Objektorientierte und Formale Programmierung
-- Java Grundlagen --

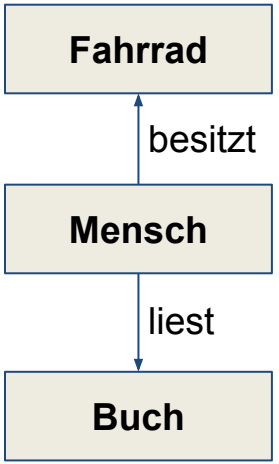
6. Objekte und Methoden

6. Objekte und Methoden

Prinzipien der Objektorientierung

Abstraktion der Wirklichkeit:
wird als Menge **interagierender Objekte** modelliert

- Ein Objekt
  - hat bestimmte **Eigenschaften (Attribute)** / einen **Zustand**
  - reagiert mit einem bestimmten **Verhalten** (beschrieben durch eine Menge von **Operationen** / **Methoden**) auf die Umgebung
  - steht in bestimmten **Beziehungen** zu anderen Objekten
- Objekte werden zu **Klassen** zusammengefaßt
  - Abstraktion von der konkreten Ausprägung eines Objekts
  - Objekte mit gleichen Attributen und gleichem Verhalten werden gemeinsam betrachtet
  - Klasse ist Stellvertreter / Bauplan für diese Objekte

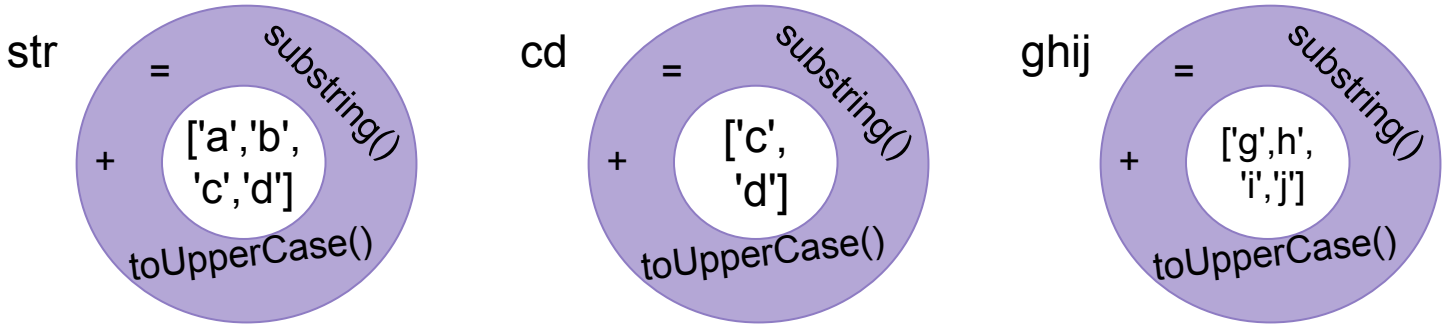


6. Objekte und Methoden

Beispiel: String Klasse

```
class StringBeispiel {
  public void beispiel( String ghij ) {
    String str = "abcd";                // str ist Objekt von Klasse String: "abcd"
    String cd = str.substring(2, 4);  // cd ist Objekt von Klasse String:  "cd"
    System.out.println(str.toUpperCase()+"ef"); // Objekt von Klasse String "abcdef"
  }
}
// ...
StringBeispiel bsp = new StringBeispiel();  bsp.beispiel("ghij");
```

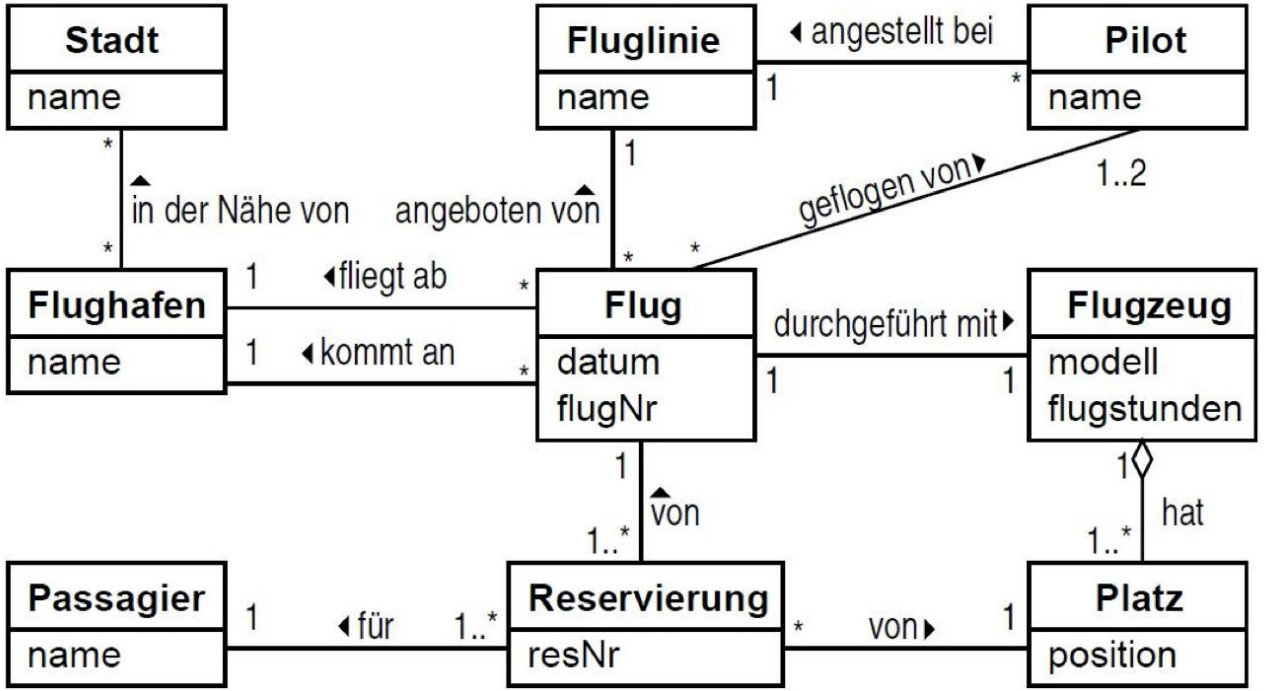
Objekte:



6. Objekte und Methoden

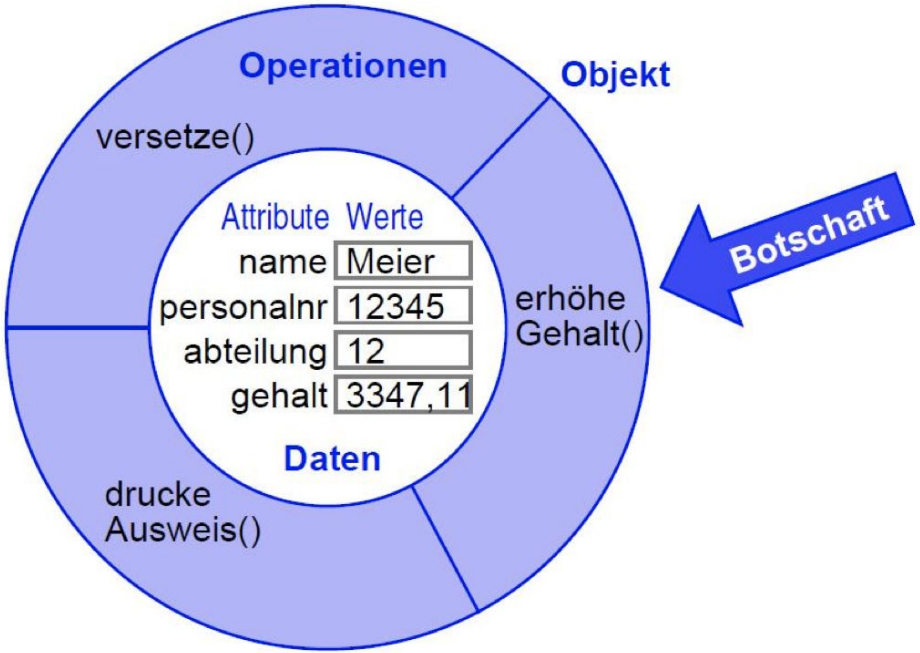
Prinzipien der Objektorientierung

- Beispiel UML Klassendiagramm: (wird später eingeführt)



Prinzipien der Objektorientierung

Objekt und Geheimnisprinzip:



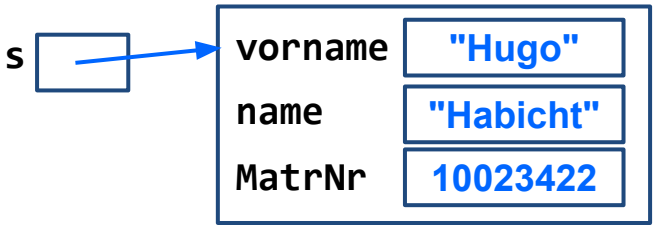
- Ein wesentliches Konzept von objektorientierter Programmierung ist, dass der innere Aufbau eines Objekts für andere Objekte zum großen Teil unzugänglich ist. Viele Teile eines Objekts bleiben geheim, befinden sich sozusagen in einer geschützten Kapsel. Man nennt das Geheimnisprinzip auch **Kapselung**.

6. Objekte und Methoden

6.1. Erzeugung von Objekten

- Objekte werden in Java nicht direkt in Variablen gespeichert (wie auch für Arrays)
- In Variablen werden nur Referenzen auf solche Speicherbereiche gespeichert:

```
class Student {
  String vorname, name;
  long matrNr;
   ...
}
...
Student s = new Student("Hugo", "Habicht", 10023422);
```



- Erzeugung eines Objekts: Der Speicherbereich für die Daten wird dynamisch durch den Operator **new** angelegt

6.1. Erzeugung von Objekten

Konstruktoren

- Eine Klasse kann spezielle Operationen, *Konstruktoren*, definieren, die bei der Erzeugung eines Objekts ausgeführt werden
- Typische Aufgaben eines Konstruktors:
  - Initialisierung der Attributwerte des neuen Objekts
  - ggf. Erzeugung existenzabhängiger Teil-Objekte
- Ein Konstruktor hat immer denselben Namen wie die Klasse
  - er kann Parameter besitzen, hat aber keinen Ergebnistyp (nicht einmal **`void`**)
  - Konstruktoren können auch überladen werden (siehe nächste Folie)
- Definiert eine Klasse keinen Konstruktor, wird automatisch ein parameterloser Konstruktor erzeugt
  - Attribute werden mit Standardwerten (**`0`** bzw. **`null`**) initialisiert

6.1. Erzeugung von Objekten
Konstruktoren

```
class Kugel {

  // Klassenattribute:
  final static double PI = 3.14159265;
  static int anzahl = 0; // zählt erzeugte Kugeln

  // Attributes:
  double xMitte, yMitte, zMitte, radius;

  // Parameterloser Konstruktor (Default-Konstruktor):
  Kugel() {
    radius = 1;      // setze Radius des neuen Objekts,
                     // andere Attribute sind mit 0 initialisiert
    anzahl++;  // Klassenvariable erhöhen
  }
```

6.1. Erzeugung von Objekten
Konstruktoren

```
  Kugel(double x, double y, double z) {
    this();             // ruft Konstruktor Kugel() auf
    xMitte = x;         // Aufruf von this() muss die
    yMitte = y;         // allererste Anweisung sein
    zMitte = z;
  }
  Kugel(double x, double y, double z, double r) {
    this(x, y, z);      // ruft Kugel(x,y,z) auf
    radius = r;         // ändert Wert des Radius auf r
  }
}
```

- this([<Parameterliste>]) als erste Anweisung führt zum Aufruf eines anderen Konstruktors für dasselbe Objekt

6. Objekte und Methoden

6.1. Erzeugung von Objekten

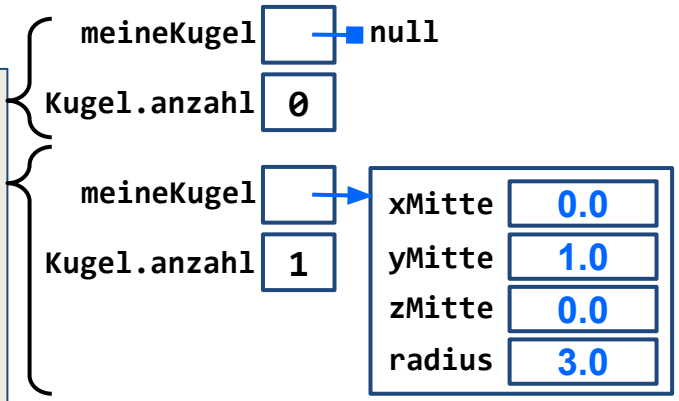
- Der Ausdruck **`new <Klassenname> ( [<Parameterliste>] )`** erzeugt (instanziiert) ein neues Objekt der angegebenen Klasse
  - Wert des Ausdrucks ist eine Referenz auf das Objekt
- Beispiele:

```
new Kugel()
new Kugel(1, 0, 0)
```

- Ablauf bei der Instanziierung:
  - Anlegen des Objekts im Speicher
    - dabei Belegung der Attribute mit Standardwerten
  - Aufruf des Konstruktors für das neue Objekt
    - passend zu Anzahl / Typen der übergebenen Parameter
  - Rückgabe der Referenz auf das Objekt

6.1. Erzeugung von Objekten

```
Kugel meineKugel;
meineKugel = new Kugel(0, 1, 0, 3);
...
  Kugel() {
        radius = 1;
        anzahl++;
  }
  Kugel(double x, double y, double z) {
        this();
        xMitte = x; yMitte = y; zMitte = z;
  }
  Kugel(double x, double y, double z, double r) {
        this(x, y, z);
        radius = r;
  }
```



Kugel(0,1,0,3)→Kugel(0,1,0)
→Kugel()

6.1. Erzeugung von Objekten

Weitere Möglichkeiten zur Initialisierung von Attributen

- Gemeinsam mit der Deklaration:

```
static int anzahl = 0;
double radius = 1.0;  // Default-Radius: 1.0
```

- In einem eigenen, speziellen Block der Klassendefinition:

```
class Kugel {
    ...
    static {          // wird nur einmal durchlaufen,
        anzahl = 0;   // wenn die Klasse geladen wird
    }
    {                 // wird für jedes erzeugte Objekt
        radius = 2.0; // ausgeführt (vor dem Konstruktor)
    }
}
```

6. Objekte und Methoden

6.1. Erzeugung von Objekten
Lebensdauer von Objekten

```
{
    Student s;
    {
        Student thomas = new Student("Thomas");
        Student tom = new Student("Tom");
        // ‘Tom’ und ‘Thomas’ sind verschiedene Studenten
        tom = thomas;
        // Hier wurde nur die Referenz kopiert
        // tom und thomas verweisen jetzt auf dasselbe Objekt
        // Das Objekt ‘Tom’ ist nicht mehr zugreifbar
        s = thomas;
    }
    // Das Objekt ‘Thomas’ existiert noch (s ist eine Referenz auf ‘Thomas’)
}
```

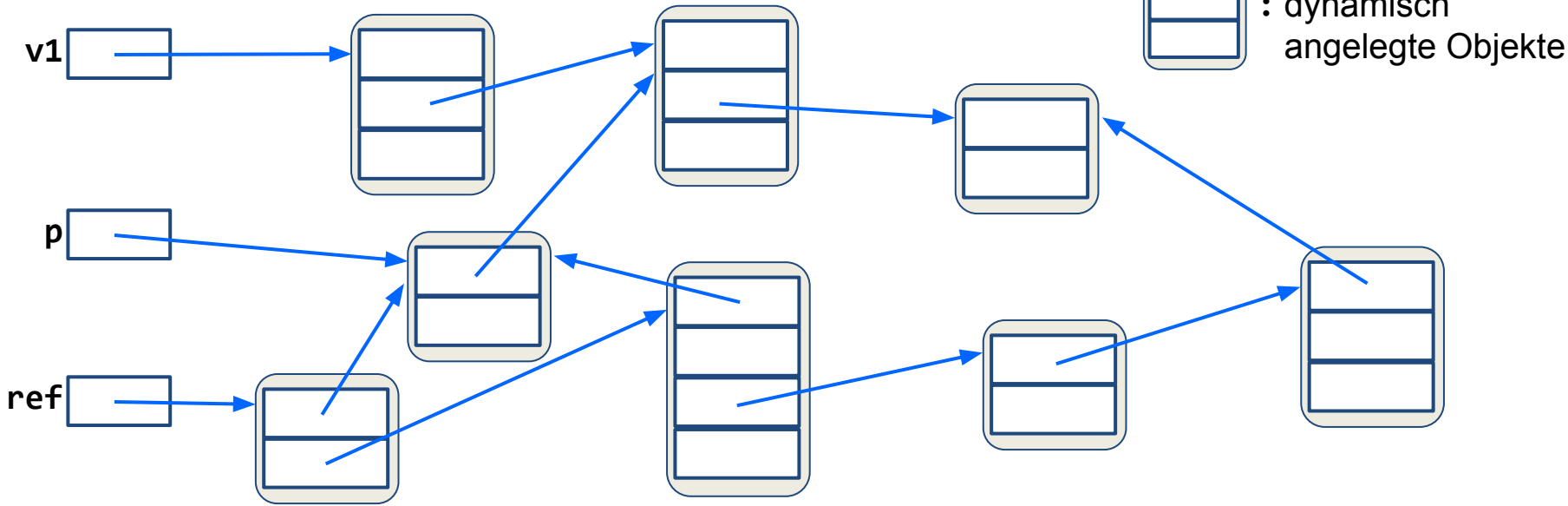
6.1. Erzeugung von Objekten

Freigabe von Speicherbereichen

- In Java existiert ein einmal erzeugtes Objekt weiter, solange es noch eine Möglichkeit gibt auf das Objekt zuzugreifen
  - d.h., solange es noch über eine Kette von Referenzen von einer Variable aus erreicht werden kann
- Ein mit **`new`** angelegter Speicherbereich (Objekt oder Array) wird von der JVM automatisch wieder freigegeben, wenn
  - Speicherplatz benötigt wird und
  - der Speicherbereich nicht mehr zugreifbar ist
- **Garbage Collection:**
  Die Suche nach solchen Speicherbereichen und deren Freigabe

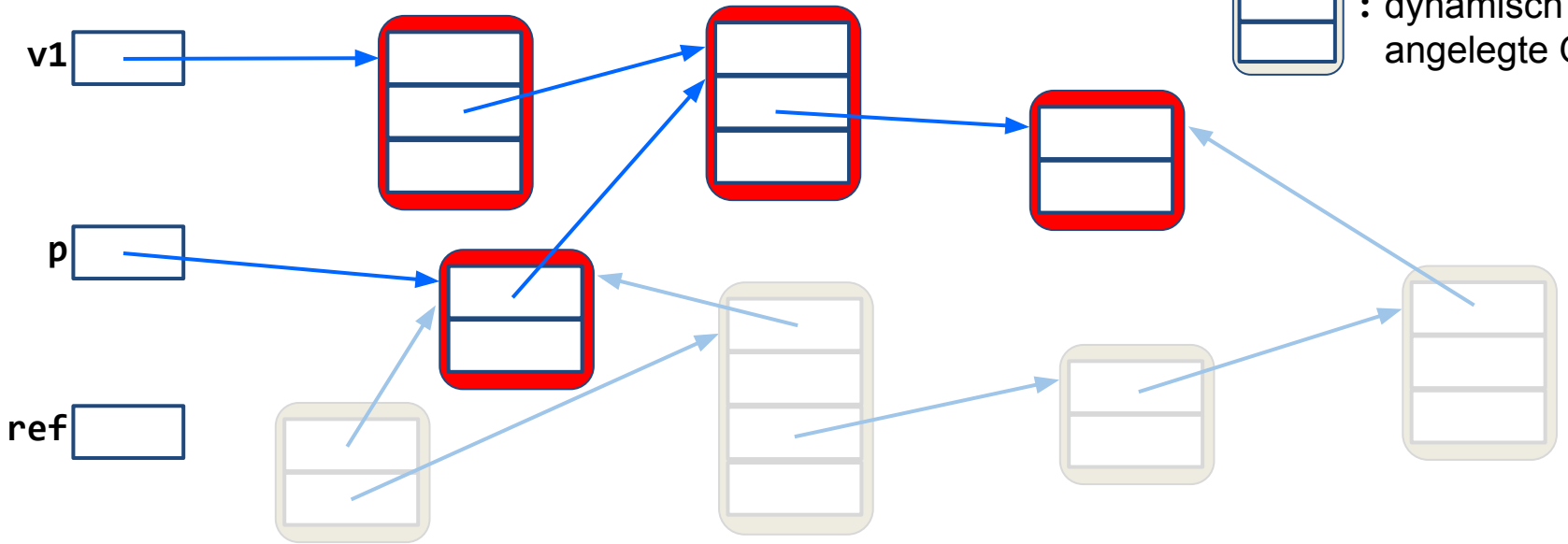
6.1. Erzeugung von Objekten

Garbage Collection



6.1. Erzeugung von Objekten

Garbage Collection



Wenn **`ref`** ungültig oder mit **`null`** belegt wird, markiert der Garbage Collector alle erreichbaren Objekte. Die unmarkierten werden gelöscht.

6.2. Zugriff auf Attribute und Methoden

- Die Attribute und Methoden eines Objekts können über eine Objektreferenz angesprochen werden:

```
<Objektreferenz> . <Attributname>
<Objektreferenz> . <Methodenname> ( [<Parameterliste>] )
```

  - gilt auch für Klassenattribute / -methoden
  - Voraussetzung: Sichtbarkeit erlaubt den Zugriff
  - Beispiele:

```
meineKugel.radius      // Attribut
meineKugel.anzahl      // Klassenattribut
meineKugel.volumen()   // Methode
```

- Klassenattribute/-methoden können unabhängig von einer Objektreferenz auch über den Klassennamen angesprochen werden
  - Beispiel: `Kugel.anzahl`

6.2. Zugriff auf Attribute und Methoden

Namen

- Ein ***qualifizierter Name*** ist ein Name mit expliziter Angabe des Objekts oder der Klasse
  - z.B.: `meineKugel.radius`, `Kugel.anzahl`
- Alle andere Namen (ohne Punkt) heißen ***einfache Namen***
  - z.B.: `radius`, `i`, `anzahl`
- Eine Methode kann auf folgende Methoden über einfache Namen zugreifen:
  - die Klassenmethoden der eigenen Klasse (`static`)
  - die Methoden des eigenen Objekts (nur, wenn die aufrufende Methode keine Klassenmethode ist)

6.2. Zugriff auf Attribute und Methoden

Namen

- Eine Methode greift immer über einfache Namen zu auf:
  - ihre Parameter
  - ihre lokalen Variablen

- Beispiel (neue Methode der Klasse Kugel):

```
double volumen() {
  return 4.0/3.0 * PI * radius * radius * radius;
}  // PI: Klassenattribut; radius: Attribut
```

6.2. Zugriff auf Attribute und Methoden

Die Referenzvariable `this`

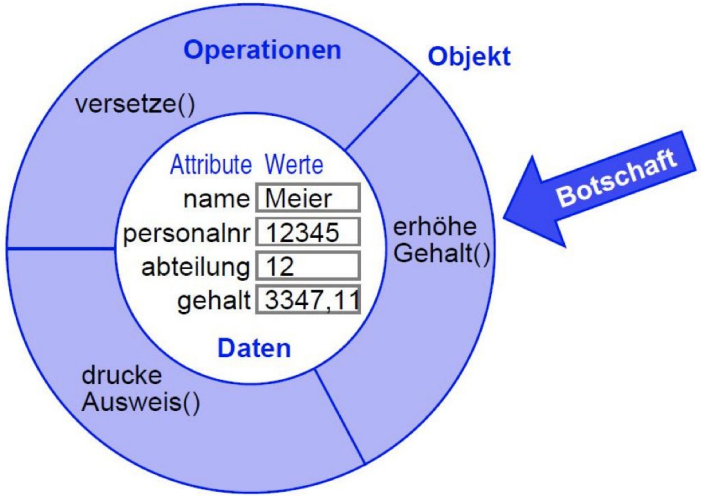
- Existiert in jeder Methode (außer in Klassenmethoden)
- Muss / darf nicht deklariert werden
- Zeigt immer auf das eigene Objekt
  - Vorstellung: `this` ist ein Parameter, in dem der Methode eine Referenz auf das eigene Objekt übergeben wird
- Nützlich z.B. zur Unterscheidung von Attributen und gleichnamigen Parametern:

```
void neuerRadius(double radius)
{
    // radius: Parameter; this.radius: Attribut
    this.radius = radius;
}
```

6.2. Zugriff auf Attribute und Methoden

Sichtbarkeit von Attributen und Methoden

- Java erlaubt die Festlegung, welche Attribute und Operationen einer Klasse ”von Außen“ sichtbar sein sollen, zur Realisierung von Geheimnisprinzip und Datenkapselung
- Wir unterscheiden drei Sichtbarkeiten:
  - **public** (öffentlich): sichtbar für alle Klassen
    - auf Attribut kann von allen Klassen aus zugegriffen werden
    - Operation kann von allen Klassen aufgerufen werden
  - **private** (privat): sichtbar nur innerhalb der Klasse
    - kein Zugriff/Aufruf durch andere Klassen möglich
  - **protected** (geschützt): sichtbar nur innerhalb der Klasse und ihren Unterklassen



6.2. Zugriff auf Attribute und Methoden

Sichtbarkeit von Attributen und Methoden: Get- und Set-Methoden

- In der Implementierung einer Klasse sollten Attribute immer private sein. Wahrung des Geheimnisprinzips: kein direkter Zugriff
- Konvention: Zugriff auf Attribute von außen nur über Get- und Set-Methoden:

```
private String name;                    // Attribut
public String getName();                // Get-Methode
public void setName(String aName);      // Set-Methode
```

- Vorteil: Kapselung der Zugriffe
  - feste Schnittstelle nach außen, unabhängig von konkreter Speicherung bzw. Darstellung der Daten
  - Get- und Set-Methoden können Prüfungen vornehmen

6.2. Zugriff auf Attribute und Methoden

Beispiel

```
class Klasse {
  Klasse1 o1;  // Klasse hat ein Objekt von Klasse1
  Klasse2 o2;  // Klasse hat ein Objekt von Klasse2
  Klasse() {
    // Erzeuge o1 und o2 als Attribute in Klasse, und
    // verlinke o2 als Attribut in o1, so dass o2.a1 == o1.k2.a1:


  }
  public static void main(String[] s) {  // main ist eine Klassenmethode
    // Erzeuge ein Objekt von Klasse:

  }
}
```

6.2. Zugriff auf Attribute und Methoden

Beispiel

```
class Klasse2 {
  public double a1 = 0.0;
  Klasse2(double a1) {
    this.a1 = a1;
    System.out.println("K2");
  }
}
class Klasse1 {
  private String a1 = "hi";
  public Klasse2 k2 = null;     // Klasse1 hat ein Objekt von Klasse2
  Klasse1(Klasse2 k2) {
    this.k2 = k2;
    System.out.println("K1");
  }
}
```

6.2. Zugriff auf Attribute und Methoden

Beispiel: Lösung

```
class Klasse {
  Klasse1 o1;  // Klasse hat ein Objekt von Klasse1
  Klasse2 o2;  // Klasse hat ein Objekt von Klasse2
  Klasse() {
    // Erzeuge o1 und o2 als Attribute in Klasse, und
    // verlinke o2 als Attribut in o1, so dass o2.a1 == o1.k2.a1:
    o2 = new Klasse2( 2.3 );
    o1 = new Klasse1( o2 );
    System.out.println( o1.k2.a1  );
  }
  public static void main(String[] s) {  // main ist eine Klassenmethode
    // Erzeuge ein Objekt von Klasse:
    Klasse k = new Klasse();
  }
}
```

6.2. Zugriff auf Attribute und Methoden

Klassenattributen und Klassenmethoden: `static`

- Im Unterschied zur Klassenmethode muss eine Objektmethode an einem konkret instanziierten Objekt aufgerufen werden, bei Klassenmethoden *nicht*
- Werden statische Variablen von einem Objekt verändert, ist diese Veränderung auch in allen anderen Objekten sichtbar
- z.B. für:
  - Variablen die zur Klasse gehören: `Kugel.anzahl`
  - Konstanten: `public static int MIN_VALUE; , Kugel.PI`
  - Hilfsmethoden: `static double parseDate(String s)`
  - Klassenverwaltung: `static int zaehleKugel() { return anzahl; }`

6.3. Aufruf von Methoden

- Syntax: <Name> ([<Parameterliste>])
  <Parameterliste> ::= <Ausdruck> {, <Ausdruck>}
- Name ist ggf. ein qualifizierter Name

```
class Kugel {
  ...
  private double kubus(double x) {  // x: formaler Parameter
    return x * x * x;
  }
  public double volumen() {
    return 4.0/3.0 * PI * kubus(radius); // kubus(radius):Methodenaufruf
                                         // radius: aktueller Parameter (Argument)
  }
}
```

6.3. Aufruf von Methoden

Ablauf eines Methodenaufrufs

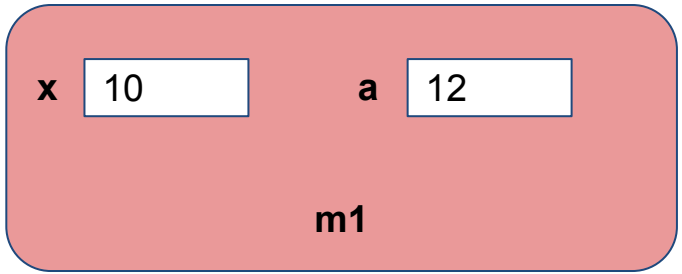
1. Die Ausdrücke in der Parameterliste werden ausgewertet
2. ***call by value***: Die Werte der aktuellen Parameter werden an die formalen Parameter der Methode zugewiesen
3. Der Methodenrumpf wird ausgeführt
   - bis zum Ende (nur bei **`void`**-Methoden erlaubt)
   - oder bis zur Anweisung **`return`** [<Ausdruck>]
     - der Wert von <Ausdruck> bestimmt ggf. den Wert des Methodenaufrufs (der selbst ein Ausdruck ist)
4. Die Abarbeitung der aufrufenden Methode wird nach dem Methodenaufruf fortgesetzt

6.3. Aufruf von Methoden

Ablauf eines Methodenaufrufs

```
int f(int b, int a) {
    a = 2 * b + a * a;
    return a + 1;
}

void m1() {
    int x = 10; int a = 12;
    a = f(a, ++x) - f(a, a + 3);
}
```



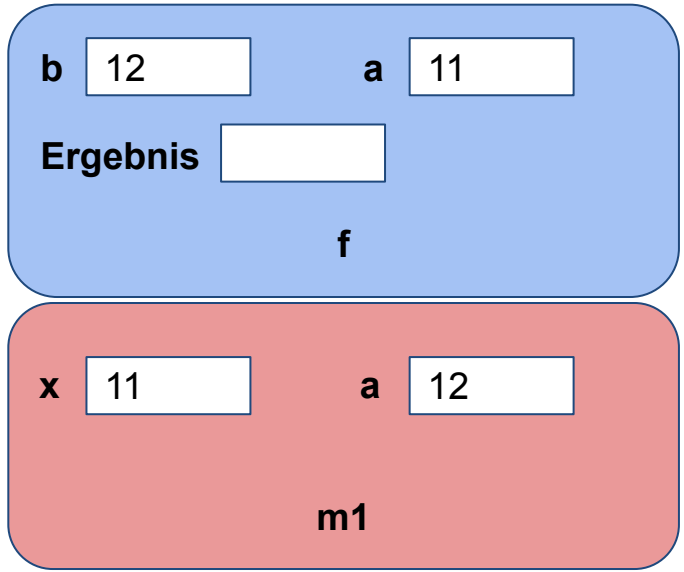
Stapel (Keller, Stack) von Aktivierungsrahmen, die lokale Variable der Methoden speichern

6.3. Aufruf von Methoden

Ablauf eines Methodenaufrufs

```
int f(int b, int a) {
    a = 2 * b + a * a;
    return a + 1;
}

void m1() {
    int x = 10; int a = 12;
    a = f(a, ++x) - f(a, a + 3);
}
```



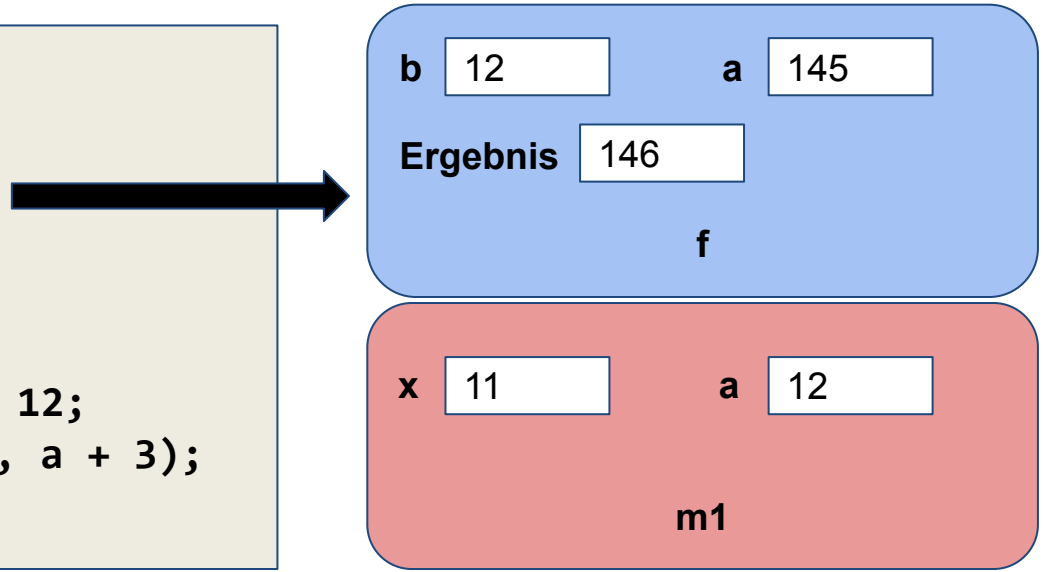
Stapel (Keller, Stack) von Aktivierungsrahmen, die lokale Variable der Methoden speichern

6.3. Aufruf von Methoden

Ablauf eines Methodenaufrufs

```
int f(int b, int a) {
    a = 2 * b + a * a;
    return a + 1;
}

void m1() {
    int x = 10; int a = 12;
    a = f(a, ++x) - f(a, a + 3);
}
```



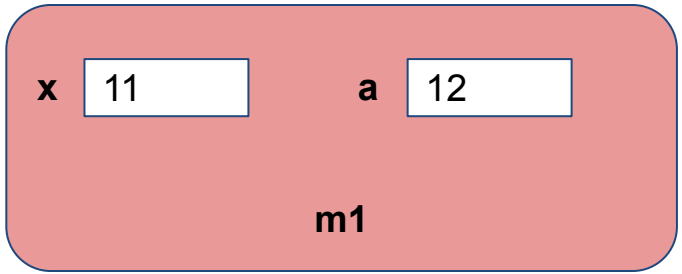
Stapel (Keller, Stack) von Aktivierungsrahmen, die lokale Variable der Methoden speichern

6.3. Aufruf von Methoden

Ablauf eines Methodenaufrufs

```
int f(int b, int a) {
    a = 2 * b + a * a;
    return a + 1;
}

void m1() {
    int x = 10; int a = 12;
    a = f(a, ++x) - f(a, a + 3);
}
```



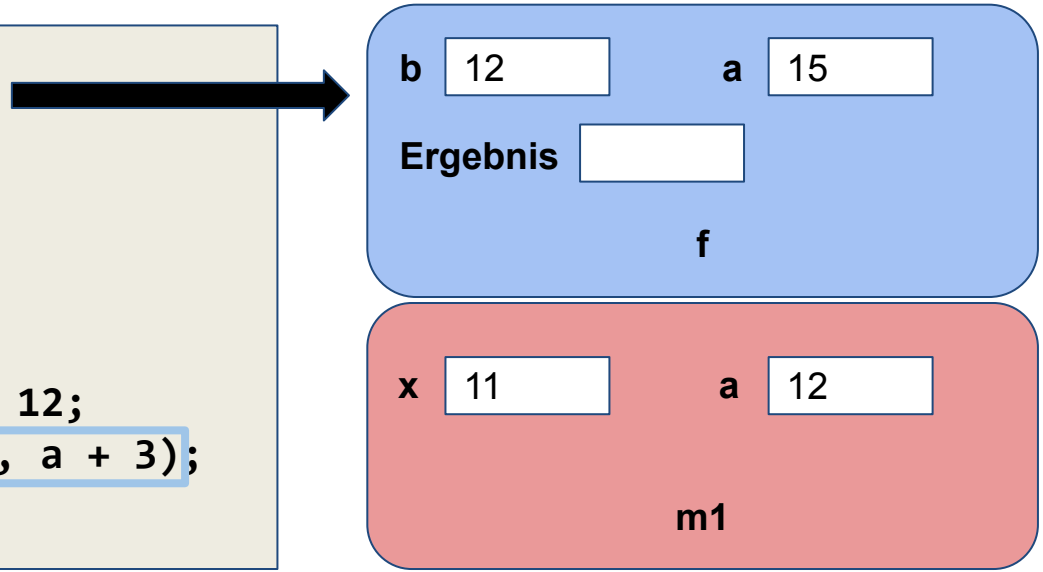
Stapel (Keller, Stack) von Aktivierungsrahmen, die lokale Variable der Methoden speichern

6.3. Aufruf von Methoden

Ablauf eines Methodenaufrufs

```
int f(int b, int a) {
    a = 2 * b + a * a;
    return a + 1;
}

void m1() {
    int x = 10; int a = 12;
    a = f(a, ++x) - f(a, a + 3);
}
```



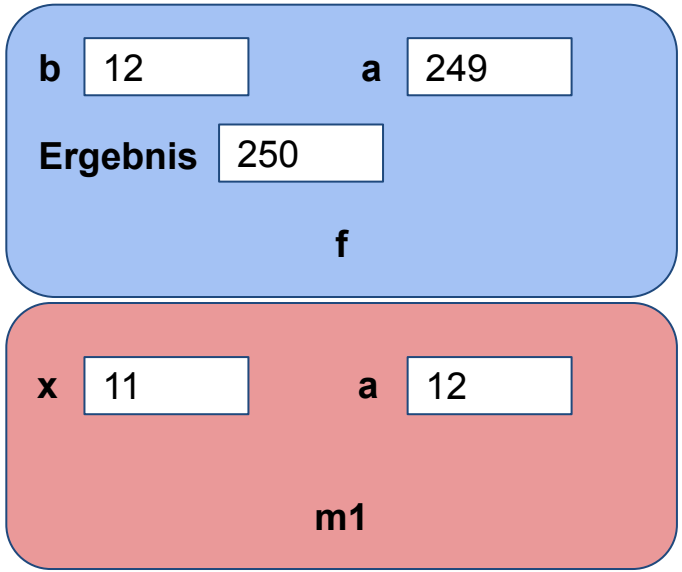
Stapel (Keller, Stack) von Aktivierungsrahmen, die lokale Variable der Methoden speichern

6.3. Aufruf von Methoden

Ablauf eines Methodenaufrufs

```
int f(int b, int a) {
    a = 2 * b + a * a;
    return a + 1;
}

void m1() {
    int x = 10; int a = 12;
    a = f(a, ++x) - f(a, a + 3);
}
```



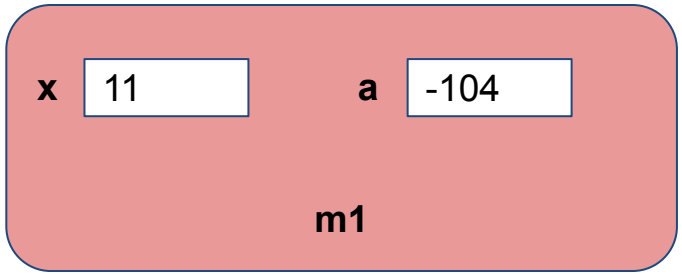
Stapel (Keller, Stack) von Aktivierungsrahmen, die lokale Variable der Methoden speichern

6.3. Aufruf von Methoden

Ablauf eines Methodenaufrufs

```
int f(int b, int a) {
    a = 2 * b + a * a;
    return a + 1;
}

void m1() {
    int x = 10; int a = 12;
    a = f(a, ++x) - f(a, a + 3);
}
```



Stapel (Keller, Stack) von Aktivierungsrahmen, die lokale Variable der Methoden speichern

6.3. Aufruf von Methoden

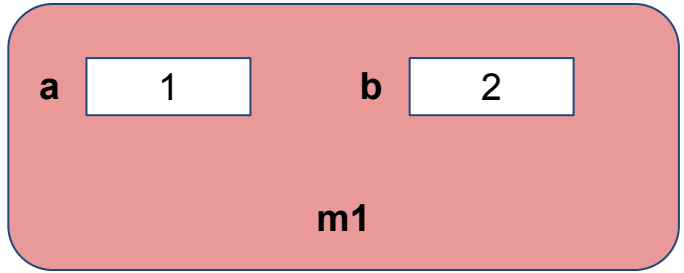
Referenzen als Parameter

- Auch Objekte können Parameter von Methoden sein
  - Übergeben werden dabei jedoch nicht die Objekte, sondern nur Referenzen auf diese Objekte: ***call by reference***
  - Übergabe der Objektreferenz mittels „call by value“ entspricht der Semantik von „call by reference“
- Die aufgerufene Methode kann dabei die referenzierten Objekte verändern
  - so lassen sich auch Ein-/Ausgabe-Parameter realisieren
  - unerwartetes Verändern übergebener Objekte sollte aber vermieden werden
- Analog können Objekt-Referenzen auch als Ergebnis einer Methode auftreten

6.3. Aufruf von Methoden

Referenzen als Parameter

```
class Example1 {
  void swap(int x, int y) {
     int temp = x;
     x = y;
     y = temp;
  }
  void m1() {
     int a, b;
     a = 1; b = 2;
     swap(a, b);  // a = 1, b = 2
  }
}
```



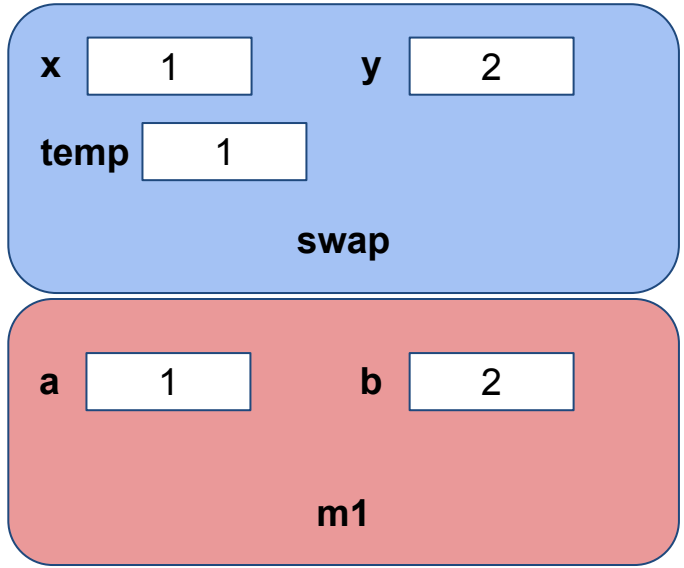
Stapel (Keller, Stack)

6.3. Aufruf von Methoden

Referenzen als Parameter

```
class Example1 {
  void swap(int x, int y) {
    int temp = x;
    x = y;
    y = temp;
  }
  void m1() {
    int a, b;
    a = 1; b = 2;
    swap(a, b);  // a = 1, b = 2
  }
}
```

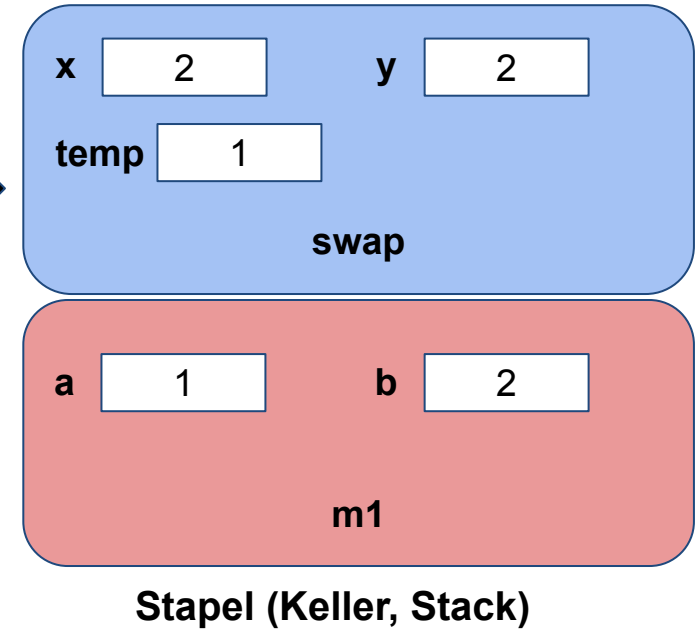
„call by value“: Die Werte der aktuellen Parameter werden an die formalen Parameter der Methode zugewiesen:



6.3. Aufruf von Methoden

Referenzen als Parameter

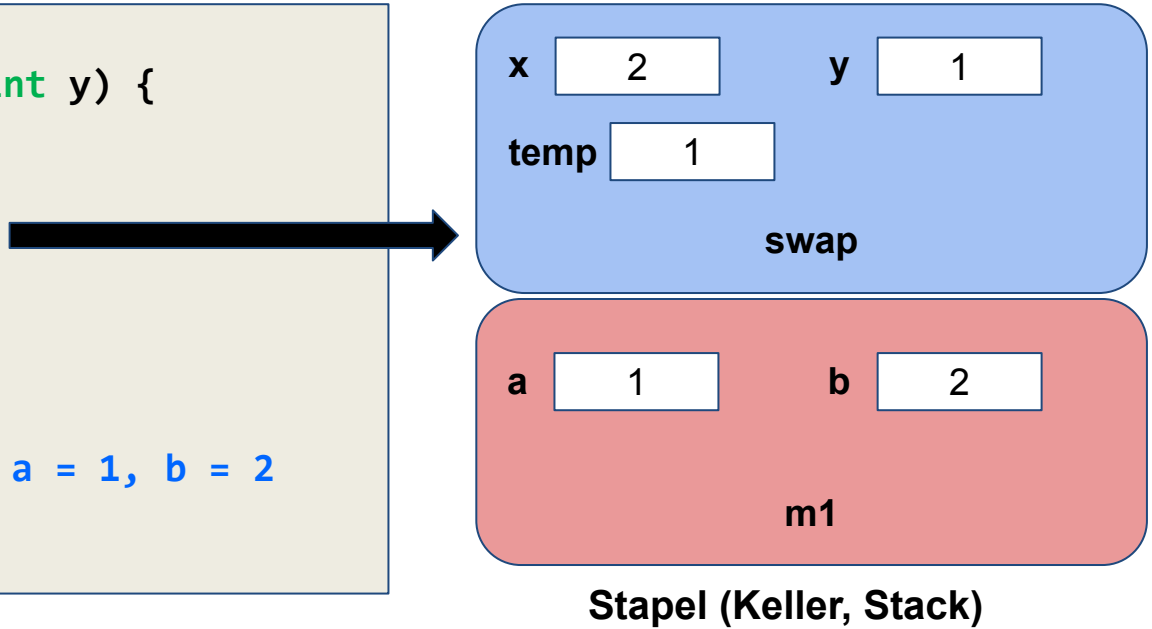
```
class Example1 {
  void swap(int x, int y) {
    int temp = x;
    x = y;
    y = temp;
  }
  void m1() {
    int a, b;
    a = 1; b = 2;
    swap(a, b);  // a = 1, b = 2
  }
}
```



6.3. Aufruf von Methoden

Referenzen als Parameter

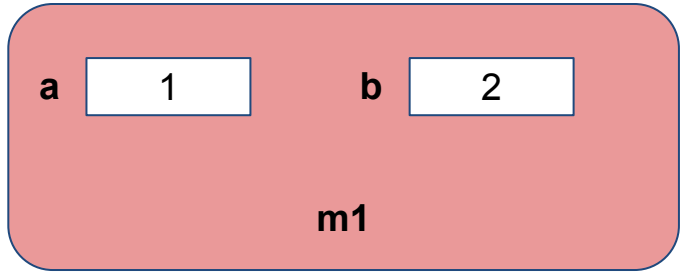
```
class Example1 {
  void swap(int x, int y) {
    int temp = x;
    x = y;
    y = temp;
  }
  void m1() {
    int a, b;
    a = 1; b = 2;
    swap(a, b);  // a = 1, b = 2
  }
}
```



6.3. Aufruf von Methoden

Referenzen als Parameter

```
class Example1 {
  void swap(int x, int y) {
     int temp = x;
     x = y;
     y = temp;
  }
  void m1() {
     int a, b;
     a = 1; b = 2;
     swap(a, b);  // a = 1, b = 2
  }
}
```



Stapel (Keller, Stack)

6.3. Aufruf von Methoden

Referenzen als Parameter

```
class Example1 {
  void swap(int x, int y) {
    int temp = x;
    x = y;
    y = temp;
  }
  void m1() {
    int a, b;
    a = 1; b = 2;
    swap(a, b);  // a = 1, b = 2
  }
}
```

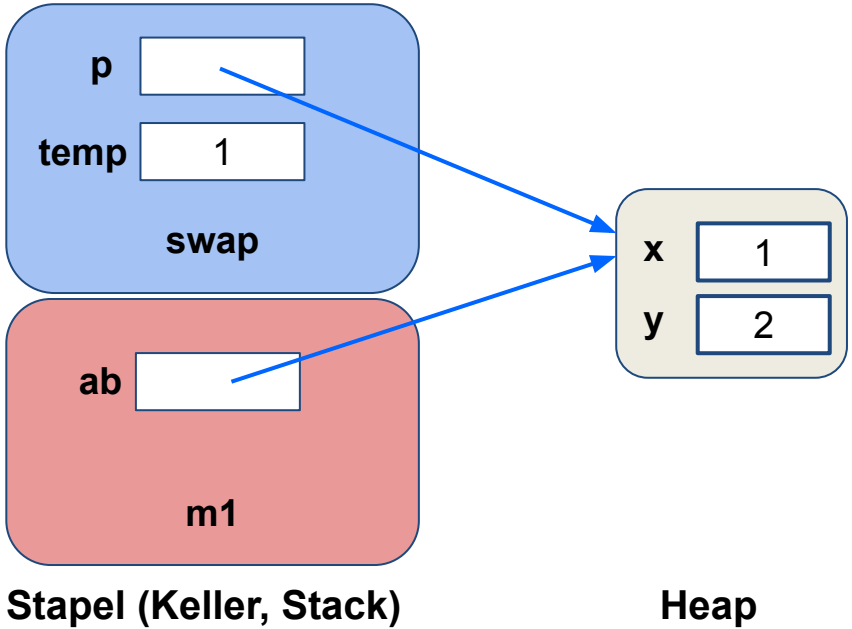
```
class Pair {
  public int x, y;
}
class Example2 {
  void swap(Pair p) {
    int temp = p.x;
    p.x = p.y;
    p.y = temp;
  }
  void m1() {
    Pair ab = new Pair();
    ab.x = 1; ab.y = 2;
    swap(ab);  // ab.x=2, ab.y=1
  }
}
```

6.3. Aufruf von Methoden

Referenzen als Parameter

```
void swap(Pair p) {
  int temp = p.x;
  p.x = p.y;
  p.y = temp;
}

void m1() {
  Pair ab = new Pair();
  ab.x = 1; ab.y = 2;
  swap(ab);  // ab.x=2, ab.y=1
}
```

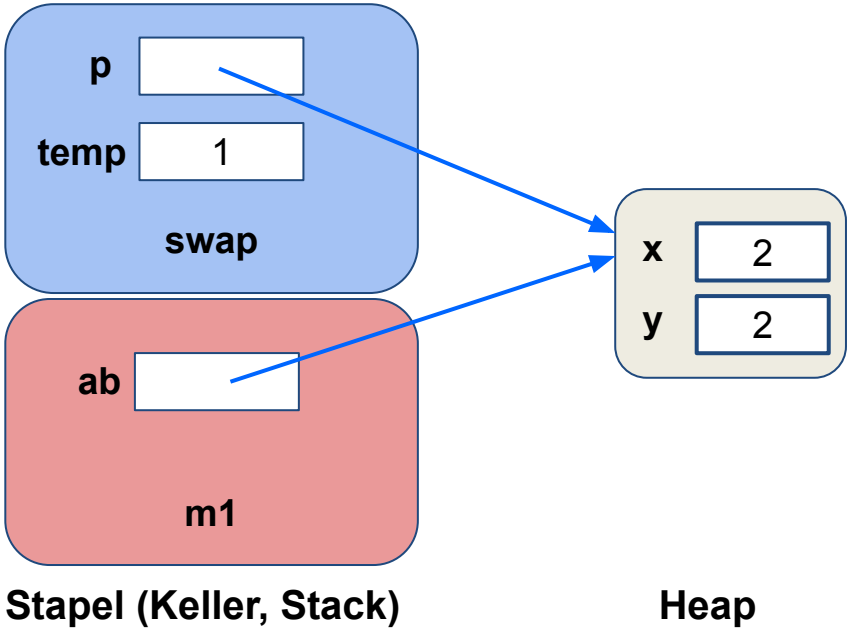


6.3. Aufruf von Methoden

Referenzen als Parameter

```
void swap(Pair p) {
  int temp = p.x;
  p.x = p.y;
  p.y = temp;
}

void m1() {
  Pair ab = new Pair();
  ab.x = 1; ab.y = 2;
  swap(ab);  // ab.x=2, ab.y=1
}
```

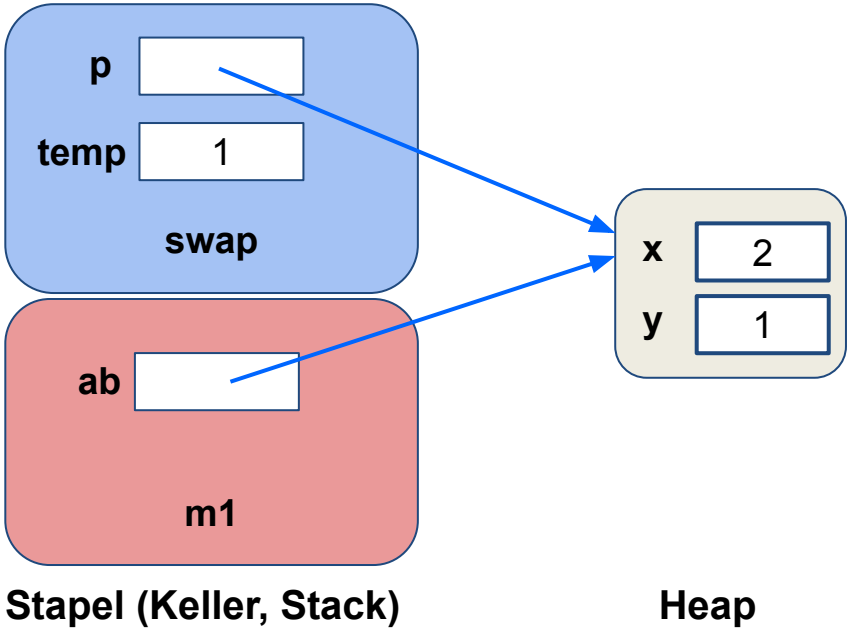


6.3. Aufruf von Methoden

Referenzen als Parameter

```
void swap(Pair p) {
  int temp = p.x;
  p.x = p.y;
  p.y = temp;
}

void m1() {
  Pair ab = new Pair();
  ab.x = 1; ab.y = 2;
  swap(ab);  // ab.x=2, ab.y=1
}
```



6.3. Aufruf von Methoden - Beispiel

```
/**  WordleClass: Implementieren Sie das einfache Wordle-Spiel jetzt als Klasse:
  */
import java.util.Scanner;  // scan strings in der Konsole

class WordleClass {
  /*...*/   // Hier kommen die Attribute

  private String dialog( String hint ) { /*...*/ }  // private Methode
  public void run() { /*...*/ }  // public Methode

  public static void main( String[] str ) {  // main ist eine Klassenmethode
    WordleClass w = new WordleClass();
    w.run();
  }
}
```

6.3. Aufruf von Methoden

Überladen von Methoden

Bei überladenen Methoden wird die zu Anzahl und Typen der aktuellen Parameter passende Methode bereits vom Compiler ausgewählt

```
class Drucker {
      static void drucke(int Zahl) {
            System.out.println(Zahl);
      }
      static void drucke(String Name) {
            System.out.println(Name);
      }
}
// …
Drucker.drucke(10);
```

Unzulässig:

```
class KursVerwaltung {
      // Suche Kurs, Ergebnis: Kursnr
       int suche(String stichwort) {
            // …
       }
      // Suche Kurs, Ergebnis: Titel
       String suche(String stichwort) {
            // …
       }
}
```

6.3. Aufruf von Methoden

Rekursion

- Eine Methode kann sich auch selbst aufrufen
- Beispiel: Fakultätsfunktion

```
static long fak(long n) {
  if (n == 0)
    return 1L;
  else if (n > 0)
    return n * fak(n-1);
}
```

Mathematische Definition:

$$n! = \begin{cases} 1 & \text{für } n = 0 \\ n \cdot (n-1)! & \text{für } n > 0 \end{cases}$$

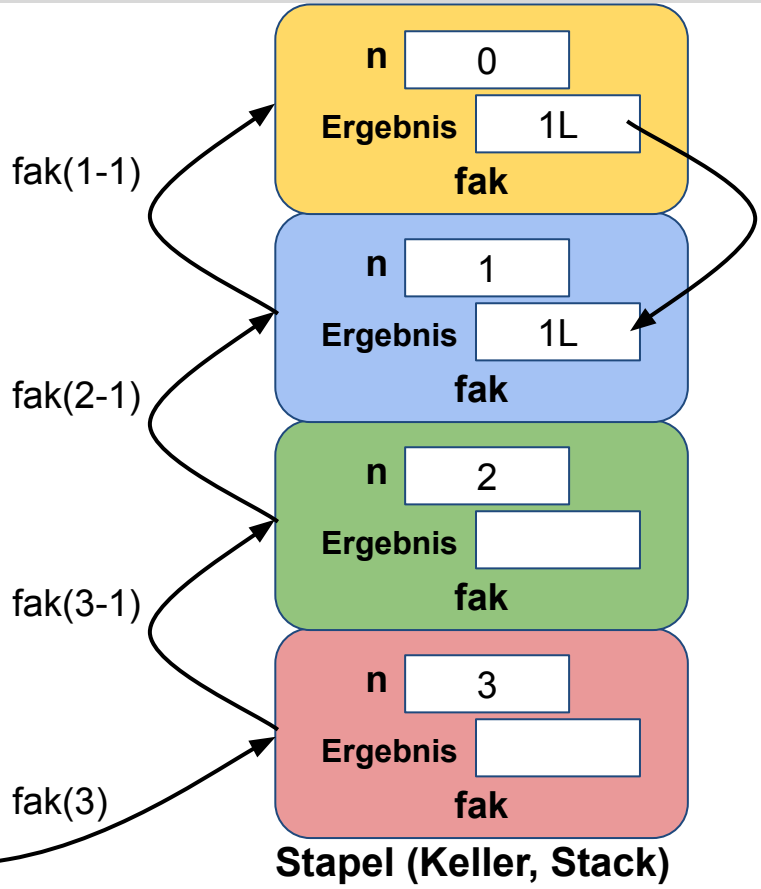
- Jede Aktivierung einer Methode bekommt einen eigenen Aktivierungs- rahmen auf dem Stapel, damit hat jede Aktivierung ihre eigenen lokalen Variablen

6.3. Aufruf von Methoden

Rekursion

- Jede Aktivierung einer Methode bekommt einen eigenen Aktivierungs-rahmen auf dem Stapel, damit hat jede Aktivierung ihre eigenen lokalen Variablen

```
static long fak(long n) {
  if (n == 0)
    return 1L;
  else if (n > 0)
    return n * fak(n-1);
}
long f = fak(3);
```

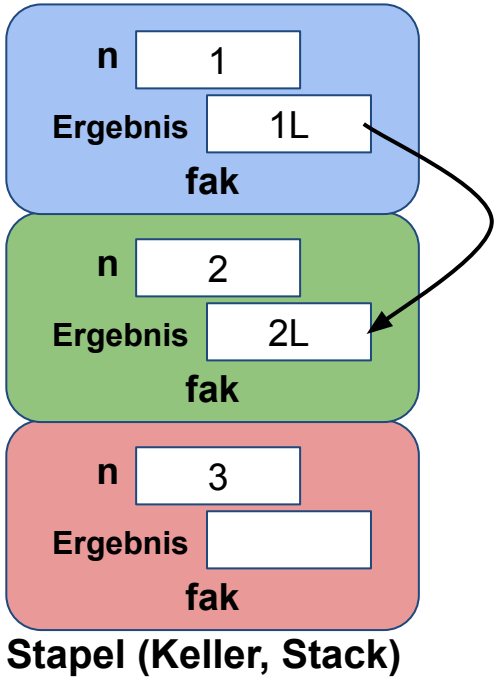


6.3. Aufruf von Methoden

Rekursion

- Jede Aktivierung einer Methode bekommt einen eigenen Aktivierungs- rahmen auf dem Stapel, damit hat jede Aktivierung ihre eigenen lokalen Variablen

```
static long fak(long n) {
  if (n == 0)
    return 1L;
  else if (n > 0)
    return n * fak(n-1);
}
long f = fak(3);
```

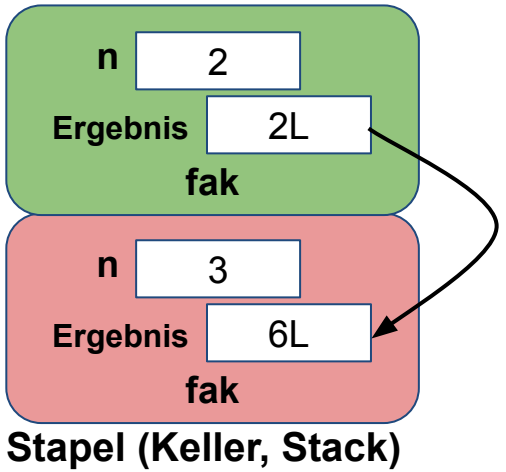


6.3. Aufruf von Methoden

Rekursion

- Jede Aktivierung einer Methode bekommt einen eigenen Aktivierungs- rahmen auf dem Stapel, damit hat jede Aktivierung ihre eigenen lokalen Variablen

```
static long fak(long n) {
  if (n == 0)
    return 1L;
  else if (n > 0)
    return n * fak(n-1);
}
long f = fak(3);
```



6.4. Arrays von Objekten

- Wie in einfachen Variablen werden in Arrays nur Objektreferenzen gespeichert
- Beispiel: Array mit 2 Studenten

```
Student[] a1 = new Student[2];  // Initialisierung mit null
a1[0] = new Student("Hans");    // Initialisierung der Elemente
a1[1] = new Student("Fritz");
```

- Kürzer:

```
// Deklaration mit Initialisierung der Elemente:
Student[] a2 = { new Student("Hans"),  new Student("Fritz") };
```

6.4. Arrays von Objekten - Beispiel: SnailGame v1 (github)

```
class Game {
  private Snail player;
  private Snail[] enemy;
  private boolean bang = false;

  public Game() { /* … */ }
  public void keyPressed(KeyEvent e) {…}
  public void keyReleased(KeyEvent e) {}
  public void keyTyped(KeyEvent e) {}
  public void paintComponent( Graphics g ) {…}
  public static void main(String[] a) {…}
}
```

```
class Snail {
  private short len = 7;
  private int[] xPos = null;
  private int[] yPos = null;

  Snail(short len) {/* … */}
  Snail(short len, int x, int y) {…}
  boolean isOverlapped( Snail snail ) {…}
  void moveBody() {/* … */}
  void moveHead(int x, int y) {/* … */}
  int getLength() {/* … */}
  int getPosX(int i) {/* … */}
  int getPosY(int i) {/* … */}
}
```

6.5. Einführung in UML

- UML = Unified Modelling Language, Entwicklung seit 1994
- standardisierte (graphische) Sprache zur objektorientierten Modellierung von (Software-)Systemen
- UML definiert eine Vielzahl von Diagrammtypen: unterschiedliche Sichtweisen des modellierten Systems
  - statische vs. dynamische Aspekte
  - unterschiedlicher Abstraktionsgrad
- UML unterstützt sowohl Objekt-Orientierte Analyse (OOA) als auch Objekt-Orientiertes Design (OOD)

6.5. Einführung in UML: Klassen, Wiederholung aus Java:

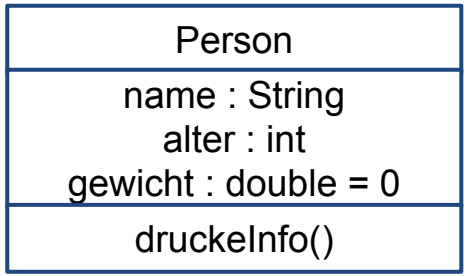
- Eine Klasse definiert für eine Kollektion gleichartiger Objekte
  - deren Struktur, d.h. die **Attribute** (nicht die Werte)
  - das Verhalten, d.h. die **Operationen**
  - die möglichen Beziehungen (**Assoziationen**) zu anderen Objekten, einschließlich der Generalisierungs-Beziehung
- Eine Klasse besitzt einen Mechanismus, um neue Objekte zu erzeugen

  Die Klasse ist der ”Bauplan“ für diese Objekte

6.5. Einführung in UML: Klassen

- Anforderungserhebung, Analyse
  Entwickler: „Was ist euch wichtig?“
  Anwender: „Der Kunde.“
  Entwickler: „Was ist denn ein Kunde, welche Merkmale sind für euch relevant?“
  Anwender: „Der Kunde hat einen Namen, eine Anschrift und eine Bonität, die wir überprüfen.“

- Design (Klasse in UML):



Implementierung (Klasse in Java):

```java
class Person {
  String name;
  int alter;
  double gewicht = 0;
  public void druckeInfo() {/*...*/}
}
```

6.5. Einführung in UML: Objekte

- Allgemeines Schema:

<Objektname>:<Klassenname>
<Attributname 1> = <Wert 1> … <Attributname n> = <Wert n>

- z.B.:

myVideo : Video
author = "R.A." length = 3.32 ytlink = "dQw4w9WgXcQ"

- Objekt− und Attributnamen klein geschrieben, Klassennamen groß geschrieben
- Die Operationen werden hier nicht angegeben, da sie für alle Objekte einer Klasse identisch sind

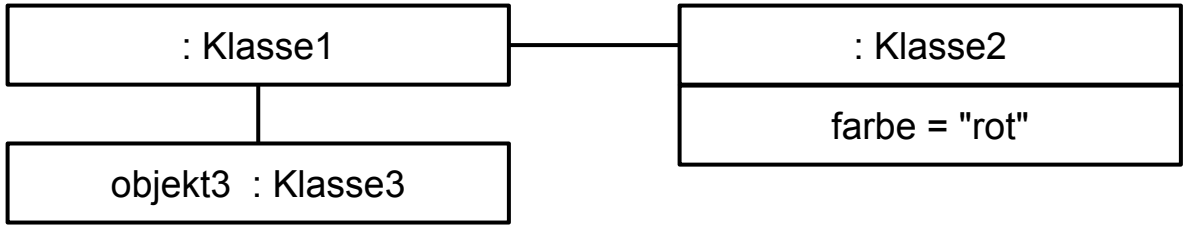
```
class Video {
  String author;
  double length = 0.0;
  String ytlink;
  /*...*/
}
Video rr;
rr = new Video("R.A.", 3.32, "dQw4w9WgXcQ");
```

6.5. Einführung in UML: Objekte

- Darstellungsvarianten:

  - Objekt ohne Klasse: Klasse geht aus Zusammenhang hervor
  - Anonymes Objekt

- Objektdiagramm mit Objektbeziehungen:
  Beziehungen werden durch Verbindungslinien zwischen Objekten dargestellt



6.5. Einführung in UML: Objekte

Identität und Gleichheit von Objekten

- Ein Objekt besitzt einen Zustand, ein wohldefiniertes Verhalten und eine Identität, die es von allen anderen Objekten unterscheidet
- Zwei Objekte sind gleich, wenn sie dieselben Attributwerte besitzen: