Programmierung 2 (SS 2021)
Universität des Saarlandes
Fakultät MI
Compiler Design Lab

Musterlösung 9
Java

Prof. Dr. Sebastian Hack
Julian Rosemann, B. Sc.
Thorsten Klößner, M. Sc.
Julia Wichlacz, M. Sc.
Maximilian Fickert, M. Sc.

Im Vorlesungskalender finden Sie Informationen über die Kapitel des Skripts, die parallel zur Vorlesung bearbeitet werden sollen bzw. dort besprochen werden. Die Übungsaufgaben dienen der Vertiefung des Wissens, das in der Vorlesung vermittelt wird und als Vorbereitung auf Minitests und Klausur.
Weitere Aufgaben zu den Themen finden Sie jeweils am Ende der Skriptkapitel.
Die Schwierigkeitsgrade sind durch Steine des 2048-Spiels gekennzeichnet, von 512 „leicht“ bis 2048 „schwer“. 4096 steht für Knobelaufgaben.

Aufgabe 9.0: Call by What?!

Konrad Klug hat nun auch Java für sich entdeckt. Er möchte herausfinden, ob in Java Argumente per *Call by Value* oder per *Call by Reference* übergeben werden. In dieser Aufgabe sollen Sie Konrad bei der Auswertung seiner Experimente helfen:

1. Was ist die Ausgabe dieses Programms?
2. Was sagt die Ausgabe des Programms über das Übergabeverhalten von Basisdatentypen im Gegensatz zu Referenzdatentypen aus? Werden diese per *Call by Value* oder per *Call by Reference* übergeben?

```java
public class Main {

    public static void main(String[] args) {
        Color color = new Color(0,0,0);

        color.incRed(color);
        color.incValue(color.green);

        System.out.println(color.red);
        System.out.println(color.green);
    }

}
```

```java
public class Color {

    public int red, green, blue;

    public void incRed(Color color) {
        color.red++;
    }

    public void incValue(int value) {
        value++;
    }

    public Color(int red, int green, int blue) {
        this.red = red;
        this.green = green;
        this.blue = blue;
    }

}
```

Lösung

Achtung - Diese Implementierung ist zwar ein gelungenes Beispiel, aber ansonsten nicht sinnvoll implementiert. Normalerweise würde man erwarten, dass `Color::incRed` den Rotwert des Objektes erhöht, auf dem sie aufgerufen wird, und keinen Parameter erwartet. Allgemein machen die gegebenen Methoden im Kontext einer Farbenklasse keinen Sinn.

1. Die Ausgabe des Programms lautet

   ```
   1
   0
   ```

2. Die Referenzen der Referenzdatentypen werden durch *Call by Value* übergeben, wodurch wir einen *Call by Reference* simulieren. Dadurch können wir zwar alle Felder des Objektes ändern, nicht aber das Objekt selbst. Es wird also bei `incRed` eine Referenz auf das Objekt `color` übergeben, das Feld `red` dieses Objektes wird um 1 erhöht. Da Basisdatentypen durch *Call by Value* übergeben werden, wird in der Methode `incValue` der Wert um 1 hochgezählt - allerdings auch nur in dieser Methode.

Aufgabe 9.1: Statische und dynamische Typen

Im Folgenden werden Sie schrittweise an die Auflösung von Funktionsaufrufen in Java herangeführt. Die Codeteile sind als zusammenhängend zu interpretieren, die Aufteilung dient der Verdeutlichung der für den Unterpunkt relevanten Programmteile.

1. Bestimmen Sie den statischen und den dynamischen Typ für alle Variablen.

```
1 A a = new A();
2 A b = new B();
3 A c = new C();
4
5 B bB = new B();
6 C cC = new C();
```

2. Wird bei der Überladung der statische oder der dynamische Typ berücksichtigt? Wie sieht es bei der Überschreibung aus?

3. Bestimmen Sie für jede Klasse die Methoden, die überladen sind und diejenigen, die überschrieben werden.

```
1 class A {
2   void f (boolean b) { System.out.println("A.f(boolean)"); }
3
4   void f (double d) { System.out.println("A.f(double)"); }
5 }
6
7 class B extends A {
8   void f (boolean b) { System.out.println("B.f(boolean)"); }
9
10  void f (int i) { System.out.println("B.f(int)"); }
11 }
12
13 class C extends B {
14  void f (boolean b) { System.out.println("C.f(boolean)"); }
15
16  void f (float f) { System.out.println("C.f(float)"); }
17
18  void f (int i) { System.out.println("C.f(int)"); }
19
20  void f (short s) { System.out.println("C.f(short)"); }
21 }
```

4. Bestimmen Sie nun die Ausgabe der folgenden Aufrufe, indem Sie sich für jeden Aufruf die maximal spezifische Signatur überlegen (unter Berücksichtigung des statischen und dynamischen Typen).

```
 1 cC.f(1);
 2 cC.f(1.0f);
 3
 4 a.f(true);
 5 b.f(true);
 6 c.f(true);
 7
 8 bB.f(1);
 9 b.f(1);
10 cC.f(1.0);
11 c.f((short) 1);
```

Lösung

1.

Variable	statischer Typ	dynamischer Typ
a	A	A
b	A	B
c	A	C
bB	B	B
cC	C	C

2. Überladen statisch, Überschreiben dynamisch.

3.

Klasse	Methoden				
A	A.f(double)	A.f(boolean)			
B		B.f(boolean)	B.f(int)		
C		C.f(boolean)	C.f(int)	C.f(short)	C.f(float)

Stehen Methoden in der Tabelle übereinander, so überschreibt die untere die obere Methode. Zum Beispiel `C.f(int)` überschreibt `B.f(int)`. Methoden mit unterschiedlichem Argumenttyp sind überladen. Beispielsweise `B.f(int)` überlädt `A.f(double)`.

4.

```
cC.f(1);          ⟹ C.f(int)
cC.f(1.0f);       ⟹ C.f(float)

a.f(true);        ⟹ A.f(boolean)
b.f(true);        ⟹ B.f(boolean)
c.f(true);        ⟹ C.f(boolean)

bB.f(1);          ⟹ B.f(int)
b.f(1);           ⟹ A.f(double)
cC.f(1.0);        ⟹ A.f(double)
c.f((short) 1);   ⟹ A.f(double)
```

Die maximal spezifische Signatur kommt bei den Aufrufen `b.f(1))` und `c.f((short) 1)` zu tragen, da hier anhand des statischen Typen `A` die Signatur `f(double)` gewählt wird. Da weder in `B` noch in `C` diese Methode definiert wird, wird die Methode aus der Klasse `A` genommen von der `B` und `C` transitiv erben.

Aufgabe 9.2: Überladen und Überschreiben

Betrachten Sie folgenden Code und begründen Sie, welche Methoden in den Zeilen 20–22 aufgerufen werden. Argumentieren Sie mit Überladung und Überschreibung.

```
 1 class A {
 2   void f (float v) {
 3     System.out.println("A.f(float)");
 4   }
 5 }
 6 class B extends A {
 7   void f (int v) {
 8     System.out.println("B.f(int)");
 9   }
10 }
11 class C extends B {
12   void f (int v) {
13     System.out.println("C.f(int)");
14   }
15 }
```

```
16 A a = new B();
17 B b = new B();
18 B c = new C();
19
20 a.f(15);
21 b.f(15);
22 c.f(15);
```

Lösung

Ausgabe:

```
A.f(float)
B.f(int)
C.f(int)
```

`a.f(15)`: a ist eine Objekt der Klasse B, welches nach A gecastet wurde. Da die Signaturen von B.f und A.f nicht identisch sind, wird A.f nicht überschrieben, sondern überladen. Aus diesem Grund ist in der Klasse A die Methode `void f(int v)` (aus Klasse B) nicht mehr sichtbar. 15 wird deshalb zu float gecastet und die Methode A.f wird aufgerufen und `"A.f(float)"` ausgegeben.
`b.f(15)`: 15 ist standardmäßig eine Ganzzahl (int). Dementsprechend wird die überladene Methode `void f (int v)` aufgerufen und `"B.f(int)"` ausgegeben.
`c.f(15)`: In der Klasse C wird mit C.f die Methode B.f überschrieben. Da Methoden in Java dynamisch gebunden werden, wird die Methode C.f aufgerufen, auch wenn c nach B gecastet wurde. Die Ausgabe ist somit `"C.f(int)"`.

Aufgabe 9.3: Vererbung

Betrachten Sie folgende Java-Klassen; sie beschreiben einige Produkte eines Möbelhauses:

```
 1 class WoodenTable {
 2   private byte   woodspecies;   // 0=unknown, 1=oak, 2=pine, ...
 3   private float  price;         // euro
 4   private float  weight;        // kg
 5   private byte   tshape;        // 0=unknown, 1=rectangular, 2=square, ...
 6   private byte   tlegs;         // table legs; >=3
 7   private byte   tfunc;         // 0=unkown, 1=dining, 2=desk, 3=drawing
 8   // constructor:
 9   public WoodenTable(byte wspecies, float price, float weight,
10                      byte tshape, byte   tlegs, byte   tfunc    ) { ... }
11   public String toString() { ... }
12 }
```

```
class WoodenChair {
  private byte   woodspecies;   // 0=unknown, 1=oak, 2=pine, ...
  private float  price;         // euro
  private float  weight;        // kg
  private byte   chshape;       // 0=unknown, 1=chair,  2=stool, 3=armchair
  private byte   chlegs;        // chair legs; >=3
  public WoodenChair(byte wspecies, float price, float weight,
                     byte   chshape, byte   chlegs            ) { ... }
  public String toString() { ... }
}
```

```
class WoodenCabinet {
  private byte   woodspecies;   // 0=unknown, 1=oak, 2=pine, 3=beech,
  private float  price;         // euro
  private float  weight;        // kg
  private byte   cashape;       // 0=unknown, 1=chair,  2=stool, 3=armchair
  private byte   cafunc;        // 0=unknown, 1=, 2=wardrobe, 3=commode, ...
  public WoodenCabinet(byte wspecies, float price, float weight,
                       byte   cashape, byte cafunc              ) { ... }
  public String toString() { ... }
}
```

Offensichtlich haben alle 3 Klassen einige Eigenschaften gemeinsam, andere sind für verschiedene Klassen unterschiedlich.

1. Notieren Sie eine neue Klasse namens `WoodenFurniture`, welche die gemeinsamen Eigenschaften aufnimmt. Erstellen Sie hierfür zunächst das Interface `WoodenFurnitureInterface`. Ferner:
   - Passen Sie die gegebenen Klassen `WoodenTable`, `WoodenChair` und `WoodenCabinet` so an, dass Sie von der neuen Klasse `WoodenFurniture` die (gemeinsamen) Eigenschaften nun erben.
   - Passen Sie auch die Konstruktoren an und überlegen Sie sich geeignete `toString`-Methoden.
   - Notieren Sie in einer `main`-Methode geeignete Variablen-Deklarationen und Zuweisungen, um ein Objekt jeder neuen Klasse zu kreieren und die Referenz abzuspeichern.
   - Schreiben Sie eine Methode `sumOfPrices` die als Argument eine Referenz auf eine Reihung von `WoodenFurniture` Referenzen nimmt und die Summe der Preise zurückgibt. Schreiben Sie Getter-Methoden, falls nötig.
2. Ausgehend vom Ergebnis in Teil 1 sollen nun in das Sortiment auch Designer-Holzmöbel aufgenommen werden; sie besitzen als weitere Eigenschaft einen Modellnamen (`String modelname`).
   - Führen Sie neue Klassen namens `DesignerWoodenTable`, `DesignerWoodenChair` und `DesignerWoodenCabinet` mit zugehörigen Konstruktoren ein, indem Sie möglichst viele Anteile per Vererbung wiederverwenden. Die `toString`-Methode dieser Klassen soll nun zusätzlich noch den Modellnamen im Rückgabewert darstellen.

Lösung

1.

```
public interface WoodenFurnitureInterface {
  public byte getWoodspecies();

  public float getPrice();

  public float getWeight();

  public String toString();

}
```

```java
public class WoodenFurniture implements WoodenFurnitureInterface {
  private byte woodspecies;
  private float price;
  private float weight;

  public byte getWoodspecies() {
    return woodspecies;
  }
  public float getPrice() {
    return price;
  }
  public float getWeight() {
    return weight;
  }
  public WoodenFurniture(byte woodspecies, float price, float weight) {
    this.woodspecies = woodspecies;
    this.price = price;
    this.weight = weight;
  }

  public String toString() {
    return "Furniture␣of␣wood␣" + woodspecies
            + "␣with␣weight␣" + weight
            + "␣for␣only␣" + price;
  }
}
```

```java
public class WoodenTable extends WoodenFurniture {
  private byte tshape;
  private byte tlegs;
  private byte tfunc;

  public WoodenTable(byte woodspecies, float price, float weight,
                     byte tshape, byte tlegs, byte tfunc) {
    super(woodspecies, price, weight);
    this.tshape = tshape;
    this.tlegs = tlegs;
    this.tfunc = tfunc;
  }
  public String toString () {
    return "Table␣with␣shape␣" + tshape + "␣and␣" + tlegs
           + "␣legs␣for␣activity␣" + tfunc
           + ",␣" + super.toString();
  }
}
```

```java
public class WoodenChair extends WoodenFurniture {
  private byte cshape;
  private byte chlegs;

  public WoodenChair(byte woodspecies, float price, float weight,
                     byte cshape, byte chlegs) {
    super(woodspecies, price, weight);
    this.cshape = cshape;
    this.chlegs = chlegs;
  }
  public String toString() {
    return "Chair␣with␣shape␣" + cshape
           + "␣and␣" + chlegs
           + "␣legs,␣" + super.toString();
  }
```

```
16 }
```

```
1 public class WoodenCabinet extends WoodenFurniture {
2   private byte cashape;
3   private byte cafunc;
4
5   public WoodenCabinet(byte woodspecies, float price, float weight,
6                        byte cashape, byte cafunc) {
7     super(woodspecies, price, weight);
8     this.cashape = cashape;
9     this.cafunc = cafunc;
10  }
11  public String toString() {
12    return "Cabinet␣with␣shape␣" + cashape
13           + "␣and␣function␣" + cafunc
14           + ",␣" + super.toString();
15  }
16 }
```

```
1 public class Main {
2   public static void main(String[] args) {
3     WoodenFurniture[] myFurniture = new WoodenFurniture[3];
4     myFurniture[0] = new WoodenChair((byte)1, 333.0f, 5.0f, (byte)3,
5                                      (byte)1);
6     myFurniture[1] = new WoodenTable((byte)2, 4.0f, 4.2f, (byte)1,
7                                      (byte)4, (byte)2);
8     myFurniture[2] = new DesignerWoodenCabinet((byte)0, 1000.0f,
9                                3.1416f, (byte)0, (byte)2,
10                               new DesignerAnnotation("Billy"));
11
12    System.out.println("My␣furniture:");
13    for (WoodenFurniture f : myFurniture) {
14      System.out.println(f);
15    }
16    System.out.println("Price␣for␣all:␣" + sumOfPrices(myFurniture));
17  }
18
19  public static int sumOfPrices (WoodenFurniture[] furniture) {
20    int result = 0;
21    for (WoodenFurniture x : furniture) {
22      result += x.getPrice();
23    }
24    return result;
25  }
26 }
```

2. Wir notieren zunächst eine Klasse `DesignerAnnotation`, die Modellnamen einheitlich behandelt und Erweiterbarkeit der Designeranmerkungen ohne Anpassung der einzelnen Möbel ermöglicht. Die Aufgabenstellung fordert, `String modelname` als Eigenschaft zu verwenden statt dieser Klasse. Diese (triviale) Lösung steht auskommentiert daneben.

```
1 public class DesignerAnnotation {
2   private String name;
3
4   public DesignerAnnotation(String name) {
5     this.name = name;
6   }
7
8   public String generateString (String oldstr) {
9     return "Special␣Design:␣\"" + name + "\",␣" + oldstr;
10  }
```

```
}
```

```
public class DesignerWoodenTable extends WoodenTable {
  private DesignerAnnotation da;
  // private String modelname;

  public DesignerWoodenTable(byte woodspecies, float price,
          float weight, byte tshape, byte tlegs, byte tfunc,
          DesignerAnnotation da) {
          // String modelname) {
    super(woodspecies, price, weight, tshape, tlegs, tfunc);
    this.da = da;
    // this.modelname = modelname;
  }

  public String toString() {
    return da.generateString(super.toString());
    // return "Model: "+modelname+", "+super.toString();
  }
}
```

```
public class DesignerWoodenChair extends WoodenChair {
  private DesignerAnnotation da;
  // private String modelname;

  public DesignerWoodenChair(byte woodspecies, float price,
          float weight, byte cshape, byte chlegs,
          DesignerAnnotation da) {
          // String modelname) {
    super(woodspecies, price, weight, cshape, chlegs);
    this.da = da;
    // this.modelname = modelname;
  }
  public String toString() {
    return da.generateString(super.toString());
    // return "Model: "+modelname+", "+super.toString();
  }
}
```

```
public class DesignerWoodenCabinet extends WoodenCabinet {
  private DesignerAnnotation da;
  // private String modelname;

  public DesignerWoodenCabinet(byte woodspecies, float price,
          float weight, byte cashape, byte cafunc,
          DesignerAnnotation da) {
          // String modelname) {
    super(woodspecies, price, weight, cashape, cafunc);
    this.da = da;
    // this.modelname = modelname;
  }
  public String toString() {
    return da.generateString(super.toString());
    // return "Model: "+modelname+", "+super.toString();
  }
}
```

Aufgabe 9.4: Scopes

Zu welchen Ausgaben führen die angegebenen Programme?

1.

```
1 int y = 5;
2 for(y = 10; y > 0; y--) {
3     System.out.println(y);
4     y--;
5 }
6 System.out.println(y);
```

2.

```
1 for(int y = 10; y > 0; y--) {
2     System.out.println(y);
3     y--;
4 }
5 System.out.println(y);
```

3. Example.java

```
1 public class Example {
2     int number = 5;
3
4     public void setNumber(int number) {
5         this.number = number;
6         System.out.println(number);
7     }
8 }
```

Main.java

```
1 public class Main {
2     public static void main(String[] args) {
3         Example ex = new Example();
4         ex.setNumber(3);
5     }
6 }
```

4. Example.java

```
1 public class Example {
2     int number = 5;
3
4     public void setNumber(int number) {
5         number = number;
6         System.out.println(this.number);
7     }
8 }
```

Main.java

```
1 public class Main {
2     public static void main(String[] args) {
3         Example ex = new Example();
4         ex.setNumber(3);
5     }
6 }
```

Lösung

1. 10, 8, 6, 4, 2, 0
2. Der letzte Sysout wirft einen Fehler, da y nur in der Schleife deklariert ist.
3. 3
4. 5

Aufgabe 9.5: Automatische Typumwandlung

Überlegen Sie sich für jede der folgenden Zuweisungen, ob Sie erfolgreich ist oder ein Fehler von Java geworfen wird. Um herauszufinden, ob Ihre Lösung stimmt, testen Sie es in einem kleinen Java-Programm.


```java
boolean b = true;
short s = 1;
int i = 0;
long l = 4;
double d = 15.0;

b = (boolean) i;
s = i;
i = (short) d;
l = b;
s = (short) l;
i = s;
```

Lösung

```java
boolean b = true;
short s = 1;
int i = 0;
long l = 4;
double d = 15.0;

// schlägt fehl: boolean ist nicht kompatibel mit anderen Typen
b = (boolean) i;

// schlägt fehl: short < int, deshalb ist keine automatische Typanpassung möglich
s = i;

// der explizite cast kürzt den double zu einem short; danach kann dieser short
// automatisch zu einem int konvertiert werden
i = (short) d;

// schlägt fehl: boolean ist nicht kompatibel mit anderen Typen
l = b;

// der explizite cast ermöglicht es, den long Wert in einen short zu speichern
s = (short) l;

// short kann implizit in int konvertiert werden
i = s;
```

Aufgabe 9.6: Equals

Ändern Sie die Implementierung von `Foo`, sodass sie sich wie gewünscht verhält. Lesen Sie über die Annotation `@Override` nach und führen Sie kurz aus, wie diese Annotation den Fehler hätte vermeiden können.

```
1 class Foo {
2   private int a;
3   public Foo(int a) { this.a = a; }
4   public boolean equals(Foo b) { return a == b.a; }
5   public static void main(String[] args) {
6     Object a = new Foo(1);
7     Object b = new Foo(1);
8     System.out.println(a.equals(b));
9     System.out.println(a.equals(a));
10  }
11 }
```

Lösung

Die equals-Methode von `Foo` wird durch die Implementierung überladen, aber nicht überschrieben, da equals als Argument ein Objekt vom Typ Object erwartet.
Verbesserung:

```
1 public boolean equals(Object b){
2     if( !(b instanceof Foo) || b == NULL ){
3         return false;
4     } else {
5         return a == ((Foo) b).a;
6     }
7 }
```

Die Annotation `@Override` teilt dem Compiler mit, dass die nachfolgende Funktion eine andere überschreibt. Wenn dann trotzdem eine Funktion folgt die nichts überschreibt, weil sie zum Beispiel wie in diesem Fall nur überlädt, erhält man beim Kompilieren einen Fehler.

Aufgabe 9.7: Matrix

In dieser Aufgabe sollen sie eine Klasse inklusive Methoden selbstständig erstellen. Ihre Klasse soll dabei eine Matrix darstellen, die sie im Konstruktor erzeugen und mit verschiedenen Methoden verändern können (Sie dürfen hierbei davon ausgehen, dass alle Inputs gültig sind). Gehen sie dabei wie folgt vor:

1. Erzeugen sie zunächst eine Klasse `Matrix` und einen Konstruktor, der mit gegebener Höhe und Breite ein Integer-Array anlegt.
2. Schreiben sie einen Setter und einen Getter, mit denen sie beliebige Werte der Matrix ablesen oder verändern können.
3. Schreiben sie eine Methode, die das Minimum aus allen Einträgen der Matrix zurückgibt.
4. Schreiben sie eine Methode, die den Durchschnitt aus allen Einträgen berechnet und zurückgibt.

Lösung

```java
public class Matrix {
  int height;
  int width;
  int[][] matrix;

  public Matrix(int height, int width){
    this.height = height;
    this.width = width;
    this.matrix = new int[height][width];
  }

  public void setValue(int h, int w, int value){
    matrix[h][w] = value;
  }

  public int getValue(int h, int w){
    return matrix[h][w];
  }

  public int getMinimum(){
    int value = matrix[0][0];
    for (int i = 0; i < height; i++){
      for (int j = 0; j < width; j++){
        if(matrix[i][j] < value)
          value = matrix[i][j];
      }
    }
    return value;
  }

  public double average(){
    int value = 0;
    int counter = 0;
    for (int i = 0; i < height; i++){
      for (int j = 0; j < width; j++){
        counter++;
        value += matrix[i][j];
      }
    }
    double res = ((double) value) / counter;
    return res;
  }
}
```

Aufgabe 9.8: Vektoren und Methodenverkettung

In dieser Aufgabe schreiben Sie eine Klasse `Vec2`, die Punkte aus $\mathbb{R}^2$ darstellt.

1. Definieren Sie die Klasse `Vec2` mit den privaten Feldern x, y vom Typ `double`. Erstellen Sie einen Konstruktor mit zwei Argumenten vom Typ `double`.

2. Fügen Sie die Methode `toString` in ihre Klasse ein, die einen String zurück gibt, der einen `Vec2` in einem für Menschen leicht lesbaren Format darstellt. Erweitern Sie dazu die folgende Implementierung um die Ausgabe des Feldes y.

```java
    public String toString() {
      return "(" + x + ")";
    }
```

3. Fügen Sie Zugriffsmethoden für x und y hinzu.

4. Testen Sie ihre Implementierung, indem Sie die folgende Methode `main` in ihre Klasse einfügen und ausführen:

```
1   static public void main(String[] args) {
2     // hier deinen Vec2 v erstellen
3     Vec2 v = new Vec2 (1.0 , 2.0);
4     // Folgende Anweisung gibt v gemäß seiner toString Methode aus.
5     System.out.println(v);
6   }
```

   Ihr Programm sollte die Ausgabe `(1.0, 2.0)` erzeugen.

5. Implementieren Sie die Methoden `addAssign` und `mulAssign` mit Rückgabetyp `void` und den folgenden Signaturen:

```
1 addAssign(double scalar)
2 addAssign(final Vec2 point)
3 mulAssign(double scalar)
4 mulAssign(final Vec2 point)
```

   `mulAssign` soll die beiden Vektoren komponentenweise multiplizieren, und nicht das Skalar- oder Kreuzprodukt bilden. Testen Sie die Implementierung, indem Sie zwei Vektoren erstellen, den zweiten auf den ersten addieren, den ersten dann mit 5 multiplizieren, und ausgeben.

6. Die Methoden aus dem vorherigen Aufgabenteil hatten den Rückgabetyp `void`. In diesem Aufgabenteil ändern wir die Methoden so ab, dass wir die Berechnung $v * 5 + w$ mit `v.mulAssign(5).addAssign(w)` bewerkstelligen können. Dieses Programmieridiom wird Methodenverkettung (engl. *method chaining*) genannt. Passen Sie `mulAssign` und `addAssign` so an, dass Sie den Vec2 selbst zurückgeben. Stellen Sie sicher, dass ihr geändertes Programm nach wie vor die selbe Ausgabe erzeugt.

7. Implementieren Sie einen Copy-Konstruktor, der gegeben eine Instanz `v` von Vec2 ein neues Objekt mit den gleichen Koordinaten wie `v` erzeugt. Testen Sie dies in ihrer `main` Methode.

Lösung

```java
public class Vec2 {

  // Aufgabe 1.
  private double x,y;

  public Vec2(double x, double y) {
    this.x = x;
    this.y = y;
  }

  // Aufgabe 2.
  @Override
  public String toString() {
    return "(" + x + ",␣" + y + ")";
  }

  // Aufgabe 3.
  public double getX() {
    return this.x;
  }

  public void setX(double x) {
    this.x = x;
  }

  public double getY() {
    return this.y;
  }

  public void setY(double y) {
    this.y = y;
  }

  @Override
  public boolean equals(Object o) {
    if (!(o instanceof Vec2))
      return false;
    Vec2 v = (Vec2) o;
    return v.getX() == getX() && v.getY() == getY();
  }

  public static void main(String[] args) {
    // Aufgabe 4.
    Vec2 v = new Vec2(1.0, 2.0);
    System.out.println(v);

    // Test zu Aufgabe 5. starten.
    test();

    // Test zu Aufgabe 7.
    Vec2 v2 = new Vec2(v);
    if (v.equals(v2)) {
      System.out.println("Korrekt.␣Der␣Klon␣ist␣gleich.");
    } else {
      System.out.println("Fehler.␣Objekt␣ist␣ungleich.");
    }
  }

  // Aufgabe 5.
```

```
60   public void addAssign(double scalar) {
61     this.x += scalar;
62     this.y += scalar;
63   }
64
65   public void addAssign(final Vec2 point) {
66     this.x += point.x;
67     this.y += point.y;
68   }
69
70   public void mulAssign(double scalar) {
71     this.x *= scalar;
72     this.y *= scalar;
73   }
74
75   public void mulAssign(final Vec2 point) {
76     this.x *= point.x;
77     this.y *= point.y;
78   }
79
80   // Testen der Implementierung von Aufgabe 5.
81   public static void test() {
82     Vec2 v1 = new Vec2(5.2, 3.2);
83     Vec2 v2 = new Vec2(3.2, 5.2);
84     v1.addAssign(v2);
85     v1.mulAssign(5);
86     System.out.println(v1);
87   }
88
89   // Aufgabe 7.
90   public Vec2(Vec2 v) {
91     this(v.x, v.y);
92   }
93 }
```

In Aufgabe 6 werden die Befehle addAssign und mulAssign folgendermaßen geändert:

```
 1 public Vec2 addAssign(double scalar) {
 2   this.x += scalar;
 3   this.y += scalar;
 4   return this;
 5 }
 6
 7 public Vec2 addAssign(final Vec2 point) {
 8   this.x += point.x;
 9   this.y += point.y;
10   return this;
11 }
12
13 public Vec2 mulAssign(double scalar) {
14   this.x *= scalar;
15   this.y *= scalar;
16   return this;
17 }
18
19 public Vec2 mulAssign(final Vec2 point) {
20   this.x *= point.x;
21   this.y *= point.y;
22   return this;
23 }
24
25 // Test für den Aufgabenteil 6. gibt das selbe aus wie für Aufgabenteil 5.
```

```
26 public static void test2() {
27   Vec2 v = new Vec2(5.2, 3.2);
28   Vec2 w = new Vec2(3.2, 5.2);
29   System.out.println(v.addAssign(w).mulAssign(5));
30 }
```

Aufgabe 9.9: Referenzen

Implementieren Sie eine Klasse `Rectangle`, die ein Rechteck mithilfe von zwei `Vec2` Objekten darstellt. Implementieren Sie einen geeigneten Konstruktor. Schreiben sie Zugriffsmethoden für den Punkt links oben und den Punkt rechts unten.
Testen Sie ihr Programm nach dem folgenden Schema, um sicherzustellen, dass die Klasse `Rectangle` ihre internen Daten richtig kapselt:

```
1 Vec2 v = new Vec2(1.0, 2.0);
2 Vec2 w = new Vec2(3.0, 4.0);
3 Rectangle r = new Rectangle(v,w);
4 System.out.println(r);
5 v.setX(10.0);
6 System.out.println(r);
7 r.getTopLeft().setX(10.0);
8 System.out.println(r);
```

Die folgende Ausgabe sollte erzeugt werden:

```
1 Rectangle: (1.0, 2.0) to (3.0, 4.0)
2 Rectangle: (1.0, 2.0) to (3.0, 4.0)
3 Rectangle: (1.0, 2.0) to (3.0, 4.0)
```

Lösung

```java
public class Rectangle {
  private Vec2 topLeft;
  private Vec2 botRight;

  /***
   * Stellt ein Rechteck im zweidimensionalen Raum dar.
   * @param topLeft - obere linke Ecke
   * @param botRight - untere rechte Ecke
   */
  public Rectangle(Vec2 topLeft, Vec2 botRight) {
    this.topLeft = new Vec2(topLeft);
    this.botRight = new Vec2(botRight);
  }

  public Vec2 getTopLeft() {
    // clone the Vec2 so noone can modify the rectangle
    return new Vec2(topLeft);
  }

  public Vec2 getBotRight() {
    // clone the Vec2 so noone can modify the rectangle
    return new Vec2(botRight);
  }

  public static void main(String[] args) {
    Vec2 v = new Vec2(1.0, 2.0);
    Vec2 w = new Vec2(3.0, 4.0);
    Rectangle r = new Rectangle (v, w);
    System.out.println(r);
    v.setX(10.0);
    System.out.println(r);
    r.getTopLeft().setX(10.0);
    System.out.println(r);
  }

  // Damit man ausreichend Informationen sieht, sollte folgende
  // Methode zum Ausgeben hinzugefügt werden.
  @Override
  public String toString() {
    return "Rectangle:␣" + this.topLeft + "␣to␣" + this.botRight;
  }
}
```

Aufgabe 9.10: Statische Methoden

In Aufgabe haben wir die Addition und Multiplikation für `Vec2` so definiert, dass das Objekt selbst verändert wurde. Jetzt implementieren wir statische Methoden, die jeweils ein neues Objekt zurückgeben.

1. Lesen Sie im Internet nach, was das Schlüsselwort `static` bedeutet und welche Auswirkungen dies auf die Methode hat.

2. Implementieren Sie `add`, `mul`, und `mod` nach dem folgenden Schema:

   ```java
   public static Vec2 add(final Vec2 v, final Vec2 w) {
     //TODO
   }
   ```

   Die Methoden sollen v und w jeweils unverändert lassen und ein neues `Vec2` Objekt zurückgeben. Testen Sie in ihrer `main`, dass `Vec2.mul(v,w)` die `Vec2` Objekte v und w tatsächlich nicht verändert.

3. Implementieren Sie jetzt eine weitere statische Methode `rotate`, die zu einem `Vec2` v und einem `double` a einen neuen Vektor liefert, der den um a rotierten Vektor v darstellt. Der Winkel a soll im Bogenmaß angegeben werden. Es kann hilfreich sein, sich über die Bibliotheksklasse `java.lang.Math` zu informieren.

Lösung

1. Zur Klasse Vec2 müssen folgende Methoden hinzugefügt werden:

```
1 public static Vec2 add(final Vec2 v, final Vec2 w) {
2   return new Vec2(v.x + w.x, v.y + w.y);
3 }
4 public static Vec2 add(final Vec2 v, final double scalar) {
5   return new Vec2(v.x + scalar, v.y + scalar);
6 }
7 public static Vec2 mul(final Vec2 v, final Vec2 w) {
8   return new Vec2(v.x * w.x, v.y * w.y);
9 }
10 public static Vec2 mul(final Vec2 v, final double scalar) {
11   return new Vec2(v.x * scalar, v.y * scalar);
12 }
13 public static Vec2 mod(final Vec2 v, final double scalar) {
14   return new Vec2(v.x % scalar, v.y % scalar);
15 }
```

   Statische Methoden gehören in Java zu einer Klasse an sich anstatt einer Instanz der Klasse. Das heißt, dass die Methode ist aufrufbar durch `Vec2.add(...);` anstatt durch `Vec2 inst = new Vec2(); inst.add(...);`.

2. Zur Klasse Vec2 muss folgende Methode hinzugefügt werden:

```
1 public static Vec2 rotate(final Vec2 v, double a) {
2   // Rotation im zweidimensionalen Raum.
3   double x = v.x * Math.cos(a) - v.y * Math.sin(a);
4   double y = v.x * Math.sin(a) + v.y * Math.cos(a);
5   return new Vec2(x,y);
6 }
```

Aufgabe 9.11: Buggy

Unter Materialien finden Sie das Projekt *Buggy*. In der Datei `BuggyClass.java` haben sich Fehler eingeschlichen. Ihre Aufgabe ist es alle Errors im Projekt zu beseitigen, sodass man es ausführen kann und dabei die folgende Ausgabe bekommt:

```
Magic value: 42
Magic value: 42
Equal: true
```

Der Rumpf der `main` Methode und die Dateien `ChildOfBuggyClass.java` und `InterfaceForBuggyClass.java` sollen dabei nicht verändert werden. Insgesamt gibt es 5 Fehler im Projekt.
Bei einigen Fehlern kann Ihnen Eclipse weiterhelfen. Diese Hilfe anzunehmen ist natürlich erlaubt und sinnvoll, hinterfragen Sie aber immer, ob die Vorschläge auch semantisch das tun, was Sie erreichen möchten.

Lösung

- Das Feld `magic` ist `private`, wodurch es von der Unterklasse nicht verfügbar ist. Man sollte die Sichtbarkeit zu `public`, `protected` setzen oder den Spezifizierer weglassen (`getter` und `setter` implementieren wäre auch möglich, allerdings müsste man dafür Code in der Unterklasse anpassen).
- Der `main` Methode fehlt der Parameter `String[] args`.

- Damit das Interface erfüllt ist, muss eine weitere Methode implementiert werden:

```
1 @Override
2 public int subtractNumbers(int num1, int num2) {
3   return num1 - num2 + this.magic;
4 }
```

- Die `equals` Methode sollte ein `Object` nehmen, um die allgemeine `equals` Methode sinnvoll zu überschreiben.

```
1 @Override
2 public boolean equals(Object o) {
3
4   if(!(o instanceof BuggyClass)) {
5     return false;
6   }
7
8   BuggyClass other = (BuggyClass) o;
9   if (other.magic == this.magic) {
10    return true;
11  }
12
13  return false;
14 }
```

- Eine Erweiterung um die Methode `hashCode` ist jetzt auch notwendig, da sonst der Vertrag zwischen `hashCode` und `equals` verletzt wird.

```
1 @Override
2 public int hashCode() {
3   return this.magic;
4 }
```