertNotNull()**
- und viele mehr…

Wie immer bei Fragen empfehlen wir, die Dokumentation hier zu überfliegen.

JUnit 5

Imports

Um in einer Klasse die Funktionalitäten von JUnit 5 nutzen zu können, müsste ihr sie **importieren**. Die Imports der Annotationen und Assertions sehen dann etwa so aus:

```
import org.junit.jupiter.api.AfterEach;
import org.junit.jupiter.api.BeforeAll;
import org.junit.jupiter.api.Test;

import static org.junit.jupiter.api.Assertions.assertEquals;
import static org.junit.jupiter.api.Assertions.assertArrayEquals;
import static org.junit.jupiter.api.Assertions.assertNotNull;
```

Achtet darauf, dass die Annotationen und Assertions von der Jupiter API importiert werden, also im import-Statement der String “jupiter” enthalten ist.

(Die Imports kommen ganz nach oben in jede Klasse, die die jeweiligen Funktionalitäten verwendet, direkt unter das package-Statement, falls vorhanden.)