Aufgabe 1 **(14 Punkte)**

a) Die nachfolgenden Programmfragmente weisen jeweils einen syntaktischen bzw. semantischen Fehler auf, so dass sie **nicht compiliert werden können**. Streichen Sie den Fehler an und begründen Sie kurz!

```
//   1.
     public class SyntaxUndSemantik1 {
       enum Wochentag {MO, DI, MI, DO, FR, SA, SO}
       public static void main (String [] args) {
         for (Wochentag.values() : Wochentag w) {
           System.out.println(w);
         }
       }
     }
```

Begründung:

Zeile 4: Falsche Syntax (umgekehrte Reihenfolge der Komponenten) in vereinfachter FOR-Schleife.

```
//   2.
     interface SyntaxUndSemantik2 {
       String a();
     }
     interface B extends SyntaxUndSemantik2 {
       String a() {}
     }
```

Begründung:

Zeile 5: Kein Methodenrumpf in Schnittstelle erlaubt.

```
//   3.
     import java.util.*;
     public class SyntaxUndSemantik3 {
       public static void main (String [] args) {
         LinkedList<String> list = new LinkedList<String>();
         list.add(new Integer("1"));
       }
     }
```

Begründung:

Zeile 5: Ein `Integer`-Objekt darf nicht in die mit `String` typisierte generische Liste eingefügt werden.

```
//   4.
     public class SyntaxUndSemantik4 {
       private static final int NUMBER = -1;
       public static void main(String[] args) {
         NUMBER++;
         int i = 1 / NUMBER;
       }
     }
```

Begründung:

Zeile 4: Der Operator `++` kann nicht auf eine Konstante (`final`) angewendet werden.

---

b) Die nachfolgenden Programme sind syntaktisch korrekt und compilierbar, führen aber zu einem **fehlerhaften Laufzeitverhalten**. Geben Sie die Art des auftretenden Fehlers an und erklären Sie kurz, wodurch er verursacht wird!

```
//   1.
     public class LaufzeitFehler1 {
       public static void doSomething(int[] feld) {
         while (feld.length > 0)
           doSomethingOther(feld);
       }
       public static void doSomethingOther(int[] feld) {
         int[] args = new int[feld.length - 1];
         for (int i = 0; i < args.length; i++)
           args[i] = feld[i] - 3;
         doSomething(args);
       }
       public static void main(String[] args) {
         doSomething(new int[] { 1, 2, 3 });
       }
     }
```

Art des Laufzeit-Fehlers:

Endlosschleife.

Ursache:

Die Bedingung der WHILE-Schleife wird nie falsch, da sich die Länge des in der lokalen Variable `feld` gespeicherten Felds durch die Aufrufe der Methode `doSomethingOther` nicht ändert.

```
//   2.
     public class LaufzeitFehler2 {
       public static void main(String args) {
         int i = 0;
         System.out.println("1/0 = " + 1.0/i);
         System.out.println("1/0 = " + 1/i);
       }
     }
```

Art des Laufzeit-Fehlers:

NoSuchMethodError

Ursache:

Die Klasse hat keine `main(String[])`-Methode.

Aufgabe 2 **(9 Punkte)**

Gegeben seien die beiden folgenden Klassen `A` und `B`.

```
public class A {
  public void methode(int x, byte y, int z) {
    System.out.println("Methode 1");
  }
  public void methode(int x, byte y, short z) {
    System.out.println("Methode 2");
  }
  public void methode(float x, long y, int z) {
    System.out.println("Methode 3");
  }
  public void methode(float x, long y, short z) {
    System.out.println("Methode 4");
  }
}
public class B extends A {
  public void methode(int x, byte y, short z) {
    System.out.println("Methode 5");
  }
}
```

Geben Sie in der folgenden Tabelle an, ob die Methoden-Aufrufe zulässig oder unzulässig sind! Kreuzen Sie unzulässige Aufrufe nur an, und geben Sie bei zulässigen Aufrufen an, was auf dem Bildschirm ausgegeben wird!

```
public class Test {
  public static void main(String[] args) {
    byte b = 1;            short s = 2;
    int i = 3;             long L = 4;
    float f = 5;           double d = 6;
    A aObjekt = new A();   B bObjekt = new B();
```

	ist unzulässig	ist zulässig und gibt aus:
`bObjekt.methode(i, b, s);`		`Methode 5`
`bObjekt.methode(i+d, L, s);`	×	
`aObjekt.methode(i, i, i);`		`Methode 3`
`aObjekt.methode(s, b, s);`		`Methode 2`
`bObjekt.methode(L, L, L);`	×	
`aObjekt.methode(s, b, s+s);`		`Methode 1`
`bObjekt.methode(f, i, b);`		`Methode 4`
`aObjekt.methode(i, f, i);`	×	
`aObjekt.methode(b, i, s);`		`Methode 4`

```
  }
}
```

Aufgabe 3 **(12 Punkte)**

Zur Speicherung der bei einem physikalischen Experiment gewonnenen Messdaten wird ein Objekt der Klasse

```
class Messung {
    public double[] gewichtungsreihe;   // Gewichtungsfaktor jedes Messwerts
    public int[]    messreihe;          // Messreihe mit Messwerten
    public double   messwertGewichtet;  // gewichteter Messwert
}
```

verwendet. Bei einem befüllten Objekt der Klasse `Messung` sind beide Felder vollständig mit Zahlen belegt und haben dieselbe Länge.

Bitte beachten Sie, dass die Gewichtungsfaktoren in `gewichtungsreihe` durch eine Besonderheit des Versuchsaufbaus leider in **umgekehrter Reihenfolge** abgespeichert wurden.

Vervollständigen Sie die Methode `gewichteterMesswert`, der ein befülltes Objekt vom Typ `Messung` übergeben wird und deren Aufgabe es ist, in der Variablen `messwertGewichtet` des übergebenen Objekts die Summe der Produkte der Messwerte mit ihrem jeweiligen Gewichtungsfaktor abzuspeichern.

```
public class ExperimentAuswertung {

    ...

    public static void gewichteterMesswert ( Messung m ) {


        double result = 0.0;
        int    j;


        for (int i = 0; i < m.messreihe.length; i++) {

            j = m.gewichtungsreihe.length - i - 1;
            result = result + m.messreihe [i] * m.gewichtungsreihe [j];

        }


        m.messwertGewichtet = result;

    }

}
```

Aufgabe 4 **(16 Punkte)**

Gegeben sei folgende Definition des Abstrakten Datentyps `Warteschlange`, in der Ganzzahlen vom Typ `int` in Form einer Warteschlange gespeichert werden:

```
TYPEN
  Warteschlange

FUNKTIONEN
  new() : Warteschlange
  isEmpty() : boolean
  isFull() : boolean
  append (int x)
  getFirst() : int
  removeFirst()

VORBEDINGUNGEN
  @Pre:  append()      : ! isFull()
  @Pre:  getFirst()    : ! isEmpty()
  @Pre:  removeFirst() : ! isEmpty()

AXIOME
  Fuer beliebige x und y vom Typ int gilt:
  Axiom 1: Nach  w = new Warteschlange()
           gilt: w.isEmpty()
  Axiom 2: Nach  w = new Warteschlange()
                 w.append (x)
           gilt: !w.isEmpty()
  Axiom 3: Nach  w = new Warteschlange()
                 w.append (x)
                 w.removeFirst()
           gilt: w.isEmpty()
  Axiom 4: Nach  w = new Warteschlange()
                 w.append (x)
                 w.append (y)
           gilt: w.getFirst() == x
  Axiom 5: Nach  w = new Warteschlange()
                 w.append (x)
                 w.append (y)
                 w.removeFirst ()
           gilt: w.getFirst() == y
```

a) Vervollständigen Sie auf Basis der obigen Definition des Abstrakten Datentyps `Warteschlange` das Interface `WarteschlangeSpec` aus dem Paket `warteschlange`. Ergänzen Sie zusätzlich mit Hilfe der Annotation `@Pre` die Vorbedingungen gemäß des Abschnitts `VORBEDINGUNGEN` des Abstrakten Datentyps `Warteschlange`. Markieren Sie zusätzlich die Methoden mit der `@Pure`-Annotation, die als rein lesende Methoden seiteneffektfrei implementiert werden sollten.

```
package warteschlange;
import jass.modern.*;

public interface WarteschlangeSpec {


    @Pure public boolean isEmpty ();


    @Pure public boolean isFull ();


    @Pre ("!isFull ()")
    public void append ( int x );


    @Pre  ("!isEmpty ()")
    @Pure public int getFirst ();


    @Pre ("!isEmpty ()")
    public void removeFirst ();


}
```

---

b) Vervollständigen Sie nun noch den Unit-Test `WarteschlangeTest` aus dem Paket `warteschlange`, der gemäß des Abschnitts `AXIOME` des Abstrakten Datentyps `Warteschlange` das grundsätzliche Verhalten der Warteschlange überprüft.

```
package warteschlange;
import org.junit.*;
import jass.modern.*;

public class WarteschlangeTest {

    private Warteschlange w = null;
```

```
    @Before public void setUp () {
        w = new Warteschlange ();
    }

    @After public void tearDown () {
        w = null;
    }

    @Test public void axiom1Test () {

        Assert.assertTrue (  w.isEmpty() );
    }

    @Test public void axiom2Test () {

        int x = 1;
        w.append (x);
        Assert.assertFalse (  w.isEmpty() );
    }

    @Test public void axiom3Test () {

        int x = 1;
        w.append (x);
        w.removeFirst ();
        Assert.assertTrue (  w.isEmpty() );
    }

    @Test public void axiom4Test () {
        int x = 1;
        int y = 2;
        w.append (x);
        w.append (y);
        Assert.assertEquals ( x, w.getFirst () );
    }

    @Test public void axiom5Test () {
        int x = 1;
        int y = 2;
        w.append (x);
        w.append (y);
        w.removeFirst ();
        Assert.assertEquals ( y, w.getFirst () );
    }
}
```

Aufgabe 5 **(16 Punkte)**

Was wird bei Ausführung der Methode `main` der Klasse `Tiere` ausgegeben?

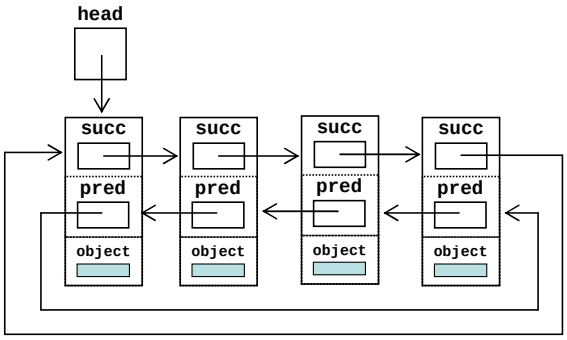
```
public class Maus {
  public int fuss;
  public int[] nase = {3};
  public static int ohr = 2;
  public Maus(int f) {
    fuss = f + ohr++;
  }
}
public class Katze {
  public Maus maus;
  public Katze(Maus tier) {
    maus = tier;
  }
}
public class Tiere {
  public static void methode(int[] a, int b) {
    a[0] = a[0] * 3;
    b    = a[0] - 1;
    System.out.println("T: " + (a[0] + b));
  }
  public static void main(String[] args) {
    Maus x = new Maus(2);
    Maus y = new Maus(1);
    Katze z = new Katze(y);
    y = new Maus(x.fuss);
    Katze u = new Katze(z.maus);
    u.maus.fuss = 0;
    methode(x.nase, x.fuss);
    System.out.print(x.fuss + " A ");
    System.out.print(x.nase[0] + " B ");
    System.out.print(x.ohr + " C ");
    System.out.print(y.fuss + " D ");
    System.out.print(y.nase[0] + " E ");
    System.out.print(y.ohr + " F ");
    System.out.println(z.maus.fuss);
  }
}
```

Ausgabe:

```
T: 17
4 A 9 B 5 C 8 D 3 E 5 F 0
```

Aufgabe 6 **(18 Punkte)**

Bei einer zyklisch doppelt verketteten Liste hat jedes Listen-Element einen Zeiger auf seinen Nachfolger (`succ`) und einen auf seinen Vorgänger (`pred`). Im Gegensatz zu einer gewöhnlichen doppelt verketteten Liste zeigt jedoch das „letzte“ Element wieder auf das „erste“ bzw. umgekehrt, so dass zwei Zyklen entstehen.



Die Klasse `ListCDL` soll eine generische Version einer solchen Liste implementieren.
Vervollständigen Sie die Klasse entsprechend der angegebenen JavaDoc-Kommentare!

```
/**
 * Generische, zyklisch doppelt verkettete Liste (typisiert mit Typ T).
 * Klassenname: ListCDL
 */
public class ListCDL<T> {
   /**
    * Statische innere Klasse für die Listen-Elemente (typisiert mit Typ T)
    * Klassenname: ListElementCDL
    */
   private static class ListElementCDL<T> {
      /** Zeiger auf Vorgänger-Element (typisiert) */
      private ListElementCDL<T> pred;
      /** Zeiger auf Nachfolger-Element (typisiert) */
      private ListElementCDL<T> succ;
      /** eigentlich gespeichertes Objekt (typisiert) */
      private T object;

      /**
       * Konstruktor.
       *
       * @param object Objekt, das im Listen-Element gespeichert werden soll
       */
      private ListElementCDL(T object) {
         this.object = object;
      }
   }
```

```
   /** Kopf-Element der Liste (typisiert) */
   private ListElementCDL<T> head;

   /**
    * Konstruktor. Erzeugt eine neue Liste und fügt ein Listen-Element mit dem
    * zu speichernden Objekt hinzu. Dieses erste Listen-Element zeigt sowohl
    * für seinen Nachfolger als auch für seinen Vorgänger auf sich selbst.
    *
    * @param object Objekt (typisiert), das hinzugefügt werden soll
    */
   public ListCDL(T object) {
      ListElementCDL<T> newElement = new ListElementCDL<T>(object);
      newElement.succ = newElement;
      newElement.pred = newElement;
      head = newElement;
   }

   /**
    * Fügt das angegebene Objekt am Anfang der Liste (Kopf-Element) hinzu.
    * Alle notwendigen Nachfolger- und Vorgänger-Zeiger werden entsprechend
    * angepasst.
    *
    * @param object Objekt (typisiert), das hinzugefügt werden soll
    */
   public void add(T object) {
      ListElementCDL<T> newElement = new ListElementCDL<T>(object);
      newElement.succ = head;
      newElement.pred = head.pred;
      head.pred.succ = newElement;
      head.pred = newElement;
      head = newElement;
   }
}
```

Aufgabe 7 **(16 Punkte)**

Gegeben sei die Klasse `Student` und der Aufzählungstyp `Fach` (siehe nächste Seite).

```
public class Student {
    protected String name;
    private int nummer;
    private Fach fach;
    private static int zaehler;
    public Student () {
        zaehler = zaehler + 1;
        this.setNummer  (1400000 + zaehler);
    }
    public void setName ( String name ) {
        this.name = name;
    }
    public String getName () {
        String result = name;
        return result;
    }
    public void setNummer ( int nummer ) {
        this.nummer = nummer;
    }
    public int getNummer () {
        int result = nummer;
        return result;
    }
    public void setFach ( Fach fach ) {
        this.fach = fach;
    }
    public Fach getFach () {
        Fach result = fach;
        return result;
    }
    public String toString () {
        String result = null;
        result = "Student : name = " + name;
        result = result + " : nummer = " + nummer;
        result = result + " : fach = " + fach;
        return result;
    }
    public static int getZaehler () {
        int result = zaehler;
        return result;
    }
}
```

```
public enum Fach {
    ARCHITEKTUR, BIOLOGIE, GERMANISTIK, GESCHICHTE, INFORMATIK,
    MATHEMATIK, PHYSIK, POLITOLOGIE, WIRTSCHAFTSWISSENSCHAFTEN;
}
```

a) Die Klasse `ArchitekturStudent` erbt von der Klasse `Student`.
Erstellen Sie diese Klasse derart, dass der Konstruktor das gleiche Verhalten aufweist wie für die Klasse `Student` und zusätzlich schon bei der Objekterzeugung das Fach auf den Wert `ARCHITEKTUR` des Aufzählungstyps `Fach` setzt. Überschreiben Sie die Methode `toString()` so, dass sie für ein Objekt der Klasse `ArchitekturStudent` mit dem Namen `Gropius` und der Nummer `1401928` einen String der Form

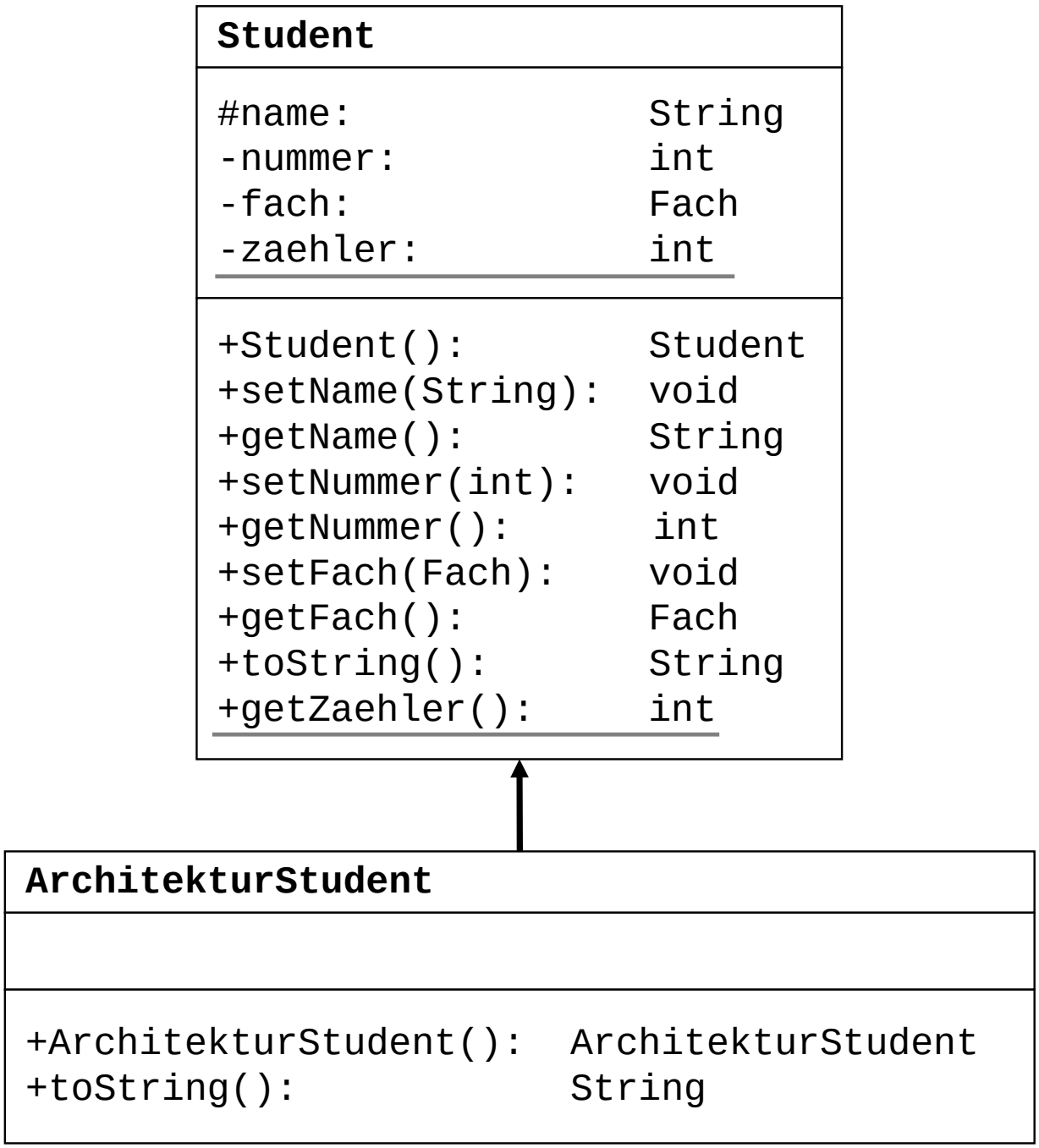
```
Architekturstudent : name = Gropius : nummer = 1401928
```

zurück gibt.

```
public class ArchitekturStudent extends Student {


    public ArchitekturStudent () {

        super ();
        setFach ( Fach.ARCHITEKTUR );

    }


    public String toString () {

        String result = null;

        result = "Architekturstudent : name = " + name;
        result = result + " : nummer = " + getNummer ();

        return result;

    }


}
```

---

b) Erstellen Sie sowohl für die Klasse `Student` als auch für die Klasse `ArchitekturStudent` ein vollständiges UML-Klassendiagramm, das alle Variablen und Methoden mit Ihren Zugriffsrechten darstellt. Klassen-Variablen und Klassen-Methoden werden durch Unterstreichungen kenntlich gemacht. Zeichnen Sie auch die Vererbungsbeziehung zwischen den beiden Klassen ein.



Aufgabe 8 **(9 Punkte)**

Geben Sie an, was beim Ablauf des Programms `Rekursion` ausgegeben wird!

```
public class Rekursion {
  public static int m(int x, int y) {
    System.out.println(x + "//" + y);
    int z;
    if (y == 2) {
      z = 2;
      System.out.println(z);
      return z;
    }
    if (x % 2 == 0) {
      // x gerade
      z = m(x+1, y-2) + y;
    } else {
      // x ungerade
      z = m(x-1, y-2) + 2;
    }
    System.out.println(z);
    return z;
  }

  public static void main(String[] args) {
    System.out.println("m(5, 8) = " + m(5, 8));
  }
}
```

Ausgabe:

```
  5 // 8
  4 // 6
  5 // 4
  4 // 2
  2
  4
  10
  12
  m(5, 8) = 12
```

Aufgabe 9 **(10 Punkte)**

a) Die goldenen Regeln der Namensgebung in Java sind eine grundlegende Voraussetzung, um Variablen, Konstanten, Methoden und Klassen in Java korrekt zu benennen. Streichen Sie in den folgenden Regeln diejenigen **fett gedruckten Worte** durch, die nicht korrekt sind, so dass die gültige Regel übrig bleibt!

---

a1) Klassennamen beginnen immer mit einem **Großbuchstaben**. Setzen sich Klassennamen aus mehr als einem Wort zusammen, wird jedes Wort mit einem **Großbuchstaben** begonnen.

---

a2) Variablen- und Methodennamen beginnen immer mit einem **Kleinbuchstaben**. Setzen sich Variablen- bzw. Methodennamen aus mehr als einem Wort zusammen, wird jedes Wort mit einem **Großbuchstaben** begonnen.

---

a3) Konstanten werden mit **Großbuchstaben** bezeichnet. Setzt sich eine Konstante aus mehreren Worten zusammen, werden diese durch **Unterstriche** getrennt.

---

a4) Methoden, die einen Wert zurückgeben, heißen **Funktionen**. Diese Methoden sollen **genau eine** `return`-Anweisung enthalten. Im Methodenrumpf wird der von der Methode zu ermittelnde Rückgabewert in der Variablen `result` gespeichert.

---

b) Die Ablaufsteuerung von Java-Programmen wird durch Befehle gesteuert, mit denen man den Ablauf von Programmen gezielt beeinflussen kann. Geben Sie zu den folgenden, häufig in Java-Programmen verwendeten Konstrukten jeweils ein passendes Beispiel an! Sie können dabei davon ausgehen, dass die von Ihnen verwendeten Variablen bereits deklariert wurden.

---

b1) `if`-Anweisung mit zweizeiligem Anweisungsblock und zweizeiligem `else`-Block:

```
if ( i < 10 ) {
    System.out.println ("Erhoehe i");
    i = 10;
}
else {
    System.out.println ("Alles O.K.");
    System.out.println ("Keine Aenderungen.");
}
```

---

b2) `for`-Anweisung (normal oder vereinfacht) mit zweizeiligen Anweisungsblock:

```
for ( i = 0; i < 50; i++ ) {
    System.out.println (i);
    System.out.println (2 * i);
}
```

---

b3) `while`-Schleife mit zweizeiligen Anweisungsblock:

```
while ( i < 10 ) {
    System.out.println (i);
    i = i + 1;
}
```

---

b4) `do-while`-Schleife mit zweizeiligen Anweisungsblock:

```
do {
    i = i + 1;
    System.out.println (i);
} while ( i < 10);
```

---

b5) `switch`-Anweisung mit zwei Fallunterscheidungen und einen `default`-Abschnitt:

```
switch (i) {
  case 1:
    j = 17;
    break;
  case 2:
    j = 23;
    break;
  default:
    j = 0;
}
```

---

b6) `try-catch`-Block mit abschließender `finally`-Anweisung:

```
try {
    i = i / j;
}
catch (NumberFormatException e) {
    System.out.println ("Achtung: j == 0");
}
finally {
    System.out.println ("Fertig");
}
```
