Übungen zu

Objektorientierte Konzepte in Verteilten Systemen und Betriebssystemen

Sommer 2002

Teil I: Java

A Übersicht

A.1 Teil I: Java

- Die Sprache Java
- OO-Grundlagen in Java: Klassen, Objekte, Vererbung
- automatische Tests mit JUnit
- weitere OO-Konzepte in Java: Packages, Innere Klassen, Abstrakte Klassen, Final-Klassen und -Methoden
- Fehlerbehandlung, Ein-/Ausgabe, AWT, Threads

A Übersicht

A.2 Teil II: Verteilte Programmierung

- Streams, Sockets
- Serialization, Classloader
- RMI
- Message Passing, RPC, ORB
- ORB Erweiterungen: Sicherheit, Replikation, Persistenz

A Übersicht

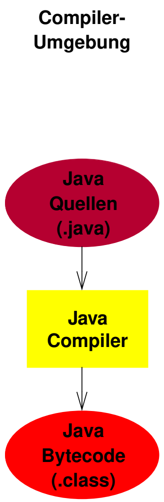
A.3 Teil III: CORBA

- CORBA Programmieren in Java
- CORBA Services
- weiterführende Themen

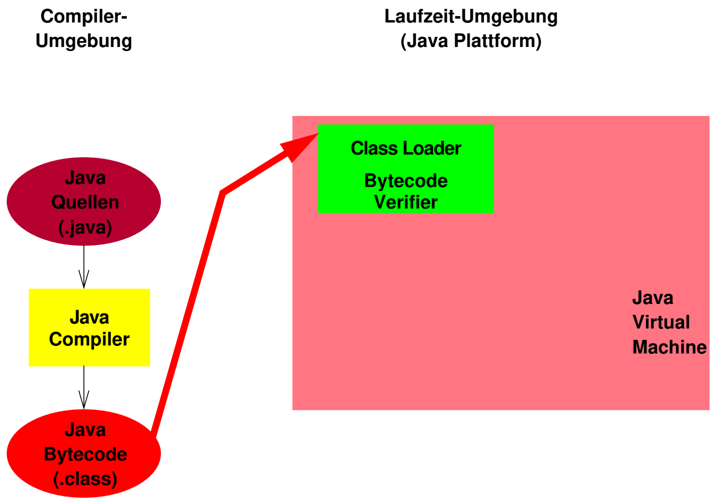
B Java - Überblick

- Was ist Java?
  - eine objektorientierte Programmiersprache mit folgenden Eigenschaften:
    - einfach
    - portabel
    - robust
  - ein sicheres Laufzeitsystem
    - beliebige Programme ohne Gefahr ausführen
    - Zugriffskontrolle auf lokale Ressourcen (z.B. Dateien) und das Netzwerk

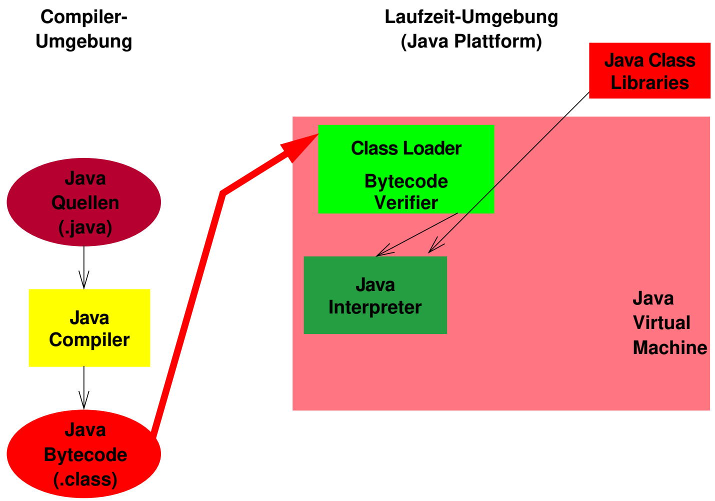
B.1 Das Java-System



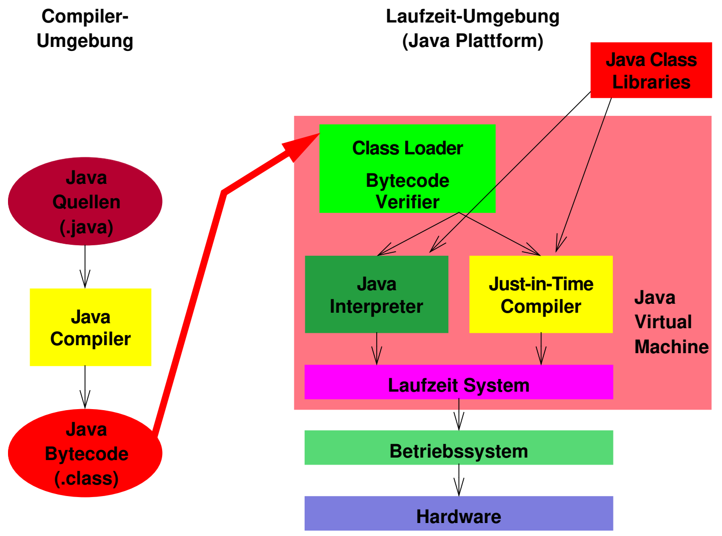
B.1 Das Java-System (2)



B.1 Das Java-System (3)



B.1 Das Java-System (4)



B.2 Java vs. C/C++

- kein Präprozessor (#define)
- kein typedef, union, enum, struct
- keine Zeiger-Arithmetik
- keine Funktionen oder Prozeduren
- keine globalen Variablen

B.3 Datentypen

- Primitive Datentypen:
  - byte, short, int, long (8/16/32/64 bit)
  - float, double (32/64 bit)
  - char (Unicode, 16 bit)
  - boolean (true, false)
  - “einfache” Operatoren: +,-,+=,<<,...
- Unterschiede zu C:
  - keine “unsigned” Datentypen
  - spezielle Datentypen “boolean” und Unicode char
  - kein `while (x--)`, stattdessen `while (x-- != 0)`
  - kein `if (ptr)`, stattdessen `if (ptr != null)`

B.4 Kommentare

- einzeilige Kommentare:

```
// Das ist ein Kommentar.
```

- mehrzeilige Kommentare:

```
/* Das ist
 * ein Kommentar.
 */
```

- Dokumentation (Erzeugt HTML-Dokumentation mit dem Tool `javadoc`):

```
/** Dieser Kommentar
 *  enthält javadoc-
 *  Dokumentation.
 */
```

B.5 Kontrollstrukturen

- Kontrollstrukturen wie in C:
  - for (... ; ... ; ...) { ... }
  - while (...) { ... } oder do { ... } while (...);
  - switch (...) { case .... }
  - if ....
  - **return**, **break**, **continue**
- Unterschiede zu C:
  - kein **`goto`**
  - **`break`** und **`continue`** mit Label:

```
outerloop: for ( .... ) {        // this is "outerloop"
    while ( .... ) {
        if ( ... ) break outerloop;// leave "outerloop"
    }
}
```


B.6 Strings

- gute Unterstützung für Stringverarbeitung
- Datentyp **`String`**:

```
String hello = "Hallo";
```

- Zusammenfügen mit “+”:

```
String world = "Welt";
String greeting = hello + " " + world + "!";
System.out.println(greeting);
```

Ausgabe:

Hallo Welt!

B.6 Strings (2)

- Zusammenfügen funktioniert auch mit **`int`**, **`float`**,...:

```
String test = "Result: " + (2 + 3);
```

Inhalt von test: "**`Result: 5`**"

- **Wichtig: Strings sind konstant.**

B.7 Arrays

- Deklaration mit ***type*** **`name[] oder`** ***type*** **`[] name`**
- Allokation mit **`new`**, keine Freigabe notwendig (garbage collector):

```
int numbers[] = new int[100];
```

- Elementzugriff wie in C: **`name[index]`**:

```
for (int n=0; n<100; n++) numbers[n] = n;
```

- automatische Bereichsüberprüfung

B.7 Arrays (2)

- Deklaration und Initialisierung:

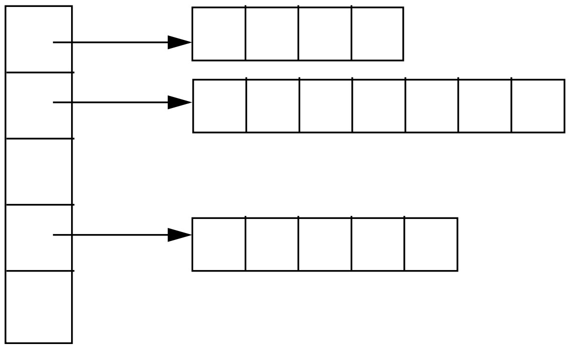
```
int [] firstPrimes = { 2, 3, 5, 7 };
```

- Ermitteln der Länge eines Arrays mit **`length`**:

```
for(int i=0; i<firstPrimes.length; i++) {
    System.out.println(firstPrimes[i]);
}
```


B.7 Mehrdimensionale Arrays

- Array-Elemente können wiederum Arrays sein:



```
int [][] matrix = new int[5][4];
```

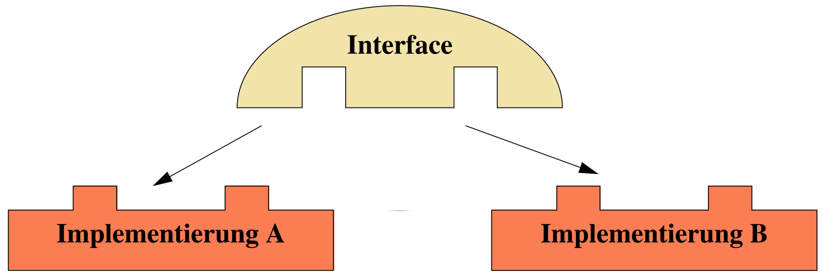

C OO-Grundlagen mit Java

- Fundamentale Konzepte der objektorientierten Programmierung:

Abstraktion
und
Kapselung

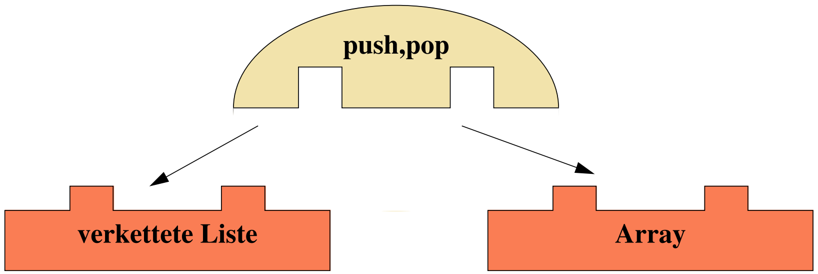
C.1 Abstraktion

- Trennung von:
  - Schnittstelle (interface): Was kann getan werden?
  - Implementierung: Wie wird es gemacht?



C.1 Abstraktion

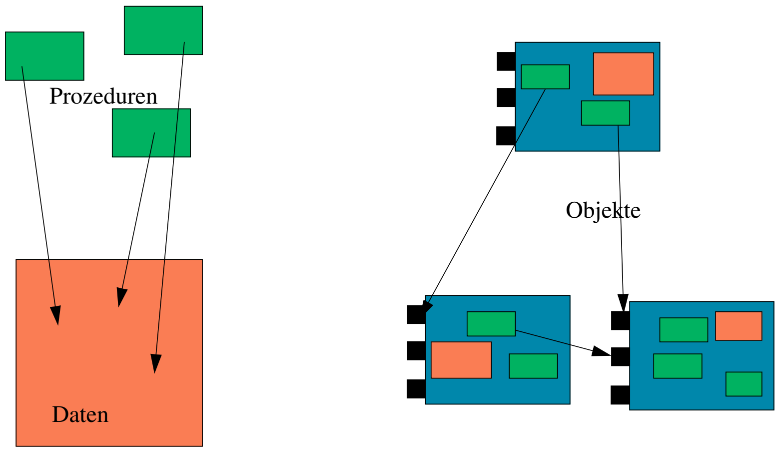
- Beispiel: stack
  - Interface: push, pop
  - Implementierung: verkettete Liste, Array



C.2 Kapselung

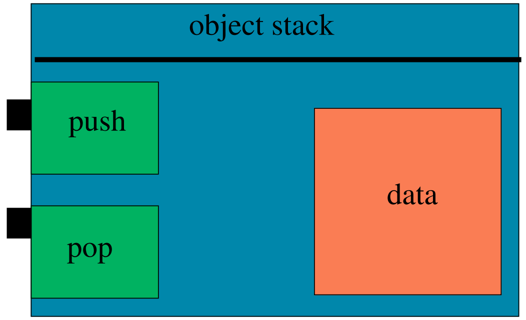
Prozedurale Programmierung

Objektorientierte Programmierung



C.2 Kapselung

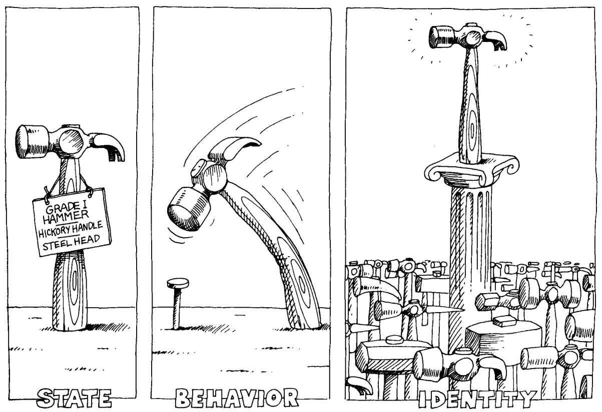
- Objekte: Gekapselte Datenstruktur, bestehend aus:
  - Daten (Instanzvariablen, Attribute)
  - Methoden (Operationen)



- Kapselung unterstützt die Bildung von Abstraktionen.

C.3 Objekte

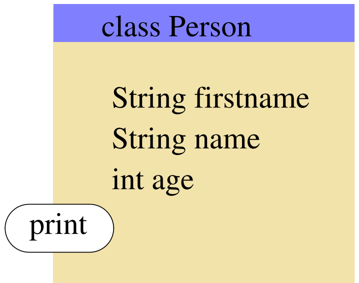
1 Eigenschaften von Objekten



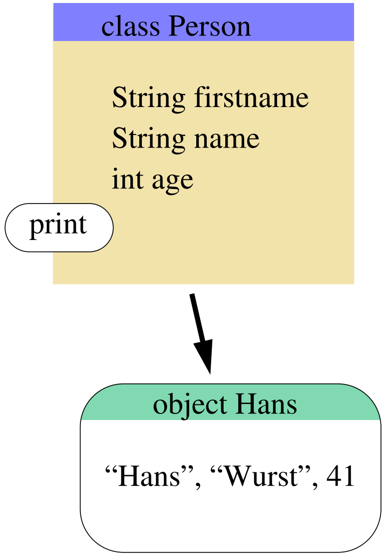
2 Klassen

- Objekte sind *Instanzen* einer Klasse.
- Die Klasse bestimmt die interne Struktur und die Schnittstelle eines Objekts.

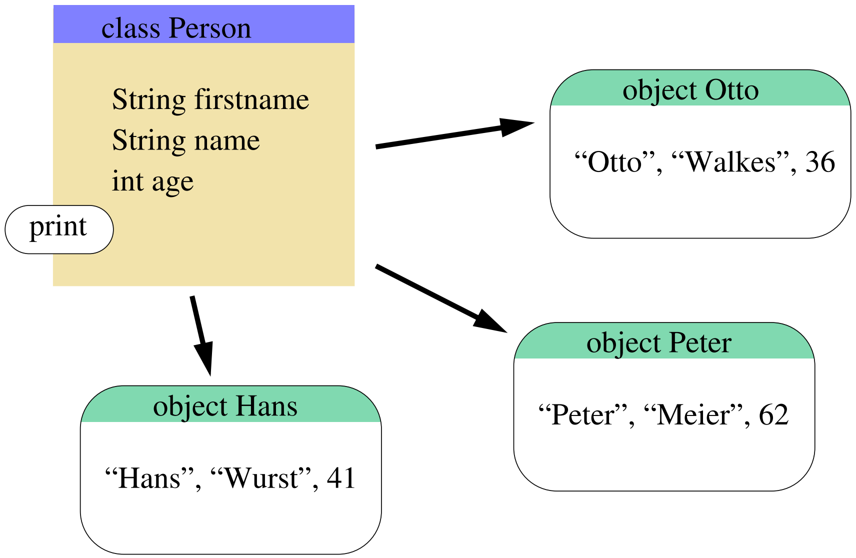
2 Klassen - Beispiel



2 Klassen - Beispiel



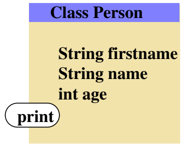
2 Klassen - Beispiel



2 Klassen - Beispiel

- Klassendefinition:

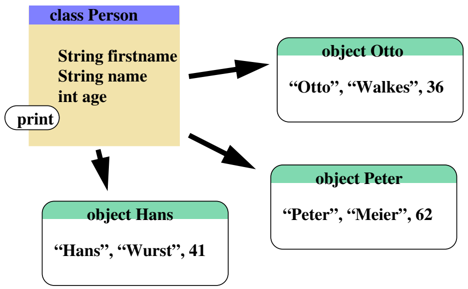
```
class Person {
    String firstname;
    String name;
    int age;
    void print() {
     System.out.println("Name: " + firstname + " " + name +
                        " Age: " + age);
    }
}
```



3 Objekterzeugung

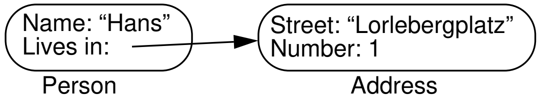
- Objekterzeugung (Instantiierung):

```
Person otto = new Person();
otto.firstname = "Otto";
otto.name = "Walkes";
otto.age = 36;
otto.print();
```

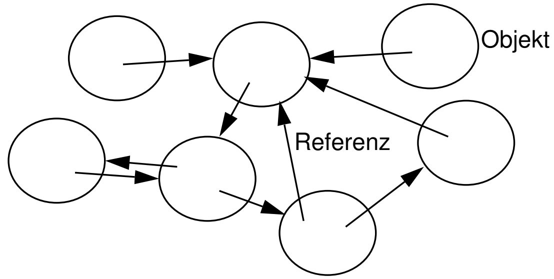


4 Das Objekt-Netz

- Objekte können andere Objekte referenzieren.
  - Beispiel: eine Person kann eine Adresse referenzieren:



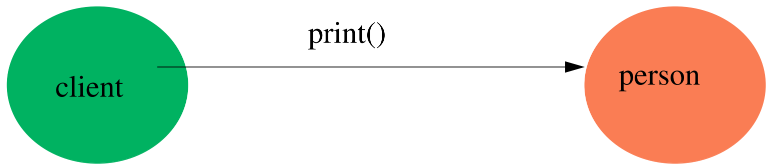
- Darstellung des Zustands eines objektorientierten Programms:



5 Nachrichten / Methoden

- Objekte kommunizieren mit Hilfe von Nachrichten.
  - Jedes Objekt legt sein eigenes Verhalten selbst fest.
  - Die Objektsemantik ist nicht über das ganze Programm verteilt.
- Nachricht = Methodenaufruf an einem Objekt

```
person.print()
```



- objektorientiertes Programm: mehrere Objekte kommunizieren miteinander, um eine bestimmte Aufgabe zu erfüllen.

6 Methoden und Variablen

- Methoden können auf 3 verschiedene Arten von Variablen zugreifen:
  - Parameter (Argumente)
  - lokale Variablen
  - Instanzvariablen

```
class Point {
    int xPosition, yPosition;
    ...
    void move(int x, int y) {

        int i;
        ...
    }
}
```

Instanzvariablen

Parameter

lokale Variablen

6 Variablen

- Zugriff auf Instanzvariablen mit:
  - `instanceName.variableName`

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

    boolean sameName(Person otherPerson) {
        if (name == otherPerson.name) return true;
        return false;
    }
}
```

Instanzvariable

Instanzvariable

6 Variablen

- Parameter und lokale Variablen müssen unterschiedliche Namen haben.
- Parameter und lokale Variablen überdecken Instanzvariablen.

```
class Test {
    int a;
    void m(int a) {
        a = 5; // does not change instance variable a
    }
}
```


7 Der Parameter *this*

- Jede Methode hat einen *impliziten* Parameter **`this`**.
- **`this`**: Referenz auf die Instanz, an der die Methode aufgerufen wurde:

```
class Person {
    String name;
    ...
    void print() {
        System.out.println(this.name);
    }
}
```

- **`this`** kann bei Eindeutigkeit weggelassen werden.
- Beispiel für Mehrdeutigkeit:

```
class Person {
    ...
    boolean compare(String name) { return this.name == name; }
}
```

8 Überladen

- Methoden mit unterschiedlichen Parametern können den gleichen Namen haben.
- Beispiel:

```
class Date {
    ...
    void print(PrintStream stream) { stream.println(...); }
    void print() { print(System.out); }
}
```

- Hinweis: Überladen funktioniert nur mit Parametern nicht mit dem Typ des Rückgabewerts:

```
class Income {
    ...
    int computeIncome() { ... }
    float computeIncome() { ... } // Error !!
}
```


9 Objekt-Initialisierung

- Erzeugen eines Objekts bedeutet Reservierung von Speicher.
- Dieser Speicher muss initialisiert werden.
- Eine Möglichkeit:
  - ◆ Explizites Aufrufen einer Initialisierungsmethode.
  - ◆ Nachteil: fehleranfällig.

10 Konstruktoren

- Konstruktoren dienen der Initialisierung des Objekts.
- Name des Konstruktors = Name der Klasse.
- Der Konstruktor wird automatisch nach der Objekterzeugung aufgerufen.

10 Konstruktoren (2)

- Initialisierung einer Person mit Name und Alter:

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

    Person(String name, int age) {
        this.name = name;
        this.age = age;
    }
    ...
}
```

10 Konstruktoren (3)

- Mehrere Konstrukturen sind möglich:

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

    Person(String name, int age) {
        this.name = name;
        this.age = age;
    }
    Person(String name) {
        this.name = name;
        this.age = 18;
    }
    ...
}
```


10 Konstruktoren (4)

- Aufruf eines anderen Konstruktors mit **`this(...)`**:

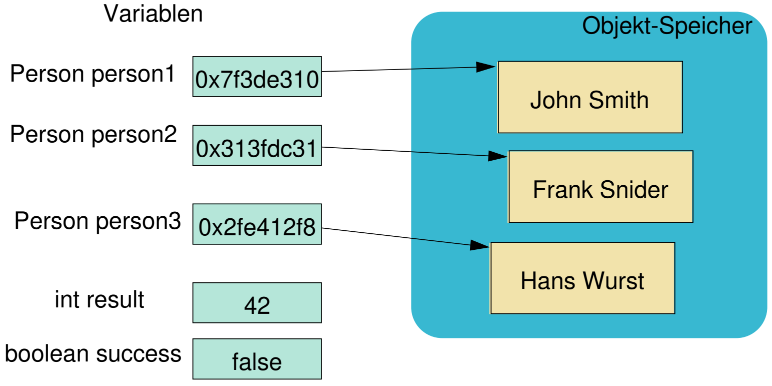
```
class Person {
    String name;
    int age;

    Person(String name, int age) {
        this.name = name;
        this.age = age;
    }
    Person(String name) {
        this(name,18);
    }
    ...
}
```


11 Objekte und Referenzen

- Java-Variablen bezeichnen keine Objekte, sondern Referenzen.

```
Person p;       // Deklaration einer Referenz auf ein Objekt
                // der Klasse Person
p.print();      // Fehler: Methodenaufruf an einer null-
```



12 Zuweisungen

- = weist einer Variable eine Referenz zu
- == vergleicht zwei Referenzen
- Einer Variable eines primitiven Datentypes kann keine Referenz zugewiesen werden.
- Einer Variable, welche eine Objektreferenz ist, kann niemals der Wert eines primitiven Datentypes zugewiesen werden.
- Beispiel:

```
Person p;       // Deklaration einer Referenz-Variablen
int i=42;     // Deklaration und Initialisierung einer
                // Variable eines primitiven Datentypes
p = i;        // Fehler: Zuweisung zwischen Referenz und
                // primitiven Datentyp
```

13 Aufrufsemantik von Methoden

- Objekt-Parameter werden als Referenz übergeben.
- Primitive Datentypen (int, float, etc.) werden als Wert übergeben.
- Beispiel:

```
void meth(int a, Person k) {
    a = 5;              // a: passed by value
    k.setAge(25);       // k: passed by reference
}
```


14 Gleichheit und Identität

- Unterschied zwischen *gleichen Objekten* und *identischen Objekten*:

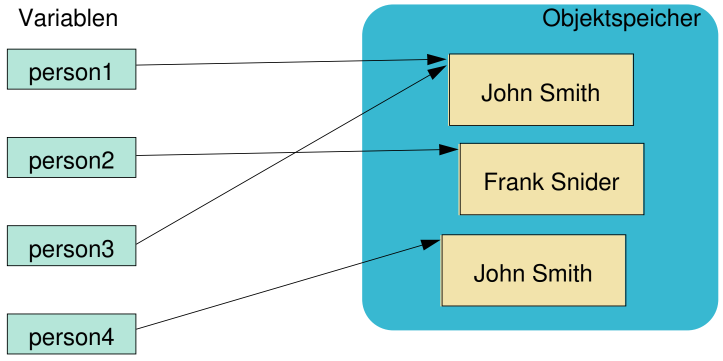
```
class Date {
  int day, month, year;
  Date(int day, int month, int year) {
    this.day = day; this.month = month; this.year = year;
  }
}
...
Date d = new Date(1,3,98);
Date d1 = new Date(1,3,98);
Date d2 = d;
```

◆ d und d1 sind gleich

◆ d und d2 sind identisch

15 Identität und Referenzen

- Identität: gleiche Referenz
- Gleichheit: Inhalt der referenzierten Objekte ist gleich



- Welche Personen sind identisch, welche gleich?

16 Testen von Gleichheit und Identität

- Identität kann mit dem Operator == getestet werden:

```
if (d == d1) { ... }
```

- Gleichheit kann mit der Methode **equals** getestet werden:

```
if (d.equals(d1)) { ... }
```

- Die Methode **equals** muss selbst implementiert werden:

```
class Date {
  ...
  public boolean equals(Object o) {
    if (! (o instanceof Date)) return false;
    Date d = (Date)o;
   return d.day == day && d.month == month && d.year == year;
  }
}
```

16 Testen von Gleichheit und Identität (2)

- Wenn zwei Objekte gleich sind, dann müssen sie den gleichen Hash-Wert besitzen:

```
class Date {
    ...
  public int hashcode() {
    return day + month + year;
  }
}
```