Prof. Dr. J. Giesl
D. Cloerkes, S. Dollase, N. Lommen, D. Meier, F. Meyer

Tutoraufgabe 1 (Überblickswissen):

a) Wie unterscheiden sich Referenzvariablen von Wertvariablen? Geben Sie ein kleines Java-Beispiel an, in dem dieser Unterschied deutlich wird.

b) Welche Schleifenarten haben Sie bisher kennengelernt? Nennen Sie je einen typischen Anwendungsfall!

c) Was ist der grundlegende Unterschied zwischen einer Klasse und einem Objekt?

d) Variablen können in einer Klasse, aber auch in einer Methode deklariert werden. Worin besteht der Unterschied und in welchen Fällen deklariert man Variablen typischerweise an welcher Stelle?

e) Was ist der Unterschied zwischen *Call By Reference* und *Call By Value*? Inwieweit unterstützt Java die beiden Konzepte?

Lösung:

a) Wertvariablen enthalten einen konkreten Wert. Wenn man einer Wertvariable `x` den Wert einer anderen Wertvariable `y` zuweist (`int x = y;`), wird dieser Wert einmalig übernommen. Direkt danach enthalten die beiden Variablen zwar denselben Wert, haben aber von da an nichts mehr miteinander zu tun. Wenn man also den Wert einer Variable ändert, dann bleibt der Wert der anderen Variable davon unberührt.
Referenzvariablen dagegen referenzieren ein Objekt, das irgendwo im Speicher liegt. Dieses Objekt kann auch von mehreren Referenzvariablen referenziert werden. Wenn dieses Objekt nun über eine der es referenzierenden Variablen verändert wird, dann ändert es sich für alle Variablen, die es referenzieren, denn alle verweisen ja auf dasselbe Objekt.
Zwei Beispielmethoden, um den Unterschied zwischen Wert- und Referenzvariablen zu verdeutlichen, sind die Folgenden:

```
public class A {
  int n;

  public static void beispielWertvariablen() {
    int x = 1;
    int y = 2;
    System.out.println(x + ", " + y); //1, 2

    y = x;
    System.out.println(x + ", " + y); //1, 1

    x = 3;
    System.out.println(x + ", " + y); //3, 1
  }

  public static void beispielReferenzvariablen() {
    A a = new A();
    a.n = 1;

    A b = new A();
    b.n = 2;
    System.out.println(a.n + ", " + b.n); //1, 2

    b = a;
    System.out.println(a.n + ", " + b.n); //1, 1
```

```
    a.n = 3;
    System.out.println(a.n + ", " + b.n); //3, 3
  }
}
```

b) Es wurden vier Arten von Schleifen eingeführt, nämlich `while`-, `do`-, `for`- und `foreach`-Schleifen. Die ersten beiden Schleifenarten sind dabei sehr ähnlich zueinander: Solange die Bedingung erfüllt ist, wird der Schleifenrumpf ausgeführt. Die `while`-Schleife prüft die Bedingung dabei immer vor Ausführung des Schleifenrumpfes, die `do`-Schleife immer danach. Man nutzt diese Art von Schleifen, wenn man die Anzahl der Durchläufe (auch zur Laufzeit) beim Betreten der Schleife noch nicht kennt und stattdessen in der Schleife bleiben möchte, bis die Bedingung nicht mehr erfüllt ist. Die Bedingung kann dabei ein beliebiger Ausdruck vom Typ `boolean` sein.
Die `for`-Schleife bietet im Gegensatz dazu mehr Struktur: So lässt sich im Kopf der Schleife erstens die Initialisierung angeben, die vor erstmaligem Betreten der Schleife ausgeführt wird, zweitens die Bedingung, die wie bei einer `while`-Schleife vor jeder Ausführung des Schleifenrumpfes überprüft wird und drittens die Fortschaltung, die nach jeder Ausführung des Schleifenrumpfes ausgeführt wird. Man nutzt die `for`-Schleife vor allen Dingen in Situationen, in denen bei Betreten der Schleife (zur Laufzeit, aber nicht notwendigerweise schon zur Kompilierzeit) bereits bekannt ist, wie oft die Schleife ausgeführt werden soll.
Die `foreach`-Schleife ist eine einfache Möglichkeit, über alle Elemente eines Arrays zu iterieren. Im Schleifenkopf wird eine Variable deklariert, die in jeder Iteration der Schleife ein Array-Element nach dem anderen durchläuft. Die `foreach`-Schleife nutzt man vor allen Dingen, wenn man mit jedem Array-Element eine bestimmte Operation durchführen möchte. Später wird die `foreach`-Schleife auch noch für andere Sammlungen mehrerer Elemente eines Typs nützlich werden.

c) Eine Klasse definiert einen Typ, ein Objekt ist eine konkrete Instanz dieses Typs. In einer Klasse werden die Attribute und Methoden festgelegt, die jedes Objekt dieser Klasse hat. Normale (d.h. nicht-statische) Attribute haben aber für eine Klasse keinen Wert, und normale (d.h. nicht-statische) Methoden kann man auch nicht auf Klassen aufrufen. Für ein konkretes Objekt dagegen haben alle Attribute stets einen Wert und man kann alle Methoden auf diesem Objekt aufrufen. Prägnant gesagt: Eine Klasse beschreibt, wie ein Objekt aussieht und was es kann. Ein Objekt dieser Klasse hat dann die in der Klasse definierten Eigenschaften.

d) Nicht-statische Variablen, die innerhalb einer Klasse deklariert werden, sind Attribute der Objekte, die später von dieser Klasse instantiiert werden können. Für jedes Objekt kann jedes Attribut dann einen anderen Wert haben. Die Werte der Attribute beschreiben sozusagen den Zustand des Objekts. Auch wenn das Objekt gerade nicht genutzt, aber noch irgendwo referenziert wird, bleiben die Attribute dieses Objekts bestehen.
Statische Variablen, die ebenfalls innerhalb einer Klasse deklariert werden, haben dagegen einen Wert *je Klasse*. Alle Objekte der Klasse teilen sich dann dasselbe Attribut, d.h. es hat für alle Objekte dieser Klasse denselben Wert.
Außerdem gibt es Variablen, die in einer Methode deklariert werden. Auf diese kann man nur solange zugreifen, bis die Methode ein `return`-Statement erreicht hat (oder bei `void`-Methoden ohne `return`-Statement die letzte Zeile abgearbeitet hat), d.h. bis sie zu Ende gelaufen ist. Variablen, die innerhalb einer Methode deklariert werden, bezeichnet man deshalb auch als lokale Variablen. Sie werden genutzt, um Zwischenergebnisse (verschiedenster Art) vorzuhalten, die aber außerhalb der Methode nicht benötigt werden.

e) Diese beiden Begriffe beschreiben zwei verschiedene Arten, wie einer Methode bei ihrem Aufruf Werte übergeben werden können: Wird *Call By Value* benutzt, wird der Wert der Variable übergeben. Bei Referenzvariablen ist dieser übergebene Wert dann die Referenz auf das entsprechende Objekt. Wird in der Methode am referenzierten Objekt etwas geändert, so ist diese Änderung auch nach Ausführung der Methode noch sichtbar, da das referenzierte Objekt ja weiterhin besteht. Es bleibt aber dabei, dass an zwei Stellen auf dasselbe Objekt verwiesen wird. Wird in der Methode die eine Referenz geändert, bleibt die andere Referenz außerhalb der Methode bestehen. Dies ist auch der Unterschied zu echtem *Call By Reference*: Hier wird in der Methode nicht eine zweite Referenz auf dasselbe Objekt genutzt, sondern es wird mit der ursprünglichen Referenz gearbeitet, die es auch vor Aufruf der Methode schon gab! Wird die

Referenz dann innerhalb der Methode geändert, ändert sich auch die Referenz außerhalb der Methode. In Java wird immer *Call By Value* genutzt. Echtes *Call By Reference* gibt es in Java nicht, auch wenn man natürlich Referenzen auch per *Call By Value* übergeben kann. Dies ist aber, wie oben erläutert, nicht dasselbe.

Tutoraufgabe 2 (Programmierung):

Bubblesort ist ein Algorithmus zum Sortieren von Arrays, der wie folgt vorgeht, um ein Array `a` zu sortieren: Das Array wird wiederholt von links nach rechts durchlaufen. Am Ende des n-ten Durchlauf gilt, dass die letzten n Array-Elemente an ihrer endgültigen Position stehen. Folglich müssen im $(n+1)$-ten Durchlauf nur noch die ersten `a.length`$-n$ Elemente betrachtet werden. In jedem Durchlauf wird in jedem Schritt das aktuelle Element mit seinem rechten Nachbarn verglichen. Falls das aktuelle Element größer ist als sein rechter Nachbar, werden sie getauscht.
Als Beispiel betrachten wir das Array `{3,2,1}`. Im ersten Durchlauf wird erst 3 mit 2 getauscht (dies ergibt `{2,3,1}`) und dann 3 mit 1, was `{2,1,3}` ergibt. Im zweiten Durchlauf wird 2 mit 1 getauscht, was zu `{1,2,3}` führt.
Implementieren Sie eine Klasse `BubbleSort` mit einer Methode `public static void sort(int[] a)`, die das Array `a` mithilfe des Algorithmus *Bubblesort* aufsteigend sortiert.

Hinweise:

- Im Moodle-Lernraum stehen die beiden Java-Dateien `BubbleSort.java` und `BubbleSortTest.java` zum Download zur Verfügung. Speichern Sie beide Dateien in einem neuen Ordner.
  Die Klasse `BubbleSort` enthält eine Methode `sort` mit leerem Rumpf. Wenn Sie die Implementierung vervollständigen und anschließend mit `javac BubbleSortTest.java` kompilieren, dann können Sie Ihre Implementierung mit `java BubbleSortTest` testen.

Lösung:

```
public class BubbleSort {
    public static void sort(int[] a) {
        for (int i = a.length - 1; i > 0; i--) {
            for (int j = 0; j < i; j++) {
                if (a[j] > a[j + 1]) {
                    int tmp = a[j];
                    a[j] = a[j + 1];
                    a[j + 1] = tmp;
                }
            }
        }
    }
}
```

[3, 2, 1]
i = 2, j = 0
[2, 3, 1]
i = 2, j = 1
[2, 1, 3]
i = 1, j = 0
[1, 2, 3]
i = 1, j = 1
[1, 2, 3]

Tutoraufgabe 4 (Verifikation mit Arrays):

Gegeben sei folgendes Java-Programm P, wobei `x` und `i` `int`-Variablen sind, `res` eine `boolean`-Variable und `a` ein Array vom Typ `int[]` ist:

a = [1, 2, 4, 2, 5] => a.length = 5
x = 2

⟨ true ⟩ (Vorbedingung)

<false = false ∧ 0 = 0>
<false = false ∧ i = 0>

```
i = 0;
res = false;
```

<res = false ∧ i = 0> <IV>

```
while(i < a.length) {
```

<res = x ∈ {a[j] | 0 ≤ j ≤ i – 1} ∧ i < a.length>

i	res
0	false
1	false
2	true
3	true
4	true

das Programm prüft, ob x in a ist

<res = x ∈ {a[j] | 0 ≤ j ≤ i – 1}>

=> x ist in a bis zum Index i - 1

```
  if(x == a[i]) {
    res = true;
  }
  i = i + 1;
}
```

<IV ∧ i < a.length, x == a[i]>
<IV ∧ i = i + 1>
<IV ∧ i = i + 1>
<IV>
<IV ∧ i >= a.length>

⟨ res = x ∈ {a[j] | 0 ≤ j ≤ a.length−1} ⟩ (Nachbedingung)

a) Vervollständigen Sie die folgende Verifikation der partiellen Korrektheit des Algorithmus im Hoare-Kalkül, indem Sie die unterstrichenen Teile ergänzen. Hierbei dürfen zwei Zusicherungen nur dann direkt untereinander stehen, wenn die untere aus der oberen folgt. Hinter einer Programmanweisung darf nur eine Zusicherung stehen, wenn dies aus einer Regel des Hoare-Kalküls folgt.

Beachten Sie bei der Anwendung der "Bedingungsregel 1" mit Vorbedingung φ, Nachbedingung ψ und `if`-Bedingung B, dass $\varphi \wedge \neg B \implies \psi$ gelten muss, d. h. die Nachbedingung ψ der `if`-Anweisung muss aus der Vorbedingung φ der `if`-Anweisung und der negierten Bedingung $\neg B$ folgen. Geben Sie beim Verwenden der Regel einen entsprechenden Beweis an.

Hinweise:

- Gehen Sie davon aus, dass keine Integer-Überlaufe stattfinden, d.h. behandeln Sie Integers als die unendliche Menge $\mathbb{Z}$.
- Sie dürfen beliebig viele Zusicherungs-Zeilen ergänzen oder streichen. In der Musterlösung werden allerdings genau die angegebenen Zusicherungen benutzt.
- Bedenken Sie, dass die Regeln des Kalküls syntaktisch sind, weshalb Sie semantische Änderungen (beispielsweise von $x+1 = y+1$ zu $x = y$) nur unter Zuhilfenahme der Konsequenzregeln vornehmen dürfen.
- Der Ausdruck `x ∈ M` hat den Wert `true`, wenn `x` in der Menge `M` enthalten ist, sonst hat der Ausdruck den Wert `false`.

b) Untersuchen Sie den Algorithmus P auf seine Terminierung. Für einen Beweis der Terminierung muss eine Variante angegeben werden und unter Verwendung des Hoare-Kalküls die Terminierung bewiesen werden.

Geben Sie auch bei dieser Teilaufgabe einen Beweis für die Aussage $\varphi \wedge \neg B \implies \psi$ bei der Anwendung der "Bedingungsregel 1" an.

Lösung:

a)

⟨true⟩
⟨0 = 0 ∧ false = false⟩
```
i = 0;
```
⟨i = 0 ∧ false = false⟩
```
res = false;
```
⟨i = 0 ∧ res = false⟩
⟨res = x ∈ {a[j] | 0 ≤ j ≤ i − 1} ∧ i ≤ a.length⟩
```
while (i < a.length) {
```
⟨res = x ∈ {a[j] | 0 ≤ j ≤ i − 1} ∧ i ≤ a.length ∧ i < a.length⟩
```
    if (x == a[i]) {
```
⟨res = x ∈ {a[j] | 0 ≤ j ≤ i − 1} ∧ i ≤ a.length ∧ i < a.length ∧ x = a[i]⟩
⟨true = x ∈ {a[j] | 0 ≤ j ≤ i + 1 − 1} ∧ i + 1 ≤ a.length⟩
```
        res = true;
```
⟨res = x ∈ {a[j] | 0 ≤ j ≤ i + 1 − 1} ∧ i + 1 ≤ a.length⟩
```
    }
```
⟨res = x ∈ {a[j] | 0 ≤ j ≤ i + 1 − 1} ∧ i + 1 ≤ a.length⟩
```
    i = i + 1;
```
⟨res = x ∈ {a[j] | 0 ≤ j ≤ i − 1} ∧ i ≤ a.length⟩

```
}
```

$\langle \texttt{res} = \texttt{x} \in \{\texttt{a}[j] \mid 0 \leq j \leq \texttt{i} - 1\} \wedge \texttt{i} \leq \texttt{a.length} \wedge \neg \texttt{i} < \texttt{a.length} \rangle$
$\langle \texttt{res} = \texttt{x} \in \{\texttt{a}[j] \mid 0 \leq j \leq \texttt{a.length} - 1\} \rangle$

Um die Bedingungsregel anwenden zu dürfen, muss noch gezeigt werden, dass aus $\texttt{res} = \texttt{x} \in \{\texttt{a}[j] \mid 0 \leq j \leq \texttt{i} - 1\} \wedge \texttt{i} \leq \texttt{a.length} \wedge \texttt{i} < \texttt{a.length}$ zusammen mit $\neg(\texttt{x} = \texttt{a[i]})$ die Nachbedingung $\texttt{res} = \texttt{x} \in \{\texttt{a}[j] \mid 0 \leq j \leq \texttt{i} + 1 - 1\} \wedge \texttt{i} + 1 \leq \texttt{a.length}$ folgt. Da sich der Wert von $\texttt{x} \in M$ nicht ändert, wenn zu M ein Wert ungleich x hinzugefügt wird, impliziert $\texttt{res} = \texttt{x} \in \{\texttt{a}[j] \mid 0 \leq j \leq \texttt{i} - 1\} \wedge \neg(\texttt{x} = \texttt{a[i]})$ die Aussage $\texttt{res} = \texttt{x} \in \{\texttt{a}[j] \mid 0 \leq j \leq \texttt{i} + 1 - 1\}$. Weiterhin folgt aus $\texttt{i} < \texttt{a.length}$ auch $\texttt{i} + 1 \leq \texttt{a.length}$. Also darf die Bedingungsregel angewendet werden.

b) Wir wählen als Variante $V = \texttt{a.length} - \texttt{i}$. Hiermit lässt sich die Terminierung von P beweisen, denn für die einzige Schleife im Programm (mit Schleifenbedingung $B = \texttt{i} < \texttt{a.length}$) gilt:

- $B \Rightarrow V \geq 0$, denn $B = \texttt{i} < \texttt{a.length} \Leftrightarrow 0 < \texttt{a.length} - \texttt{i}$ und
- die folgende Ableitung ist korrekt:

$\langle \texttt{a.length} - \texttt{i} = m \wedge \texttt{i} < \texttt{a.length} \rangle$
$\langle \texttt{a.length} - (\texttt{i} + 1) < m \rangle$
`if (x == a[i]) {`
$\langle \texttt{a.length} - (\texttt{i} + 1) < m \wedge \texttt{x} = \texttt{a[i]} \rangle$
$\langle \texttt{a.length} - (\texttt{i} + 1) < m \rangle$
`res = true;`
$\langle \texttt{a.length} - (\texttt{i} + 1) < m \rangle$
`}`
$\langle \texttt{a.length} - (\texttt{i} + 1) < m \rangle$
`i = i + 1;`
$\langle \texttt{a.length} - \texttt{i} < m \rangle$

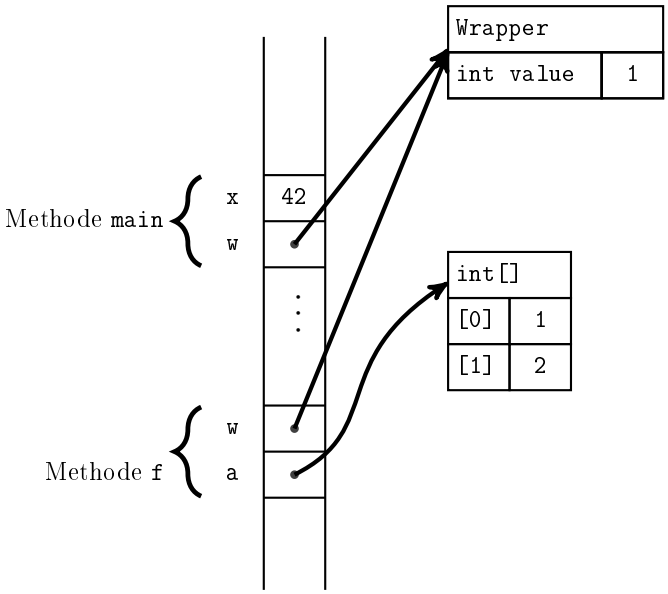
Die Vorbedingung der `if`-Anweisung ist gleich der Nachbedingung. Da $\neg \texttt{x} = \texttt{a[i]}$ nichts über die Variablen in der Nachbedingung aussagt, wird sie von der Vorbedingung impliziert. Also darf die Bedingungsregel angewendet werden.

In den Aufgaben 6 bis 8 sollen Sie Speicherzustände zeichnen. Angenommen wir haben folgenden `Java` Code:

```
public class Wrapper {
    int value;
}
```

```
public class Main {
    public static void main(String[] args) {
        int x = 42;
        Wrapper w = new Wrapper();
        w.value = 0;
        f(w);
    }

    public static void f(Wrapper w) {
        int[] a = {1,2};
        w.value = 1;

        // Speicherzustand hier gezeichnet
    }
}
```

Dann sieht der Speicher an der markierten Stelle wie folgt aus:



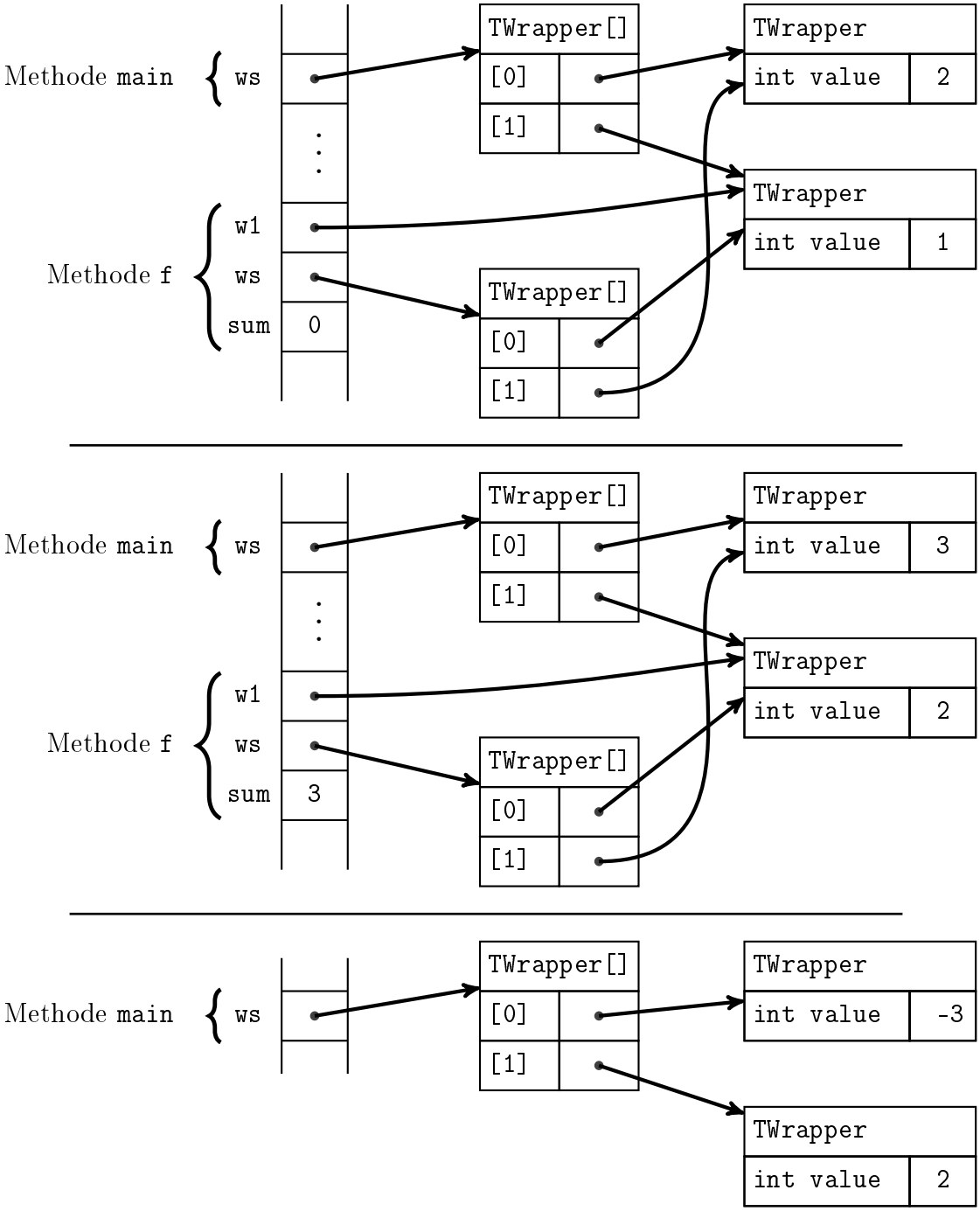
Tutoraufgabe 6 (Seiteneffekte):

Betrachten Sie das folgende Programm:

```
public class TSeiteneffekte {
    public static void main(String[] args) {
        TWrapper[] ws = new TWrapper[2];
        ws[0] = new TWrapper();
        ws[1] = new TWrapper();

        ws[0].value = 2;
        ws[1].value = 1;

        f(ws[1], new TWrapper[] { ws[1], ws[0] });

        // Speicherzustand hier zeichnen
    }

    public static void f(TWrapper w1, TWrapper[] ws) {
        int sum = 0;

        // Speicherzustand hier zeichnen

        for (int j = 0; j < ws.length; j++) {
            TWrapper w = ws[j];
            sum += w.value;
            w.value = j + 2;
        }

        // Speicherzustand hier zeichnen

        w1 = ws[1];
        w1.value = -sum;
    }
}

public class TWrapper {
    int value;
}
```

Es wird nun die Methode `main` ausgeführt. Stellen Sie den Speicher an allen drei markierten Programmzuständen graphisch dar. Achten Sie darauf, dass Sie alle (implizit) im Programm vorkommenden Arrays (außer `args`) und alle Objekte sowie die zu dem Zeitpunkt existierenden Programmvariablen darstellen.

Lösung:



Tutoraufgabe 7 (Seiteneffekte (Video)):

Betrachten Sie das folgende Programm:

```
public class VSeiteneffekte {
    public static void main(String[] args) {
        VWrapper w1 = new VWrapper();
        VWrapper w2 = w1;

        w1.i = 1;
        w2.i = 2;

        int x = 3;
        int[] a = { 1, 2 };

        f(w1, x, new int[] { 4, 5 });
        f(w2, x, a);
        //Speicherzustand hier zeichnen
    }

    public static void f(VWrapper w, int x, int[] a) {
        //Speicherzustand hier zeichnen
        x = a[0];
        a[0] = w.i;
        w.i = x;
    }
}

public class VWrapper {
    int i;
}
```

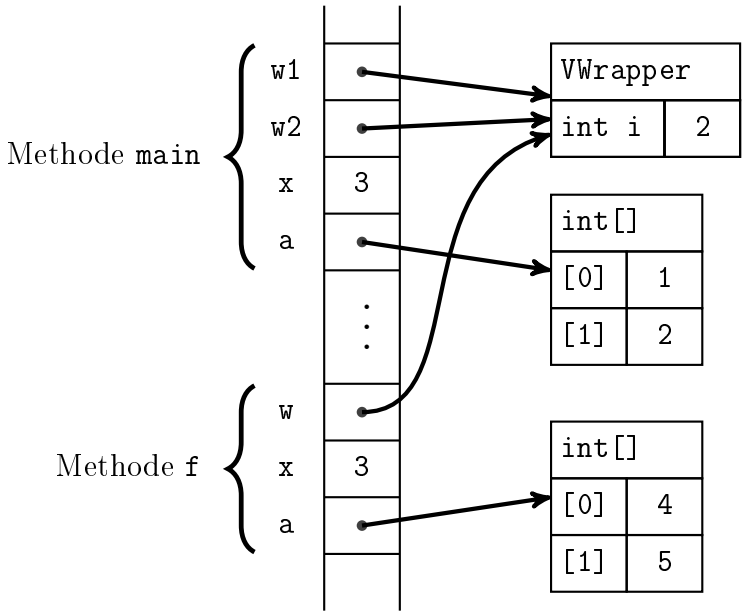
w1 → wrapper
i = (1 => 2 => 4 => 1)
w2
x = 3
a = ({1, 2} => {4, 2})

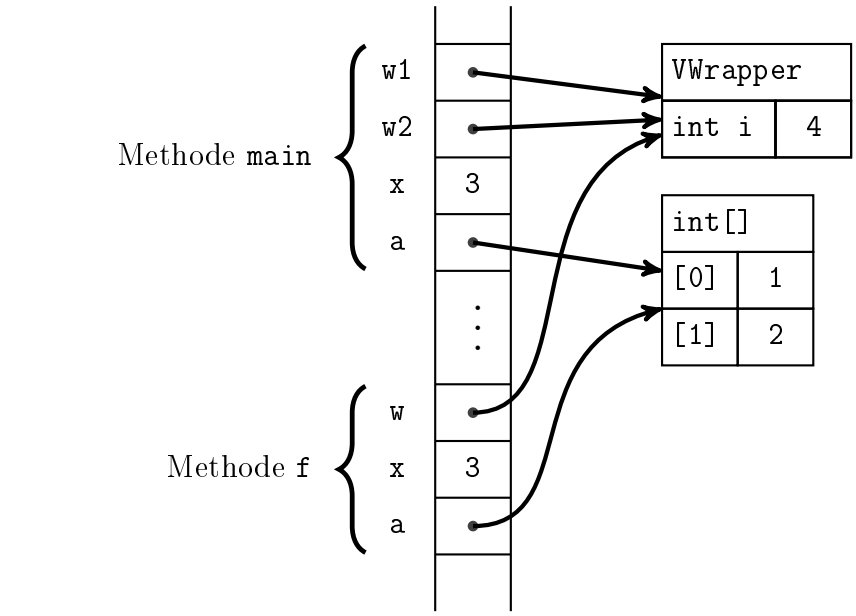
f(w1, x, {4, 5})
x = 4
a = ({4, 5} => {2, 5})

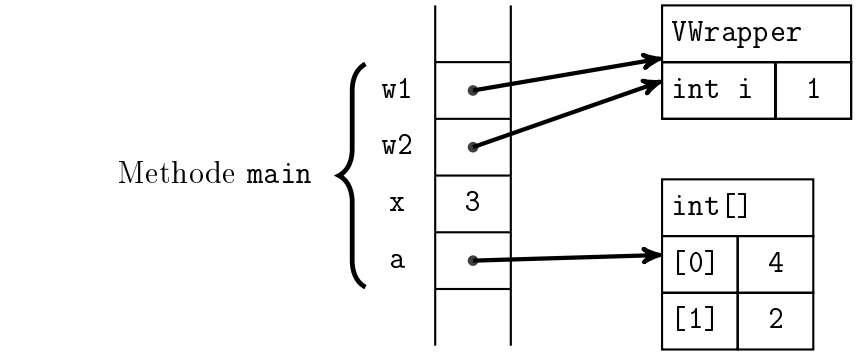
f(w2, x, a)
x = 1

Es wird nun die Methode `main` ausgeführt. Stellen Sie den Speicher an allen drei markierten Programmzuständen graphisch dar. Achten Sie darauf, dass Sie alle (implizit) im Programm vorkommenden Arrays (außer `args`) und alle Objekte sowie die zu dem Zeitpunkt existierenden Programmvariablen darstellen.

Lösung:




Methode main
w1
w2
x
3
a
Methode f
w
x
3
a
VWrapper
int i
4
int[]
[0]
1
[1]
2


Methode main
w1
w2
x
3
a
VWrapper
int i
1
int[]
[0]
4
[1]
2