Javakurs FSS 2012

Lehrstuhl Stuckenschmidt

Tag 1 - Variablen und Kontrollstrukturen

main Methode

- Startpunkt jeder Java Anwendung
- String[] args ist ein Array aus Parametern, die beim Aufruf über die Kommandozeile übergeben wurden

```
public class MainClass {
  public static void main(String[] args){
    System.out.println("Ausgabe aus der main()-Methode");
  }
}
```

Kommentare

- Codeteile die nicht vom Compiler in Maschinencode umgewandelt werden
- Für Leser des Quellcodes
- Bessere Verständlichkeit des Codes
- Anmerkungen

Art	Beginn	Ende
Zeilenkommentar	//	
Blockkommentar	/*	*/
JavaDoc-Kommentar	/**	*/

Ausgaben

- Standardmäßig per System.out
- System.out.println() schreibt eine neue Zeile in der Ausgabe
- Konkatinierung möglich mit "+"
- Ausgaben eignen sich um Fehler im Programm zu finden

Beispiel Ausgaben

```
System.out.println("Ausgabe");
//Ausgabe

System.out.println("Wert der Variablen a:  " + a);
// Wert der Variablen a:  x

System.out.print("a");
System.out.print("b");
// ab
```

Deklaration

- Erstellung einer Variablen
- Zuweisung eines festen Datentyps (z.B. int)

```
int a;
String b;
```

Initialisierung

- Erste Zuweisung eines Wertes zu einer Variablen
- Kann mit Deklaration zusammengefasst werden

```
a = 10;
b = "String";

int c = 5;
```

Klammern

- () runde Klammern werden für Parameterübergabe und Bedingungen benutzt, außerdem können sie auch für Schachtelung verwendet werden
- {} geschweifte Klammern werden zur Eingrenzung von Codeblöcken benutzt, z.B. bei Methoden oder Schleifen, außerdem werden sie auch zum definieren von Mengen z.B. bei Arrays verwendet
- [] eckige Klammern geben mehrdimensionale Felder an
- <> spitze Klammern werden zur Typisierung von Objekten verwendet

Klammern

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

  int a = 3;

  if (a == 3) {
    a = (a + 1) * 2;
  }

  int c[] = new int[2];

  ArrayList<String> x = new ArrayList<String>();

}
```

Primitive Datentypen

- Numerische Datentypen sind vorzeichenbehaftet
- Datentypen haben eine festgesetzte Länge

$\Rightarrow$Overflows sind möglich

Primitive Datentypen

Typ	Länge in Byte	Belegung (Wertebereich)
boolean	1	true oder false
char	2	16-Bit Unicode Zeichen (0x0000 – 0xffff)
byte	1	-2^7 bis 2^7-1 (-128 – 127)
short	2	-2^15 bis 2^15-1 (-32768 – 32767)
int	4	-2^31 bis 2^31-1 (-2147483648 – 2147483647)
long	8	–2^63 bis 2^63 – 1 (–9.223.372.036.854.775.808 - 9.223.372.036.854.775.807)
float	4	1,40239846E-45 - 3,40282347E+38
double	8	4,94065645841246544E-324 - 1,79769131486231570E+308

Strings

- Kein primitiver Datentyp
- Symbiose von Klasse und Datentyp
- Kette von chars

```
String str = "abc";

//Äquivalent zu:
char data[] = {'a', 'b', 'c'};
String str = new String(data);

char a = str.charAt(0);           //a
boolean e = str.equals("abc");    //true
int l = str.length();             //3
str = str.replace('b', 'd');      //adc
```

null

- Referenzwert, falls kein Objekt referenziert wird

```
String a = null;
```

Cast (Typumwandlung)

- Primitive Datentypen außer boolean können in andere primitive Datentypen umgewandelt werden
- Unterscheidung zwischen implizit (automatisch) und explizit

Expliziter Cast

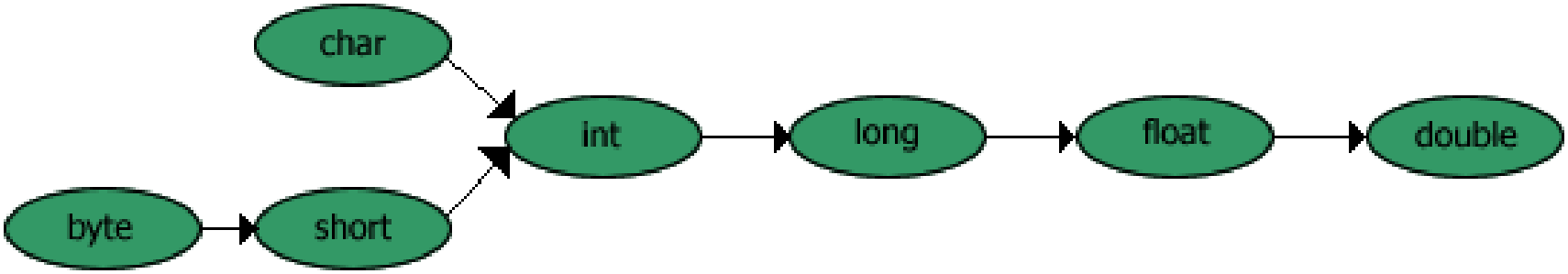
- Möglicher Informationsverlust
- Umwandlung indem neuer Datentyp in runden Klammern dem zu wandelnden Ausdruck vorangestellt wird

```
char c = 'c';
int i = (int) c;

double d = 1.8;
int e = (int) d;     //e ist 1, Verlust der Nachkommastelle
```

Impliziter Cast

- Konvertierung in den "größeren" Datentypen
- byte und short werden automatisch in int umgewandelt



Random

- Instanzen der Klasse Random generieren Zufallszahlen
- Verwendung:

```
Random random = new Random();
boolean b = random.nextBoolean();
int i = random.nextInt();
i = random.nextInt();
long l = random.nextLong();
float f = random.nextFloat();
double d = random.nextDouble();
```

Mathematische Operatoren

- Addition: +

```
int a = 2 + 3;          //5
a = a + 3;              //8
```

- Inkrement: ++

```
int a = 5;
int b = 5;

System.out.println(a++);    //5
Systen.out.println(a);      //6

System.out.println(++b);    //6
System.out.println(b);      //6
```

Mathematische Operatoren

- Subtraktion: -

```
int a = 3 - 2;          //1
a = a - 3;              //-2
```

- Dekrement: - -

```
int a = 5;
int b = 5;

System.out.println(a--);    //5
System.out.println(a);      //4

System.out.println(--b);    //4
System.out.println(b);      //4
```

Mathematische Operatoren

- Multiplikation:*

```
int a = 3 * 4;          //12
a = a * 3;              //36
```

- Division: /

```
int a = 36 / 3;         //12
a = a / 4;              //3
```

- Modulo: %

```
int a = 105 % 10;       //5
a = a % 10;             //5
```

Vergleichsoperatoren

- = vs. == vs. equals

  = ist kein Vergleichsoperator sondern ein Zuweisungsoperator

  == ist ein Vergleich von primitiven Datentypen

```
boolean a = (4 == 5);       //false
a = (5 == 5);               //true
```

equals vergleicht komplexe Datentypen und Objekte

```
String s = "abc";
s.equals("cde");     //false
s.equals("abc");     //true
```

Vergleichsoperatoren

- Ungleich: !=

```
int a = 2;
if (a != 0) {
  System.out.println("Dieser Block wird ausgeführt, wenn a
ungleich 0 ist.");
}
```

- Größer: >
- Grösser gleich: >=
- Kleiner: <
- Kleiner gleich: <=

Logische Operatoren

- & UND ohne Short-Circuit-Evaluation
- && UND mit Short-Circuit-Evaluation
- | ODER ohne Short-Circuit-Evaluation
- || ODER mit Short-Circuit-Evaluation
- ! Logisches Nicht
- ^ Exklusives Oder

- Bei Short-Circuit-Evaluation wird ein weiter rechts ausstehender Teilausdruck nur ausgewertet, falls er noch von Bedeutung ist

Logische Operatoren

```
boolean a = false;
boolean b = true;

boolean c = a & b;   //false, beide Teilausdrücke werden
                     //ausgewertet
boolean d = a && b; //false, nur der linke Teilausdruck
                     //(a) wird ausgewertet
```

```
boolean a = true | false;  //true, beide Teilausdrücke
                           //werden ausgewertet
boolean b = true || false; //true, nur der linke
                           //Teilausdruck wird ausgewertet
```

```
boolean a = true;
a = !a;                    //false
```

```
boolean a = true ^ false   //true
boolean b = true ^ true    //false
```

Bitlogische Operatoren

- & Und
- | Oder
- ^ Exklusives Oder
- ~ Komplement

```
byte i1 = 1;          // 00000001   1
byte i2 = 2;          // 00000010   2

byte i3 = i1 & i2;    // 00000000   0
byte i4 = i1 | i2;    // 00000011   3
byte i5 = i1 ^ i2;    // 00000011   3
byte i6 = ~i1;        // 11111110   -2
```

Verschiebungsoperatoren

- << Linksverschiebung
- >> Rechtsverschiebung mit Vorzeichen
- >>> Rechtsverschiebung ohne Vorzeichen

```
int i1 = -10;          //11111111111111111111111110110
                       //-10
int i2 = i1 << 4;      //11111111111111111111101100000
                       //-160
int i3 = i1 >> 4;      //11111111111111111111111111111
                       //-1
int i4 = i1 >>> 4;     //00001111111111111111111111111
                       //268435455
```

Zuweisungsoperatoren

- = Einfache Zuweisung

```
a = b;  //a wird der Wert von b zugewiesen
```

- += Additionszuweisung

```
a += b; //a wird der Wert von a + b zugewiesen
```

- -= Subtraktionszuweisung

```
a -= b; //a wird der Wert von a - b zugewiesen
```

- *= Multiplikationszuweisung

```
a *= b; //a wird der Wert von a * b zugewiesen
```

- /= Divisionzuweisung

```
a /= b; //a wird der Wert von a / b zugewiesen
```

- % = Modulozuweisung

```
a %= b; //a wird der Wert von a % b zugewiesen
```

Zuweisungsoperatoren

- &= UND-Zuweisung

```
a &= b; //a wird der Wert von a & b zugewiesen
```

- |= ODER-Zuweisung

```
a |= b; //a wird der Wert von a | b zugewiesen
```

- ^= EXKLUSIV-ODER-Zuweisung

```
a ^= b; //a wird der Wert von a ^ b zugewiesen
```

- <<= Linksschiebezuweisung

```
a <<= b; //a wird der Wert von a << b zugewiesen
```

- >>= Rechtsschiebezuweisung

```
a >>= b; //a wird der Wert von a >> b zugewiesen
```

- >>>= Rechtsschiebezuweisung mit Nullexpansion

```
a >>>= b; //a wird der Wert von a >>> b zugewiesen
```

Wichtige Operatoren

- Folgende Operatoren sollte man kennen:
  - Grundrechenarten: +, -, * und /
  - ++ und -- vor allem bei Schleifen nützlich
  - Modulo %
  - Vergleiche: ==, !=, <, <=, >, >=
  - && und ||
  - Normale Zuweisung =

Kontrollstrukturen

- if
- switch
- while
- for
- break
- continue

if

- Überprüfung einer Bedingung

```
if (i > 5) {
   i = i + 1;
}
```

```
if (i > 5) {
   i = i - 1;
}
else {
   i = i + 1;
}
```

if

```
if (i > 5) {
  i = i - 1;
}
else if (i < 3) {
  i = i + 1;
}
else {
  i = 10;
}
```

switch

- Vergleich mit primitiven Datentypen
- Keine größeren Datentypen (long, float, double) oder Objekte benutzbar

```
switch (i) {
  case 2:
    System.out.println("i == 2");
  case 3:
    System.out.println("i == 3");
  default:
    System.out.println("i != 2 und i != 3");
}
```

while

- Schleife um eine Abfolge von Anweisungen mehrfach auszuführen, bis die Bedingung erfüllt ist
- Verwendung bei dynamischer Abbruchbedingung
- do-while-Schleife um die Anweisung mindestens einmal auszuführen

```
int a = 0;
Random r = new Random();
while(a <100){
        a = a + r.nextInt();
}

do{
         a = a + r.nextInt();
} while(a < 100);
```

for

- for-Schleifen werden meist bei vorher definierter Anzahl an Durchläufen verwendet

```
int a = 0;
Random r = new Random();
for(int i = 0; i < 10; i++){
        a = a + r.nextInt();
}
```

break

- Ausbruch aus Schleifen und switch Anweisung

```
int i = 10;

while ( true ){
        i = i-1;
        if ( i == 0 ) {
                break;
        }
}

switch(i){
        case 0:
                System.out.println("i = 0");
                break;
        case 1:
                System.out.println("i = 1");
                break;
}
```

continue

- continue springt in Schleifen zum Schleifenkopf zurück

```
for ( int i = 0; i <= 10; i++ )
{
  if ( i % 2 == 1 ) {
    continue;
  }
  System.out.println( i + " ist eine gerade Zahl" );
}
```

Exceptions

- Wichtig um Fehler zu erkennen und zu behandeln
- Bei Exceptions ist man sich meistens bewusst, dass ein Fehler auftreten kann
- Werfen von exceptions ist mit throw möglich
- Behandlung von exceptions in einem „try“ Block durch „catch“ möglich
- „finally“ Block nach „try catch“ Block wird sowohl bei korrekter Ausführung als auch bei Fehler ausgeführt

Exceptions

```
public static void main(String[] args) throws Exception{
  int i = 2;
  if (i > 0) {
    throw(new Exception());
  }
}
```

```
String a = null;
try {
  a.length();
}
catch (NullPointerException e) {
  e.printStackTrace();
}
finally{
  System.out.println("finally Block wird immer ausgeführt,
falls nicht benötigt kann er weggelassen werden");
}
```

Fragen?

Vielen Dank für ihre Aufmerksamkeit!

Kursmaterialien sind zu finden unter:

http://web.informatik.uni-mannheim.de/java10000/