Grundlagen der Informatik für Ingenieure I

5. Klassen, Objekte und Methoden

Klassen, Objekte und Methoden

5.1 Klassen (classes)

- Klassen sind die Baupläne (Muster, Templates) der zur Laufzeit zu erzeugenden Objekte (Instanzen).
- Das Schreiben eines Programms in Java ist gleichbedeutend mit der Definition einer Menge von Klassen.
- Beim Entwurf eines Programms werden - soweit möglich - bereits vorhandene Teilproblemlösungen als Klassen aus den verschiedenen Klassenbibliotheken (Siehe Kapitel *"packages"*) benutzt (Wiederverwendbarkeit).
- Bisher haben wir nur eigene Klassen definiert und darüberhinaus andere Klassen (z.B. String, Array, System) intuitiv benutzt.

1 Definition einer Klasse

- Eine Klassendefinition hat folgende Form

```
[modifier] class classname {
        ....
    }
```

  - Ist die neue Klasse außerdem Unterklasse einer anderen Klasse, muss dies mit dem Schlüsselwort ***extends*** deklariert werden:

```
[modifier] class classname extends superclassname{
        ....
    }
```

  - Ein Beispiel war bereits unser "HelloWorld - Applet":

```
public class HelloWorldApplet extends Applet {
....
}
```

2 Definition von Objekt-Variablen

- Objekt-Variable (Instanz-Variable) sind Variable die objektglobal gültig sind.
- Die Werte aller Objektvariablen bestimmen den Zustand des Objekts.
- Aufgrund des Gültigkeitsbereichs werden diese Variablen unmittelbar nach der Klassendeklaration definiert:

Beispiel:

```
class DampfLokomotive {
    private String baureihe;
    private int leistung;
    private int anzahlAntriebsRaeder;
    private boolean heusingerSteuerung;
    ...
}
```


3 Definition von Klassenvariablen

- Klassenvariable sind **gemeinsame (*shareable*)** Variable aller Objekte einer Klasse.

  - **Objektvariable:**
    - Bei der Erzeugung (Instantiierung) eines neuen Objekts erhält das Objekt seine “eigene” Kopie der Objektvariablen.
    - Veränderung dieser Kopie haben keinen Effekt auf Objektvariable anderer Objekte.

  - **Klassenvariable:**
    - **Im Gegensatz dazu** existieren **Klassenvariable** in der Klasse nur einmal.
    - Alle Objekte dieser Klasse referenzieren die **gleiche Variable**.

3 Definition von Klassenvariablen

◆ Eine Klassenvariable wird mit dem Schlüsselwort ***static*** vereinbart:

Beispiel:

```
class FamilienMitglied {
static String familienName = "Wurm"
// Klassenvariable
String vorName;
// Objektvariable
int alter;
// Objektvariable
...
}
/*  Der Familienname ist allen Objekten der Klasse
    FamilienMitglied gemeinsam */
```


4 Definition von Methoden

◆ Eine **Methodendefinition** setzt sich aus folgenden Teilen zusammen:

- **Methodenname**
- Typ des **Returnparameters** (Objekt oder prim. Typ)
- Eine Liste von **Parametern**
- **Rumpf** der Methode
- ***Modifier*** *(public, private, ...*- siehe Kapitel 5.3)
- ***Exceptions*** *(throw* - siehe Kapitel 10)

Methodenname und Parameterliste werden auch als **Signatur** der Methode bezeichnet

◆ Eine Methodendefinition hat folgende Form:

```
[modifier] returntype name( type1 arg1, type2 arg2, ... ) {
...
}
```

4 Definition von Methoden

- **Der Typ einer Methode** wird durch den Typ des Rückgabewertes (Returnparameters) bestimmt.
- Das Schlüsselwort ***void*** drückt aus, dass kein Wert zurückgegeben wird.
- Die Rückgabe eines Wertes erfolgt mit Hilfe des ***return-Statements:***

```
return [varname];
```


4 Definition von Methoden

- Beispiel:

```
double kontoStand(){
        ...
    // Ermittle saldo
    ...
    return saldo;
}
```

- Diese Methode könnte wie folgt benutzt werden:

```
 ....
meinKontoStand = meinKonto.kontoStand();
....
```

- Am "Ende" einer Methode kann das *return-Statement* fehlen, falls kein Parameter zurückzugeben ist.

4 Definition von Methoden

◆ Mit dem ***Return-Statement*** ohne Wert kann man den Rücksprung von einer beliebigen Stelle einer Methode realisieren, wenn dies aufgrund interner Zustandsbedingungen eines Objekts notwendig ist.

Beispiel:

```
....
if ( bedingung == true)
    return;
....
```

oder:

```
....
if ( bedingung )
    return;
....
```


4 Definition von Methoden

◆ Warum benötigt man - in Java - den Begriff der Signatur?

- Andere Programmiersprachen unterscheiden Funktionen (Subroutinen, Unterprogramme oder Proceduren) nur nach ihrem Namen.
- In Java können verschiedene Methoden den gleichen Namen haben, aber verschiedene Parameter (Anzahl, Typ) besitzen.
- Zur Identifikation dient die Signatur (Name, Paramerliste) der Methode.
- Näheres hierzu siehe: 5.2.8 Überladen von Methoden

5 Aufruf von Methoden

◆ Methoden eines Objektes ruft man üblicherweise mit der sog. *Dot-Notation* auf:

```
objectName.methodName( arg1, arg2,... )
```

Beispiel:

```
meinKonto.getSaldo()
eKlasse.getVorderAchse( produktionsJahr )
```

- Parameter werden in runden Klammern übergeben; ist die Parameterliste leer, erscheinen nur die Klammern.

◆ Bei lokalem Aufruf (innerhalb der Klasse) ist die Dot-Notation nicht notwendig:

```
methodName( arg1, arg2,... )
```

Beispiel:

```
getVorderAchse( 1996 )
getVorderAchse()
```


5 Aufruf von Methoden

- Ist der Returnparameter einer Methode selbst wieder ein Objekt, so kann man Methodenaufrufe aneinanderhängen.

Beispiel:

```
eKlasse.getVorderAchse( 1996 ).getArtikelNr()
```

– Es wird die Methode ***getArtikelNr()*** aufgerufen.

– ***getArtikelNr()*** ist eine Methode des Objektes, das von der Methode ***getVorderachse()*** als Returnparameter geliefert wird.

– ***getVorderachse()*** ist eine Methode des Objektes ***eKlasse***.

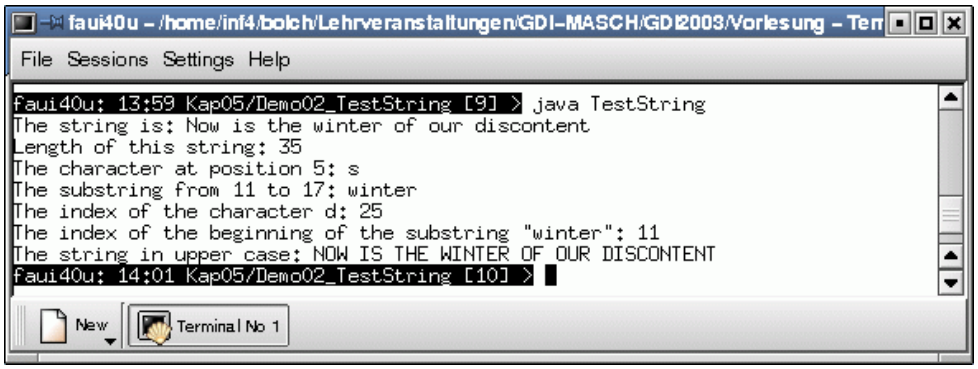
5 Aufruf von Methoden

◆ Beispiel (Demo1_TestString): "Verschiedene Methoden der Klasse ***String***"

```
class TestString {
    public static void main( String args[] ) {
        String str = "Now is the winter of our discontent";
        System.out.println( "The string is: " + str );
        System.out.println( "Length of this string: "
                            + str.length() );
        System.out.println( "The character at position 5: "
                            + str.charAt(5) )
        System.out.println( "The substring from 11 to 17: "
                            + str.substring( 11, 17 ) );
        System.out.println( "The index of the character d: "
                            + str.indexOf( 'd' ) );
        System.out.print( "The index of the beginning of the " );
        System.out.println( "substring \"winter\": "
                            + str.indexOf( "winter" ) );
        System.out.println( "The string in upper case: "
                            + str.toUpperCase() );
    }
}
```


5 Aufruf von Methoden

- Ergebnis (Demo1_TestString): "Aufruf verschiedener Methoden der Klasse String"



6 Das *this* - Schlüsselwort

- Das ***this*** - Schlüsselwort:
  - Referenzen auf Methoden oder Objektvariablen des *"current objects"* kann mit dem ***this*** - Schlüsselwort in *Dot-Notation* erfolgen, was u. a. die Lesbarkeit des Programmcodes steigern kann.

    Beispiele:

    – Der Variablen ***t*** wird die Instanzvariable ***x*** **dieses** Objekts zugewiesen:

    ```
    t = this.x; // entspricht: t = x;
    ```

    – **Dieses** Objekt (genauer: eine Referenz für dieses Objekt) kann auch als Returnparameter zurückgegeben:

    ```
    return this;
    ```

    – Aufruf der Methode *firstmethod*, definiert in **dieser** Klasse; als Parameter wird das *"current object"* übergeben:

    ```
    this.firstmethod(this);// entspricht: firstmethod(this);
    ```

  - *this* kann nur in Objektmethoden nicht in Klassenmethoden verwendet werden.

7 Gültigkeitsbereiche von Variablen

- Der Gültigkeitsbereich einer Variablen ist vom Ort der Definition *(default)* und zusätzlich von verwendeten *Modifiern(siehe 5.3)* abhängig.
  Wir unterscheiden:
  - Klassenvariable (für alle Instanzen dieser Klasse global)
  - Objektvariable (für das Objekt global)
  - lokale Variable (methodenlokal, blocklokal)

- In Java (nicht nur) wird nach der Variablendefinition vom Ort der Vewendung her gesehen gesucht:
  - zunächst blocklokal,
  - dann methodenlokal,
  - dann auf Instanzebene
  - und schließlich auf Klassenebene.

7 Gültigkeitsbereiche von Variablen

- Wird ein Variablenname mehrfach verwendet (was mit Einschränkungen zulässig ist!), verdeckt (*overrides*) die innere Definition die äußere.

- Will man sowohl auf eine lokale Variable ***x*** als auch auf die Objektvariable ***x*** zugreifen können, so kann man auf die “verdeckte” Objektvariable mit Hilfe des ***this***-Schlüsselworts zugreifen:

  this.x

7 Gültigkeitsbereiche von Variablen

Beispiel (Demo2_ScopeTest): "Methodenlokale Variable verdeckt Objektvariable... "

```
class Scope {
    private int test = 10;        // Objektvariable
    static int kVarTest = 30;     // Klassenvariable

    void printTest() {
        int test = 20;     // methodenlokale Variable
        System.out.println( "test = " + test );
        System.out.println( "this.test = " + this.test );
        System.out.println( "kVarTest = " + kVarTest );
    }
}
```

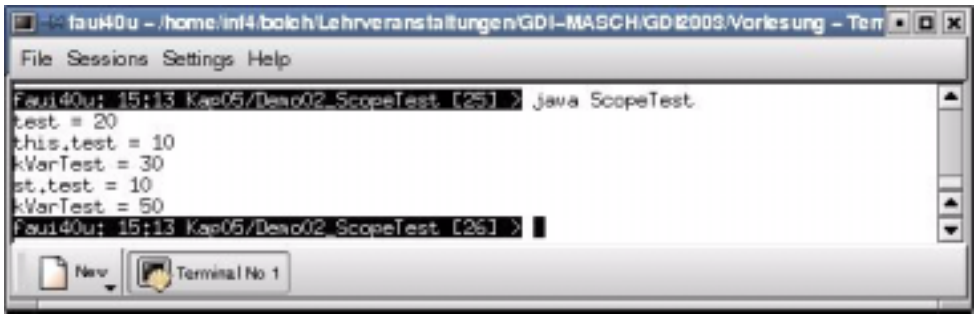
7 Gültigkeitsbereiche von Variablen

Aus Klassenmethoden heraus wird auf Klassenvariablen zugegriffen; auf Objektvariablen wird dieses versucht.

```
class ScopeTest {
    int test = 40;                  // Objektvariable
    static int kVarTest = 50;       // Klassenvariable

    public static void main( String args[] ) {
        Scope st = new Scope();
        st.printTest();
        System.out.println( "st.test = " + st.test );
//      System.out.println( "test = " + test );
        System.out.println( "kVarTest = " + kVarTest );
    }
}
/* Can't make a static reference to nonstatic variable
    test in class ScopeTest.
    System.out.println( "test = " + test );
*/
```


7 Gültigkeitsbereiche von Variablen

- Ergebnis (Demo2_ScopeTest):



8 Parameterübergabe

◆ Pass by Value

- Primitive Typen werden als Wert übergeben. Änderungen dieser Größen haben also nur lokalen Effekt.

◆ Pass by Reference

- Objekte werden als Referenz übergeben; d. h. sämtliche Änderungen erfolgen im Original.

- Beispiel (Demo3_PassByReference): "Objekt als Parameter"

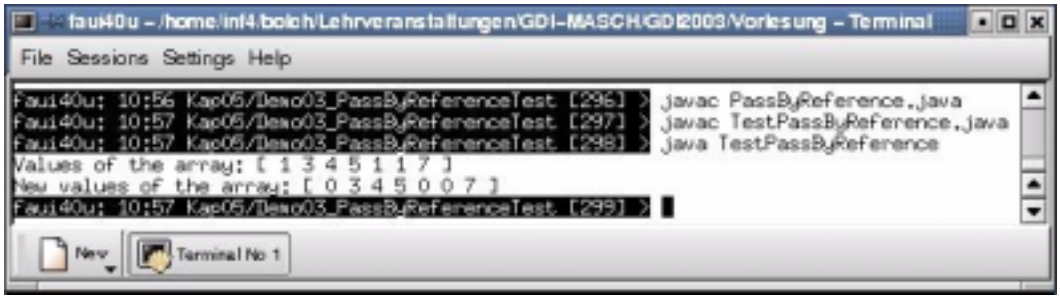
```
class PassByReference {
// ersetzt in einem, als Parameter übergebenem, Feld
// alle 1 durch 0

    void oneToZero( int vector[] ) {
           for ( int i = 0; i < vector.length; i++ ) {
               if ( vector[i] == 1 )
                   vector[i] = 0;
           }
    }
}
```


8 Parameterübergabe

```
class TestPassByReference {

    public static void main( String args[] ) {

        int arr[] = { 1, 3, 4, 5, 1, 1, 7 };

        PassByReference pbr = new PassByReference();

        System.out.print( "Values of the array: [ " );
        for ( int i = 0; i < arr.length; i++ ) {
               System.out.print( arr[i] + " " ); //ohne Vorschub!
        }
        System.out.println( "]" );

        pbr.oneToZero( arr );

        System.out.print( "New values of the array: [ " );
        for ( int i = 0; i < arr.length; i++ ) {
               System.out.print( arr[i] + " " );
        }
        System.out.println( "]" );
    }
}
```

8 Parameterübergabe

- Ergebnis (Demo3_PassByReference):



9 Definition von Klassenmethoden

◆ Klassenmethoden:

- In Analogie zu den Klassenvariablen gibt es auch Klassenmethoden
- Wie bei den Klassenvariablen wird auch hier das Schlüsselwort ***static*** verwendet:

```
static type methodname (typ1 arg1, typ2 ...)
```

- Eine Klassenmethode ist allen Objekten dieser Klasse zugänglich.
- Andere Objekte können eine Klassenmethode über den Klassennamen in der Dot-Notation aufrufen:

```
classname.methodname
```

- Klassenmethoden können nur auf Klassenvariablen und -parameter zugreifen nicht auf Objektvariablen!

9 Definition von Klassenmethoden

- Beispiel (Demo4_Classmethod):

```
class PassByReference {

        static void onetoZero( int vector[] ) {       // Klassenmethode

            for ( int i = 0; i < vector.length; i++ ) {
                if ( vector[i] == 1 )
                    vector[i] = 0;
            }
        }
        public static void main ( String args[] ) {
             int arr[] = { 1, 3, 4, 5, 1, 1, 7 };
            System.out.print( "Values of the array: [ " );
            for ( int i = 0; i < arr.length; i++ ) {
                System.out.print( arr[i] + "" );
            }
            System.out.println( "]" );

            onetoZero( arr );       // Aufruf der Klassenmethode

            System.out.print( "New values of the array: [ " );
            for  ( int i = 0; i < arr.length; i++ ) {
                System.out.print( arr[i] + "" );
            }
            System.out.println( "]" );
        }
}
```


9 Definition von Klassenmethoden

- Ein Beispiel für die Verwendung von Klassenmethoden ist die Klasse ***Math***:
  - Die Klasse *Math* stellt eine große Anzahl mathematischer Operationen zur Verfügung .
  - Diese Methoden kann man aufrufen, ohne dass man ein Objekt erzeugen muss.
  - Beispiele:

```
double root = Math.sqrt( 512. );

System.out.println( "Maximum von x,y ist: "
                                    + Math.max( x,y ) );
```

9 Definition von Klassenmethoden

Die Klassenmethode *main()* :

◆ Die Signatur der Methode *main()* sieht immer wie folgt aus:

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

- ***public***: Die Methode *main()* ist überall (d.h. allen Klassen bzw. Objekten anderer *packages*) zugänglich.
- ***static***: *main()* ist eine Klassenmethode
- ***void***: *main()* gibt keine Parameter zurück

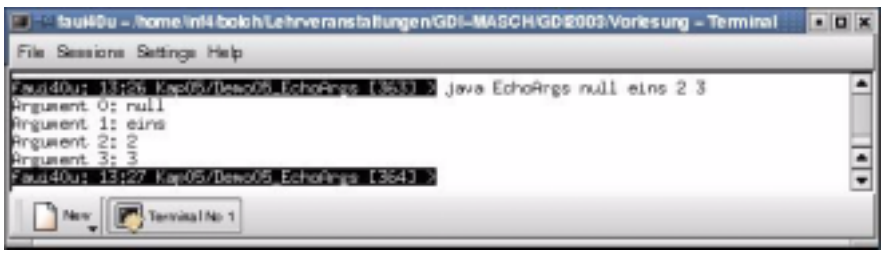
9 Definition von Klassenmethoden

◆ *main()* übernimmt aus der Kommandozeile Parameter (Zeichenketten getrennt durch <b>) und stellt sie im array *args[]* zur Verfügung.

- Beispiel (Demo5_EchoArgs):

```
class EchoArgs {
  public static void main( String args[] ) {
    for ( int i = 0; i < args.length; i++ ) {
        System.out.println( "Argument " + i + ":" + args[i] );
    }
  }
}
```

- Ergebnis (Demo5_EchoArgs):



9 Definition von Klassenmethoden

- Parameterübernahme aus der Kommandozeile:
  - Mit dem Paramater args[] in der main-Methode können Parameter aus der Kommandozeile übernommen werden (siehe Kapitel 4.8)
    z.B. **`java DreiecksTest 3 5 7`**
  - Nur *String*-Variable können aus der Kommandozeile übernommen werden.
  - Bei anderen Datentypen muss eine Konvertierung vorgenommen werden.
  - Dazu bieten die **Klassen!!** der primitiven Typen geeignete Methoden an (siehe auch Kapitel 4.8).

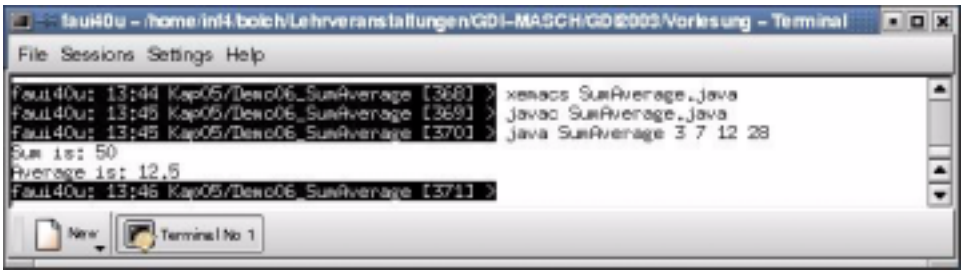
9 Definition von Klassenmethoden

- Eine weiteres Beispiel (Demo6_SumAverage):

```
class SumAverage {

    public static void main( String args[] ) {
           int sum = 0;

           for ( int i = 0; i < args.length; i++ ) {
               sum += Integer.parseInt( args[i] );
           }

           System.out.println( "Sum is: " + sum );

           System.out.println( "Average is: "
                                    +(float)sum/args.length );
    }
}
```

9 Definition von Klassenmethoden

- Ergebnis (Demo6_SumAverage):



5.2 Objekte (Instanzen)

1 Erzeugung von Objekten, new-Operator

◆ **new**-Operator:

```
[Classname] varname = new Classname( par1, par2, ...);
```

- Nach der Erzeugung (Instantiierung) enthält die Variable `varname` die Referenz auf ein Objekt der Klasse `Classname`.

- Beispiel1:

```
String str = new String();
```

oder die Definition der Variablen getrennt von der Erzeugung:

```
String str;
....
str = new String();
```

1 Erzeugung von Objekten, new-Operator

- Beispiel 2:

```
Car vw = new Car();

bzw.
Car vw;
...
vw = new Car();
```

◆ Vorbesetzung (Initialisierung) von Objektvariablen:

- Die Argumente in den runden Klammern dienen dazu, Objektvariable auf einen Anfangswert zu setzen.
- Die Anzahl, der Typ und die Zuordnung der Argumente werden in der Klasse definiert.

Beispiele:

```
Date datum = new Date(90,4,1,4,30);
Point punkt = new Point(1,1);
```

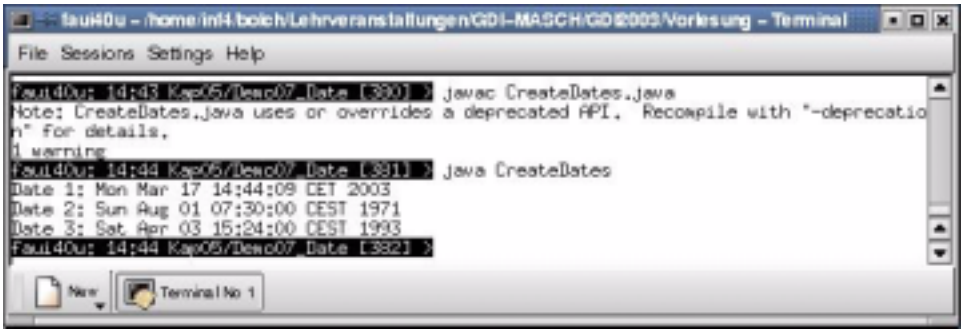

1 Erzeugung von Objekten, new-Operator

- Ein vollständiges Beispiel (Demo7_CreateDates):

```
import java.util.Date;

class CreateDates {

    public static void main( String args[] ) {
     Date d1, d2, d3;

     d1 = new Date();
     System.out.println( "Date 1: " + d1 );

     d2 = new Date( 71, 7, 1, 7, 30 );
     System.out.println( "Date 2: " + d2 );

     d3 = new Date( "April 3 1993 3:24 PM" );
     System.out.println( "Date 3: " + d3 );
    }
}// mostly deprecated!!!
```

1 Erzeugung von Objekten, new-Operator

- Ergebnis (Demo7_CreateDates):



1 Erzeugung von Objekten, new-Operator

◆ Was löst der ***new***-*Operator* tatsächlich aus?

- ***new*** erzeugt aus dem *Template* der Klasse ein Objekt (eine Instanz).
- Üblicherweise wird ein Objekt durch eine Variable repräsentiert.
- Dieser Variablen wird durch die Anweisung:
  `var = new classname()`
  eine Referenz auf das neu erzeugte Objekt zugewiesen.
- ***new*** initialisiert das Objekt mit Hilfe eines speziellen in der Klasse definierten Methodentyps. Dieser Methodentyp wird als ***Konstruktor*** *(siehe Kapitel 5.2.9)* bezeichnet.
- Eine Klasse kann **keinen**, **einen** oder **mehrere *Konstruktoren*** besitzen, mit verschiedener Anzahl und verschiedenen Typen von Argumenten (siehe Beispiel: *Date*).
- Jedes erzeugte Objekt hat seinen eigenen Speicherbereich für die Daten, der Code selbst ist nur einmal vorhanden, da er nur ausgeführt wird.

2 Zugriff und Zuweisungen auf Klassen- und Objektvariablen

- Der lokale Zugriff auf Objekt- bzw. Klassenvariablen unterscheidet sich nicht von dem auf lokale Variablen.

- Der nichtlokale Zugriff auf Klassen- und auf Objektvariablen erfolgt in der sogenannten *Dot-Notation*:

```
objectname.variablenname
```

Beispiele:

```
vw.vorderAchse
myArray.length
```

- **Der nichtlokale Zugriff auf Objektvariable ist eher die Ausnahme!**

2 Zugriff und Zuweisungen auf Klassen- und Objektvariablen

- Wichtig:
  - Durch den Zugriff wird ein Wert “zurückgegeben” , es handelt sich also um einen Ausdruck.
  - Dadurch ist es möglich auch an Objektvariable "geschachtelter" Objekte, in einfacher Weise heranzukommen.
  - Beispiel:

```
vw.vorderAchse.achsSchenkel
```

- Zuweisen eines Wertes an eine Objekt - bzw. Klassenvariable:
  - Beispiel:

```
vw.vorderAchse = vorderAchseGolf;
```

3 Referenzen auf Objekte

◆ Durch die Anweisung

```
var = new Konstruktor
```

wird einer Variablen eine Referenz (ein Verweis) auf das erzeugte Objekt zugewiesen.

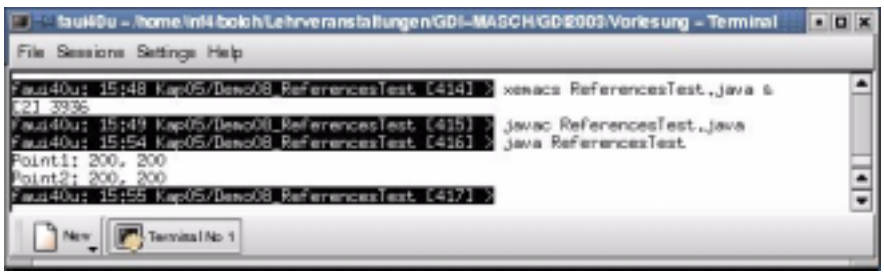
◆ Die Variable selbst enthält nicht das Objekt, sondern es handelt sich um eine Referenz (einen Verweis) auf das Objekt.

◆ Weisen wir diese Objekte auch anderen Variablen zu oder übergeben wir Objekte als Argumente, so übergeben wir jeweils nur die Referenz auf das Objekt; es finden keine Kopiervorgänge statt!

3 Referenzen auf Objekte

• Beispiel (Demo8_ReferencesTest): "Wann referieren Variable identische Objekte?"

```
import java.awt.Point;
class ReferencesTest {
   public static void main( String args[] ) {
     Point pt1, pt2;
     pt1 = new Point( 100, 100 );
     pt2 = pt1; //pt1 und pt2 referenzieren dasselbe Objekt
     pt1.x = 200;
     pt1.y = 200;
     System.out.println( "Point1:" + pt1.x + "," + pt1.y );
     System.out.println( "Point2:" + pt2.x + "," + pt2.y );
  }
}
```



4 Casting

◆ ***Casting*** (Typwandlung) heißt, einen Wert von einem Typ in einen Wert eines anderen Typs umzuwandeln, wobei das Original unverändert bleibt.
Casting wird angewendet bei primitiven Typen oder bei Objekten.

- Beispiel:

```
i = (int) 99.999     // double-Wert 99.999 wird nach
                     // int gewandelt (i = 99)
                     // der Nachkommateil wird abgeschnitten
```

◆ *Casting* primitiver Typen:

- Solange man in Richtung eines “größeren” Typs wandelt in dem Sinne, dass keine Information verloren geht, erfolgt das *Casting* automatisch.
- Dies ist der Fall bei: *byte --> int; int --> long; float -->double;*

4 Casting

◆ *Casting* primitiver Typen (cont.):

- In umgekehrten Fällen muß man explizit *casten*.
- Gleiches wird bei ganzzahlig nach *float* oder *double* empfohlen.

```
(typname) expression
```

Beispiel:

```
float x, y;
int i;
...
i = (int) (x / y)
```

Da die *“Precedence”* des *Castings* höher liegt, als die der arithmetischen Operationen, ist die Klammerung des Ausdrucks wichtig!

4 Casting

◆ *Casting Objects:*

- *Casting* von Objekten ist möglich unter der Voraussetzung, dass ihre Klassen in einer Vererbungsbeziehung zueinander stehen.

- Beispiel:

```
class GreenApple extends Apple {
...
}

...

Apple a = new Apple();
GreenApple agreen = new GreenApple();

a = agreen;           // kein casting, da “upward use”
agreen = (GreenApple) a; // casting da “downward use”

...
```


5 Relationale Operation auf Objekte

◆ Für die primitiven Typen habe wir eine Anzahl relationaler Operatoren kennengelernt (siehe Kapitel 4.6).

◆ Für die Anwendung auf Objekte sind nur zwei davon sinnvoll und erlaubt:

- “ == “
- “ != “

Tatsächlich wird die **Identität** getestet:
Referenzieren verschiedene Variable dasselbe Objekt?

– Zwei *String* - Objekte mit dem gleichen String als Inhalt ergeben beim “==” - Test den Wert *“false”*!

– Für das Testen auf Gleichheit des Inhalts verschiedener String-Variabler stellt die Klasse *String* geeignete Methoden bereit, wie das nächste Beispiel demonstriert.

◆ **Hinweis:** Werden zwei String - Variable mit dem gleichen literalen String instantiiert, dann referenzieren beide Variable das identische Objekt!

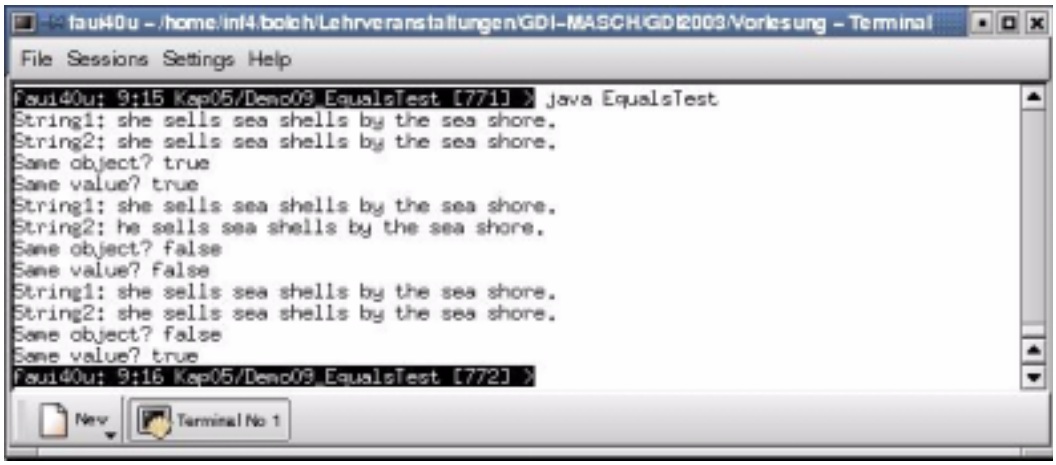
5 Relationale Operation auf Objekte

◆ Test auf Gleichheit von Strings am Beispiel (Demo9_EqualsTest):

```
class EqualsTest {
public static void main(String args[]) {
     String str1, str2;
     str1 = "she sells sea shells by the sea shore.";
     str2 = "she sells sea shells by the sea shore.";

     System.out.println( "String1: " + str1 );
     System.out.println( "String2: " + str2 );
     System.out.println( "Same object? " + ( str1 == str2 ) );
     System.out.println( "Same value? " + str1.equals( str2 ) );

     str2 = "he sells sea shells by the sea shore.";

     System.out.println( "String1: " + str1 );
     System.out.println( "String2: " + str2 );
     System.out.println( "Same object? " + ( str1 == str2 ) );
     System.out.println( "Same value? " + str1.equals( str2 ) );

     str2 = new String(str1);

     System.out.println( "String1: " + str1 );
     System.out.println( "String2: " + str2 );
     System.out.println( "Same object? " + ( str1 == str2 ) );
     System.out.println( "Same value? " + str1.equals( str2 ) );
     }
}
```


5 Relationale Operation auf Objekte

• Ergebnis (Demo9_EqualsTest):



6 Zugehörigkeit eines Objekts zu einer Klasse

◆ Die **Zugehörigkeit eines Objekts zu einer Klasse** kann mit dem *instanceof* - Operator wie folgt ermittelt werden:

- Beispiel1:

```
if (“Viele Worte” instanceof String)    // true
...
```

- Beispiel2:

```
Point pt = new Point(10,10);
if (pt instanceof String)              // false
...
```


7 Java - Klassen - Bibliotheken

◆ Die Java-Klassen-Bibliotheken sind unterteilt in sogenannte *class packages*. Die wichtigsten davon sind:

- java.lang
  Die Sprache selbst, *Object class, String class, System class*
- java.util
  Nützliche Hilfsmittel; z. B. *Date class; Calendar class*
- java.io
  Das Java-I/O-System
- java.net
  Networking classes, z. B. sockets, url, ...
- java.awt
  Abstract Windowing Toolkit (Klassen für grafische Oberflächen (Kapitel 8)
- java.applet
  (siehe Kapitel 6)

7 Java - Klassen - Bibliotheken

◆ Services zur Info-Beschaffung:

- Wir sind nicht in der Lage jede Methode jeder verwendeten Klasse auch nur annähernd zu besprechen, noch können wir Ihnen die Informationen gedruckt zur Verfügung stellen.

- Sie können sich die erforderlichen Informationen wie folgt beschaffen:

  `http://www4.informatik.uni-erlangen.de/Services/Doc/Java`

  `und dort JDK-1.4`

  (Nur von Rechnern an der Uni, nicht von extern)

8 Überladen von Methoden *(Overloading Methods)*

◆ Von *Overloading* spricht man, wenn man mehrere Methoden **gleichen Namens,** jedoch **unterschiedlicher Signatur** definiert.

◆ Unterschiedliche Signatur heißt:

- verschiedene Anzahl von Parametern
- und/oder Parameter verschiedenen Typs.

◆ Der Typ des Returnparameters wird bei der Unterscheidung von Signaturen **nicht** ausgewertet.

8 Überladen von Methoden *(Overloading Methods)*

◆ Beispiel (Demo10_MyRect):

```
import java.awt.Point;

class MyRect {

    private int x1 = 0;
    private int y1 = 0;
    private int x2 = 0;
    private int y2 = 0;

    void buildRect( int x1, int y1, int x2, int y2 ) {
       this.x1 = x1;
       this.y1 = y1;
       this.x2 = x2;
       this.y2 = y2;
    }
```


8 Überladen von Methoden *(Overloading Methods)*

◆ Beispiel (Demo10_MyRect), cont.:

```
    void buildRect( Point topLeft, Point bottomRight ) {
       x1 = topLeft.x;
       y1 = topLeft.y;
       x2 = bottomRight.x;
       y2 = bottomRight.y;
    }
    void buildRect( Point topLeft, int w, int h ) {
       x1 = topLeft.x;
       y1 = topLeft.y;
       x2 = ( x1 + w );
       y2 = ( y1 + h );
    }
    void printRect(){
       System.out.print( "MyRect: <" + x1 + "," + y1 );
       System.out.println( "," + x2 + "," + y2 + ">" );
    }
}
```

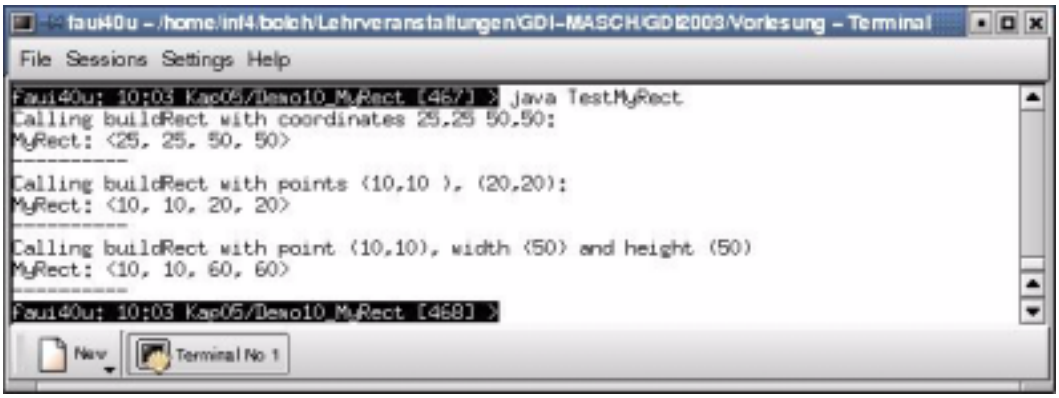
8 Überladen von Methoden *(Overloading Methods)*

```
class TestMyRect {

    public static void main(String args[]) {

      MyRect rect = new MyRect();

      System.out.println( "Calling buildRect with coordinates 25,25,50,50:" );
      rect.buildRect( 25, 25, 50, 50 );
      rect.printRect();
      System.out.println( "----------" );

      System.out.println( "Calling buildRect with points (10,10), (20,20):" );
      rect.buildRect( new Point( 10,10 ), new Point( 20,20 ) );
      rect.printRect();
      System.out.println( "----------" );

      System.out.print( "Calling buildRect with point( 10,10 )," );
      System.out.println( " width (50) and height (50)" );

      rect.buildRect( new Point( 10,10 ), 50, 50 );
      rect.printRect();
      System.out.println( "----------" );
    }
}
```


5.2 Objekte (Instanzen)

8 Überladen von Methoden *(Overloading Methods)*

◆ Ergebnis (Demo10_MyRect):



9 Konstruktoren *(Constructor Methods)*

◆ *Constructor Methods* sind die Methoden, die ein Objekt (Instanz) bei der Instantiierung in einen initialen Zustand setzen.

- Ist kein Konstruktor explizit definiert, kann die Instantiierung (*new*) ohne Parameter erfolgen. Die Objektvariablen werden default wie folgt vorbesetzt:
  - primitive numerische Typen: 0 bzw. 0.0
  - boolean: false
  - char: '\0'
  - Objektreferenzen: null
- *Constructor-Methods* haben den gleichen Namen wie die Klasse
- *Constructors* geben keinen Parameter zurück - trotzdem darf der Modifier *"void"* **nicht** verwendet werden.
- *"public"* wird angegeben, wenn der *Constructor* auch von außerhalb eines *packages* (siehe später) erreichbar sein soll.
- Constructor-Methods können nicht explizit aufgerufen werden.

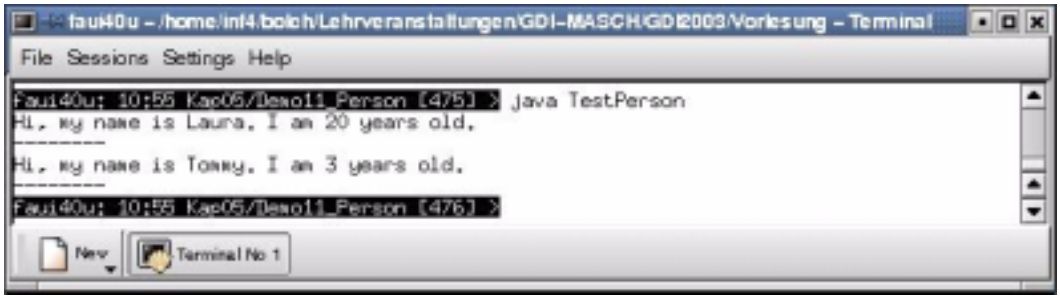
9 Konstruktoren *(Constructor Methods)*

- Beispiel (Demo11): (Zusammengefasst in einer Datei: *Person.java*)

```
class Person {
  private String name;
  private int age;

  Person( String n, int a ) {           // Konstruktor
    name = n;
    age = a;
  }
  void printPerson() {
    System.out.print( "Hi, my name is " + name );
    System.out.println( ". I am " + age + " years old." );
  }
}
public class TestPerson {
  public static void main( String args[] ) {
    Person p;
    p = new Person( "Laura", 20 );
    p.printPerson();
    System.out.println( "--------" );
    p = new Person( "Tommy", 3 );
    p.printPerson();
    System.out.println( "--------" );
  }
}
```

9 Konstruktoren *(Constructor Methods)*

- Ergebnis (Demo11_Person):



9 Konstruktoren *(Constructor Methods)*

◆ Das Überladen von Konstruktoren ist analog dem Überladen von Methoden möglich. Beispiel (Demo12_MyRect2):

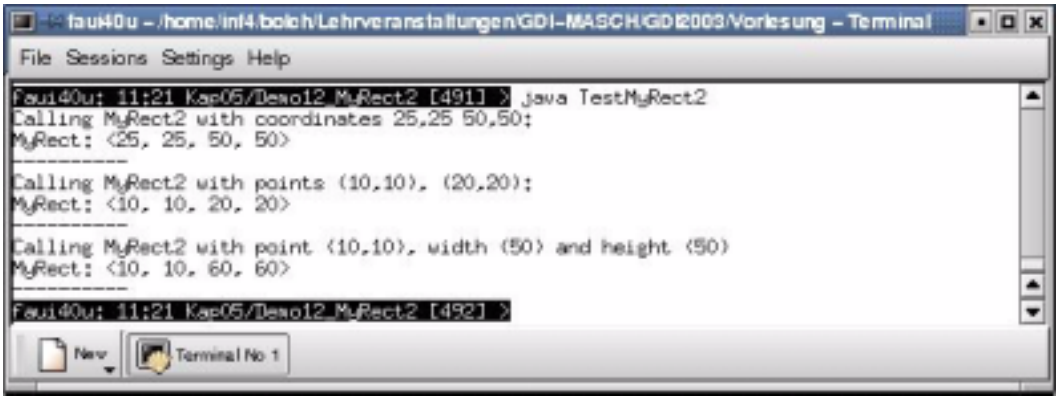
```
import java.awt.Point;

class MyRect2 {
  private int x1, y1;
  private int x2, y2;
    MyRect2( int x1, int y1, int x2, int y2 ) {
        this.x1 = x1;
        this.y1 = y1;
        this.x2 = x2;
        this.y2 = y2;
    }
    MyRect2( Point topLeft, Point bottomRight ) {
        x1 = topLeft.x;
        y1 = topLeft.y;
        x2 = bottomRight.x;
        y2 = bottomRight.y;
    }
    MyRect2( Point topLeft, int w, int h ) {
        x1 = topLeft.x;
        y1 = topLeft.y;
        x2 = (x1 + w);
        y2 = (y1 + h);
    }
```

9 Konstruktoren *(Constructor Methods)*

```
void printRect() {
        System.out.print( "MyRect: <" + x1 + ", " + y1 );
        System.out.println( ", " + x2 + ", " + y2 + ">" );
    }
}
class TestMyRect2{

    public static void main( String args[] ) {
        MyRect2 rect;

        System.out.println( "Calling MyRect2 with coordinates 25,25 50,50:" );
        rect = new MyRect2( 25, 25, 50,50 );
        rect.printRect();
        System.out.println( "----------" );

        System.out.println( "Calling MyRect2 with points (10,10), (20,20):" );
        rect= new MyRect2( new Point( 10,10 ), new Point( 20,20 ) );
        rect.printRect();
        System.out.println( "----------" );

        System.out.print( "Calling MyRect2 with point (10,10)," );
        System.out.println( " width (50) and height (50)" );
        rect = new MyRect2( new Point( 10,10 ), 50, 50);
        rect.printRect();
        System.out.println( "----------" );
    }
}
```


9 Konstruktoren *(Constructor Methods)*

- Ergebnis (Demo12_MyRect2):



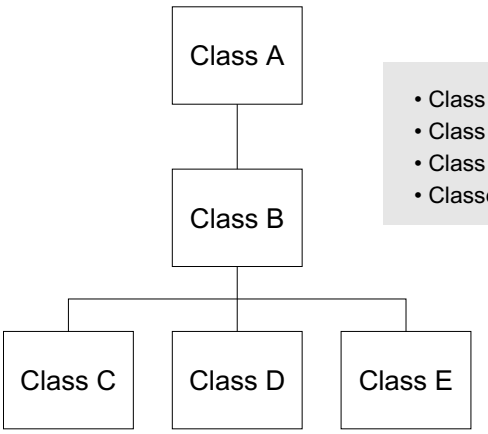
10 Vererbung *(Inheritance)*

◆ *Inheritance* ist ein grundlegendes Konzept der **O**bjekt-**O**rientierten **P**rogrammierung, das folgendes beinhaltet:

- Alle Klassen sind hierachisch geordnet. Jede Klasse hat
  - (in Java!) exakt eine Oberklasse (*superclass*)
  - und kann Unterklassen (*subclasses*) haben.
- Die höchste Klasse in der Hierachie ist eine Klasse mit dem Namen *Object*; sie ist automatisch Oberklasse aller Klassen.
- Eine Unterklasse ist im allgemeinen eine Erweiterung der Oberklassen; bei der Definition einer neuen Klasse müssen Sie also nur die "Erweiterung" zur Oberklasse definieren, den *"Rest erben Sie".*

10 Vererbung *(Inheritance)*

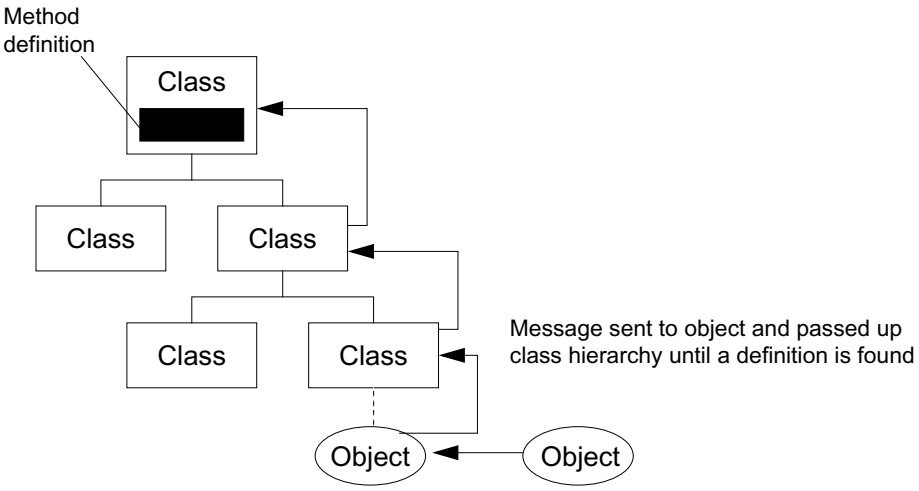
◆ Der erste Schritt beim Entwurf eines größeren Programmsystems ist der Entwurf eines Klassen-Hierachie-Diagrammes für die Aufgabenstellung:



- Class A is the superclass of B
- Class B is a subclass of A
- Class B is the superclass of C, D and E
- Classes C, D and E are subclasses of B

10 Vererbung *(Inheritance)*

◆ Wie funktioniert Inheritance?



Bei der Suche einer Methode in einem Objekt wird der Klassenbaum zur Wurzel hin durchlaufen, bis die Definition der Methode gefunden wird.

10 Vererbung *(Inheritance)*

◆ Beispiel (Demo13_Temperatur):

- Zur Erweiterung (Vererbung) von Klassen verwendet man das Schlüsselwort ***"extends"***

```
class Temperatur {
        double celsius;
        public void setCelsius( double grad ) {
            if ( grad < -273.15 )
                    grad = -273.15;
            celsius = grad;
            }
        public double getCelsius() {
            return celsius;
        }
}

class TemperaturFahrenheit extends Temperatur {
        public double getFahrenheit() {
            return 9.0/5.0 * celsius + 32.0;
        }
}
```

10 Vererbung *(Inheritance)*

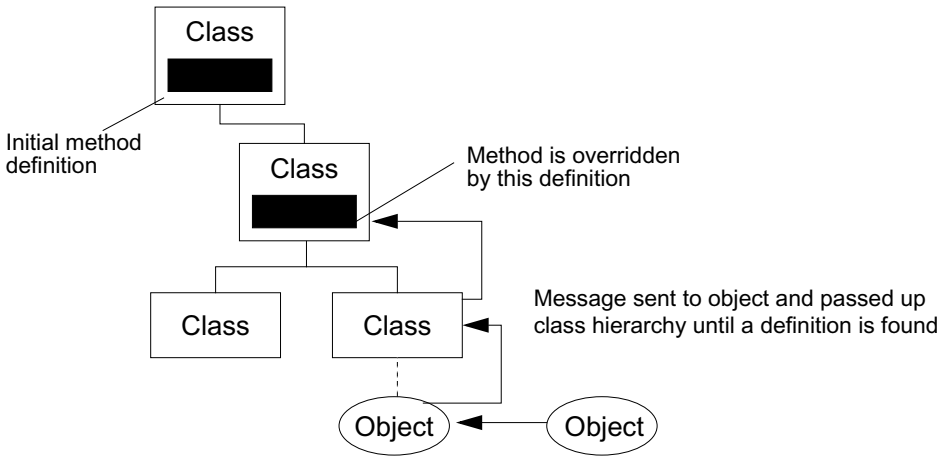
```
public class TemperaturTest {
        public static void  main ( String args[] ) {
            TemperaturFahrenheit temp;
            temp = new TemperaturFahrenheit();
            temp.setCelsius( 49.4 );
            System.out.println( "Temperatur in Grad Celsius: "
                                          + temp.getCelsius() );
            System.out.println( "Temperatur in Grad Fahrenheit: "
                                          +  temp.getFahrenheit() );
        }
}
```

- Ergebnis (Demo13_Temperatur):

```
faui40u: 13:41 Kap05/Demo13_Temperatur [571] > java TemperaturTest
Temperatur in Grad Celsius: 49.4
Temperatur in Grad Fahrenheit: 120.92
faui40u: 13:41 Kap05/Demo13_Temperatur [572] >
```


11 Überschreiben von Methoden *(Overriding Methods)*

◆ Man spricht von Overriding von Methoden dann, wenn bei der Definition einer Methode in einer Subklasse die **identische Signatur** einer Methode in einer der Superklassen verwendet wird.



11 Überschreiben von Methoden *(Overriding Methods)*

- Beispiel (Demo14_PrintClass):

```
class PrintClass {
    protected int x = 0;
    protected int y = 1;

    void printMe() {
        System.out.println( "x is " + x + ", y is " + y );
        System.out.println( "I am an instance of the class " +
                            this.getClass().getName() );
    }
}

class PrintSubClass extends PrintClass {
    protected int z = 3;

    void printMe() {
        System.out.println( "x is " + x + ", y is " + y ", z is " + z);
        System.out.println( "I am an instance of the class " +
                            this.getClass().getName() );
    }
}
```


11 Überschreiben von Methoden *(Overriding Methods)*

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

        PrintClass obj1 = new PrintClass();

        obj1.printMe();

        PrintSubClass obj2 = new PrintSubClass();

        obj2.printMe();
    }
}
```

- Ergebnis (Demo14_PrintClass):

```
faui40u: 16:11 Kap05/Demo14_PrintClass [585] > java OverridingTest
x is 0, y is 1
I am an instance of the class PrintClass
x is 0, y is 1, z is 3
I am an instance of the class PrintSubClass
faui40u: 16:11 Kap05/Demo14_PrintClass [586] >
```

11 Überschreiben von Methoden *(Overriding Methods)*

- Häufig will man bei Anwendung der *Overriding* - Technik vorhandene Methoden nicht vollständig ersetzen, sondern nur ihre Funktionalität erweitern.
- Ziel ist es dann, die Originalmethode in der "*Overriding*-Methode" zu benutzen und nur die zusätzliche Funktionalität zu definieren.
- Ähnlich wie man mit dem *this - operator* das *"current object"* referenziert, kann man mit dem *super -Operator* den Methodennamen in einer der Superklassen referenzieren.

11 Überschreiben von Methoden *(Overriding Methods)*

- Beispiel (Demo15_PrintAnotherClass):

```
class PrintAnotherClass {
    private int x = 0;
    private int y = 1;

    void printMe() {
        System.out.println( "I am an instance of the class " +
                            this.getClass().getName() );
        System.out.println( "x is " + x );
        System.out.println( "y is " + y );
    }
}
```

11 Überschreiben von Methoden *(Overriding Methods)*

```
class PrintAnotherSubClass extends PrintAnotherClass {
    int z = 3;

    void printMe() {
        super.printMe();
        System.out.println("z is " + z);
    }
    public static void main( String args[] ) {
        PrintAnotherSubClass obj =
                  new  PrintAnotherSubClass();
        obj.printMe();
    }
}
```

- Ergebnis (Demo15_PrintAnotherClass):

```
faui40u: 17:34 Kap05/Demo15_PrintAnotherClass [607] > java PrintAnotherSubClass
I am an instance of the class PrintAnotherSubClass
x is 0
y is 1
z is 3
faui40u: 17:35 Kap05/Demo15_PrintAnotherClass [608] >
```


12 Überschreiben von Konstruktoren *(Overriding Constructors)*

- *Overriding* von Konstruktoren ist grundsätzlich **NICHT** möglich, da der Konstruktor den Namen der Klasse hat.
- Es ist jedoch möglich bei der Initialisierung einer Subklasse den Konstruktor der Superklasse aufzurufen.
  - Dies geschieht mit

    `super( arg1, arg2, ... );`
  - **`super`** muß das 1. ausführbare Statement des neuen Konstruktors sein.
  - Andernfalls wird automatisch

    `super();`

  ohne Parameter aufgerufen, also die *"default"* - Initialisierung ausgeführt.

12 Überschreiben von Konstruktoren

(Overriding Constructors)

- Beispiel (Demo16_NamedPoint) :

```
import java.awt.Point;

class NamedPoint extends Point {
    private String name;

    NamedPoint( int x, int y, String name ) {
        super( x, y );
        this.name = name;
    }
    String getName(){
        return name;
      }
}
```


12 Überschreiben von Konstruktoren

(Overriding Constructors)

```
class TestNamedPoint{

    public static void main( String args[] ) {
        NamedPoint np = new NamedPoint( 5, 5, "SmallPoint" );
        System.out.println( "x is" + np.x );
        System.out.println( "y is" + np.y );
        System.out.println( "Name is" + np.getName() );
    }
}
```

- Ergebnis (Demo16_NamedPoint):

```
faui40u: 18:26 Kap05/Demo16_NamedPoint [627] > java TestNamedPoint
 x is 5
 y is 5
 Name is SmallPoint
faui40u: 18:26 Kap05/Demo16_NamedPoint [628] >
```

5.3 Access-, Final- and Static- Modifiers

- *Packages* sind Zusammenfassungen von Klassen mit eigenem **Namensraum**.
- *Access Modifier* sind spezielle *keywords* mit denen man Gültigkeitsbereiche gegenüber den *default* - Spezifikationen einschränken oder erweitern kann.

◆ Für Variable gilt:

- *default*: Siehe Kapitel 5.1, innerhalb eines *packages*
- *public*: zugreifbar auch über *packages* hinweg
- *protected*: zugreifbar wie *"default"* und zusätzlich auch von Subklassen außerhalb des *packages*
- *private*: zugreifbar nur für Methoden der Klasse selbst

- *final*: Namenskonstante
- *static:* Klassenvariable

5.3 Access-, Final- and Static- Modifiers

◆ Für Methoden gilt:

- *default*: aufrufbar von allen Klassen innerhalb eines *"packages"* es sei denn, die Methode ist überschrieben worden.
- *public*: ausführbar auch über *packages* hinweg
- *protected:* wie bei Variablen
- *private*: nur innerhalb der Klasse selbst aufrufbar
- *final*: nicht überschreibbar
- *static*: Klassenmethode (Klassenmethoden sind **nicht** überschreibbar.)

◆ Für Klassen gilt:

- *final*: keine Subklasse möglich
- *public*: auch von Klassen außerhalb des *packages* erreichbar

◆ Weitere Modifier zu behandeln in späteren Kapiteln:

- *abstract*
- *synchronized, volatile*
- *native*

5.4 Zusammenfassung

- Klassen
  - Definition
  - Klassenvariable
  - Klassenmethoden
  - Vererbung
- Objekte (Instanzen)
  - Erzeugung (Instantiierung)
    - Parameterübergabe
    - Konstruktoren
    - Initialisierung
  - Objektvariable, Modifier, Gültigkeitsbereiche
  - Zugehörigkeit zu einer Klasse
  - Relationale Operatoren

5.4 Zusammenfassung

- Objekte (Instanzen) (cont.)
  - Methoden, Modifier
    - Signatur
    - Aufruf
    - Parameterübergabe
      - by reference
      - by value
    - lokale Variable, Gültigkeitsbereiche
    - Returnparameter
    - Overloading
    - Overriding
  - Konstruktoren
    - Initialisierung
    - Overloading

5.4 Zusammenfassung

- Variable
  - Klassenvariable
  - Objektvariable
  - methodenlokale Variable
  - blocklokale Variable
  - Gültigkeitsbereiche
  - Casting

Notizen