Prof. Dr. J. Giesl
S. Dollase, M. Hark, D. Korzeniewski

Allgemeine Hinweise:

- Die **Hausaufgaben** sollen in Gruppen von je **2 Studierenden** aus der **gleichen Kleingruppenübung (Tutorium)** bearbeitet werden. **Namen und Matrikelnummern** der Studierenden sind auf jedes Blatt der Abgabe zu schreiben. **Heften bzw. tackern Sie die Blätter oben links!**
- Die **Nummer der Übungsgruppe** muss **links oben** auf das **erste Blatt** der Abgabe geschrieben werden. Notieren Sie die Gruppennummer gut sichtbar, damit wir besser sortieren können.
- Die Lösungen müssen **bis Freitag, den 07.12.2018 um 12:00** in den entsprechenden Übungskasten eingeworfen werden. Sie finden die Kästen am Eingang Halifaxstr. des Informatikzentrums (Ahornstr. 55). Alternativ können Sie die Lösungen auch vor der Abgabefrist direkt bei Ihrer Tutorin/Ihrem Tutor abgeben.
- In einigen Aufgaben müssen Sie in Java programmieren und `.java`-Dateien anlegen. **Drucken** Sie diese aus **und** schicken Sie sie per **E-Mail** vor Freitag, den 07.12.2018 um 12:00 an Ihre Tutorin/Ihren Tutor. Stellen Sie sicher, dass Ihr Programm von `javac` **akzeptiert** wird, ansonsten werden keine Punkte vergeben.
- Benutzen Sie in Ihrem Code keine Umlaute, auch nicht in Strings und Kommentaren. Diese führen oft zu Problemen, da diese Zeichen auf verschiedenen Betriebssystemen unterschiedlich behandelt werden. Dadurch kann es dazu führen, dass Ihr Programm bei Ihrer Tutorin/Ihrem Tutor bei Verwendung von Umlauten nicht mehr von `javac` akzeptiert wird.
- Halten Sie sich beim Lösen von Programmieraufgaben an die auf der Website zur Vorlesung verfügbaren Codekonventionen. Verstöße, die zu unleserlichem Code führen, können zu Punktabzug führen.
- Für einige Programmieraufgaben benötigen Sie die Java Klasse `SimpleIO`. Diese können Sie auf der Website herunterladen.
- Einige Hausaufgaben müssen im Spiel Codescape gelöst werden:
  https://codescape.medien.rwth-aachen.de/progra/
  Diese Aufgaben werden getrennt von den anderen Hausaufgaben gewertet.

---

## Tutoraufgabe 1 (Überschreiben, Überladen und Verdecken):

Betrachten Sie die folgenden Klassen:

Listing 1: X.java

```
1  public class X {
2      public int a = 23;
3
4      public X(int a) {                          // Signatur: X(I)
5          this.a = a;
6      }
7
8      public X(float x) {                        // Signatur: X(F)
9          this((int) (x + 1));
10     }
11
12     public void f(int  i, X o) { }             // Signatur: X.f(IX)
13     public void f(long lo, Y o) { }            // Signatur: X.f(LY)
14     public void f(long lo, X o) { }            // Signatur: X.f(LX)
15 }
```

Listing 2: Y.java

```
1  public class Y extends X {
2      public float a = 42;
3
4      public Y(double a) {                       // Signatur: Y(D)
5          this((float) (a - 1));
```

```
6       }
7
8       public Y(float a) {                             // Signatur: Y(F)
9           super(a);
10          this.a = a;
11      }
12
13      public void f(int i, X o) { }                   // Signatur: Y.f(IX)
14      public void f(int i, Y o) { }                   // Signatur: Y.f(IY)
15      public void f(long lo, X o) { }                 // Signatur: Y.f(LX)
16  }
```

Listing 3: Z.java

```
1   public class Z {
2       public static void main(String [] args) {
3                                                       // a)
4           X xx1 = new X(42);                          // (1)
5           System.out.println("X.a: " + xx1.a);
6           X xx2 = new X(22.99f);                      // (2)
7           System.out.println("X.a: " + xx2.a);
8           X xy = new Y(7.5);                          // (3)
9           System.out.println("X.a: " + ((X) xy).a);
10          System.out.println("Y.a: " + ((Y) xy).a);
11          Y yy = new Y(7);                            // (4)
12          System.out.println("X.a: " + ((X) yy).a);
13          System.out.println("Y.a: " + ((Y) yy).a);
14                                                      // b)
15          int i = 1;
16          long lo = 2;
17          xx1.f(i, xy);                               // (1)
18          xx1.f(lo, xx1);                             // (2)
19          xx1.f(lo, yy);                              // (3)
20          yy.f(i, yy);                                // (4)
21          yy.f(i, xy);                                // (5)
22          yy.f(lo, yy);                               // (6)
23          xy.f(i, xx1);                               // (7)
24          xy.f(lo, yy);                               // (8)
25          //xy.f(i, yy);                              // (9)
26      }
27  }
```

In dieser Aufgabe sollen Sie angeben, welche Methoden- und Konstruktoraufrufe stattfinden. Verwenden Sie hierzu keinen Computer, sondern nur die aus der Vorlesung bekannten Angaben zum Verhalten von Java. Verwenden Sie zur eindeutigen Bezeichnung die Funktionssignatur, die jeweils als Kommentar hinter jeder Funktionsdefinition steht. Begründen Sie Ihre Antwort kurz.

**a)** Geben Sie für die mit `(1)`-`(4)` markierten Konstruktoraufrufe in der Klasse `Z` jeweils an, welche Konstruktoren in welcher Reihenfolge von Java aufgerufen werden. Notieren Sie auch die von Java implizit aufgerufenen Konstruktoren. Bedenken Sie, dass die Oberklasse von `X` die Klasse `Object` ist. Erklären Sie außerdem, welche Attribute mit welchen Werten belegt werden und welche Werte durch die `println`-Anweisungen ausgegeben werden.

**b)** Geben Sie für die mit `(1)`-`(9)` markierten Aufrufe der Methode `f` in der Klasse `Z` jeweils an, welche Variante der Funktion von Java verwendet wird. Geben Sie hierzu die jeweilige Signatur an.

## Aufgabe 2 (Überschreiben, Überladen und Verdecken): (4 + 3 = 7 Punkte)

Betrachten Sie die folgenden Klassen:

Listing 4: A.java

```
1   public class A {
2       public final String x;
3
4       public A() {                                    // Signatur: A()
5           this("written in A()");
6       }
7
8       public A(int p1) {                              // Signatur: A(int)
9           this("written in A(int)");
10      }
11
```

```
12      public A(String x) {                                    // Signatur: A(String)
13          this.x = x;
14      }
15
16      public void f(A p1) {                                   // Signatur: A.f(A)
17          System.out.println("called A.f(A)");
18      }
19  }
```

Listing 5: B.java

```
1   public class B extends A {
2       public final String x;
3
4       public B() {                                            // Signatur: B()
5           this("written in B()");
6       }
7
8       public B(int p1) {                                      // Signatur: B(int)
9           this("written in B(int)");
10      }
11
12      public B(A p1) {                                        // Signatur: B(A)
13          this("written in B(A)");
14      }
15
16      public B(B p1) {                                        // Signatur: B(B)
17          this("written in B(B)");
18      }
19
20      public B(String x) {                                    // Signatur: B(String)
21          super("written in B(String)");
22          this.x = x;
23      }
24
25      public void f(A p1) {                                   // Signatur: B.f(A)
26          System.out.println("called B.f(A)");
27      }
28
29      public void f(B p1) {                                   // Signatur: B.f(B)
30          System.out.println("called B.f(B)");
31      }
32  }
```

Listing 6: C.java

```
1   public class C {
2       public static void main(String[] args) {
3                                                               // a)
4           A v1 = new A(100);                                  // (1)
5           System.out.println("v1.x: " + v1.x);
6
7           A v2 = new B(100);                                  // (2)
8           System.out.println("v2.x: " + v2.x);
9           System.out.println("((B) v2).x: " + ((B) v2).x);
10
11          B v3 = new B(v2);                                   // (3)
12          System.out.println("((A) v3).x: " + ((A) v3).x);
13          System.out.println("v3.x: " + v3.x);
14
15          B v4 = new B();                                     // (4)
16          System.out.println("((A) v4).x: " + ((A) v4).x);
17          System.out.println("v4.x: " + v4.x);
18
19                                                              // b)
20          v1.f(v1);                                           // (1)
21          v1.f(v2);                                           // (2)
22          v1.f(v3);                                           // (3)
23          v2.f(v1);                                           // (4)
24          v2.f(v2);                                           // (5)
25          v2.f(v3);                                           // (6)
26          v3.f(v1);                                           // (7)
27          v3.f(v2);                                           // (8)
28          v3.f(v3);                                           // (9)
29      }
30  }
```

In dieser Aufgabe sollen Sie angeben, welche Methoden- und Konstruktoraufrufe stattfinden. Verwenden Sie hierzu keinen Computer, sondern nur die aus der Vorlesung bekannten Angaben zum Verhalten von Java. Verwenden Sie zur eindeutigen Bezeichnung die Funktionssignatur, die jeweils als Kommentar hinter jeder Funktionsdefinition steht. Begründen Sie Ihre Antwort kurz.

**a)** Geben Sie für die mit `(1)-(4)` markierten Konstruktoraufrufe in der Klasse `C` jeweils an, welche Konstruktoren in welcher Reihenfolge von Java aufgerufen werden. Notieren Sie auch die von Java implizit aufgerufenen Konstruktoren. Bedenken Sie, dass die Oberklasse von `A` die Klasse `Object` ist. Erklären Sie außerdem, welche Werte aus welchem Grund durch die `println`-Anweisungen ausgegeben werden.

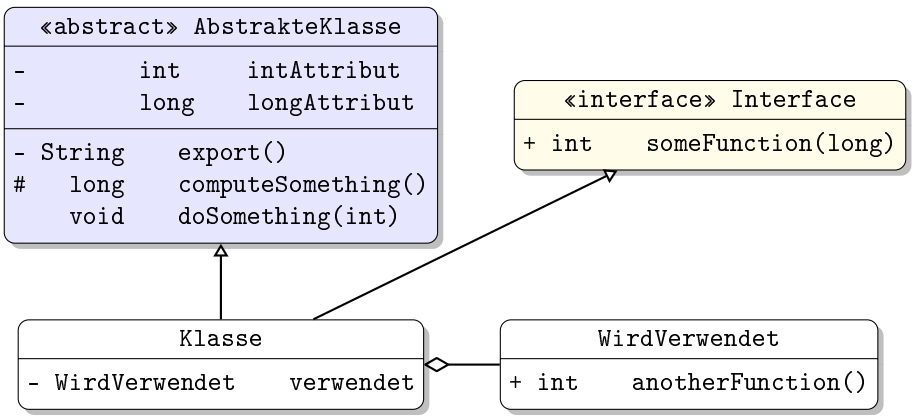
**b)** Geben Sie für die mit `(1)-(9)` markierten Aufrufe der Methode `f` in der Klasse `C` jeweils an, welche Variante der Funktion von Java verwendet wird. Geben Sie hierzu die jeweilige Signatur an.

## Tutoraufgabe 3 (Entwerfen von Klassenhierarchien):

In dieser Aufgabe soll der Zusammenhang verschiedener Getränke zueinander in einer Klassenhierarchie modelliert werden. Dabei sollen folgende Fakten beachtet werden:

- Jedes Getränk hat ein bestimmtes Volumen.
- Wir wollen Apfelsaft und Kiwisaft betrachten. Apfelsaft kann klar oder trüb sein.
- Alle Saftarten können auch Fruchtfleisch enthalten.
- Wodka und Tequila sind zwei Spirituosen. Spirituosen haben einen bestimmten Alkoholgehalt.
- Wodka wird häufig aromatisiert hergestellt. Der Name dieses Aromas soll gespeichert werden können.
- Tequila gibt es als silbernen und als goldenen Tequila.
- Ein Mischgetränk ist ein Getränk, das aus verschiedenen anderen Getränken besteht.
- Mischgetränke und Säfte kann man schütteln, damit die Einzelteile (bzw. das Fruchtfleisch) sich gleichmäßig verteilen. Sie sollen daher eine Methode `schuetteln()` ohne Rückgabe zur Verfügung stellen.
- In unserer Modellierung gibt es keine weiteren Getränke.

Entwerfen Sie unter Berücksichtigung der Prinzipien der Datenkapselung eine geeignete Klassenhierarchie für die Getränke. Notieren Sie keine Konstruktoren, Getter und Setter. Sie müssen nicht markieren, ob Attribute `final` sein sollen. Achten Sie darauf, dass gemeinsame Merkmale in Oberklassen bzw. Interfaces zusammengefasst werden.
Verwenden Sie hierbei die folgende Notation:




Eine Klasse wird hier durch einen Kasten beschrieben, in dem der Name der Klasse sowie Attribute und Methoden in einzelnen Abschnitten beschrieben werden. Weiterhin bedeutet der Pfeil $B \twoheadrightarrow A$, dass $A$ die Oberklasse von $B$ ist (also `class B extends A` bzw. `class B implements A`, falls $A$ ein Interface ist) und $A \diamond\!\!- B$, dass $A$ den Typ $B$ verwendet (z.B. als Typ eines Attributs oder in der Signatur einer Methode). Benutzen sie `+`, `-` und `#` um `public`, `private` und `protected` abzukürzen.
Tragen Sie keine vordefinierten Klassen (String, etc.) oder Pfeile dorthin in ihr Diagramm ein.

## Tutoraufgabe 4 (Programmieren in Klassenhierarchien):

In einer kleinen Stadt kommt es gehäuft zu Diebstählen. Sperren Sie die Diebe weg! Verwenden Sie hierbei die Hilfsklasse `Zufall` (auf der Homepage), die zwei statische Methoden `int zahl(int)` und `String name()` enthält. Ein Aufruf `Zufall.zahl(i)` (für `i` größer 0) gibt eine zufällige Zahl zwischen 0 und `i` − 1 zurück. Ein Aufruf `Zufall.name()` gibt einen zufällig gewählten Namen zurück.

**a)** In der Stadt sind viele Menschen. Schreiben Sie das Interface `Mensch`. Jeder Mensch hat einen Namen, welcher über die Methode `String getName()` abrufbar ist. Außerdem kann jeder Mensch über die Methode `void aktion(Mensch[] menschen)` eine Aktion ausführen, welche sich ggf. auf andere Menschen in der Stadt auswirkt.

**b)** Jeder `Tourist` ist ein `Mensch`. Implementieren Sie die Klasse `Tourist`, welche den `name` als Attribut speichert. Die Klasse hat einen Konstruktor, der einen Namen übergeben bekommt. Die Aktion, welche ein `Tourist` ausführt ist, die Stadt zu besuchen. Geben Sie das auf der Konsole aus.

**c)** Schreiben Sie die abstrakte Klasse `Buerger`, welche jeden Einwohner der Stadt modelliert. Jeder `Buerger` ist ein `Mensch`. Somit hat auch jeder Bürger einen Namen (als `String`-Attribut), der an den Konstruktor übergeben wird und bei einem Aufruf der `toString`-Methode zurückgegeben wird. Da sich der Name eines Bürgers nicht ändern soll, definieren Sie nur eine `get`-Methode, aber keine `set`-Methode. Weiterhin hat jeder Bürger die abstrakte Methode `boolean hatDiebesgut()`.

**d)** Ein reicher Bürger wird in der Klasse `ReicherBuerger` repräsentiert, welche die Klasse `Buerger` erweitert. Ein reicher Bürger hat einen gewissen Reichtum in Euro, der durch ein Attribut `reichtum` vom Typ `int` dargestellt wird. Der Konstruktor eines reichen Bürgers bekommt Namen und Reichtum als Parameter übergeben und initialisiert das Objekt entsprechend. Ein reicher Bürger hat kein Diebesgut. Die Aktion (implementiert in der Methode `void aktion(Mensch[] menschen)`) eines reichen Bürgers besteht darin, mit einem (zufälligen) Teil seines Geldes (seinen letzten Euro behält der reiche Bürger allerdings selbst) Politiker zu bestechen, was als Mitteilung ausgegeben werden soll. Dadurch schrumpft sein Reichtum um den entsprechenden Betrag.

**e)** Ein Dieb ist ebenfalls ein Bürger und wird durch die Klasse `Dieb` repräsentiert. Ein Dieb hat ein Attribut `int diebesgut`, welches auf 0 initialisiert wird. Somit hat der Konstruktor nur den Parameter `name`. Die Methode `boolean hatDiebesgut()` soll zurückgeben, ob das Diebesgut größer als 0 ist. Bei einem Aufruf der Methode `void aktion(Mensch[] menschen)` hat der Dieb 5 Versuche, um Menschen zu bestehlen. In jedem Versuch soll ein Mensch aus dem Array `menschen` zufällig ausgewählt werden. Ist es ein reicher Bürger, der mindestens einen Euro besitzt, so klaut der Dieb ihm einen zufälligen Teil seines Reichtums (aber nicht seinen letzten Euro). Hierbei wird das Attribut `reichtum` des reichen Bürgers um den Betrag verringert, durch den sich das Attribut `diebesgut` des Diebes erhöht. Trifft der Dieb bei den Versuchen auf einen Polizisten, so bricht er die Aktion ab (die restlichen Versuche verfallen).

**f)** Ein Gefangener ist ein Dieb, der kein Diebesgut besitzt und im Gefängnis sitzt. Schreiben Sie die Klasse `Gefangener`. Diese verfügt über einen Konstruktor, der den Namen des Gefangenen als einziges Argument entgegen nimmt. Außerdem verfügt sie über eine Methode `void aktion(Mensch[] menschen)`, die auf der Konsole ausgibt, dass sich der Gefangene mit dem Namen, der dem Konstruktor als Argument übergeben wurde, im Gefängnis ärgert.

**g)** Ein Polizist ist ein Bürger, der Verbrecher jagt. Er hat kein Diebesgut. Bei einem Aufruf der Methode `void aktion(Mensch[] menschen)` sucht der Polizist bei den Bürgern im Array `menschen` nach Diebesgut. Hierzu durchläuft er das Array von vorne nach hinten und verwendet die Methode `hatDiebesgut` der Bürger. Findet der Polizist Diebesgut, so ersetzt er den Dieb an der Stelle im Array durch einen Gefangenen gleichen Namens. Außerdem verfügt die Klasse Polizist über einen Konstruktor, der den Namen des Polizisten als einziges Argument entgegen nimmt.

**h)** Erstellen Sie die Klasse `Stadt`, welche das als `private` markierte Attribut `Mensch[] menschen` besitzt. Der Konstruktor dieser Klasse bekommt das Argument `int anzahl` und legt ein Array mit entsprechend vielen Menschen an, deren Namen (mit der Methode `Zufall.name()`) zufällig bestimmt werden sollen. Verwenden Sie die Methode `Zufall.zahl(int)` so, dass es jeweils etwa 20% Diebe, reiche Bürger, Polizisten, Gefangene und Touristen gibt. Der Reichtum eines reichen Bürgers soll zwischen 1 und 1000 Euro

liegen. In der statischen `main`-Methode der Klasse soll eine neue Stadt mit 10 Menschen erstellt werden. Danach wird 10 mal ein zufälliger Mensch ausgewählt und seine Methode `aktion` mit allen Menschen der Stadt aufgerufen.

Ein Lauf des Programms könnte beispielsweise die folgende Ausgabe erzeugen:

```
Gefangener Wibke aergert sich im Gefaengnis.
Reicher Buerger Alina besticht einen Politiker mit 286 Euro.
Dieb Lina sucht nach Diebesgut.
Dieb Lina bricht die Suche ab.
Gefangener Rebecca aergert sich im Gefaengnis.
Dieb Luis sucht nach Diebesgut.
Dieb Luis klaut Alina 6 Euro.
Dieb Luis sucht nach Diebesgut.
Dieb Luis klaut Torsten 110 Euro.
Polizist Till geht auf Verbrecherjagd.
Polizist Till entlarvt Dieb Luis.
Dieb Luis wurde eingesperrt.
Gefangener Luis aergert sich im Gefaengnis.
Tourist Yannick kommt die Stadt besuchen.
Reicher Buerger Torsten besticht einen Politiker mit 549 Euro.
```

Hinweise:

- Berücksichtigen Sie in der gesamten Aufgabe die Prinzipien der Datenkapselung.

## Aufgabe 5 (Klassenhierarchie): (2 + 14 + 4 + 6 + 0 = 26 Punkte)

Eine der grundlegenden Funktionalitäten des Betriebssystems ist es, einen einfachen und einheitlichen Zugriff auf gespeicherte Daten zu liefern. Dabei muss es dem Benutzer Ordner (Directories) und Dateien (Files) präsentieren. Im folgenden sehen Sie ein Beispiel für einen Teil eines typischen Linux Dateisystems.

```
/
/boot/
/boot/kernel
/etc/
/etc/motd
```

Wir sehen den Wurzelordner `/`, die beiden Unterordner `boot` und `etc`, sowie die beiden Dateien `kernel` und `motd`[1].
Intern wird der Inhalt einer Datei oder eines Unterordners nicht unter ihrem Namen abgelegt, sondern unter einem sogenannten inode, einem Integer. Der Eintrag im Ordner enthält dann nur die Information, unter welchem inode der Inhalt zu finden ist. Falls beispielsweise die Datei `motd` unter dem inode 5 abgelegt ist, dann ist der Inhalt selbst unter dem inode 5 zu finden. Im Ordner steht nur, dass im Unterordner `etc` eine Datei mit dem Namen `motd` existiert und dass dessen Inhalt unter dem inode 5 zu finden ist.
Dies ist ein nützlicher Mechanismus, denn er erlaubt es, auf einfache Art und Weise den Inhalt zweier Dateien gleich zu halten. Dazu wird ein zweiter Eintrag im Ordner angelegt, welcher auf denselben inode verweist wie ein bereits existierender Eintrag. So ist es möglich, einen zweiten Eintrag `friendly-message.txt` im Ordner `etc` zu erstellen, welcher ebenfalls auf den inode 5 verweist. Wird nun der Inhalt von `motd` verändert, so ändert sich automatisch auch der Inhalt von `friendly-message.txt`, und umgekehrt. Man sagt, `friendly-message.txt` ist ein *Hardlink* auf den Inhalt von `motd`. Auch `motd` ist ein Hardlink auf seinen Inhalt. In der Tat sind alle Einträge im Ordner gleichberechtigte Hardlinks auf ihren Inhalt. Ein inode wird erst dann gelöscht, wenn der letzte auf ihn verweisende Hardlink gelöscht wurde.
In dieser Aufgabe modellieren wir ein Dateisystem wie folgt:

[1] message of the day

- Jeder Hardlink, also jeder Eintrag im Ordner, wird durch ein Objekt der Klasse `Entry` dargestellt. Jedes `Entry`-Objekt hat einen `name` sowie eine Referenz auf einen `Node`.
- Jeder inode, also jeder Inhalt, wird durch ein Objekt der abstrakten Klasse `Node` dargestellt. Jedes `Node`-Objekt hat das Attribut `lastModified`, welches den Zeitpunkt der letzten Änderung enthält und über die Methode `long getLastModified()` abgerufen werden kann. Dieses Attribut wird beim Erstellen eines Objekts auf den aktuellen Zeitpunkt gesetzt. Außerdem kann das Attribut durch die Methode `void touch()` auf den aktuellen Zeitpunkt gesetzt werden.
- Ein Dateiinhalt ist ein `Node` welcher durch ein Objekt der Klasse `File` dargestellt wird. Jedes `File`-Objekt hält seinen Inhalt in einem `String content`, welcher über die Methoden `String readContent()` sowie `void writeContent(String content)` gelesen und geschrieben werden kann.
- Ein Ordnerinhalt ist ein `Node` welcher durch ein Objekt der Klasse `Directory` dargestellt wird. Jedes `Directory`-Objekt hält seine Dateien und Unterordner in einem Array von `Entry`s, welches über die Methode `Entry[] getEntries()` abgerufen werden kann. Außerdem kann über die Methoden `boolean containsEntry(String name)` und `Entry getEntry(String name)` geprüft werden, ob der Ordner einen gegebenen Eintrag enthält bzw. dieser Eintrag abgerufen werden. Im letzteren Fall wird bei Nicht-Existenz des Eintrags `null` zurückgegeben. Die Methode `Entry createDirectory(String name)` erstellt im aktuellen Ordner einen leeren Unterordner mit dem gegebenen Namen. Die Methode `Entry createFile(String name, String content)` erstellt im aktuellen Ordner eine Datei mit dem gegebenen Namen und Inhalt. Die Methode `Entry createHardlink(String name, Entry entry)` erstellt im aktuellen Ordner einen neuen Hardlink mit dem gegebenen Namen, welcher auf denselben `Node` verweist wie `entry`. Alle drei Methoden geben den neu erstellten `Entry` zurück. Existiert im aktuellen Ordner bereits ein Eintrag mit dem selben Namen, so geben diese drei Methoden einen Fehler auf der Konsole aus und liefern `null` zurück.
- Die Klasse `Entry` bietet ebenfalls Methoden. Die Methode `String getName()` gibt das `name`-Attribut zurück. Die Methode `File getAsFile()` liefert ihren `Node` als `File`, falls dieser tatsächlich ein `File` ist, ansonsten wird `null` zurückgegeben. Die Methode `Directory getAsDirectory()` arbeitet analog dazu. Die Methode `Entry createHardlink(String newName)` erstellt einen neuen `Entry` mit dem Namen `newName`, welcher auf denselben `Node` zeigt wie der aktuelle `Entry`.

**a)** Entwerfen Sie unter Berücksichtigung der Prinzipien der Datenkapselung eine geeignete Klassenhierarchie. Notieren Sie keine Konstruktoren. Getter und Setter sollen notiert werden. Sie müssen nicht markieren, ob Attribute `final` sein sollen. Achten Sie darauf, dass gemeinsame Merkmale in Oberklassen bzw. Interfaces zusammengefasst werden.

Verwenden Sie hierbei die gleiche Notation wie in Tutoraufgabe 3.

**b)** Implementieren Sie die Klassen entsprechend Ihrer Beschreibung. Fügen Sie geeignete Konstruktoren hinzu. Sie dürfen, falls nötig, weitere Hilfsmethoden einfügen. Implementieren Sie außerdem die statische Methode `Directory.createEmpty()`, welche ein neues und leeres `Directory`-Objekt zurückliefert. Achten Sie darauf, durch den Aufruf der Methode `touch()` den Zeitpunkt der letzten Änderung bei jeder Schreiboperation auf den aktuellen Zeitpunkt zu setzen.

**c)** Erstellen Sie ein Interface `Visitor` mit den Methoden

- `void visitFile(String name, File file)`
- `void visitDirectory(String name, Directory directory)`
- `void visitedDirectory()`

Fügen Sie außerdem die Methode `void accept(String name, Visitor visitor)` für das `Directory` hinzu. Implementieren Sie die Methode, sodass diese wie folgt arbeitet:

- Bevor die Einträge des `Directory`s verarbeitet werden, wird `visitor.visitDirectory(name, this)` aufgerufen.
- Anschließend wird für jeden Eintrag `e` des `Directory`s die Methode `visitor.visitFile(n, f)` mit dem Namen `n` des Eintrags `e` aufgerufen, falls der Eintrag `e` ein `File f` ist. Falls der Eintrag `e` ein `Directory` ist, so wird dessen `accept`-Methode aufgerufen und ihr der Name des Eintrags `e` und der `visitor` übergeben.

- Nachdem alle Einträge des `Directory`s verarbeitet wurden, wird die Methode `visitor.visitedDirectory()` aufgerufen.

Die `accept`-Methode läuft also rekursiv über den kompletten Ordner inklusive aller Unterordner und Dateien.

**d)** Implementieren Sie die Klasse `Printer`, welche den `Visitor` implementiert, um das Dateisystem wie folgt auf der Konsole auszugeben. Pfade im Dateisystem beginnen mit /. Unterordner werden ebenfalls mit / angehängt. Der Pfad zum Unterordner `local` des Ordners `usr` im Wurzelordner würde also, wie üblich, `/usr/local/` heißen.

Für jedes `Directory` wird zunächst der Zeitpunkt der letzten Änderung gefolgt von einem Leerzeichen und dem Pfad des Ordners ausgegeben, wobei dieser mit einem / endet. Anschließend werden alle Einträge in beliebiger Reihenfolge ausgegeben.

Für ein `File` wird zunächst der Zeitpunkt der letzten Änderung gefolgt von einem Leerzeichen und dem Pfad der Datei ausgegeben, ohne dass dabei ein / angehängt wird. In einer neuen Zeile wird zunächst > gefolgt von einem Leerzeichen und dem Inhalt der Datei ausgegeben.

**e)** Testen Sie Ihre Implementierung mit folgender `main`-Methode, welche von unserer Website heruntergeladen werden kann. Sie nutzt den Aufruf `Thread.sleep(42)`, um die Ausführung der `main`-Methode an dieser Stelle für 42 Millisekunden zu unterbrechen. Dies ist nötig, damit in der Ausgabe verschiedene Zeitpunkte der letzten Änderung sichtbar werden. Dieser Aufruf erfordert des Weiteren den Zusatz `throws InterruptedException` bei der Deklaration der `main`-Methode. Sie müssen die Konzepte der Exceptions und Threads also nicht verstehen, um die `main`-Methode zu verstehen. Es reicht zu wissen, dass `Thread.sleep(42)` 42 Millisekunden wartet.

Listing 7: Main.java

```
1  public class Main {
2      public static void main(String[] args) throws InterruptedException {
3          Directory root = Directory.createEmpty();
4          root.createDirectory("boot").getAsDirectory()
5                  .createFile("kernel", "1101001001001101001");
6          Entry etc = root.createDirectory("etc");
7          Entry motd = etc.getAsDirectory().createFile("motd", "");
8          Entry friendlyMessage = etc.getAsDirectory()
9                  .createHardlink("friendly-message.txt", motd);
10
11         root.accept("", new Printer());
12
13         System.out.println();
14         Thread.sleep(42);
15         friendlyMessage.getAsFile().writeContent("Welcome =)");
16
17         root.accept("", new Printer());
18     }
19 }
```

Beispielausgabe:

```
1542901393722 /
1542901393722 /boot/
1542901393722 /boot/kernel
> 1101001001001101001
1542901393722 /etc/
1542901393722 /etc/motd
>
1542901393722 /etc/friendly-message.txt
>
```

```
1542901393722 /
1542901393722 /boot/
1542901393722 /boot/kernel
> 1101001001001101001
1542901393722 /etc/
1542901393764 /etc/motd
> Welcome =)
1542901393764 /etc/friendly-message.txt
> Welcome =)
```

Hinweise:

- Die Java Methode `System.currentTimeMillis()` liefert den aktuellen Zeitpunkt als `long`-Wert, welcher die Anzahl der Millisekunden seit dem ersten Januar 1970 um 00:00 Uhr darstellt.
- Die Java Methode `String.substring(int beginIndex, int endIndex)` kann auf einem `String`-Objekt aufgerufen werden, um daraus einen Teilstring zu extrahieren. So gibt der Aufruf `"Hallo Welt!".substring(3, 7)` den `String` `"lo W"` zurück.
- Ihre `Printer`-Implementierung muss bis zu einer Verschachtelungstiefe von 100 Unterordnern korrekt funktionieren. Bei einer größeren Verschachtelungstiefe darf sich Ihr Code beliebig verhalten.
- Berücksichtigen Sie in der gesamten Aufgabe die Prinzipien der Datenkapselung und verwenden Sie Implementierungen in Oberklassen soweit möglich.

## Aufgabe 6 (Codescape): (Codescape)

Lösen Sie die Räume von **Deck 5** des Spiels Codescape.
Ihre Lösung für Räume dieses Codescape Decks wird nur dann für die Zulassung gezählt, wenn Sie die Lösung bis Freitag, den 07.12.2018 um 12:00 abschicken.