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());
      }
    }
  }
}
```

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 9: 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 10: Tool.java

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

Listing 11: 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.