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

Tutoraufgabe 1 (Überblickswissen):

a) Welche Methoden kann man nicht nur auf Objekten, sondern auch auf Klassen aufrufen? Wann ergibt das Sinn?

b) Gibt es einen Unterschied zwischen `f(int... args)` und `f(int[] args)`? Wenn es einen gibt, worin besteht er?

c) Warum ist es gerade bei Attributen sinnvoll, diese, soweit möglich, mit dem `final`-Schlüsselwort zu deklarieren?

d) In der Vorlesung wurde das neue Java-Feature `record`-Klassen eingeführt. Was sind die Vorteile und was sind die Nachteile von diesen Klassen gegenüber gewöhnlichen Klassen?

Lösung:

a) Methoden, die mit dem Schlüsselwort `static` gekennzeichnet sind, weichen davon ab, dass Attribute nur für Objekte wirklich mit Werten belegt sind und dass man Methoden nur auf Objekten aufrufen kann. Statische Methoden kann man auch auf der Klasse aufrufen, d.h. ohne dass es dafür ein Objekt dieser Klasse braucht. Das ergibt immer dann Sinn, wenn die Funktionalität der Methode nichts mit einem bestimmten Objekt bzw. dessen Attributen zu tun hat. Trotzdem sollte die Methode natürlich in einem Bezug zur Klasse stehen, sonst sollte sie an anderer Stelle stehen. Damit aus dem Aufruf klar wird, dass gerade eine statische Methode aufgerufen wird, sollte man statische Methoden immer auf der Klasse und nicht auf einem Objekt der Klasse aufrufen.
Ein gutes Beispiel sind Factory-Methoden, die eine neue Instanz der Klasse zurückgeben: Offenbar muss man sie nicht auf einem Objekt der Klasse aufrufen, denn ein solches will man mit der Methode ja erst erzeugen. Auch ist der Bezug zur Klasse klar, da ja Objekte der Klasse erzeugt werden. Die `main`-Methode einer Klasse ist übrigens immer statisch.

b) Beide Methodenköpfe sind sich sehr ähnlich, denn die varargs-Schreibweise `int...` wird intern in ein `int`-Array übersetzt. Deshalb darf man auch nur eine der beiden Methoden deklarieren, nicht beide zugleich. Dennoch gibt es einen Unterschied beim Aufruf der Methode: Eine varargs-Methode kann ich mit beliebig vielen Objekten des angegeben Typs (hier `int`) aufrufen. Eine Methode, die ein Array erwartet, muss dagegen auch ein solches übergeben bekommen. Habe ich nur die einzelnen Objekte, muss ich das Array vorher selbst erstellen. Diese Arbeit wird mir abgenommen, wenn ich die varargs-Methode aufrufe.

c) Allgemein ist es sinnvoll, im Quellcode zu notieren, dass man als Entwickler annimmt, dass eine Variable/ein Parameter/ein Attribut seinen Wert nach der ersten Zuweisung nicht mehr ändert. In Java geschieht dies über das Schlüsselwort `final`. Wenn der Wert eines Attributes geändert werden soll, so bedeutet dies dann, dass man hierfür ein neues Objekt braucht.

Es ist aus zwei Gründen sinnvoll, diese Information im Quellcode zu verankern. Erstens überprüft der Compiler so, ob tatsächlich nach der ersten Zuweisung keine weiteren Zuweisungen mehr stattfinden. Ist dies doch der Fall, wird ein Compilerfehler generiert. Es passiert also nicht so schnell, dass man versehentlich den Wert einer Variablen ändert, der eigentlich nicht mehr geändert werden sollte, denn dazu muss man explizit das `final`-Schlüsselwort von der Deklaration entfernen. Das `final`-Schlüsselwort drückt also die Designentscheidung aus, die Variable nach der ersten Zuweisung nicht erneut zuzuweisen, und der Compiler hilft uns dabei, diese Designentscheidung nicht versehentlich zu ändern.

Zweitens kann man sich während der Entwicklung eines Programms auf diese Garantie verlassen. Wenn die Deklaration das `final`-Schlüsselwort enthält, so ist es schlicht nicht möglich, dass sich der Wert der Variablen nach der ersten Zuweisung noch einmal ändert, da der Compiler dies sicherstellt. Beim Lesen des Quellcodes muss der Entwickler also nicht selbst suchen, ob die Variable irgendwo erneut zugewiesen wird, was gerade bei größeren Methoden oft nicht offensichtlich ist. Es genügt die Deklaration zu betrachten. Daher erhöht die Nutzung von `final` die Lesbarkeit von Quellcode.

Aufgrund des größeren Scopes von Attributen ist hier die Nutzung des `final`-Schlüsselworts besonders hilfreich. Wurde es nicht genutzt, so muss der Entwickler zunächst die ganze Klasse lesen, um herauszufinden, ob das Attribut irgendwo neu zugewiesen wird.

Tatsächlich ist diese Markierung so hilfreich, dass in jüngeren Sprachen alle unmarkierten Variablen `final` sind. Falls es hingegen möglich sein soll, nach der ersten Zuweisung einen neuen Wert zuzuweisen, so muss die Variable explizit markiert werden, beispielsweise als `mut` (für mutable).

d) Der große Vorteil von `record`-Klassen ist, dass viele Methoden wie Getter, `toString` oder `equals` nicht mehr manuell implementiert werden müssen. Mittlerweile können viele Entwicklungsumgebungen diese zwar automatisch generieren. Möchte man aber z.B. nachträglich ein Attribut hinzufügen oder ein Attribut entfernen, so muss man die von der Entwicklungsumgebung automatisch generierten Methoden manuell abändern. Dies ist bei `record`-Klassen nicht nötig. Da die oben genannten Methoden nicht mehr implementiert werden müssen, ist der Code in den meisten Fällen auch kürzer und übersichtlicher. Der Nachteil gegenüber "Standard"-Klassen ist die fehlende Flexibilität. Insbesondere sind alle Attribute einer `record`-Klasse `final`.

Tutoraufgabe 2 (Einfache Klassen):

In dieser Aufgabe beschäftigen wir uns mit den beiden Freunden *Pettersson und Findus*. Wenn die beiden nicht gerade eine von Findus' verrückten Ideen in die Tat umsetzen, gehen sie gerne im Wald Pilze sammeln. Jeder dieser (beiden) Pilzsammler hat einen Korb, in den eine feste Anzahl von Pilzen passt. Weiterhin hat jeder Pilzsammler einen Namen. Wie Sie in der Klasse `Pilzsammler` sehen können, gibt es hierfür drei Attribute. Das Attribut `anzahl` gibt hierbei an, wie viele Pilze bereits im Korb enthalten sind.
Zu jedem `Pilz` kennen wir die Pilzart, von denen es in dieser Aufgabe genau vier gibt: Champignons, Hallimasche, Pfifferlinge und Steinpilze.

Hinweise:

- Wir verwenden hier die Klassen `Main`, `Pilzsammler`, `Pilzart` und `Pilz`, die Sie im Moodle-Lernraum herunterladen können.
- Um sich zunächst auf die Erstellung der Klassen zu konzentrieren, müssen Sie in dieser Aufgabe die Prinzipien der Datenkapselung noch nicht beachten.

a) Schreiben Sie eine Record-Klasse `Pilz` an der Stelle `TODO a)`. Ein `Pilz` hat hierbei die beiden Attribute

- `Pilzart art` und
- `boolean reif`. Letzteres Attribut ist genau dann `true`, wenn der Pilz reif ist.

b) Vervollständigen Sie die Klasse `Main` wie folgt:

- Ergänzen Sie an den mit `TODO b.1)` markierten Stellen den Code so, dass die Variablen `steinpilz1`, `steinpilz2`, `champignon` und `pfifferling` auf unterschiedliche Pilz-Objekte der passenden Pilzart verweisen. Bis auf den Champignon sind alle vier Pilze unreif.
- Ergänzen Sie an den mit `TODO b.2)` markierten Stellen den Code so, dass die Variablen `pettersson` und `findus` auf passende Pilzsammler-Objekte zeigen. Setzen Sie hierfür jeweils den passenden Namen und sorgen Sie dafür, dass in Findus' Korb maximal 7 Pilze Platz haben. Bei Pettersson passen 8 Pilze in den (noch leeren) Korb.
- Nun stellen Pettersson und Findus fest, dass auch der erste Steinpilz reif ist. Da jedoch die Attribute einer Record-Klasse `final` sind, erzeugen Sie ein neues Pilz-Objekt an der mit `TODO b.3)` markierten Stelle.

c) Gehen Sie in dieser und den folgenden Teilaufgaben davon aus, dass die Attribute der Objekte bereits alle auf vernünftige Werte gesetzt sind. In dieser Aufgabe soll jeder leere Platz im `Pilz`-Array `korb` ein `null`-Element enthalten. In diesem Fall betrachten wir also auch `null` als vernünftigen Wert.

- Ergänzen Sie die Klasse `Pilzsammler` um eine Methode `hatPlatz()`, die genau dann `true` zurückgibt, wenn im Korb Platz für einen weiteren Pilz ist. Anderenfalls wird `false` zurückgegeben.

- Schreiben Sie in der Klasse `Pilzsammler` eine Methode `ausgabe()`. Diese gibt kein Ergebnis zurück, aber sie gibt den Namen und eine lesbare Übersicht der von der Person gesammelten Pilze aus. Geben Sie in der ersten Zeile den Namen der Person gefolgt von der Anzahl der gesammelten Pilze und der Größe des Korbes getrennt durch einen Schrägstrich („/“) in Klammern und einem abschließenden Doppelpunkt („:“) aus. Schreiben Sie pro Pilz im Korb eine weitere Zeile, in der die Art und der Reifegrad des jeweiligen Pilzes steht.

  Eine Beispielausgabe von einer Pilzsammlerin mit Namen „Prillan“ und einem Korb der Größe 5, der einen unreifen Pilz der Art „Hallimasch“ und einen reifen Pilz der Art „Steinpilz“ enthält, könnte folgendermaßen aussehen:

  ```
  Prillan(2/5):
  Pilz[art=HALLIMASCH, reif=false]
  Pilz[art=STEINPILZ, reif=true]
  ```

Hinweise:

- Da Sie eine `record`-Klasse verwenden, steht eine sinnvolle `toString`-Methode automatisch zur Verfügung. Verwenden Sie diese, um die Pilze als Strings auszugeben.

d) Ergänzen Sie die von Ihnen zuvor geschriebene Record-Klasse `Pilz` um die Methode `pilzlichtung` mit dem Methodenkopf `public static Pilz[] pilzlichtung()`. Auf einer Lichtung wachsen die verschiedenen Pilze jeweils genau einmal.
Die Methode `pilzlichtung` soll ein Array mit Elementen vom Typ `Pilz` zurückgeben. Dabei soll von jeder `Pilzart` genau ein Pilz im Array enthalten sein. Ein Pilz auf einer Lichtung ist immer reif. Gestalten Sie Ihre Implementierung so, dass die Ausgabe auch dann noch korrekt ist, wenn sich die zugrundeliegende `enum`-Klasse `Pilzart` ändert.

e) Ergänzen Sie die von Ihnen zuvor geschriebene Record-Klasse `Pilz` um die Methode `anzahlUnreif` mit dem Methodenkopf `public static int anzahlUnreif(Pilz[] pilze)`. Diese Methode bekommt ein Array von Pilzen und gibt die Anzahl der unreifen Pilze zurück, welche in diesem Array enthalten sind. Das Array kann das Objekt `null` beinhalten. Ein solches Objekt ist niemals reif.

Hinweise:

- Da Sie eine `record`-Klasse verwenden, stehen sinnvolle Getter-Methoden automatisch zur Verfügung. Für das Attribut `reif` wäre dies zum Beispiel `public boolean reif()`.
- Beachten Sie hier und im Folgenden, dass auf das Objekt `null` keine solche Methode angewendet werden kann.

f) In der Klasse `Pilzsammler` sehen Sie den Kopf einer Methode `public Pilz[] sammlePilze(Pilz... pilze)`. Beim Aufruf dieser Methode sollen die als Parameter übergebenen Pilze vom Pilzsammler gesammelt werden, auf dem die Methode aufgerufen wurde. Alle Pilze, für die der Pilzsammler leider keinen Platz mehr in seinem Korb hat oder die nicht reif sind, sollen zurückgegeben werden.
Schreiben Sie an die mit `TODO f)` markierte Stelle den Rumpf der Methode. Diese soll für jeden Pilz im Array `pilze` prüfen, ob noch Platz im Korb ist und ob der Pilz reif ist. Wenn das der Fall ist, soll der Pilz dem Korb des Pilzsammlers hinzugefügt werden und die Anzahl der gesammelten Pilze um eins erhöht werden. Außerdem soll mittels `System.out.println` eine Ausgabe der Form `"Pettersson sammelt einen Pilz[art=STEINPILZ, reif=true]"` erfolgen, wobei statt `"Petersson"` der Name des Pilzsammlers und statt `"STEINPILZ"` der Name des soeben gesammelten Pilzes stehen soll. Wenn im Korb kein Platz mehr ist oder der Pilz nicht reif ist, soll der Pilz dem Array hinzugefügt werden, das am Ende zurückgegeben wird. Außerdem soll wie oben mittels `System.out.println` eine Ausgabe der Form `"Pettersson nimmt Pilz[art=STEINPILZ, reif=true] nicht mit."` erfolgen.
Das zurückgegebene Array soll keine `null`-Elemente enthalten. Überlegen Sie, wie sich bestimmen lässt, für wie viele Elemente es Platz bieten muss.

g) Nun schicken wir Pettersson und Findus auf Pilzsammlung. Da Findus viel schneller als Pettersson ist, versucht er immer als erstes, neu gesichtete Pilze einzusammeln. Danach kommt Pettersson dazu und sammelt die Pilze auf, die nicht mehr bei Findus in den Korb gepasst haben. Die beiden beenden ihre Pilzsammlung erst, wenn keiner mehr Platz in seinem Korb hat. Pettersson und Findus sammeln hierbei nur reife Pilze ein. Dabei finden Sie zuerst die Pilze, die in Teilaufgabe a) erstellt worden sind. Danach entdecken Sie solange neue Pilzlichtungen, bis die Pilzsammlung endet.

Schreiben Sie an die mit `TODO g)` markierte Stelle u.a. eine Schleife, die dieses Vorgehen abbildet.

Rufen Sie vor Beginn der Schleife und am Ende jeder Schleifeniteration die Methode `ausgabe()` zuerst für Findus und anschließend für Pettersson auf. Geben Sie anschließend jeweils eine Zeile aus, in der nur „---“ (drei Bindestriche) steht.

Listing 1: Main.java

```
public class Main {
  public static void main(String[] args) {
    Pilz steinpilz1 = // TODO b.1)

    Pilz steinpilz2 = // TODO b.1)

    Pilz champignon = // TODO b.1)

    Pilz pfifferling = // TODO b.1)

    Pilzsammler pettersson = // TODO b.2)

    Pilzsammler findus = // TODO b.2)

    steinpilz1 = //TODO b.3)

    // TODO g)
  }
}
```

Listing 2: Pilzart.java

```
enum Pilzart {
    CHAMPIGNON, HALLIMASCH, PFIFFERLING, STEINPILZ;
  }
```

Listing 3: Pilz.java

```
//TODO a) Record-Klasse Pilz
//TODO d) public static Pilz[] pilzlichtung()
//TODO e) public static int anzahlUnreif(Pilz[] pilze)
```

Listing 4: Pilzsammler.java

```
public class Pilzsammler {
  String name;
  Pilz[] korb;
  int anzahl = 0;

  public Pilz[] sammlePilze(Pilz... pilze) {
    //TODO f)
  }

  public boolean hatPlatz() {
    //TODO c.1)
  }

  public void ausgabe() {
    //TODO c.2)
  }
}
```

Lösung:

Listing 5: Main.java

```
public class Main {
  public static void main(String[] args) {
    Pilz steinpilz1 = new Pilz(Pilzart.STEINPILZ, false);

    Pilz steinpilz2 = new Pilz(Pilzart.STEINPILZ, false);

    Pilz champignon = new Pilz(Pilzart.CHAMPIGNON, true);

    Pilz pfifferling = new Pilz(Pilzart.PFIFFERLING, false);

    Pilzsammler pettersson = new Pilzsammler();
    pettersson.name = "Pettersson";
    pettersson.korb = new Pilz[8];

    Pilzsammler findus = new Pilzsammler();
    findus.name = "Findus";
    findus.korb = new Pilz[7];

    steinpilz1 = new Pilz(Pilzart.STEINPILZ, true);

    // TODO g)
    Pilz[] uebrigePilze = findus.sammlePilze(steinpilz1,steinpilz2,champignon,pfifferling);
    pettersson.sammlePilze(uebrigePilze);
    findus.ausgabe();
    pettersson.ausgabe();
    System.out.println("---");

    while (findus.hatPlatz() || pettersson.hatPlatz()) {
      Pilz[] neueLichtung = Pilz.pilzlichtung();
      uebrigePilze = findus.sammlePilze(neueLichtung);
      pettersson.sammlePilze(uebrigePilze);
      findus.ausgabe();
      pettersson.ausgabe();
      System.out.println("---");
    }
  }
}
```

Listing 6: Pilzart.java

```
enum Pilzart {
  CHAMPIGNON, HALLIMASCH, PFIFFERLING, STEINPILZ;
}
```

Listing 7: Pilz.java

```
//TODO a) Record-Klasse Pilz
public record Pilz (Pilzart art, boolean reif) {
  //TODO d) public static Pilz[] pilzlichtung()
  public static Pilz[] pilzlichtung() {
    Pilzart[] pilzarten = Pilzart.values();
    Pilz[] res = new Pilz[pilzarten.length];
    for (int i = 0; i < pilzarten.length; ++i) {
      res[i] = new Pilz(pilzarten[i], true);
    }
    return res;
  }

  //TODO e) public static int anzahlUnreif(Pilz[] pilze)
  public static int anzahlUnreif(Pilz[] pilze) {
```

```
    int anzahl_unreif = 0;
    for (Pilz pilz : pilze) {
        if (pilz == null || !pilz.reif())
            anzahl_unreif++;
    }
    return anzahl_unreif;
  }
}
```

Listing 8: Pilzsammler.java

```
public class Pilzsammler {
  String name;
  Pilz[] korb;
  int anzahl = 0;

  public Pilz[] sammlePilze(Pilz... pilze) {
    int anzahlUnreif = Pilz.anzahlUnreif(pilze);
    int restReifePilze = (pilze.length - anzahlUnreif) - (korb.length - anzahl);
    if (restReifePilze < 0) {
      restReifePilze = 0;
    }
    Pilz[] rest = new Pilz[restReifePilze + anzahlUnreif];

    int iRest = 0;
    for (Pilz pilz : pilze) {
      if(pilz != null) {
        if(hatPlatz() && pilz.reif()) {
            System.out.println(name + " sammelt einen " + pilz.toString());
                korb[anzahl] = pilz;
                ++anzahl;
          }
          else {
            System.out.println(name + " nimmt " + pilz.toString() + " nicht mit.");
            rest[iRest] = pilz;
            ++iRest;
        }
      }
    }
    return rest;
  }

  public boolean hatPlatz() {
    return anzahl < korb.length;
  }

  public void ausgabe() {
    System.out.println(name + "(" + anzahl + "/" + korb.length + "): ");
    for (Pilz pilz : korb) {
      if (pilz != null) {
        System.out.println(pilz.toString());
      }
    }
  }
}
```

Aufgabe 3 (Einfache Klassen): (2 + 2 + 4 + 3 + 4 + 3 + 2 + 6 + 2 + 2 = 30 Punkte)

In dieser Aufgabe geht es um das bekannte Kartenspiel Mau-Mau. Dieses Spiel wird mit einem Skatblatt aus 32 Karten gespielt: Es gibt acht verschiedene Werte (Sieben, Acht, Neun, Zehn, Bube, Dame, König und Ass)

in vier verschiedenen sog. Farben (Kreuz, Pik, Herz und Karo). Jede Kombination aus Wert und Farbe kommt in einem Skatblatt genau einmal vor. Wir konzentrieren uns hier hauptsächlich auf den Aspekt des *Bedienens*, d.h. der Festlegung, welche Karten aufeinander gespielt werden dürfen: Eine Karte k' darf genau dann auf eine Karte k gespielt werden (auch: k' bedient k), wenn mindestens eine der folgenden Bedingungen erfüllt ist:

- Der Wert von k stimmt mit dem Wert von k' überein.
- Die Farbe von k stimmt mit der Farbe von k' überein.
- Der Wert von k' ist Bube.

Beispiel: Das Herz-Ass bedient u.a. die Herz-Neun und das Kreuz-Ass, nicht aber den Karo-König. Der Pik-Bube bedient jede Karte, aber nicht jede Karte bedient den Pik-Buben.
Die weiteren Regeln von Mau-Mau sind in dieser Aufgabe nicht von Bedeutung.

Hinweise:

- Sie dürfen in allen Teilaufgaben beliebige Hilfsmethoden schreiben.
- Um sich zunächst auf die Erstellung der Klassen zu konzentrieren, müssen Sie in dieser Aufgabe die Prinzipien der Datenkapselung noch nicht beachten.

a) Schreiben Sie jeweils einen Aufzählungstyp (d.h. eine `enum`-Klasse) `Farbe` und `Wert` für die vier verschiedenen Farben bzw. die acht verschiedenen Werte. Verwenden Sie dabei für die Bezeichner der einzelnen Objekte ausschließlich Großbuchstaben. Umgehen Sie Umlaute, indem sie diese wie in Kreuzworträtseln üblich durch zwei Buchstaben kodieren.

b) Schreiben Sie eine `record`-Klasse `Karte` mit je einem Attribut vom Typ `Farbe` und `Wert`. Überschreiben Sie außerdem in dieser Klasse die Methode `String toString()`, die für jedes Objekt vom Typ `Karte` einen `String` ausgibt: Dazu sollen die Farbe und der Wert in dieser Reihenfolge konkateniert werden. So soll bspw. für das Herz-Ass (mit dem `Farbe`-Attribut `Farbe.HERZ` und dem `Wert`-Attribut `Wert.ASS`) der String `HERZASS` ausgegeben werden. Sie können hier und in allen folgenden Teilaufgaben davon ausgehen, dass die Attribute eines Objekts vom Typ `Karte` stets sinnvoll gesetzt sind, wenn auf diesem eine Methode aufgerufen wird oder es als Parameter übergeben wird.

Hinweise:

- Um die String-Repräsentation eines `enum`-Objekts zu erhalten, können Sie die bereits vorhandene Methode `toString()` auf Objekten eines `enum`-Typs nutzen. Bspw. ist `Farbe.HERZ.toString()` der String `"HERZ"`.

c) Ergänzen Sie die Klasse `Karte` um eine statische Methode `Karte neueKarte(Farbe f, Wert w)`, die ein neues Objekt vom Typ `Karte` mit den übergebenen Attributen erzeugt und zurückgibt. Nutzen Sie diese Methode, um eine weitere statische Methode `Karte neueKarte(String f, String w)` in der Klasse `Karte` zu schreiben. Diese soll ebenfalls ein neues Objekt vom Typ `Karte` zurückgeben, wobei diesmal je ein `String` für die zu setzende Farbe und den zu setzenden Wert übergeben werden. Sie können davon ausgehen, dass nur solche `String`s übergeben werden, für die auch ein zugehöriges `enum`-Objekt existiert.

d) Ergänzen Sie die Klasse `Karte` um die statische Methode `int kombinationen()`, die die Anzahl der verschiedenen Farbe-Wert-Kombinationen zurückgibt. Gestalten Sie Ihre Implementierung so, dass die Ausgabe auch dann noch korrekt ist, wenn sich die zugrundeliegenden `enum`-Klassen ändern. Nutzen Sie die Methode `kombinationen()`, um eine weitere statische Methode `Karte[] skatblatt()` in der Klasse `Karte` zu schreiben. Diese soll ein Array mit Elementen vom Typ `Karte` zurückgeben, in dem sich für jede Farbe-Wert-Kombination genau eine entsprechende Karte befindet. Das Array soll keine weiteren Elemente haben, insbesondere keine `null`-Elemente.

e) Ergänzen Sie die Klasse `Karte` um eine Methode `boolean bedient(Karte other)`. Beim Aufruf dieser Methode auf einer Karte `this` soll genau dann `true` zurückgegeben werden, wenn die Karte `this` die Karte `other` bedient. Wann eine Karte eine andere bedient, haben wir zu Beginn dieser Aufgabe definiert.
Angenommen, das Objekt `k1` enthält `Farbe.HERZ` im Attribut `farbe` und `Wert.BUBE` im Attribut `wert`. Außerdem enthalte das Objekt `k2` `Farbe.KARO` im Attribut `farbe` und `Wert.KOENIG` im Attribut `wert`.

Der Aufruf `k1.bedient(k2)` soll dann `true` zurückgeben und der Aufruf `k2.bedient(k1)` soll `false` zurückgeben.
Nutzen Sie die Methode `bedient`, um eine weitere Methode `boolean bedienbar(Karte... kn)` in der Klasse `Karte` zu schreiben. Diese soll genau dann `true` zurückgeben, wenn mindestens eines der übergebenen `Karte`-Objekte in `kn` dasjenige `Karte`-Objekt bedient, auf dem die Methode aufgerufen wurde. Sie können hierbei davon ausgehen, dass `kn` nicht `null` ist.

f) Ergänzen Sie die Klasse `Karte` um eine statische Methode `void druckeDoppelBedienungen()`. Diese Methode soll alle Paare von `Karte`-Objekten mit unterschiedlichen Attributen durchgehen und für jedes Paar (`k1`,`k2`) eine Meldung ausgeben, wenn sowohl `k1.bedient(k2)` als auch `k2.bedient(k1)` gilt. Die Meldung soll mit `System.out.println` ausgegeben werden und folgende Form haben: `KARODAME bedient KAROKOENIG und KAROKOENIG bedient KARODAME.`

Hinweise:

- Um zu überprüfen, ob die Attribute von zwei Karten übereinstimmen, verwenden Sie die Methode `boolean equals(Karte k)`. Für zwei Karten `k1` und `k2` können Sie beispielsweise durch den Aufruf `k1.equals(k2)` überprüfen, ob alle Attribute übereinstimmen. Da Sie eine `record`-Klasse verwenden, steht die `equals`-Methode bereits automatisch zur Verfügung.

g) Schreiben Sie eine Klasse `Spieler` mit einem Attribut `kartenhand`, das ein Array mit Elementen vom Typ `Karte` ist. Außerdem soll ein zweites Attribut den Namen des Spielers enthalten und ein drittes Attribut `gespielteKarte` die Karte beinhalten, welche der Spieler zuletzt gespielt hat. Wählen Sie für diese beiden Attribute sinnvolle Typen. Schreiben Sie außerdem die Methode `String toString()` in der Klasse `Spieler`, die für jedes Objekt vom Typ `Spieler` den Namen des Spielers als `String` ausgibt. Sie können hier und in allen folgenden Teilaufgaben davon ausgehen, dass die Attribute eines Objekts vom Typ `Spieler` stets sinnvoll gesetzt sind, wenn auf diesem eine Methode aufgerufen wird oder es als Parameter übergeben wird.

h) Ergänzen Sie die Klasse `Spieler` außerdem um eine Methode `void spieleKarte(Karte k)`. Beim Aufruf dieser Methode soll geprüft werden, ob der Spieler mit seinen Karten die Karte `k` bedienen kann. Ist dies der Fall, dann soll sein Attribut `gespielteKarte` auf die erste Karte des Arrays `kartenhand` gesetzt werden, die `k` bedient, und die gespielte Karte soll aus seiner Hand entfernt werden. Geben Sie außerdem mit `System.out.println` eine geeignete Ausgabe aus (z.B.: `Max bedient KAROKOENIG mit KARODAME`). Kann der Spieler nicht bedienen, setzen Sie das Attribut `gespielteKarte` auf `null`. Geben Sie in diesem Fall keine Ausgabe aus.

Hinweise:

- Um die Karte aus der Hand zu entfernen, ist es sinnvoll, zuerst die Position zu bestimmen, an der sich die Karte befindet (z.B. mit einer `while`-Schleife). In einer zweiten Schleife wird nun ein neues Array mit allen Karten außer eben der zuvor bestimmten Karte gefüllt.

i) Ergänzen Sie die Klasse `Spieler` um eine `main`-Methode mit der bekannten Signatur. Erstellen Sie in dieser zuerst zwei `Spieler`-Objekte für die Spieler Elisabeth und Klaus. Elisabeth hat die Karten Herz-Neun, Herz-Zehn und Pik-Bube in ihrer Hand. Klaus hingegen hat die Karten Herz-Zehn, Pik-Bube und Herz-Neun. (Beachten Sie die Reihenfolge!) Erstellen Sie hierzu jeweils ein dreielementiges Karten-Array für die `kartenhand`. Lassen Sie das Attribut `gespielteKarte` uninitialisiert.

j) Erweitern Sie die Klasse `Spieler` um eine Methode `void spiele(Karte k)`. Diese Methode simuliert ein vereinfachtes Mau-Mau-Spiel. Begonnen wird mit der Karte `k`. Eine Partie hat hierbei den folgenden Ablauf. Der Spieler versucht, die Karte `k` zu bedienen. Hierzu wird die Methode `void spieleKarte(Karte k)` verwendet. Kann der Spieler bedienen, so wird die Karte, mit welcher er bedienen konnte, aus seiner Hand entfernt. Anschließend versucht der Spieler, nun die gerade entfernte Karte zu bedienen. Dies wird solange wiederholt, bis der Spieler entweder nicht mehr bedienen kann oder seine Kartenhand leer ist. Kann der Spieler nicht mehr bedienen, so wird `Elisabeth hat verloren!` ausgegeben für einen Spieler, der Elisabeth heißt. Hingegen wird `Elisabeth hat gewonnen!` ausgegeben für einen Spieler, der Elisabeth heißt, falls die Kartenhand zum Schluss leer ist. Sollte beim Aufruf der Methode `spiele` die Kartenhand bereits leer sein, so kann sich Ihre Methode beliebig verhalten. In der `main`-Methode soll zuerst Elisabeth und dann Klaus beginnend mit der Karo-Zehn spielen.

Hinweise:

- Die erzeugte Ausgabe sieht wie folgt aus:

```
Elisabeth bedient KAROZEHN mit HERZZEHN
Elisabeth bedient HERZZEHN mit HERZNEUN
Elisabeth bedient HERZNEUN mit PIKBUBE
Elisabeth hat gewonnen!
Klaus bedient KAROZEHN mit HERZZEHN
Klaus bedient HERZZEHN mit PIKBUBE
Klaus hat verloren!
```

Lösung:

Listing 9: Farbe.java

```
enum Farbe {
  KREUZ, PIK, HERZ, KARO;
}
```

Listing 10: Wert.java

```
enum Wert {
  SIEBEN, ACHT, NEUN, ZEHN, BUBE, DAME, KOENIG, ASS;
}
```

Listing 11: Karte.java

```
public record Karte (Farbe farbe, Wert wert) {

  public static int kombinationen() {
    return Farbe.values().length * Wert.values().length;
  }

  public static Karte[] skatblatt() {
    Karte[] res = new Karte[kombinationen()];
    int i = 0;
    for (Farbe f : Farbe.values()) {
      for (Wert w : Wert.values()) {
        res[i] = neueKarte(f,w);
        ++i;
      }
    }
    return res;
  }

  public static Karte neueKarte(Farbe f, Wert w) {
    Karte k = new Karte(f,w);
    return k;
  }

  public static Karte neueKarte(String farbstring, String wertstring) {
    Farbe farbe = Farbe.valueOf(farbstring);
    Wert wert = Wert.valueOf(wertstring);
    return Karte.neueKarte(farbe,wert);
  }

  public static void druckeDoppelBedienungen() {
    for (Karte k1 : skatblatt()) {
      for (Karte k2 : skatblatt()) {
        if (k1.bedient(k2) && k2.bedient(k1) && !k1.equals(k2)) {
```

```
          System.out.println(k1 + " bedient " + k2 + " und " +
                             k2 + " bedient " + k1 + ".");
        }
      }
    }
  }

  public boolean bedient(Karte other) {
    return (this.farbe == other.farbe || this.wert == other.wert
                                      || this.wert == Wert.BUBE);
  }

  public boolean bedienbar(Karte... kartenhand) {
    for (Karte k : kartenhand) {
      if (k.bedient(this)) {
        return true;
      }
    }
    return false;
  }

  public String toString() {
    return (farbe.toString() + wert.toString());
  }
}
```

Listing 12: Spieler.java

```
public class Spieler {
  String name;
  Karte[] kartenhand;
  Karte gespielteKarte;


  public void spieleKarte(Karte k) {
    int pos = 0;
    while(pos < kartenhand.length && !kartenhand[pos].bedient(k)) {
        pos++;
    }

    if(pos == kartenhand.length) {
        gespielteKarte = null;
        return;
    }

    gespielteKarte = kartenhand[pos];
    System.out.println(this + " bedient " + k + " mit " + gespielteKarte);

    Karte[] res = new Karte[kartenhand.length - 1];
    for(int i = 0; i < kartenhand.length; i++) {
        if(i < pos)
            res[i] = kartenhand[i];
        else if (i > pos)
            res[i - 1] = kartenhand[i];
    }
    kartenhand = res;
  }

  public void spiele(Karte k) {
      do {
        spieleKarte(k);
        k = gespielteKarte;
```

```
        } while(gespielteKarte != null && kartenhand.length != 0);
        if(kartenhand.length == 0)
            System.out.println(this + " hat gewonnen!");
        else
            System.out.println(this + " hat verloren!");
    }

    public String toString() {
      return name;
    }

    public static void main(String[] args) {
      Spieler s1 = new Spieler();
      s1.name = "Elisabeth";

      Karte k0 = Karte.neueKarte("HERZ","NEUN");
      Karte k1 = Karte.neueKarte("HERZ","ZEHN");
      Karte k2 = Karte.neueKarte("PIK","BUBE");

      s1.kartenhand = new Karte[3];
      s1.kartenhand[0] = k0;
      s1.kartenhand[1] = k1;
      s1.kartenhand[2] = k2;

      Spieler s2 = new Spieler();
      s2.name = "Klaus";

      s2.kartenhand = new Karte[3];
      s2.kartenhand[0] = k1;
      s2.kartenhand[1] = k2;
      s2.kartenhand[2] = k0;

      Karte k3 = Karte.neueKarte("KARO","ZEHN");

      s1.spiele(k3);
      s2.spiele(k3);

    }
}
```

Tutoraufgabe 4 (Programmierung mit Datenabstraktion):

In dieser Aufgabe wird eine Klasse implementiert, die eine Werkzeugkiste verwaltet. In einer Werkzeugkiste können Materialien (Schrauben, Nieten, etc.), einfache Werkzeuge (Zangen, Schraubendreher, etc.) und Elektrowerkzeuge (Bohrmaschinen, Schleifgerät, etc.) sein. Die meisten Materialien sind sehr klein. Deswegen können alle Materialien zusammen in einem einzelnen Fach der Werkzeugkiste untergebracht werden. Jedes einfache Werkzeug belegt ein Fach der Werkzeugkiste. Elektrowerkzeuge hingegen sind sehr groß. Jedes Elektrowerkzeug nimmt daher immer je drei benachbarte Fächer ein.
Beachten Sie in allen Teilaufgaben die *Prinzipien der Datenkapselung*.

a) Schreiben Sie einen Aufzählungstyp (d.h. eine `enum`-Klasse) `Tool` für die drei Arten von Werkzeugen: `PowerTool`, `SimpleTool` und `Materials`.

b) Schreiben Sie eine Klasse `Toolbox`, die vier Attribute hat: ein Array von `Tool`-Objekten als Fächer der Werkzeugkiste, eine ganzzahlige Variable für die freie Kapazität der Werkzeugkiste, ein String als Name der Werkzeugkiste und eine Konstante, die angibt, wie viele Fächer ein Elektrowerkzeug belegt.

Schreiben Sie außerdem zwei Konstruktoren:

- Ein Konstruktor, der eine Kapazität übergeben bekommt und eine leere Werkzeugkiste mit entsprechender Kapazität erstellt.
- Ein Konstruktor, der eine beliebige Anzahl `Tool`-Objekte übergeben bekommt und eine Werkzeugkiste erstellt, die genau diese Werkzeuge enthält und keine zusätzlichen freien Fächer hat. Freie Fächer entstehen hier also nur, falls auch `null` als `Tool`-Objekt übergeben wird. Die Einschränkung, dass Elektrowerkzeuge immer drei Fächer benötigen, kann hier ignoriert werden, denn bei idealer Platzeinteilung kann man oft viel mehr auf gleichem Raum unterbringen.

Beide Konstruktoren bekommen außerdem einen String übergeben, der als Name der Werkzeugkiste gesetzt wird.

c) Schreiben Sie Selektor-Methoden, um die freie Kapazität zu lesen, um den Namen der Kiste zu lesen und um das Werkzeug in Fach `i` zu lesen. Falls `i` keine gültige Fachnummer ist, soll `null` zurückgegeben werden. Schreiben Sie außerdem eine Methode, um den Namen zu ändern.

d) Schreiben Sie eine Hilfsmethode `checkRoomForPowerTool`, die den ersten Index `i` ermittelt, an dem ein Elektrowerkzeug in die Werkzeugkiste passen würde. Als Rückgabewert hat die Methode einen `boolean`, der angibt, ob drei freie Plätze in Folge gefunden werden konnten. Als Eingabe bekommt die Methode ein Objekt der Klasse `Wrapper`, die im Moodle-Lernraum zur Verfügung steht. Sie speichert einen `int`-Wert und bietet Getter und Setter Methoden für diesen `int`-Wert. Als Seiteneffekt soll dieses `Wrapper`-Objekt so geändert werden, dass es den gefundenen Index `i` speichert.

e) Diskutieren Sie die Sichtbarkeit der Methode `checkRoomForPowerTool`.

f) Schreiben Sie eine Methode `void addTool(Tool t)`, die ein Werkzeug `t` zu einer Werkzeugkiste hinzufügt, falls dafür Platz ist. Elektrowerkzeuge werden an die erste Stelle gespeichert, an der drei Fächer in Folge frei sind. Das Objekt wird in jedes dieser drei Fächer geschrieben. Normale Werkzeuge werden an den ersten freien Platz geschrieben. Materialien werden nur dann neu hinzugefügt, wenn kein Fach mit Materialien gefunden wurde, bevor ein freies Fach gefunden wurde. Die Methode sollte außerdem die Kapazität aktualisieren.

g) Schreiben Sie ausführliche `javadoc`-Kommentare für die gesamte Klasse `Toolbox`.

Lösung:

Listing 13: Toolbox.java

```
/**
 * Objekte dieser Klasse repraesentieren eine beschriftete Werkzeugkiste.
 *
 * In den Faechern koennen je ein einfaches Werkzeug, eine grosse Menge Materialien
 * oder, in drei nebeneinander liegenden Faechern, ein Elektrowerkzeug untergebracht werden.
 */
public class Toolbox {
  /**
   * Anzahl Faecher, die ein Elektrowerkzeug belegt.
   */
  public static final int PowerToolSize = 3;

  /**
   * Array, das die Faecher der Werkzeugkiste repraesentiert.
   */
  private Tool [] tools;
  /**
   * Anzahl freier Faecher in der Werkzeugkiste.
   */
  private int capacity;
  /**
   * Beschriftung der Werkzeugkiste.
```

```
 */
private String name;

/**
 * Erstelle eine neue, leere Werkzeugkiste mit einer bestimmten Anzahl Faecher
 * @param name Beschriftung der Kiste
 * @param capacity Anzahl Faecher fuer die Kiste
 */
public Toolbox (String name, int capacity) {
  this.name = name;
  this.capacity = capacity;
  this.tools = new Tool[capacity];
}
/**
 * Erstelle eine neue Werkzeugkiste mit festgelegtem Inhalt.
 * @param name Beschriftung der Kiste
 * @param tools Werkzeuge, die in der Kiste enthalten sein sollen.
 */
public Toolbox (String name, Tool... tools) {
  this.name = name;
  this.capacity = 0;
  this.tools = tools;
  for(Tool tool : tools) {
    if(tool == null) {
      this.capacity += 1;
    }
  }
}

/**
 * Lese das Werkzeug im i-ten Fach.
 * @param i Nummer des Fachs
 * @return Das Werkzeug im i-ten Fach
 */
public Tool getTool(int i) {
  if(0 <= i && i < this.tools.length) {
    return this.tools[i];
  } else {
    return null;
  }
}

/**
 * Lese Anzahl freier Faecher
 * @return Anzahl freier Faecher
 */
public int getCapacity() {
  return this.capacity;
}

/**
 * Lese Beschriftung
 * @return Beschriftung der Werkzeugkiste
 */
public String getName() {
  return this.name;
}
/**
 * Setze Beschriftung
 * @param name Neue Beschriftung
 */
```

```
public void setName(String name) {
  this.name = name;
}

/**
 * Finde den ersten moeglichen Platz fuer ein Elektrowerkzeug.
 * @param index Ausgabeparameter fuer den freien Platz. Falls Rueckgabewert
 *   false ist, dann ist dieser Wert ungueltig.
 * @return ob ein gueltiger Index gefunden wurde.
 */
private boolean checkRoomForPowerTool(Wrapper index) {
  index.set(0);
  boolean room = true;
  while(index.get() <= this.tools.length - Toolbox.PowerToolSize) {
    for(int j = index.get(); j < index.get() + Toolbox.PowerToolSize; ++j) {
      if(this.tools[j] != null) {
        room = false;
        break;
      }
      room = true;
    }
    if(room) {
      return true;
    }
    index.set(index.get()+1);
  }
  return false;
}

/**
 * Fuege ein Werkzeug an erster passender Stelle zur Kiste hinzu, falls Platz ist.
 * @param t Das Werkzeug, das hinzugefuegt werden soll.
 */
public void addTool(Tool t) {
  switch(t) {
    case PowerTool -> {
      Wrapper i = new Wrapper(0);
      if(this.checkRoomForPowerTool(i)) {
        for(int k = 0; k < Toolbox.PowerToolSize; ++k) {
          this.tools[i.get() + k] = t;
        }
        this.capacity -= Toolbox.PowerToolSize;
      }
    }
    case Materials -> {
      for(int k = 0; k < this.tools.length; ++k) {
        if(this.tools[k] == null) {
          this.tools[k] = t;
          this.capacity -= 1;
          break;
        }
        if(this.tools[k] == Tool.Materials) {
          break;
        }
      }
    }
    case SimpleTool -> {
      for(int k = 0; k < this.tools.length; ++k) {
        if(this.tools[k] == null) {
          this.tools[k] = t;
          this.capacity -= 1;
```

```
            break;
          }
        }
      }
    }
  }
}
```

Listing 14: Tool.java

```
public enum Tool {
  PowerTool, SimpleTool, Materials
}
```

Listing 15: Wrapper.java

```
public class Wrapper {
  private int i;

  public Wrapper(int i) {
    this.i = i;
  }

  public void set(int i) {
    this.i = i;
  }
  public int get() {
    return this.i;
  }
}
```

e) Die Methode sollte `private` sein, da sie stark mit der internen Struktur verknüpft ist. Eine Änderung der internen Struktur der Klasse könnte eine Änderung der Signatur nach sich ziehen.

Dass der Rückgabewert "existiert ein Platz für ein Elektrowerkzeug" aber auch für den Benutzer der Klasse interessant ist, reicht nicht als Grund, die Methode öffentlich zu machen. Es wäre besser, eine zusätzliche öffentliche Methode `checkRoomForPowerTool` ohne Parameter zu ergänzen.

Aufgabe 5 (Programmierung mit Datenabstraktion): (1 + 3 + 2 + 7 + 3 + 4 = 20 Punkte)

In dieser Aufgabe soll eine Java-Klasse erstellt werden, mit der sich Rechtecke repräsentieren lassen. Ein solches Rechteck lässt sich mit den Koordinaten x und y für die linke obere Ecke, der Breite *width* und der Höhe *height* beschreiben, wobei x, y, *width* und *height* ganze Zahlen sind. Die Breite und die Höhe eines Rechtecks können nicht negativ sein.
Ihre Implementierung sollte mindestens die folgenden Methoden beinhalten. Sie sollten dabei die *Prinzipien der Datenkapselung* berücksichtigen. Hilfsmethoden müssen als `private` deklariert werden. In dieser Aufgabe dürfen Sie die in der Klasse `Utils` zur Verfügung gestellten Hilfsfunktionen, aber keine Bibliotheksfunktionen verwenden. Sie finden die Klasse im Moodle-Lernraum.
Um Ihre Implementierung selbst zu testen, können sie in der Klasse `Rectangle` eine `main`-Methode schreiben, um verschiedene Szenarien zu überprüfen. Vergessen Sie nicht, Randfälle zu betrachten.

a) Erstellen Sie eine Klasse `Rectangle` mit den Attributen `x`, `y`, `width` und `height`.

b) Erstellen Sie die folgenden öffentlichen Methoden, um Objekte des Typs `Rectangle` erzeugen zu können. Entscheiden Sie dabei selbst, welche Methoden Sie als statisch deklarieren:

```
Rectangle(int xInput, int yInput, int widthInput, int heightInput)
Rectangle(int xInput, int yInput, int sidelengthInput)
Rectangle copy(Rectangle toCopy)
```

Beachten Sie dabei folgende Punkte:

- Der Konstruktor `Rectangle` mit vier Argumenten soll ein Rechteck erzeugen, dessen Attribute jeweils die von den entsprechenden Parametern angegebenen Werte haben.
- Der Konstruktor `Rectangle` mit drei Argumenten soll ein Quadrat erzeugen, dessen Höhe und Breite den Wert des Parameters `sidelengthInput` annehmen und dessen übrige Attribute jeweils die von den entsprechenden Parametern angegebenen Werte haben.
- Falls bei den ersten beiden Methoden eines der Argumente einen unzulässigen Wert hat, muss eine geeignete Fehlermeldung ausgegeben und direkt im Anschluss `return` ausgeführt werden. Zur Ausgabe einer Fehlermeldung kann die Methode `Utils.error(String msg)` genutzt werden.
- Die Methode `copy` soll ein Rechteck kopieren, d.h., es soll ein neues Rechteck zurückgeliefert werden, das die gleichen Attribut-Werte wie das Rechteck `toCopy` hat.

c) Erstellen Sie Selektoren, um die Koordinaten, die Breite und die Höhe eines Rechtecks setzen und auslesen zu können. Entscheiden Sie dabei selbst, welche Methoden Sie als statisch deklarieren. Falls ein Attribut auf einen unzulässigen Wert gesetzt werden soll, darf der Wert des Attributes nicht verändert werden. Stattdessen muss eine geeignete Fehlermeldung ausgegeben werden.

Hinweise:

- Um einen Fehler auszugeben, kann die Methode `Utils.error(String msg)` genutzt werden.

d) Erstellen Sie die folgenden öffentlichen Methoden. Entscheiden Sie dabei selbst, welche Methoden Sie als statisch deklarieren und begründen Sie Ihre Entscheidung kurz:

```
boolean areSquares(Rectangle... rectangles)
int area()
Rectangle intersection(Rectangle... rectangles)
```

Bei den in diesem Aufgabenteil geforderten Methoden muss der Aufrufer (und nicht Sie als Implementierer der Klasse `Rectangle`) sicherstellen, dass die Parameter, mit denen die Methoden aufgerufen werden, nicht den Wert `null` haben bzw. enthalten.

Beachten Sie dabei folgende Punkte:

- Die Methode `boolean areSquares(Rectangle... rectangles)` soll `true` zurückgegeben, falls jedes Rechteck in `rectangles` ein Quadrat ist.
- Die Methode `int area()` soll die Fläche des Rechtecks zurückgeben, auf dem sie aufgerufen wird.
- Die Methode `intersection(Rectangle... rectangles)` gibt das größte Rechteck zurück, das *vollständig* in *allen* als Argument übergebenen Rechtecken enthalten ist. Wenn `rectangles` leer ist, soll `null` zurückgegeben werden. Falls der Schnitt der Rechtecke leer ist, soll ebenfalls `null` zurückgegeben werden.
  Hinweis: Es empfiehlt sich, eine Hilfsmethode zu implementieren, die den Schnitt *zweier* Rechtecke berechnet.
  Beispiel: Wir betrachten die beiden Rechtecke, die mit den Aufrufen
  `new Rectangle(1,4,2,3)` und `new Rectangle(2,5,3,3)` erzeugt werden. Das ausgegebene Rechteck beim Aufruf von `intersection` soll ein Rechteck mit den Werten $2, 4, 1, 2$ zurückgegeben werden. Die Werte stehen jeweils in der Reihenfolge $x, y, width, height$.

  Hinweise:

  - Sie finden in der Klasse `Utils` zwei Methoden `min` und `max`, die das Minimum bzw. Maximum von beliebig vielen Werten des Typs `int` berechnen.

e) Erstellen Sie ebenfalls eine Implementierung für die öffentliche Methode

```
String toString()
```

Entscheiden Sie dabei selbst, ob Sie die Methode als statisch deklarieren. Die Methode `toString()` erstellt eine textuelle Repräsentation des aktuellen Rechtecks. Dies geschieht über die Eckpunkte des Rechtecks, die, beginnend bei der oberen linken Ecke, gegen den Uhrzeigersinn ausgegeben werden sollen.

Zum Beispiel stellt der String `(3|5),(3|1),(6|1),(6|5)` das Rechteck mit den Koordinaten 3 und 5, der Breite 3 und der Höhe 4 dar.

f) Dokumentieren Sie alle Methoden, die als `public` markiert sind, indem Sie die Implementierung mit `javadoc`-Kommentaren ergänzen. Diese Kommentare sollten eine allgemeine Erklärung der Methode sowie weitere Erklärungen jedes Parameters und des `return`-Wertes enthalten. Verwenden Sie innerhalb des Kommentars dafür die `javadoc`-Anweisungen `@param` und `@return`.

Benutzen Sie das Programm `javadoc`, um Ihre `javadoc`-Kommentare in das HTML-Format zu übersetzen. Überprüfen Sie mit einem Browser, ob das gewünschte Ergebnis generiert wurde. Falls `javadoc` Ihre Abgabe nicht kompiliert, werden **keine** Punkte vergeben.

Lösung:

Listing 16: Rectangle.java

```
/**
 * Ein Objekt der Klasse Rectangle repraesentiert ein Rechteck.
 */
public class Rectangle {

    private int x;
    private int y;
    private int width;
    private int height;

    /**
     * Konstruktor fuer ein neues Rechteck.
     * @param xInput der x-Anteil der oberen linken Ecke
     * @param yInput der y-Anteil der oberen linken Ecke
     * @param widthInput die nicht negative Breite
     * @param heightInput die nicht negative Hoehe
     */
    public Rectangle(int xInput, int yInput, int widthInput, int heightInput) {
        if (widthInput < 0 || heightInput < 0) {
            Utils.error("Trying to create rectangle with invalid dimensions: height " +
                        heightInput + ", width " + widthInput);
            return;
        }
        x = xInput;
        y = yInput;
        width = widthInput;
        height = heightInput;
    }

    /**
     * Konstruktor fuer ein neues Rechteck, das ein Quadrat ist.
     * @param xInput der x-Anteil der oberen linken Ecke
     * @param yInput der y-Anteil der oberen linken Ecke
     * @param sidelengthInput die nicht negative Breite
     */
    public Rectangle(int xInput, int yInput, int sidelengthInput) {
        if (sidelengthInput < 0) {
            Utils.error("Trying to create rectangle with invalid dimension: sidelength " +
                        sidelengthInput);
            return;
        }
        x = xInput;
        y = yInput;
        width = sidelengthInput;
        height = sidelengthInput;
    }

    /**
     * Erzeugt eine Kopie des uebergebenen Rechtecks.
     * @param toCopy das zu kopierende Rechteck
     * @return eine Kopie des uebergebenen Rechtecks
     */
    public static Rectangle copy(Rectangle toCopy) {
        return new Rectangle(toCopy.getX(), toCopy.getY(), toCopy.getWidth(), toCopy.getHeight());
    }

    /**
     * Liefert die Hoehe dieses Rechtecks.
     * @return die Hoehe dieses Rechtecks
     */
    public int getHeight() {
        return height;
    }

    /**
```

```
 * Setzt die Hoehe dieses Rechtecks.
 * @param height die neue, nicht negative Hoehe dieses Rechtecks
 */
public void setHeight(int height) {
    if (height < 0) {
        Utils.error("Trying to set height to negative value " + height + "!");
    } else {
        this.height = height;
    }
}

/**
 * Liefert die Breite dieses Rechtecks.
 * @return die Breite dieses Rechtecks
 */
public int getWidth() {
    return width;
}

/**
 * Setzt die Breite dieses Rechtecks.
 * @param width die neue, nicht negative Breite dieses Rechtecks
 */
public void setWidth(int width) {
    if (width < 0) {
        Utils.error("Trying to set width to negative value " + width + "!");
    } else {
        this.width = width;
    }
}

/**
 * Liefert den x-Anteil der linken oberen Ecke des Rechtecks.
 * @return den x-Anteil der linken oberen Ecke des Rechtecks
 */
public int getX() {
    return x;
}

/**
 * Setzt den x-Anteil der linken oberen Ecke des Rechtecks.
 * @param x der neue x-Anteil der linken oberen Ecke des Rechtecks
 */
public void setX(int x) {
    this.x = x;
}

/**
 * Liefert den y-Anteil der linken oberen Ecke des Rechtecks.
 * @return den y-Anteil der linken oberen Ecke des Rechtecks
 */
public int getY() {
    return y;
}

/**
 * Setzt den y-Anteil der linken oberen Ecke des Rechtecks.
 * @param y der neue y-Anteil der linken oberen Ecke des Rechtecks
 */
public void setY(int y) {
    this.y = y;
}

/* Alle Rechtecke spielen die gleiche Rolle, ohne static
 * suggeriert die Signatur eine "Sonderrolle" von this.
 * Das Rechteck "this" spielt fuer die Methode ausserdem keine Rolle.
*/
/**
 * Berechnet, ob alle Rechtecke Quadrate sind.
 * @param rectangles alle Rechtecke die betrachtet werden
 * @return true, iff. alle Rechtecke sind Quadrate.
 */
public static boolean areSquares(Rectangle... rectangles) {
    for (Rectangle r : rectangles) {
        if (r.width != r.height)
            return false;
    }
    return true;
}

 /* Nicht-static. Methode bezieht sich auf dieses konkrete Objekt.
/**
 * Berechnet die Flaeche dieses Rechteckes
 * @return die Flaeche des Rechteckes
```

```
     */
    public int area() {
        return this.width * this.height;
    }

    /*
     * Liefert den Schnitt zweier Rechtecke zurueck.
     * Wenn der Schnitt leer ist, wird null zurueckgegeben.
     */
    private static Rectangle singleIntersection(Rectangle aRect, Rectangle anotherRect) {
        int nx = Utils.max(aRect.x, anotherRect.x);
        int ny = Utils.min(aRect.y, anotherRect.y);
        int nw = Utils.min(aRect.x + aRect.width, anotherRect.x + anotherRect.width) - nx;
        int nh = ny - Utils.max(aRect.y - aRect.height, anotherRect.y - anotherRect.height);
        if (nw < 0 || nh < 0) {
            return null;
        }
        return new Rectangle(nx, ny, nw, nh);
    }

    /* Alle Rechtecke spielen die gleiche Rolle, ohne static
     * suggeriert die Signatur eine "Sonderrolle" von this.
     * Das Rechteck "this" spielt fuer die Methode ausserdem keine Rolle.
    */
    /**
     * Liefert den Schnitt aller uebergebenen Rechtecke zurueck.
     * @param rectangles jene Rechtecke, deren Schnitt berechnet werden soll
     * @return den Schnitt der uebergebenen Rechtecke oder null, wenn der Schnitt leer ist
     */
    public static Rectangle intersection(Rectangle... rectangles) {
        if (rectangles.length == 0) {
            return null;
        }
        Rectangle res = rectangles[0];
        for (int i = 1; i < rectangles.length; i++) {
            res = singleIntersection(res, rectangles[i]);
            if (res == null) {
                return null;
            }
        }
        return res;
    }

    /**
     * Gibt eine String-Repraesentation dieses Rechtecks zurueck.
     * @return die String-Repraesentation dieses Rechtecks
     */
    public String toString() {
        String res = "";

        //Eckpunkt links oben
        res += "(" + x + "|" + y + "),";

        //Eckpunkt links unten
        res += "(" + x + "|" + (y - height) + "),";

        //Eckpunkt rechts unten
        res += "(" + (x + width) + "|" + (y - height) + "),";

        //Eckpunkt rechts oben
        res += "(" + (x + width) + "|" + y + ")";

        return res;
    }
}
```

Aufgabe 6 (Deck 4): (Codescape)

Lösen Sie die Missionen von Deck 4 des Codescape Spiels. Ihre Lösung für die Codescape Missionen wird nur dann für die Zulassung gezählt, wenn Sie Ihre Lösung vor der einheitlichen Codescape Deadline am Samstag, den 22.01.2022, um 23:59 Uhr abschicken.

Lösung: