

# Grundzüge der Informatik

## V9. Die Programmiersprache Java (Teil 3)

Prof. Dr. Mira Mezini  
FG Softwaretechnik  
TU-Darmstadt



V6 - 2

## Agenda

- Fünf Wege, über die ein Objekt Zugriff auf Daten bekommt
  - Attribute
  - Lokale Variablen
    - Rückgabewerte von Operationsaufrufen
    - Neu erzeugte Objekte
  - Parameter der Operationen
  - Sichtbarkeitsbereiche und -Dauer von lokalen Variablen und Attributen
  - Klassenattribute und -Operationen
    - Statische Initialisierungsblöcke



V6 - 3

## Sichtbare Daten eines Objektes

- Fünf Wege, über die ein Objekt **b** vom Typ **B** Zugriff auf ein Datum **a** vom Typ **A** bekommt
  - ① **a** ist der Wert eines Attributs von **b**, bzw. ein Referenzattribut von **b** verweist auf **a**
  - ② **a**, bzw. ein Referenz auf **a**, wird als aktuelles Parameterwert einer Operation übergeben, die an **b** vorgenommen wird
  - ③ **a** wird innerhalb der Implementierung einer Operation erzeugt, die an **b** vorgenommen wird
  - ④ **a** wird als Wert eines Operationsaufrufs innerhalb der Implementierung einer Operation zurückgegeben, die an **b** vorgenommen wird
  - ⑤ **a** ist der Wert eines Klassenattributs bzw. ein Klassenattribut verweist auf **a**

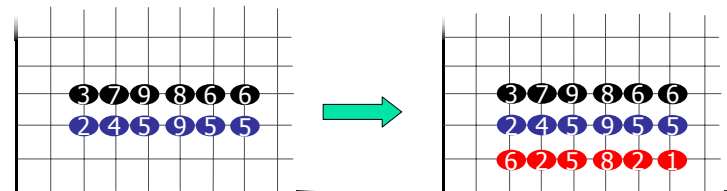


V6 - 4

## Sichtbare Daten eines Objektes

### Beispiel für die ersten vier Fälle

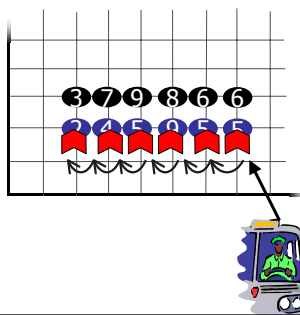
- Es sollen Roboter entwickelt werden, die zwei Zahlen addieren
- Die Zahlen sind mithilfe von Piepsern in den Ecken der 2. und 3. Straße zwischen einer beliebigen Avenue X und der Avenue 2 kodiert
- Das Resultat soll in der 1. Straße zwischen der Avenue X und der Avenue 1 kodiert werden



## Sichtbare Daten eines Objektes

### Beispiel für die ersten vier Fälle

- Wir benutzen einen **Adder**-Roboter für jede Avenue
- Der **Adder** für die Avenue **x** befindet sich ursprünglich nordwärts in der Kreuzung (2, X)

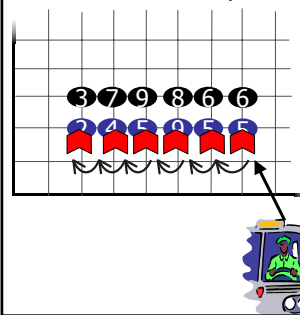


- Jeder **Adder** ist zuständig für die Addition der Ziffern in seiner Avenue
- Jeder **Adder**, außer dem am weitesten links, kennt (erzeugt) seinen linken Nachbarn
- Der Pilot kennt nur den ersten **Adder**-Roboter rechts

## Sichtbare Daten eines Objektes

### Beispiel für die ersten vier Fälle

- Pilot beauftragt den ersten **Adder** rechts, die Ziffer in seiner Avenue zu addieren und dann **rekursiv die Rolle des Piloten für die restlichen Ziffern links zu übernehmen**, d.h.



- Jeder **Adder**, außer dem am weitesten links, beauftragt seinen linken Nachbarn:
  - die Ziffer in seiner Avenue zu addieren und
  - dann die Rolle des Piloten für die restlichen Ziffern links zu übernehmen

## Sichtbare Daten eines Objektes

### Beispiel für die ersten vier Fälle

**Addieren einer Avenue** → add(): Carrier

- Adder** sind faule Wesen: Sie zählen nur die Piepser in den beiden Ecken der eigenen Avenue
  - `cornerCount(): int` → **Adder** zählt die Piepser in der Ecke, in der er sich befindet
  - `addCiphers(): int` → **Adder** bewegt sich vorwärts und ruft `cornerCount()` in jede Ecke auf

```

Adder
add(): Carrier
addCiphers(): int
cornerCount(): int
next(): void
start(): void
    
```

## Sichtbare Daten eines Objektes

### Beispiel für die ersten vier Fälle

**Addieren einer Avenue** → add(): Carrier

- Das Resultat der Zählerei lassen **Adder** von **ResultWriter**-Roboter in der ersten Straße setzen
  - Jeder **Adder** hat ein `helfer`-Attribut vom Typ **ResultWriter**, (`helfer` verweist auf ein **ResultWriter**-Roboter)

- Rückgabewert von `add()` ist ein **Carrier**-Roboter:
  - ist in der Lage, den Überlauf zu übertragen, wenn es einen gibt
  - wird innerhalb von `add()` erzeugt

```

Adder
add(): Carrier
addCiphers(): int
cornerCount(): int
next(): void
start(): void
    
```

# Sichtbare Daten eines Objektes

Beispiel für die ersten vier Fälle

## Vorbereitung und Durchführung einer Addition

→ next()

- erzeugt einen linken **Adder**-Nachbarn und übergibt ihm dabei den eigenen Helfer
- verlangt vom erzeugten Nachbarn, die Addition in seiner Avenue durchzuführen (ruft **add()** an den Nachbar auf)
- Bekommt einen **Carrier** zurück und verlangt von ihm seinen Job zu machen (ruft **carry()** an den **Carrier** auf)
- Verlangt die Vorbereitung der nächsten Addition vom Nachbar (ruft **next()** an dem Nachbar auf)

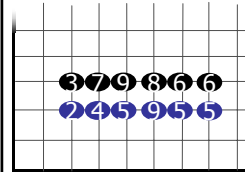
Adder
add(): Carrier
addCiphers(): int
cornerCount(): int
next():void
start(): void

# Sichtbare Daten eines Objektes

Beispiel für die ersten vier Fälle

## Starten des Additionsprozesses

→ start()



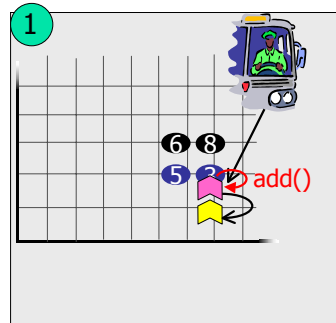
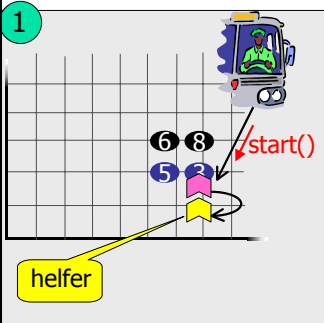
```
task {
    Adder adder = new Adder(7);
    adder.start();
    adder.turnOff();
}
```

```
public void start() {
    Carrier carrier = add();
    carrier.carry();
    next();
}
```

Adder
add(): Carrier
addCiphers(): int
cornerCount(): int
next():void
start(): void

# Sichtbare Daten eines Objektes

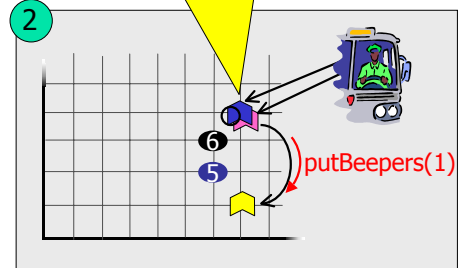
Beispiel für die ersten vier Fälle



# Sichtbare Daten eines Objektes

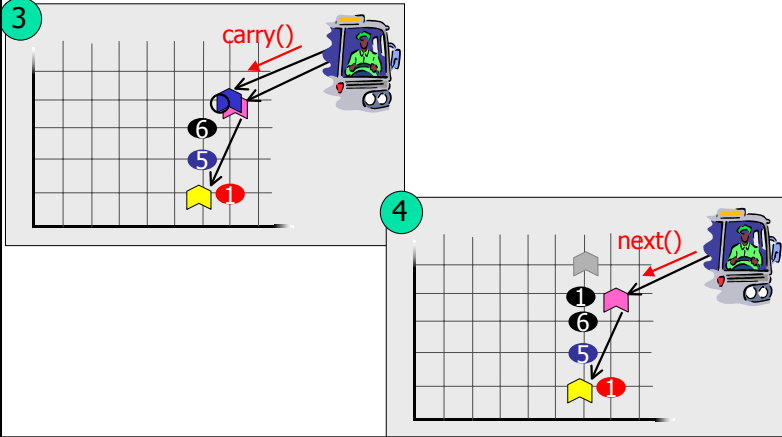
Beispiel für die ersten vier Fälle

Carrier, mit einem Piepser zu übertragen



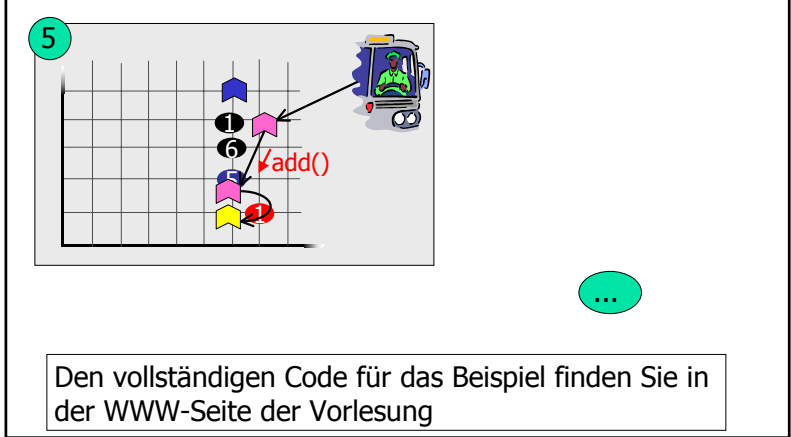
# Sichtbare Daten eines Objektes

Beispiel für die ersten vier Fälle



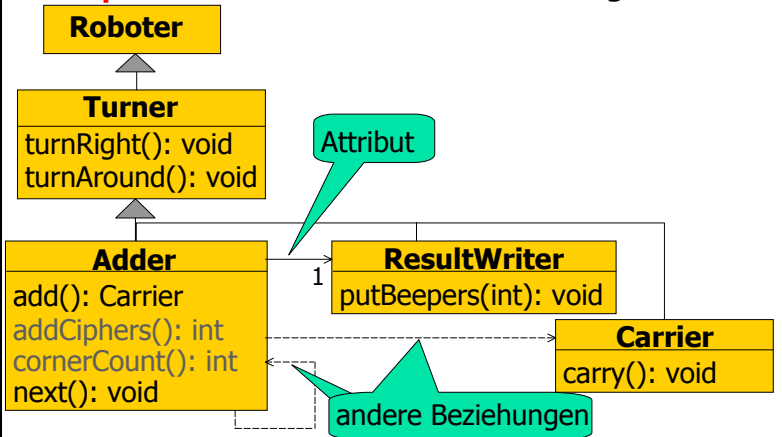
# Sichtbare Daten eines Objektes

Beispiel für die ersten vier Fälle



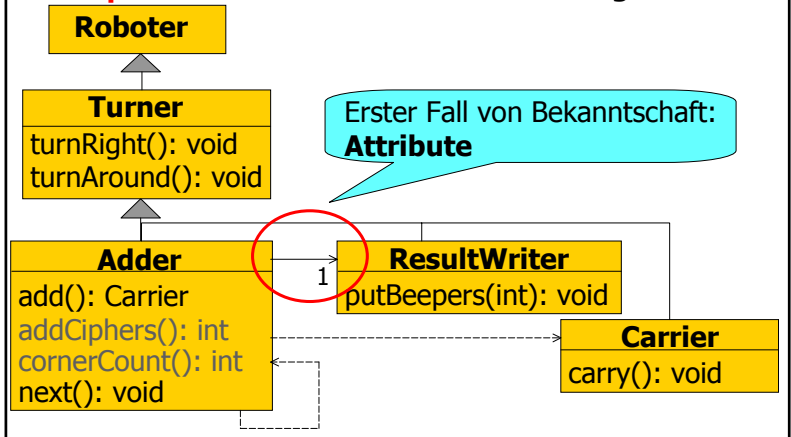
# Sichtbare Daten eines Objektes

Beispiel für die ersten vier Fälle Bisheriger Entwurf



# Sichtbare Daten eines Objektes

Beispiel für die ersten vier Fälle Bisheriger Entwurf



## Sichtbare Daten eines Objektes

### Attribute

- Werden im Klassenrumpf deklariert
  - alle Exemplare einer Klasse haben die gleiche Menge und Typen der Attribute
- Jedes Exemplar (Instanz) einer Klasse speichert die eigenen Werte der Attribute.
  - Deswegen werden Attribute im Englischen auch *Instance Variables* genannt.
- Drei Möglichkeiten der Initialisierung:
  - Direkt bei der Deklaration des Attributs
  - Innerhalb eines Konstruktors
  - Default-Werte → **null** für Referenztyp-Attribute
    - Soll vermieden werden

## Sichtbare Daten eines Objektes

### Attribute

```
class Adder extends Turner {
    private ResultWriter helper =
        new ResultWriter(1, ???);
}
```

Deklaration und direkte Initialisierung.

Nicht immer sinnvoll, da die Initialisierungswerte für die Referenztyp-Attribute eines Objektes von den Initialisierungswerten des Objektes selbst abhängig sein könnten.

In diesem Fall soll z.B. der Helfer ursprünglich in der gleichen Avenue gestellt werden, wie der **Adder**-Objekt, dem er dienen soll

## Sichtbare Daten eines Objektes

### Attribute

```
class Adder extends Turner {
    private ResultWriter helper;
    public Adder(int avenue) {
        super(2, avenue, North, 0);
        helper = new ResultWriter(1, avenue);
    }
}
```

Deklaration

Gleiche Werte

Initialisierung im Konstruktor.  
Initialisierungswert im Konstruktor erzeugt

## Sichtbare Daten eines Objektes

### Attribute

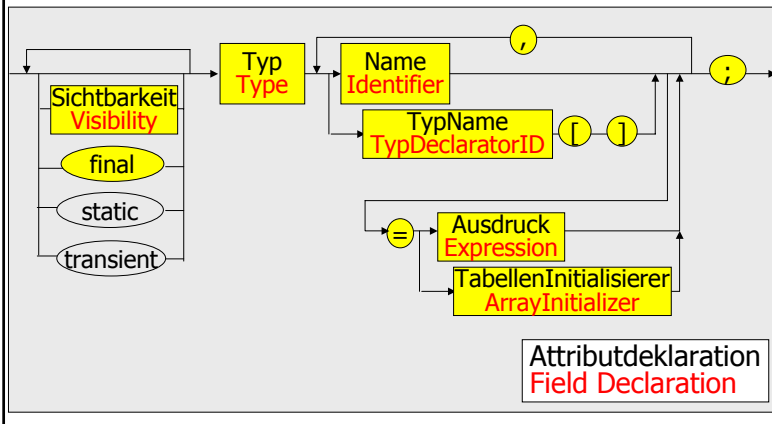
```
class Adder extends Turner {
    private ResultWriter helper;
    public Adder(int avenue) {
        super(2, avenue, North, 0);
        helper = new ResultWriter(1, avenue);
    }
    public Adder(int avenue, ResultWriter h) {
        super(2, avenue, North, 0);
        helper = h;
    }
}
```

Deklaration

Initialisierung im Konstruktor.  
Initialisierungswert als Parameter übergeben

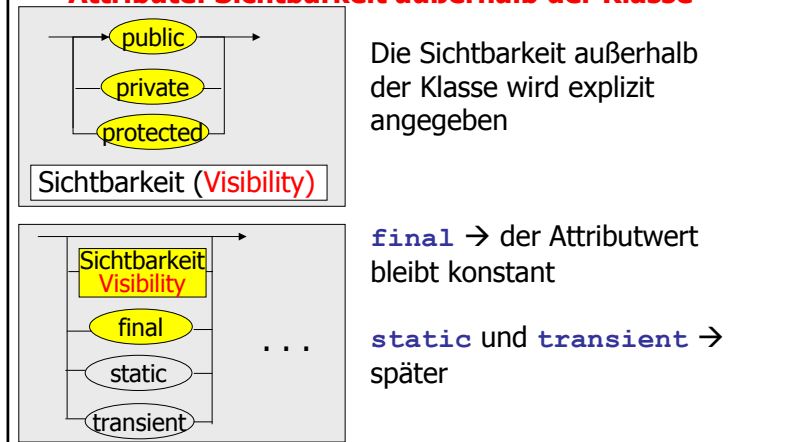
## Sichtbare Daten eines Objektes

### Struktur der Attributdeklaration



## Sichtbare Daten eines Objektes

### Attribute: Sichtbarkeit außerhalb der Klasse



## Sichtbare Daten eines Objektes

### Attribute: Sichtbarkeit innerhalb der Klasse

Attribute sind innerhalb der Implementierung aller Operationen der gleichen Klasse sichtbar

```

public Carrier add() {
    int c = addCiphers();
    if (c > 9) {
        helper.putBeepers(c - 10);
        return new Carrier(..., 1);
    } else {
        helper.putBeepers(c);
        return new Carrier(..., 0);
    }
}

```

**class Adder**

Ein **Adder** nimmt Services der ihm als Attribute bekannten Objekte in Anspruch

## Sichtbare Daten eines Objektes

### Attribute: Sichtbarkeit innerhalb der Klasse

Attribute sind innerhalb der Implementierung aller Operationen der gleichen Klasse sichtbar

```

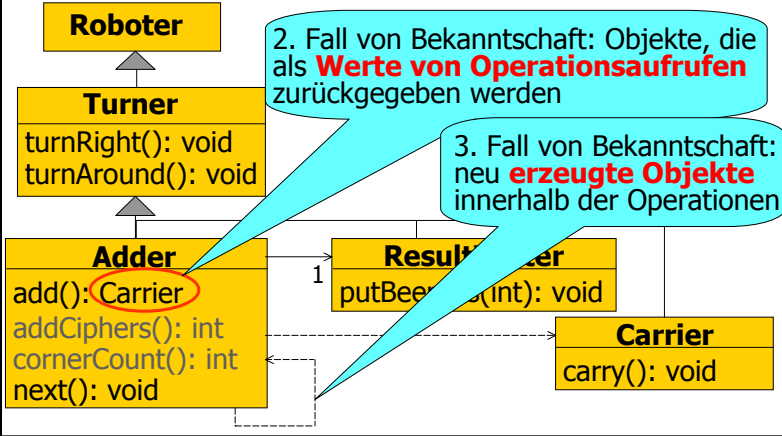
public void next() {
    turnLeft();
    if (frontIsClear()) {
        Adder neighbor =
            new Adder(avenue() - 1, helper);
        helper = null; // don't need anymore
        ...
    } else { ... }
}

```

**class Adder**

# Sichtbare Daten eines Objektes

Beispiel für die ersten vier Fälle Bisheriger Entwurf



# Sichtbare Daten eines Objektes

Rückgabewerte und neu erzeugte Objekte

```

public void next() {
    turnLeft();
    if (frontIsClear()) {
        Adder neighbor =
            new Adder(avenue()-1, helper);
        Carrier carrier = neighbor.add();
        carrier.carry();
        neighbor.next();
    }
    turnOff();
}
    
```

Ein Objekt, das als Wert eines Operationsaufrufs zurückgeliefert wird, ist ein Bekannter, dessen Services in Anspruch genommen werden können

# Sichtbare Daten eines Objektes

Rückgabewerte und neu erzeugte Objekte

```

public void next() {
    turnLeft();
    if (frontIsClear()) {
        Adder neighbor =
            new Adder(avenue()-1, helper);
        Carrier carrier = neighbor.add();
        carrier.carry();
        neighbor.next();
    }
    turnOff();
}
    
```

Ein in einer Methode neu erzeugtes Objekt ist ein Bekannter, dessen Services in Anspruch genommen werden können

# Sichtbare Daten eines Objektes

Rückgabewerte und neu erzeugte Objekte

- Neu erzeugte Objekte und Rückgabewerte der Operationsaufrufe werden innerhalb einer Methode in sogenannten **lokale Variablen** gespeichert
- Bezeichner, die innerhalb der Implementierung einer Operation deklariert und benutzt werden, um auf Zwischenergebnisse zu verweisen.
- neu erzeugte Objekte und Objekte, die als Werte von Operationsaufrufen zurückgeliefert werden, sind Zwischenergebnisse einer Methode

## Sichtbare Daten eines Objektes

### Deklaration und Verwendung lokaler Variablen

Deklaration einer lokalen Variable

class Adder

```
private int addCiphers() {
    int result;
    result = cornerCount();
    move();
    result += cornerCount();
    move();
    return result + cornerCount();
}
```

Lese-Zugriff auf die lokalen Variablen

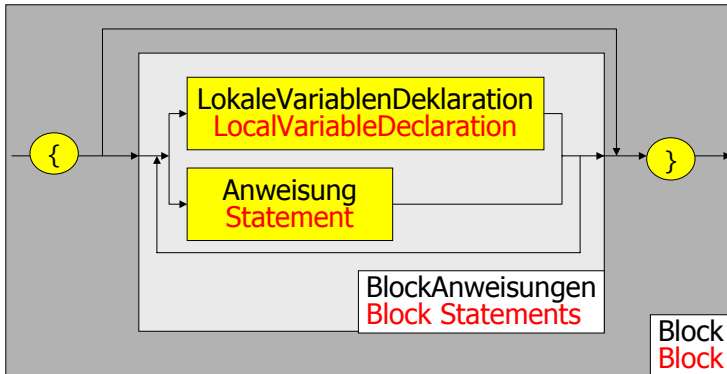
Initialisierung

Lese- und Schreibzugriff auf die lokalen Variablen

## Sichtbare Daten eines Objektes

### Definition eines Blocks

Der prinzipielle Aufbau eines Blocks ist in dem folgenden Syntax-Diagramm definiert



## Sichtbare Daten eines Objektes

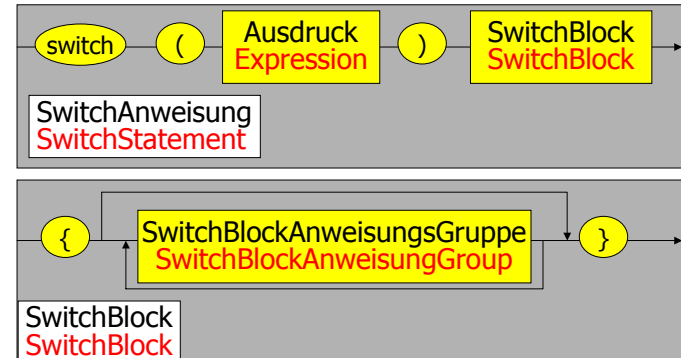
### Rückgabewerte und neu erzeugte Objekte

- Lokale Variablen werden nicht nur innerhalb der Methoden, sondern i.a. **innerhalb eines beliebigen Blocks** verwendet
- Außer Klassen-, Operation- und Konstruktor-Rümpfen haben wir Blöcke bisher auch als Teile der Definition von Kontrollanweisungen gesehen:
  - **if (Bedingung) then-Block else else-Block**
  - **while (Bedingung) while-Block**
- Wir werden bald neue Blöcke kennen lernen

## Sichtbare Daten eines Objektes

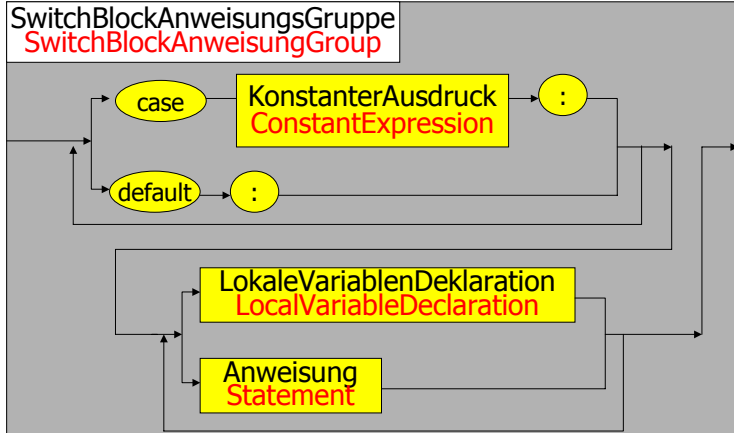
### Einschub: Blöcke in der switch-Anweisung

Die **switch**-Anweisung kann als verallgemeinerte Form der Programmverzweigung aufgefasst werden



## Sichtbare Daten eines Objektes

### Einschub: Blöcke in der `switch`-Anweisung



## Sichtbare Daten eines Objektes

### Einschub: Blöcke in der `switch`-Anweisung

```

public class SwitchAnweisung1 {
    public static void main (String [] args) {
        int farbeingabe = 2;
        switch ( farbeingabe ) {
            case 0: System.out.println("schwarz");
            case 1: System.out.println("rot");
            case 2: System.out.println("gelb");
            default: System.out.println
                ("Farbe nicht identifizierbar");
        }
    }
}

```

Beispiel

## Sichtbare Daten eines Objektes

### Einschub: Blöcke in der `switch`-Anweisung

- Eine `switch`-Anweisung darf maximal eine `default`-Alternative besitzen
- `ConstantExpression` muss eine Konstante des gewählten Integer-Typs sein, die zur Übersetzungszeit berechnet werden kann → `ConstantExpression` darf z.B. keinen Operationsaufruf enthalten

## Sichtbare Daten eines Objektes

### Einschub: Blöcke in der `switch`-Anweisung Semantik der `switch`-Anweisung

- Auswertung von Expression und Durchsuchen der `case`-Labels nach einem passenden Wert `ConstantExpression`
  - Passendes `case`-Label gefunden
    - Ausführung aller folgenden Anweisungen
    - Beachte: das sind auch die Anweisungen der eventuell folgenden `case`- und `default`-Labels
  - Kein passendes `case`-Label gefunden
    - Sprung zum (optimal) vorhandenen `default`-Label
    - Ausführung aller folgenden Anweisungen: Das sind auch die Anweisungen der eventuell folgenden `case`- und `default`-Labels

## Sichtbare Daten eines Objektes

### Einschub: Blöcke in der switch-Anweisung

#### Beispiel

```
public class SwitchAnweisung1 {
    public static void main (String [] args) {
        int farbeingabe = 1;
        switch ( farbeingabe ) {
            case 0: System.out.println("schwarz");
            case 1: System.out.println("rot");
            case 2: System.out.println("gelb");
            default: System.out.println
                ("Farbe nicht identifizierbar");
        }
    }
}
```

Aufruf: `java SwitchAnweisung1` liefert:  
rot  
gelb  
Farbe nicht identifizierbar

Nicht was wir wollten ☹️

## Sichtbare Daten eines Objektes

### Einschub: Blöcke in der switch-Anweisung Semantik der switch-Anweisung

- Bemerkungen:
  - Die **break**- (oder auch **return**-) Anweisung kann verwendet werden, um den **switch**-Block zu verlassen
  - Ist kein default-Label vorhanden und keines der **case**-Labels passt, dann wird der **switch**-Block übersprungen
  - Üblicherweise ist **break** die letzte Anweisung eines **case**-Labels
  - Das **default**-Label sollte das letzte Label sein

## Sichtbare Daten eines Objektes

### Einschub: Blöcke in der switch-Anweisung

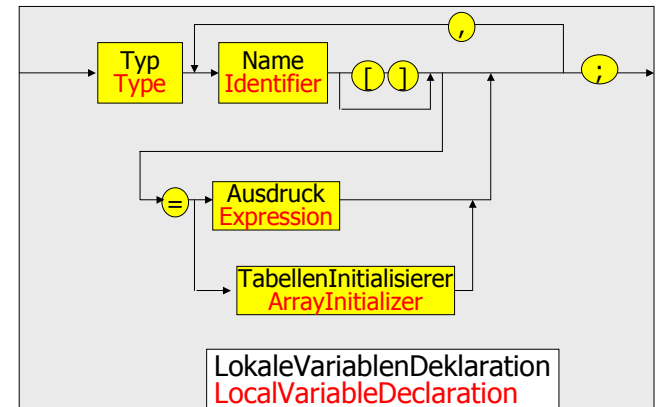
#### Beispiel

```
public class SwitchAnweisung2 {
    public static void main (String [] args) {
        int farbeingabe = 1;
        switch ( farbeingabe ) {
            case 0: System.out.println("schwarz");
                break;
            case 1: System.out.println("rot");
                break;
            case 2: System.out.println("gelb");
                break;
            default: System.out.println
                ("Farbe nicht identifizierbar");
        }
    }
}
```

Aufruf: `java SwitchAnweisung2` liefert: rot 😊

## Sichtbare Daten eines Objektes

### Struktur der Deklaration lokaler Variablen



## Sichtbare Daten eines Objektes

### Eigenschaften der lokalen Variablen

- **Lokale Variablen** besitzen folgende Eigenschaften:
  - Sie existieren nur solange, wie sich der zugehörige Block in Ausführung befindet
  - Sie sind nur innerhalb des Blocks sichtbar
  - Sie werden nicht mit Defaultwerten initialisiert. Stattdessen stellt der Compiler sicher, dass vor dem ersten lesenden Zugriff eine Initialisierung der Variablen bereits stattgefunden hat

## Sichtbare Daten eines Objektes

### Sichtbarkeitsbereich von `neighbor`, `carrier`

```

public void next() {
    turnLeft();
    if (frontIsClear()) {
        Adder neighbor =
            new Adder(avenue()-1, helper);
        Carrier carrier = neighbor.add();
        carrier.carry();
        neighbor.next();
    }
    ... neighbor ...
    ... carrier ...
}

```

**class Adder**

Kein Zugriff mehr

## Sichtbare Daten eines Objektes

### Formale Parameter der Operationen

- Formale Parameter einer Operation sind der 4. Weg, wodurch ein Objekt Zugriff auf andere Objekte und primitive Werte bekommt.
- Das Beispiel illustriert, wie Objekte Zugriff auf primitive Werte mit Hilfe formaler Parameter bekommen
  - die Operation `putBeepers(int)` erwartet einen Wert vom primitiven Typ `int` als Parameter.
- Die Herstellung von Bekanntschaftsbeziehungen zwischen Objekten mit Hilfe formaler Parametern, wird von dem Beispiel in der jetzigen Form nicht illustriert.

## Sichtbare Daten eines Objektes

### Formale Parameter der Operationen

### Erweiterung des `Adder`-Beispiels **Hausaufgabe!**

- Um die Herstellung von Bekanntschaftsbeziehungen zwischen Objekten mit Hilfe formaler Parametern zu illustrieren, sollen Sie das Beispiel erweitern, so dass das Resultat der Addition verschlüsselt gestellt wird.
- Ausgangspunkt der Hausaufgabe:
  - Der Code des Beispiels von der WWW-Seite der Vorlesung
  - Die Skizze der Erweiterung, die in den folgenden Folien diskutiert wird

## Sichtbare Daten eines Objektes

### Formale Parameter der Operationen Erweiterung des `Adder`-Beispiels

- Die Verschlüsselung jeder Ziffer im Resultat wird von einem Roboter vom Typ `Encrypter` kurz vor dem Stellen der Ziffer in der entsprechenden Ecke der 1. Straße durchgeführt
- Die Klasse `Encrypter` implementiert die Operation `int encrypt(int)`:
  - Liefert einen Wert zurück, der aus dem Parameterwert nach einer bestimmten Strategie gewonnen wird
    - Beispiel: eine Zahl  $x < 10$  könnte durch die Zahl  $10-x$  kodiert werden

## Sichtbare Daten eines Objektes

### Formale Parameter der Operationen Erweiterung des `Adder`-Beispiels

- Die Klasse `Adder` wird so modifiziert, dass der Additionsoperation ein `Encrypter` übergeben werden kann  $\rightarrow$  `add(Encrypter enc)`
  - Der `Adder` übergibt dem `ResultWriter` eine kodierte Version der Anzahl der Piepser, die gesetzt werden sollen
- Der `Encrypter` wird von jedem `Adder` an seinen linken Nachbarn beim Aufruf der modifizierten Operation `next(Encrypter enc)` weitergereicht
- Ursprünglich wird der `Encrypter` vom Piloten erzeugt

## Sichtbare Daten eines Objektes

### Formale Parameter der Operationen Erweiterung des `Adder`-Beispiels

```
task {
    Adder adder = new Adder(7);
    Encrypter enc = new Encrypter(...);
    adder.start(enc);
    adder.turnOff();
}
```

```
public void start(enc) {
    Carrier carrier = add(enc);
    carrier.carry();
    next(enc); // start recursion
}
```

## Sichtbare Daten eines Objektes

### Formale Parameter der Operationen Erweiterung des `Adder`-Beispiels

```
public Carrier add(Encrypter enc) {
    int encrypted, c = addCiphers();
    int if (c > 9) {
        encrypted = enc.encrypt(c - 10);
        helper.putBeepers(encrypted);
        return new Carrier(..., 1);
    } else {
        encrypted = enc.encrypt(c);
        helper.putBeepers( encrypted );
        return new Carrier(..., 0);
    }
}
```

`class Adder`

## Sichtbare Daten eines Objektes

### Formale Parameter der Operationen Erweiterung des Adder-Beispiels

```
public void next(Encrypter enc) {
    turnLeft();
    if (frontIsClear()) {
        Adder neighbor =
            new Adder(avenue()-1, helper);
        Carrier carrier = neighbor.add(enc);
        carrier.carry();
        neighbor.next(enc);
    }
    turnOff();
}
```

class Adder

## Sichtbare Daten eines Objektes

### Formale Parameter der Operationen

- Formale Parameter sind im Prinzip lokale Variablen der Operation → deren Sichtbarkeit ist der ganze Operationsrumpf.
- Sie werden nicht verwendet um Zwischenergebnisse zu speichern, sondern **um der Operationsausführung Daten von außen zu übergeben**
- Die Parameterübergabe erfolgt grundsätzlich als Wertübergabe (*call by value*)
  - Beim Operationsaufruf wird von dem aktuellen Parameter eine **Kopie** angelegt. Bei Referenzattributen wird eine Kopie der Objektreferenz übergeben
  - Die Methode arbeitet **ausschließlich mit der Kopie**

## Sichtbare Daten eines Objektes

### Formale Parameter der Operationen

- Folge der call-by-value-Semantik: Schreibender Zugriff auf die Originalversion des Parameters ist unmöglich
- Nur der Zustand der Objekte, auf die durch die Originalversion und Kopie verwiesen wird, kann durch Operationsaufrufe geändert werden
- Nach der Rückkehr aus der Methode steht der Originalwert wieder zur Verfügung, selbst wenn innerhalb der Methode eine Änderung vorgenommen wurde

## Sichtbare Daten eines Objektes

### Formale Parameter der Operationen

```
public class RefTypeParameter {
    private String name;
    private int a;
    public RefTypeParameter
        (String initName, int initA) {
        name = initName; a = initA;
    }
    public void print() {
        System.out.println("name=" + name
            + " a=" + a);
    }
    public void setA(int newA) { a = newA; }
}
```

Automatische  
Konvertierung nach String

## Sichtbare Daten eines Objektes

### Formale Parameter der Operationen

```
class RefTypeClient {
    public void
        wertÄnderung(RefTypeParameter rtp) {
            rtp.print();
            rtp.setA(5555);
            rtp = new RefTypeParameter("Obj2", 3);
            rtp.print();
        } // wertÄnderung
}
```

## Sichtbare Daten eines Objektes

### Formale Parameter der Operationen

```
class Program {
    public static void main(String[] args) {
        RefTypeParameter obj1 =
            new RefTypeParameter ("Obj1", 4);
        RefTypeClient client =
            new RefTypeClient();
        obj1.print();
        client.wertÄnderung(obj1);
        obj1.print();
    } // main
}
```

## Sichtbare Daten eines Objektes

### Formale Parameter der Operationen

- Der Aufruf `java Program` liefert folgendes Ergebnis:

```
name=Obj1  a=4
// 1. Aufruf von obj1.print() in main
name=Obj1  a=4
// 1. Aufruf von rtp.print() während
// der Ausführung von
// client.wertÄnderung() in main
name=Obj2  a=3
// 2. Aufruf von rtp.print()
// während der Ausführung von
// client.wertÄnderung() in main
name=Obj1  a=5555
// 2. Aufruf von obj1.print() in main
```

## Sichtbare Daten eines Objektes

### Formale Parameter der Operationen

- Folge der Parameterübergabe als Wertübergabe: **unerwünschte Seiteneffekte werden vermieden**
- Seiteneffekt: die Ausführung einer Operation an einem Objekt ändert den Zustand anderer Objekte
- Seiteneffekte sollen verhindert werden, weil sie das Verständnis, die Wartung und die Weiterentwicklung von Software erschweren
  - Grund: die Wirkung einer Veränderung ist in Präsenz von Seiteneffekten schwer zu lokalisieren
- Im folgenden wird mit Hilfe eines Beispiels illustriert, wie durch schreibenden Zugriff auf die Originalversion der Parameter Seiteneffekte entstehen können



# Sichtbare Daten eines Objektes

**Formale Parameter der Operationen**  
**Seiteneffekten durch direkte Zugriff auf Parameter**

```
class AnObject {
    ...
}
```

Nehmen wir für ein Moment an, direkte schreibender Zugriff auf Parameter ist erlaubt

```
class Other {
    ...
    void getObject(AnObject obj) {
        obj = new AnObject(...);
    }
    ...
}
```



# Sichtbare Daten eines Objektes

**Formale Parameter der Operationen**  
**Seiteneffekten durch direkte Zugriff auf Parameter**

```
class Client {
    AnObject anObject;
    public Client() {
        anObject = new AnObject(...);
    }
    void op(Other other) {
        other.getObject(anObject);
    }
}
```

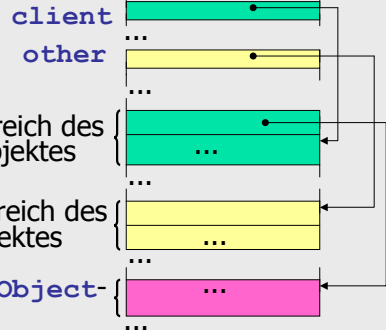
Die Ausführung von `other.getObject(anObject)` ändert den Attribut-Wert des aufrufenden Client-Objekt !



# Sichtbare Daten eines Objektes

**Formale Parameter der Operationen**  
**Seiteneffekten durch direkte Zugriff auf Parameter**

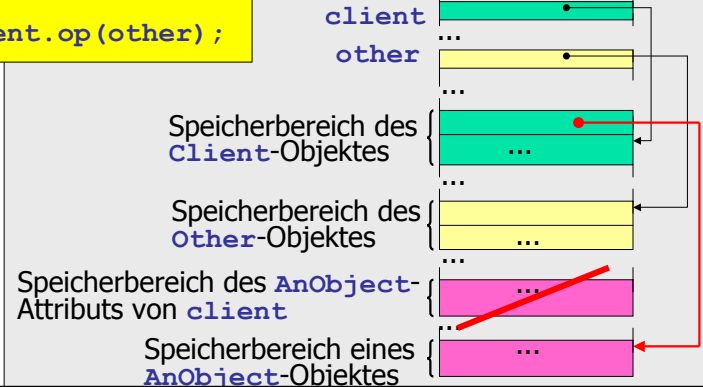
```
Client client =
    new Client(...);
Other other =
    new Other();
```



# Sichtbare Daten eines Objektes

**Formale Parameter der Operationen**  
**Seiteneffekten durch direkte Zugriff auf Parameter**

```
...
client.op(other);
```





# Sichtbare Daten eines Objektes

## Formale Parameter der Operationen Seiteneffekten

Referenzbezeichner

Objekttabelle

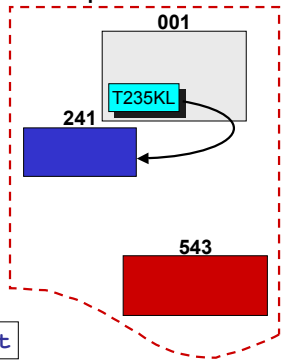
Heap

client  
T156UT

anObject  
T235KL

other  
T135KL

T156UT		001
T235KL		241
T135KL		543



```
void op(Other other) {
  other.getObject(anObject);
}
```

Klasse Client

other bekommt Zugriff auf den Eintrag T235KL ...



# Sichtbare Daten eines Objektes

## Formale Parameter der Operationen Seiteneffekten

Referenzbezeichner

Objekttabelle

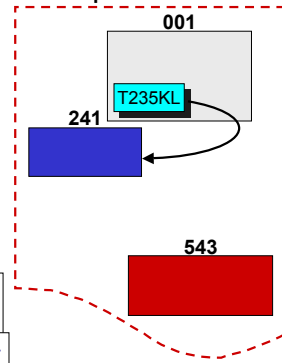
Heap

client  
T156UT

anObject  
T235KL

other  
T135KL

T156UT		001
T235KL		241
T135KL		543



```
void getObject(AnObject obj) {
  obj = new AnObject(...);
}
```

Klasse Other

obj enthält den Index für den Eintrag T235KL ...



# Sichtbare Daten eines Objektes

## Formale Parameter der Operationen Seiteneffekten

Referenzbezeichner

Objekttabelle

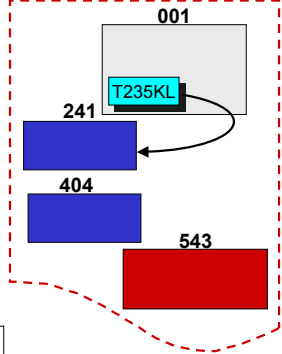
Heap

client  
T156UT

anObject  
T235KL

other  
T135KL

T156UT		001
T235KL		241
T135KL		543



```
void getObject(AnObject obj) {
  obj = new AnObject(...);
}
```

Klasse Other

Neues AnObject-Objekt wird erzeugt in 404 ...



# Sichtbare Daten eines Objektes

## Formale Parameter der Operationen Seiteneffekten

Referenzbezeichner

Objekttabelle

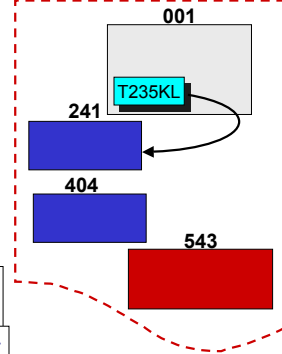
Heap

client  
T156UT

anObject  
T235KL

other  
T135KL

T156UT		001
T235KL		404
T135KL		543



```
void getObject(AnObject obj) {
  obj = new AnObject(...);
}
```

Klasse Other

Der Eintrag T235KL wird modifiziert ...



# Sichtbare Daten eines Objektes

Formale Parameter der Operationen Seiteneffekten

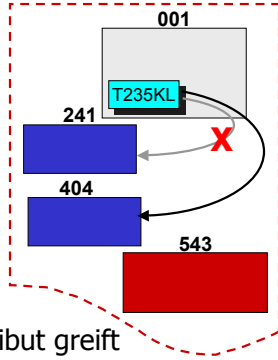
Referenzbezeichner

Objekttabelle

Heap

- client  
T156UT
- anObject  
T235KL
- other  
T135KL

T156UT		001
T235KL		404
T135KL		543



Beim nächsten Zugriff auf sein Attribut greift **client** plötzlich auf ein anderes Objekt



# Sichtbare Daten eines Objektes

Formale Parameter der Operationen Call by value

```
class AnObject {
    ...
}
```

Betrachten wir das gleiche Beispiel mit der call-by-value Semantik der Parameterübergabe

```
class Other {
    ...
    void getObject(AnObject obj) {
        obj = new AnOject(...);
    }
    ...
}
```



# Sichtbare Daten eines Objektes

Formale Parameter der Operationen Call by value

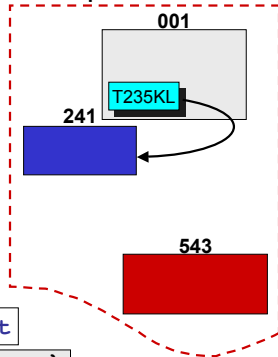
Referenzbezeichner

Objekttabelle

Heap

- client  
T156UT
- anObject  
T235KL
- other  
T135KL

T156UT		001
T235KL		241
T135KL		543



```
void op(Other other) {
    other.getObject(anObject);
}
```

Klasse Client

```
void getObject(AnObject obj) { ... }
```

Klasse Other



# Sichtbare Daten eines Objektes

Formale Parameter der Operationen Call by value

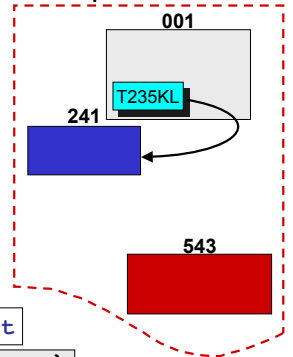
Referenzbezeichner

Objekttabelle

Heap

- client  
T156UT
- anObject  
T235KL
- other  
T135KL
- obi  
T435KL

T156UT		001
T235KL		241
T135KL		543
T435KL		241



```
void op(Other other) {
    other.getObject(anObject);
}
```

Klasse Client

```
void getObject(AnObject obj) { ... }
```

Klasse Other

## Sichtbare Daten eines Objektes

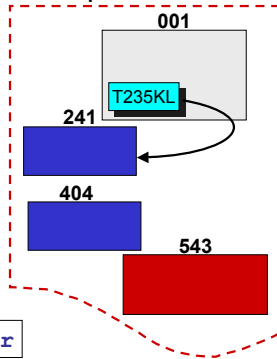
### Formale Parameter der Operationen Call by value

Referenzbezeichner

Objekttabelle

Heap

client		
T156UT		
anObject	T156UT	001
T235KL	T235KL	241
other	T135KL	543
T135KL	T435KL	404
obj		
T435KL		



```
void getObject(AnObject obj) {
    obj = new AnObject(...);
}

```

Klasse Other

Der Eintrag T435KL wird modifiziert. client unverändert.

## Sichtbare Daten eines Objektes

### Verbergen von Attributen

- **Problem:** ein Attribut, das in einer Klasse deklariert ist, hat denselben Namen, wie ein lokales Attribut oder ein Parameter einer Operation
- Innerhalb der Operationsdefinition überdeckt bzw. verbirgt die lokale Variable das Attribut der Klasse
  - Das Attribut der Klasse ist in der Operation nicht mehr sichtbar → Zugriff immer auf das lokale Attribut
  - Zugriff auf das Attribut der Klasse entweder durch Verwendung verschiedener Namen oder durch Qualifizierung möglich: `this.attributName`

## Sichtbare Daten eines Objektes

### Verbergen von Attributen

- Beispiel Kundenverwaltung: Unterscheidung von `name` und `firmenName`, um eine Überdeckung zu vermeiden

```
public class Kunde {
    private String firmenName,
    private firmenAdresse;
    public Kunde(String name)
    { firmenName = name; }
    public void setFirmenName (String name)
    { firmenName = name; }
    ...
}

```

## Sichtbare Daten eines Objektes

### Verbergen von Attributen

```
public class Kunde {
    private String firmenName, firmenAdresse;
    public Kunde(String firmenName)
    { // Konstruktor
        this.firmenName = firmenName;
    }
    public void setFirmenName (String firmenName)
    { // Schreibende Operationen
        this.firmenName = firmenName;
    } ...
}

```

Kundenverwaltung: Unterscheidung durch `this`

## Sichtbare Daten eines Objektes

### Verbergen von Attributen

Zugriff auf das Attribut der Klasse durch Qualifizierung am Beispiel `Adder`

```
class Adder {
    private Helper helper;
    public Adder(int avenue, Helper helper) {
        super(2, avenue, North, 0);
        this.helper = helper;
    }
    ...
}
```

Vergleichen Sie mit dem Konstruktor in Folie 20

## Sichtbarkeitsbereiche: Attribute & lokale Variablen

```
class Bsp {
    int einInt, j;
    char einChar;
    void g( int n ) {
        int i, j;
        ... i ... j ... n ... this.j ... einInt
    } // g()
} // class Bsp
```

## Sichtbarkeitsbereiche: Attribute & lokale Variablen

- Zusammenfassung:
  - Der Klassenname ist "außerhalb" (im Paket) deklariert
  - Methoden & Attribute werden im Klassenrumpf deklariert
  - Argumente und lokale Variablen werden innerhalb der Methode deklariert
  - Sichtbarkeit bestimmt auch die Lebensdauer

## Sichtbarkeitsbereiche: Attribute & lokale Variablen

```
class Bsp { Im Paket: Klassenname
    int einInt, j;
    char einChar;
    void g( int n ) {
        int i, j;
        ... i ... j ... n ... this.j ... einInt
    } // g()
} // class Bsp
```

## Klassenmethoden und -attribute

- Manche Eigenschaften sollen zwischen allen Instanzen einer Klasse gemeinsam genutzt werden:
  - Zähler, wie viele Objekte einer Klasse erzeugt wurden
  - Die nächste zu vergebende Identifikationsnummer für die Exemplare einer Klasse
  - Konstanten, die von allen Objekten einer Klasse gemeinsam genutzt werden sollen
- Klassenvariablen und Klassenoperationen sind der Klasse zugeordnet, im Gegensatz zu Instanz-Variablen und Instanz-Methoden, die nur in Objekten existieren, bzw. an Objekten vorgenommen werden können

## Klassenmethoden und -attribute

- Klassenoperationen und -Variablen werden mit dem Schlüsselwort `static` deklariert
- Sie werden als `Klasse.nameDerEigenschaft` angesprochen, aber auch `einObjekt.nameDerEigenschaft` ist möglich, wobei `einObjekt` ein Exemplar von `Klasse` ist
  - Dadurch sind sie "global" sichtbar: überall, wo die Klasse auch sichtbar ist
- Bereits bekannte Beispiele
  - Klassenattribut: `System.out`
  - Klassenoperation: `static void main(...)`

## Klassenmethoden und -attribute

```
class Person {
    static int bevölkerungsZähler = 0;
```

**static** Variable existiert für alle Objekte nur **einmal**

**Initialisierung** erfolgt vor Ausführung irgendeiner Methode der Klasse oder des Objektes

```
public Person ( .. ) {
    bevölkerungsZähler++;
    ...
}
```

Beim Erzeugen einer neuen Person zählt der Konstruktor mit

## Klassenmethoden und -attribute

```
// class Person fortsetzung
...
public static int bevölkerungsGröße ( ) {
    return bevölkerungsZähler;
}
...

```

Anzahl der erzeugten Objekte erfragen

## Klassenmethoden und -attribute

```
// class Person fortsetzung
```

```
...
```

Die Operation `finalize` wird an jedes Objekt durch den Garbage Collector (GC) aufgerufen, bevor das GC das Objekt endgültig löscht

```
public void finalize( ) {
    bevölkerungsZähler--;
}
```

beim Freigeben eines Personen-Objektes den Zähler entsprechend um eins verringern

```
} // class Person
```

## Klassenmethoden und -attribute

- Zugriff über den Klassennamen

```
int zähler = Person.bevölkerungsZähler;
System.out.println(zähler.toString()+"
Objekte");
```

- typische Anwendung

```
public boolean größer( Kreis b ) { .. }
public static boolean
    größer( Kreis a, Kreis b ) {..}
```

vergleicht sich selbst mit einem anderen Kreisobjekt

vergleicht 2 Kreise

## Klassenmethoden und -attribute

- Zugriff über den Klassennamen **Einschub: `toString()`**

```
int zähler = Person.bevölkerungsZähler;
System.out.println(zähler.toString()+"
Objekte");
```

Eine Operation, die in der Wurzelklasse `Object` implementiert ist und somit an Objekten jedes Typs vorgenommen werden kann. Liefert einen String zurück, der den Empfänger-Objekt als eine Zeichenkette darstellt. `Object` realisiert eine Standard-Darstellung aller Objekte. `toString()` kann aber in den Unterklassen von `Object` überschrieben werden, um spezifischere Darstellungen zu ermöglichen

## Klassenmethoden und -attribute

- Klassenmethoden kennen nur Klassenattribute
- Folgendes Beispiel wird vom Compiler zurückgewiesen. Beachte `main()` ist eine Klassenmethode!

```
class Mistake {
    public int i;
    static void main() {
        something( i );
        ...
    } // main
} // class Mistake
```

Eine statische Methode kann nicht auf Instanz-Variablen zugreifen!

## Klassenmethoden und -attribute

- Auch das Hauptprogramm ist innerhalb einer Klasse definiert (als Klassenmethode)
- Beim Aufruf des Interpreters mit der Klasse, wird keine Instanz erzeugt, sondern lediglich die Klassenmethode `main()` aufgerufen

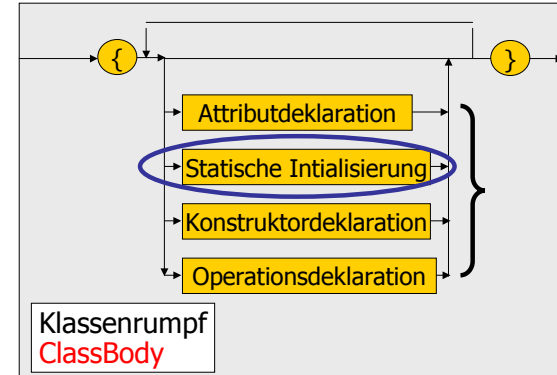
```
public class Prog {
    public static void main( .. ) {
        /* Instanzen der benötigten
           Objekte erzeugen und deren
           Methoden aufrufen */
    }
} // class Prog
```

Alternative  
Zeichen für  
Kommentar

## Klassenmethoden und -attribute

### Syntax der Klassen in Java

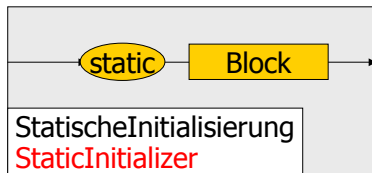
### Statische Initialisierung



## Klassenmethoden und -attribute

### Syntax der Klassen in Java

### Statische Initialisierung



- Die Anweisungen des Blocks dienen zur **Initialisierung** von **Klassenattributen**
- Innerhalb des Blocks sind Objektattribute nicht verwendbar
- **Statische Initialisierungsblöcke** werden angewandt, wenn andere Initialisierungsmöglichkeiten von ihrer Funktionalität her nicht ausreichen oder unpraktisch sind

## Klassenmethoden und -attribute

### Syntax der Klassen in Java

### Statische Initialisierung

```
public class Initialisierungsblock {
    private static double[] vektor
        = new double[6];
    // statische Initialisierungsblock
    static {
        for (int i = 0; i < vektor.length; i++) {
            vektor[i] = 1.0 / (i + 1);
        } // for
    } // static
    ...
}
```

Klassenattribute werden  
beim Laden der Klasse  
initialisiert