Klassen

Motivation

Wie würde man ein Datum speichern (z. B. **11. Oktober 2024**)?

3 Variablen

```
int day;
String month;
int year;
```

Unbequem, wenn man mehrere Exemplare davon braucht:

```
int day1;
String month1;
int year1;
int day2;
String month2;
int year2;
...
```

Idee: die 3 Variablen zu einem eigenen Datentyp zusammenfassen

Datentyp Klasse

Speicherung verschiedenartiger Werte unter einem gemeinsamen Namen

Deklaration

```
class Date {
    int day;
    String month;
    int year;
}
```

Felder der Klasse *Date*

Verwendung als Typ

```
Date x, y;
```

Zugriff

```
x.day = 13;
x.month = "November";
x.year = 2002;
```

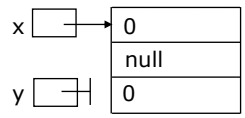
Date- Variablen sind Zeiger auf Objekte

Objekte

Objekte einer Klasse müssen vor ihrer ersten Benutzung erzeugt werden.

Date x, y; Reserviert nur Speicher für die Zeigervariablen.

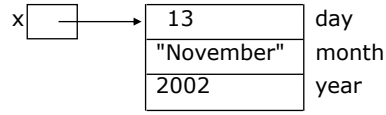
x = **new Date();** Erzeugt ein Date- Objekt und weist seine Adresse x zu.



neu erzeugte Objekte werden mit 0, null, false, '\ u0000' initialisiert

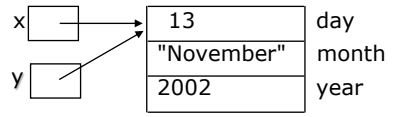
Eine Klasse ist wie eine Schablone, von der beliebig viele Objekte erzeugt werden können.

```
x. day = 13;
x. month = "November";
x. year = 2002;
```



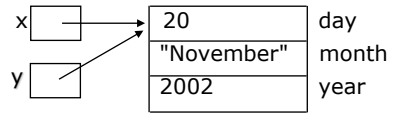
Zuweisungen

y=x;



Zeigerzuweisung!

y.day=20;



ändert auch x.day!

Zuweisungen sind erlaubt, wenn die Typen gleich sind.

```
class Date {
      int day;
      String month;
      int year;
}
```

```
class Address{
      int number;
      String steet;
      int zipCode;
}
```

```
Date d1, d2 = new Date();
Address a1, a2 = new Address();
```

```
d1 = d2;        // ok, gleiche Typen
a1 = a2;        // ok, gleiche Typen
d1 = a1;        // verboten: verschiedene Typen trotz gleicher Struktur!
```

Vergleiche

Zeigervergleich

```
x == y          vergleicht nur Zeiger
x != y

x < y           nicht erlaubt
x <= y
x > y
x >= y
```

Wertvergleich muß mittels Vergleichsmethode selbst implementiert werden

```
static boolean equalDate (Date x, y) {
    return x.day == y.day && x.month.equals(y. month) && x.year == y.year;
}
```

Wo werden Klassen deklariert

Auf äußerster Ebene eines Programms (einer Datei)

```
class C1 {
    ...
}
```

```
class C2 {
    ...
}
```

```
class MainProgram {
    ...
    public static void main (String[] arg) {
        ...
    }
}
```

Beispiel: Polygone

```
class Point { int x, y; }
class Polygon { Point[] pt; }
```

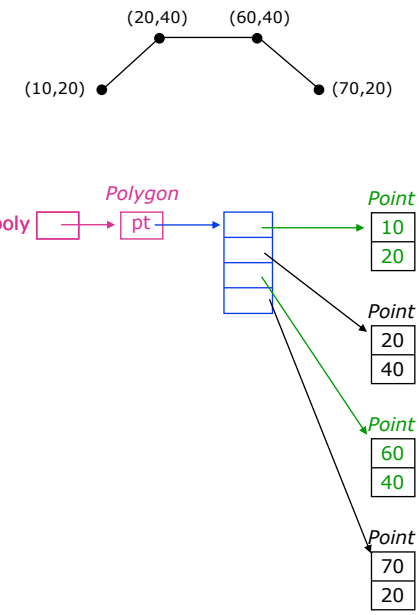
```
.......
public static void main (String[] arg) {
    Polygon poly ;
    poly = new Polygon();
    poly.pt = new Point[ 4];

    Point p = new Point();  p. x = 10; p. y = 20;
    poly.pt[ 0] = p;

    p = new Point();          p.x = 20; p.y = 40;
    poly.pt[ 1] = p;

    p = new Point();          p.x = 60; p.y = 40;
    poly.pt[ 2] = p;

    p = new Point();          p.x = 70; p.y = 20;
    poly.pt[ 3] = p;
  } ...
```



Methoden mit mehreren Rückgabewerten

Java- Funktionen haben nur 1 Rückgabewert.
Will man mehrere Rückgabewerte, muß man sie zu einer Klasse zusammenfassen.

Beispiel: Umrechnung von Sekunden auf Std, Min, Sek

```
class Time {
   int h, m, s;
}
```

```
class Program {

   static Time convert (int sec) {
      Time t = new Time();
      t. h = sec / 3600;     t. m = (sec % 3600) / 60;     t. s = sec % 60;
      return t;
   }

   public static void main (String[] arg) {
      Time t = convert( 10000);
      System.out. println( t. h + ":" + t. m + ":" + t. s);
   }
}
```

Kombination von Klassen mit Arrays

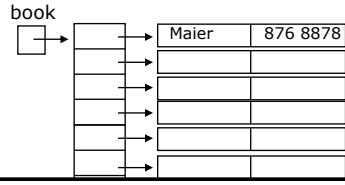
Beispiel: Telefonbuch

	name	phone
0	Maier	876 8878
	Mayr	543 2343
	Meier	656 2332
99		

zweidimensionales Array
kann hier nicht verwendet werden

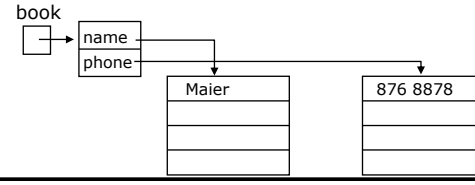
Array von Objekten

```
class Person {
    String name;
    int phone;
}
Person[] book = new Person[ 100];
```



Objekt bestehend aus 2 Arrays

```
class PhoneBook {
    String[] name;
    int[] phone;
}
PhoneBook book = new PhoneBook();
book. name = new String[ 100];
book. phone = new int[ 100];
```



Implementierung

```
class Person {
   String name;
   int phone;
}
```

```
class PhoneBookSample {
   static Person[] book;
   static int nEntries = 0; // current number of entries in book
```

```
   static void enter (String name, int phone) {
      if (nEntries >= book. length) System.out. println("--- phone book full");
      else {
         book[ nEntries] = new Person();
         book[ nEntries]. name = name;
         book[ nEntries]. phone = phone;
         nEntries++;
      }
   }
```

```
   static int lookup (String name) {
      int i = 0;
      while (i < nEntries  &&   !name.equals(  book[ i]. name )  ) i++;
      // i >= nEntries || name. equals( book[ i]. name)
      if (i == nEntries) return -1; else return book[ i]. phone;
   }
```

Implementierung (Fortsetzung)

```
...
   public static void main (String[] arg) {
      book = new Person[ 1000];
      //----- read the phone book from a file
      In. open(" phonebook. txt");
      String name = In. readWord();
      int phone;
      while (In. done()) {
         phone = In. readInt();
         enter( name, phone);
         name = In. readWord();
      }
      In. close();
```

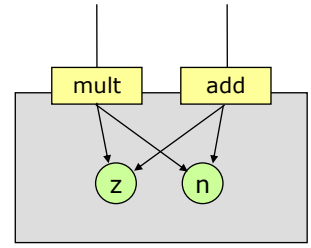
```
      //----- search in the phone book
      for (;;) {
         System.out. print(" Name: "); name = In. readWord();
         if (! In. done()) break;
         phone = lookup( name);
         if (phone > 0)   System.out. println(" Number = " + phone);
         else             System.out. println( name + " unknown");
      }
   }
} // end PhoneBookSample
```

Objektorientierung

Klasse = Daten + Methoden

Beispiel: Bruchzahlenklasse

```
class Fraction {
  int z;   // Zähler
  int n;   // Nenner
  void mult (Fraction f) {
      this. z = this. z * f. z;
      this. n = this. n * f. n;
   }
   void add (Fraction f) {
      this. z = this. z * f. n + f. z * this. n;
      this. n = this. n * f. n;
   }
}
```



abgeschlossener Baustein

- *mult* und *add* sind lokal zu *Fraction* (können auf *Fraction*- Objekte angewendet werden)
- *this* bezeichnet "dieses Objekt", auf das die Operation angewendet wird
- Methoden hier ohne *static* deklariert (siehe später)

Aufruf von Methoden

```
Fraction a = new Fraction();    a. z = 1;   a. n = 2;      // a == 1/ 2
Fraction b = new Fraction();    b. z = 3;   b. n = 5;      // b == 3/ 5

a. mult( b);
```

Auf das Objekt *a* wird die Operation *mult* angewendet (mit Parameter b)

Man sagt:

- *a* bekommt die Meldung (message) *mult*
- *a* ist der Empfänger der Meldung *mult*

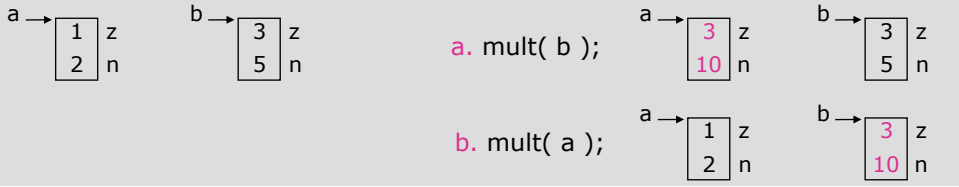
Was passiert dabei?

Aufruf von Methoden

```
                a. mult( b );
                ↓
void mult (/* Fraction this, */ Fraction f) {
   this. z = this. z * f. z;
   this. n = this. n * f. n;
}
```

Was passiert?

- *Parameterübergabe:*
  this = a; (this ist ein versteckter Parameter jeder Methode)
  f = b;
- *a* ist der Empfänger der Meldung *mult*



Weglassen von **this** (Normalfall)

```
class Fraction {
    int z;
    int n;

    void mult (Fraction f) {
        z = z * f. z;
        n = n * f. n;
    }

    void add (Fraction n) {
        z = z * n. n + n. z * this. n;
        this. n = this. n * n. n;
    }
}
```

z und *n* sind eindeutig.
Compiler fügt *this* automatisch ein

n wäre nicht eindeutig..
Qualifikation mit *this* nötig

this kann weggelassen werden, wenn der restliche Name eindeutig ist

Grafische Notation für Klassen

UML- Notation (Unified Modeling Language)

Fraction	*Klassenname*
int z int n	*Felder*
void mult (Fraction f) void add (Fraction f)	*Methoden*

Vereinfachte Form

Fraction
int z int n
mult (f) add (f)

falls weniger Details gewünscht oder nötig

Konstruktoren

Spezielle Methoden, die beim Erzeugen eines Objekts automatisch aufgerufen werden

```
class Fraction {
  int z , n ;
  Fraction ( int z , int n) {
     this.z = z;   this.n = n;
  }
  Fraction ( ) {
     z = 0;  n = 1;
  }
  void mult (Fraction f) {...}
  void add (Fraction f) {...}
}
```

- *dienen zur Initialisierung eines Objekts.*
- *heißen wie die Klasse*
- *ohne Funktionstyp oder void*
- *können Parameter haben*
- *können überladen werden*

Aufruf eines Konstruktors mit new

```
Fraction f  = new Fraction( 3, 5);
Fraction g = new Fraction();
```

- *legt neues Fraction- Objekt an.*
- *ruft den Konstruktor auf*

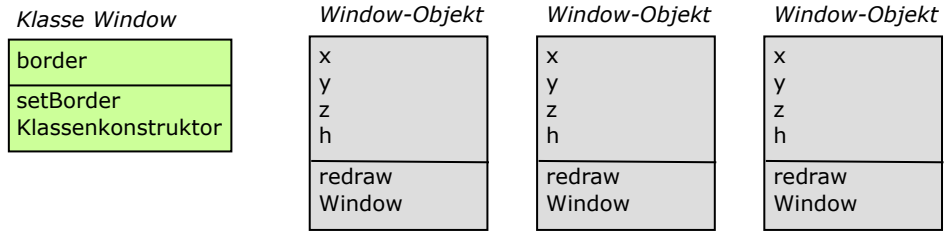
static

```
class Window {

   int x, y, w, h;                          // Objektfelder (in jedem Window- Objekt vorhanden)
   static int border;                       // Klassenfeld (nur einmal pro Klasse vorhanden)

   void redraw () {...}                     // Objektmethode (auf Objekte anwendbar)
   static void setBorder (int n) {border = n;}  // Klassenmethode (auf Klasse Window anwendbar)

   Window( int w, int h) {...}              // Objektkonstruktor (zur Initialisierung von Objekten)
   static { /* read border from config file */}  // Klassenkonstruktor (zur Initialisierung der Klasse)
}
```



- Objektmethoden haben Zugriff auf Klassenfelder (*redraw* kann auf *border* zugreifen)
- Klassenmethoden haben keinen Zugriff auf Objektfelder (*setBorder* kann nicht auf *x* zugreifen)

static (Forts.)

Was geschieht wann?

Beim Laden der Klasse Window

- Klassenfelder werden angelegt (border)
- Klassenkonstruktor wird aufgerufen

Beim Erzeugen eines Window- Objekts

- Objektfelder werden angelegt (x, y, w, h)
- Objektkonstruktor wird aufgerufen

Zugriffe

Zugriff auf static- Elemente über den Klassennamen

- Window. border = ...; Window. setBorder(3);
- Methoden der Klasse Window können Klassennamen weglassen (border = ...; setBorder(3);)
- Klassenkonstruktor wird nie explizit aufgerufen

Zugriff auf nonstatic- Elemente über einen Objektnamen

- Window win = new Window(100, 50);
  win. x = ...; win. redraw();
- Methoden der Klasse Window können auf eigene Elemente direkt zugreifen (x = ...; redraw();)

Primitive Datentypen & Objekte als Datentyp

primitive Typen

byte	8- Bit- Zahl	$-2^7 .. 2^7-1$	(- 128 .. 127)
short	16- Bit- Zahl	$-2^{15} .. 2^{15}-1$	(- 32768 .. 32767)
int	32- Bit- Zahl	$-2^{31} .. 2^{31}-1$	(- 2 147 483 648 .. 2 147 483 647)
long	64- Bit- Zahl	$-2^{63} .. 2^{63}-1$	
float	32 Bit		
double	64 Bit		
char	16 Bit		
boolean	true oder false		

Objekttypen

Alle anderen Typen sind Objekttypen, dies sind Klassen, Interface-Klassen, Strings und alle vom Nutzer definierten Typen.
Klassen sind die "Templates" für ein Objekt.
Variablen eines Objekttyps enthalten eine Referenz ("pointer") auf eine Instanz des Objekts.

Wrapper Klassen

Zu jedem primitiven Typ existiert eine Klasse vom Objekttyp.

byte	Byte
short	Short
int	Integer
long	Long
float	Float
double	Double
char	Character
boolean	Boolean