RWTH Aachen
Lehrgebiet Theoretische Informatik
Frohn–Reidl–Rossmanith–Sánchez–Ströder

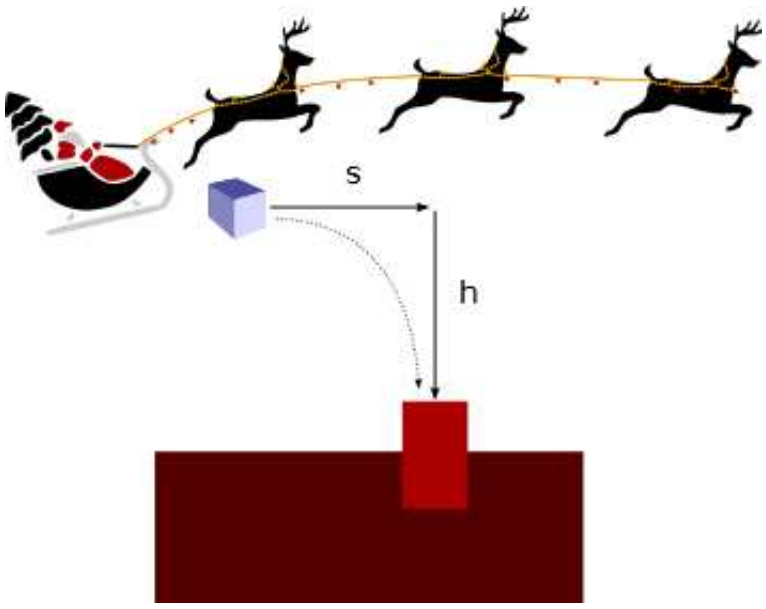
WS 2013/14
Übungsblatt 9
23.12.2013

Übung zur Vorlesung Programmierung

Dieses Übungsblatt ist ein reines Bonus-Blatt. Das bedeutet, dass die Punkte auf diesem Blatt nicht zur maximal erreichbaren Punktzahl hinzu zählen, die wir zur Berechnung des prozentualen Punkteanteils verwenden. Die durch die Bearbeitung dieses Blatts gesammelten Punkte zählen wir jedoch ganz normal zu Ihren Punkten hinzu.

Aufgabe T17

Helfen Sie dem Weihnachtsmann! Aufgrund der stetig wachsenden Erdbevölkerung hat der Weihnachtsmann keine Zeit mehr, um Geschenke noch direkt in die Häuser zu tragen. Stattdessen muss er auf eine effizientere Methode der Verteilung ausweichen. Sein Plan: Die Geschenke schon während des Fluges aus dem Rentierschlitten fallen zu lassen, sodass sie im richtigen Schornstein landen. Hierfür muss der Weihnachtsmann den genauen Zeitpunkt wissen, zu dem er das Geschenk loslassen muss.



Ihre Aufgabe: Implementieren Sie eine Methode, die berechnet, wie viele Meter vor der Überquerung des Schornsteins das Geschenk fallen gelassen werden muss.

Damit bei der Berechnung mit physikalischen Formeln weniger Fehler passieren, bietet es sich an, physikalische Einheiten durch Java-Typen abzubilden. Hierdurch kann der Compiler überprüfen, ob die Formel auf Werte mit korrekten Einheiten angewendet wird. Wie das genau funktioniert, soll durch diese Aufgabe erläutert werden.

a) Definieren Sie ein Interface *Einheit*, welches – wie alle Interfaces dieser Aufgabe – keine Methoden enthält. Definieren Sie nun die Interfaces *Meter* und *Sekunden*, welche *Einheit* erweitern. Definieren Sie außerdem die Interfaces *Multiplikation* und *Division*, welche jeweils zwei generische Typ-Parameter bekommen, die *Einheit* erweitern. Multiplikationen und Divisionen von Einheiten sind ebenfalls Einheiten. Nun können Sie die in dieser Aufgabe benötigten Einheiten als Java Typen darstellen:

Einheit	**Bedeutung**	**Java Typ**
m	Strecke	*Meter*
s	Dauer	*Sekunden*
m/s	Geschwindigkeit	$Division\langle Meter, Sekunden\rangle$
m/s^2	Beschleunigung	$Division\langle Meter, Multiplikation\langle Sekunden, Sekunden\rangle\rangle$

Da es keine Klassen gibt, die diese Interfaces implementieren (und somit auch keine entsprechenden Objekte), nennt man solche Typen auch *Phantom Types*.

b) Schreiben Sie die Klasse *Wert*, welche einen **double**-Wert kapselt und per Selektor zur Verfügung stellt. Diese Klasse bekommt einen generischen Typ-Parameter, der das Interface *Einheit* erweitern soll. Ein Objekt, welches eine Strecke von 10 m repräsentieren soll, hat also den Typ $Wert\langle Meter\rangle$ und kapselt den **double** Wert 10.0.

c) Erstellen Sie eine Klasse *Physik*. Definieren Sie eine statische Methode *berechneFalldauer* in der Klasse *Physik*, welche eine Strecke h (in Metern) sowie eine Beschleunigung g (in Metern pro Sekunde zum Quadrat) übergeben bekommt und die Dauer (in Sekunden) zurückgibt, die ein Gegenstand braucht, um aus einer solchen Höhe geworfen am Boden anzukommen, wenn die Beschleunigung g auf ihn wirkt. Die Formel zur Berechnung der Falldauer t lautet $t = \sqrt{\frac{2h}{g}}$. Verwenden Sie (auch in den Methoden der folgenden Aufgabenteile) für die Parameter und den Rückgabewert den Typ *Wert* mit korrekten Einheiten.

d) Definieren Sie eine statische Methode *berechneStrecke* in der Klasse *Physik*, welche eine Geschwindigkeit v (in Metern pro Sekunde), sowie eine Zeit t (in Sekunden) übergeben bekommt und die Strecke (in Metern) berechnet, die ein Gegenstand mit der Geschwindigkeit v in der Zeit t zurücklegt.

e) Definieren Sie eine statische Methode *berechneAbwurfentfernung* in der Klasse *Physik*, welche eine Strecke h (in Metern, Flughöhe des Rentierschlittens über dem Schornstein) und eine Geschwindigkeit v (in Metern pro Sekunde, Geschwindigkeit des Schlittens) übergeben bekommt und berechnet, in welcher Entfernung s der Weihnachtsmann ein Geschenk fallen lassen muss, damit es im Schornstein landet. Nehmen Sie hierzu an, dass sich das Geschenk in Fahrtrichtung mit gleichbleibender Geschwindigkeit fortbewegt, dabei aber nach unten fällt. Die Erdbeschleunigung beträgt $9,81\ m/s^2$.

Aufgabe T18

In dieser Aufgabe geht es darum, ein einfaches Labyrinth-Spiel um Exception-Handling zu erweitern. Hierdurch soll erreicht werden, dass der Benutzer verständliche Fehlermeldungen gezeigt bekommt, falls er eine falsche Eingabe macht.

Sie benötigen das Zip-Archiv `LabyrinthException.zip` von der Homepage. Dieses enthält die folgenden Dateien:

LabyrinthFeld.java Aufzählungstyp für Feldtypen eines Labyrinths.

LabyrinthGenerator.java Generator für Labyrinthe.

Richtung.java Aufzählungstyp für Himmelsrichtungen.

Labyrinth.java Das Labyrinth-Spiel.

Beim Ausführen der *main*-Methode der Klasse *Labyrinth* wird der Spieler zuerst aufgefordert, die Breite und Höhe des Labyrinths sowie die Anzahl der zu platzierenden Schätze anzugeben. Die Spielfigur (repräsentiert durch `<>`) startet in der linken oberen Ecke. Durch Drücken der Tasten `w`=oben, `a`=links, `s`=unten oder `d`=rechts und Bestätigung mit der Enter-Taste bewegt sich die Spielfigur in die entsprechende Richtung. Wird `q` eingegeben, beendet sich das Spiel. Hat der Spieler alle Schätze (`GG`) gefunden, gewinnt er das Spiel.

Beispiel:

```
Breite des Labyrinths: 37
Hoehe des Labyrinths: 13
Zahl der Schaetze: 5
##<>#################################################################
##        GG##      ##  ##          ##              ##  ##  ##  ##  ##  ##
##  ######  ##  ##  ##  ######  ##  ######  ##  ######  ##  ##  ##  ##  ##
##  ##  ##      ##      ##      ##          ##  ##  ##  ##  ##  ##      ##
##  ##  ##  ##  ##########  ##################  ##  ##  ##  ##  ##  ##  ##
##  ##      ##                      ##  ##  ##  ##  ##  ##  ##      ##  ##
######  ##  ##  ##  ##  ##############  ##  ######  ##  ##  ##########  ##
##  ##  ##  ##  ##GG##  ##    GG##  ##  ##      ##      ##      ##GG    ##
##  ##  ##  ##  ##  ##  ##  ######  ##  ##  ##  ##  ##  ######  ##  ##  ##
##      ##  ##  ##  ##  ##                  ##      ##    GG##  ##  ##  ##
##  ##  ##  ##  ##  ##  ######  ##  ##  ##  ##  ##  ##  ##  ##  ##  ##  ##
##  ##  ##  ##  ##  ##          ##  ##  ##  ##  ##  ##  ##          ##  ##
#####################################################################

Buchstaben und danach Enter drücken. w/a/s/d = Norden/Westen/Sueden/Osten, q = Beenden
>
```

a) Schreiben Sie die Exception-Klassen *GroesseZuKlein*, *BreiteZuKlein*, *HoeheZuKlein* und *ZuVieleSchaetze*. Hierbei soll die Klasse *GroesseZuKlein* die Elternklasse der Klassen *BreiteZuKlein* und *HoeheZuKlein* sein. Alle vier Klassen benötigen keine Attribute, Methoden oder Konstruktoren. Da nie Instanzen der Klasse *GroesseZuKlein* geworfen werden sollen, ist diese als **abstract** zu markieren.

b) Erweitern Sie die Methode *generateLabyrinth* der Klasse *LabyrinthGenerator*, indem Sie an den drei markierten Stellen Instanzen der Exception-Klassen *BreiteZuKlein*, *HoeheZuKlein* und *ZuVieleSchaetze* werfen. Erweitern Sie die Methodensignatur geeignet.

Hinweis: Wenn Sie in einer Methode mehrere Exceptions werfen, die eine gemeinsame Elternklasse haben, dann ist es eventuell sinnvoll, die Elternklasse (statt der einzelnen Kind-Klassen) in der throws-Deklaration der Methodensignatur anzugeben.

c) Die Exceptions, die durch die Methode *generateLabyrinth* der Klasse *LabyrinthGenerator* geworfen werden, sollen *nicht* in der Methode *labyrinthErstellen* der Klasse *Labyrinth* behandelt werden. Erweitern Sie daher ebenfalls die Signatur der Methode *labyrinthErstellen* geeignet.

d) Erweitern Sie die *main*-Methode der Klasse *Labyrinth*, sodass diese dem Benutzer bei falschen Eingaben die passende der folgenden Fehlermeldungen anzeigt:

- `Die angegebene Breite ist zu klein!`
- `Die angegebene Hoehe ist zu klein!`

- `Es wurden zu viele Schaetze fuer die Groesse des Labyrinths angegeben!`

Falls der Benutzer eine falsche Eingabe gemacht hat, soll nach Anzeige der Fehlermeldung das Programm beendet werden.

e) Schreiben Sie die Exception-Klassen *AusserhalbSpielfeld* und *GegenWandGelaufen*. Beide Klassen benötigen keine Attribute, Methoden oder Konstruktoren.

f) Erweitern Sie die Methode *geheNach* der Klasse *Labyrinth*, indem Sie an den beiden markierten Stellen Instanzen der Exception-Klassen *AusserhalbSpielfeld* und *GegenWandGelaufen* werfen. Erweitern Sie die Methodensignatur geeignet.

g) Erweitern Sie die Methode *spielen* der Klasse *Labyrinth*, sodass diese dem Benutzer eine Meldung anzeigt, falls dieser einen ungültigen Zug macht. Verwenden Sie hierbei nur einen **try-catch**-Block für alle vier *geheNach*-Aufrufe zusammen.

Aufgabe H22 (10 Punkte)[1]

Auf unserer Webseite finden Sie einen Parser für boolesche Formeln. Die Zeichenketten, die dieser Parser akzeptiert, sind boolesche Formeln, wobei & ein logisches Und repräsentiert, | ein logisches Oder und ! eine logische Negation. Wie in booleschen Formeln üblich hat die logische Negation höchste Präzedenz und das Und höhere Präzedenz als das Oder. Die Literale können entweder *true*, *false* oder ein Variablenname sein. Dabei dürfen Variablennamen nur aus Buchstaben und Zahlen bestehen. Jegliche Arten von Leerzeichen und Zeilenumbrüchen werden ignoriert. Wenn Sie die Funktion *Parse.parse*(*String input*) mit einer solchen Zeichenkette als Parameter aufrufen, erhalten Sie als Rückgabewert einen *Abstract Syntax Tree*, der die Formel darstellt. Ein solcher Abstract Syntax Tree stellt die Struktur der gegebenen Formel als einen Baum dar. Jeder Knoten dieses Baums ist eine Instanz einer Klasse, die das Interface *LogicExpression* implementiert. Es gibt eine Klasse für jeden Operator, für Variablen und für jede Konstante. Diese Klassen finden sie im Paket *parser.ast*.

Dieser Code kompiliert nicht, weil den Klassen, die das Interface *LogicExpression* implementieren, folgende Methoden fehlen:

- **public boolean** *evaluate*(*Map*⟨*String*, *Boolean*⟩ *variableValues*)
  throws EvaluationException;
- **public** *LogicExpression removeUnnecessaryNegations*();
- **public** *LogicExpression negateByDeMorgan*();

Implementieren Sie diese Methoden entsprechend der Kommentare in *LogicExpression*. Dazu ist es nicht notwendig, den Parsingprozess zu verstehen. Benutzen sie die Funktion *Parse.parse*(*String input*) als eine *black box*, um Zeichenketten in ASTs umzuwandeln.

Zur Erinnerung, die De Morganschen Gesetze lauten wie folgt:

$$\neg(P \wedge Q) = (\neg P \vee \neg Q)$$
$$\neg(P \vee Q) = (\neg P \wedge \neg Q)$$

Kniffelaufgabe ohne Punkte: Versuchen Sie zu verstehen, wie der Parser funktioniert. Als Hinweis: Präzedenz funktioniert in diesem Parser, weil es nur zwei binäre Operatoren gibt.

Aufgabe H23 (10 Punkte)[1]

In der Vorlesung wurde ein Programm entwickelt, das ein Tonsignal entschlüsseln kann, welches Morsesignale enthält. Dieses Programm enthält an mehreren Stellen Aufrufe an eine Methode, die Daten mithilfe des k-Means-Algorithmus klassifiziert. Zum Beispiel wird dieses Verfahren verwendet, um typische Längen der Töne für Punkte und Striche zu finden.

Im letzten Übungsblatt entwickelten Sie ein Programm für den k-Medians-Algorithmus. Auf der Webseite steht Ihnen das Programm `Morse.java` zur Verfügung. Finden Sie alle Aufrufe des k-Means-Algorithmus und ersetzen Sie sie durch geeignete Aufrufe an Ihren k-Medians-Algorithmus. Dabei müssen Sie natürlich die verschiedenen Aufrufkonventionen anpassen.

Weiter finden Sie auf der Webseite auch drei Tonsignale mit den Dateinamen `audio.wav`, `withnoise.wav` und `army.wav`. Es sollte möglich sein, sich diese anzuhören. Weiterhin gibt es entsprechende Dateien `audio.raw`, `withnoise.raw` und `army.raw`, welche die Audiodaten als Folge von **double**s enthalten. Dieses Format kann vom Programm `Morse.java` gelesen werden.

Testen Sie Ihr neues Programm an `audio.raw` und `withnoise.raw`. Es sollte beide fehlerfrei in Klartext übersetzen können.

Challenge-Aufgabe: Beim dritten Tonsignal schwanken die Pausenlängen zwischen kodierten Einzelbuchstaben sehr stark. Daher versagt das Programm in seiner jetzigen einfachen Form.

Können Sie das Programm so verbessern, dass es auch mit solchen realistischen Morsesignalen zurechtkommt?

Aufgabe H24 (20)[1]

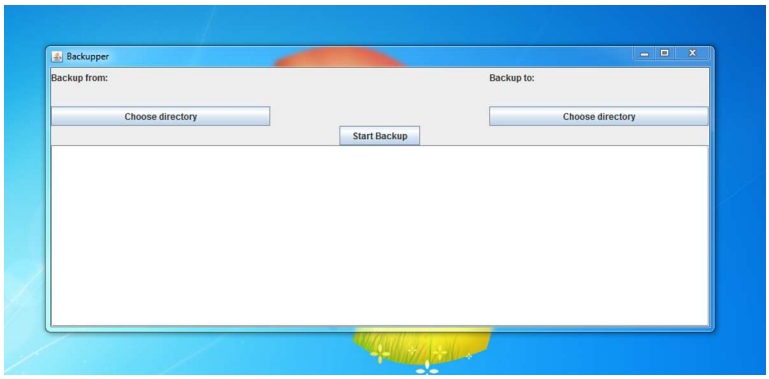
In dieser Aufgabe sollen Sie eine echte Anwendung mit grafischer Benutzeroberfläche in Java erstellen. Diese Anwendung soll einen Backup-Mechanismus implementieren, welcher für zwei vom Benutzer spezifizierte existierende Verzeichnisse auf der Festplatte (möglicherweise auch auf verschiedenen Speichermedien) FROM und TO folgendes tut:

- Alle Dateien und Verzeichnisse in FROM sollen in genau gleicher Form in TO existieren.
- In TO sollen keine anderen Dateien oder Verzeichnisse existieren.
- Falls einige Dateien bereits in TO mit gleichem Pfad, gleicher Dateigröße und gleichem Zeitpunkt ihrer letzten Veränderung wie in FROM existieren, sollen diese nicht verändert werden, um den Backup-Vorgang zu beschleunigen.
- Alle diese Eigenschaften sollen auch für alle Unterverzeichnisse in FROM und TO gelten.

Dazu soll bei Start des Programms ein Fenster für den Benutzer angezeigt werden, welches folgende Komponenten beinhalten soll:

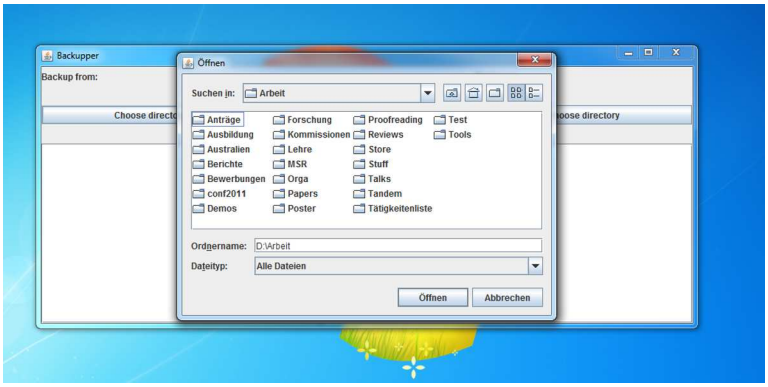
- Eine Schaltfläche zur Auswahl des FROM Verzeichnisses.
- Eine Schaltfläche zur Auswahl des TO Verzeichnisses.
- Eine Anzeige des momentan ausgewählten FROM Verzeichnisses.
- Eine Anzeige des momentan ausgewählten TO Verzeichnisses.
- Eine Schaltfläche zum Starten des Backup-Prozesses.
- Ein scrollbares Textfeld für Status- und Fehlermeldungen.

Darüber hinaus soll das Fenster die üblichen betriebssystemabhängigen Schaltflächen zum Maximieren, Minimieren und Schließen des Fensters aufweisen. Beim Schließen des Fensters soll das Programm beendet werden. Der folgende Screenshot zeigt eine mögliche Realisierung eines solchen Fensters:



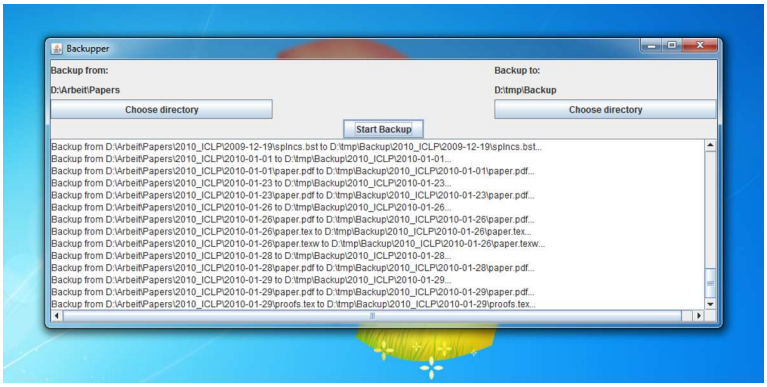
Das Fenster soll durch ein *JFrame* Objekt realisiert werden (schlagen Sie seine Verwendung in der API nach – über Suchmaschinen lassen sich auch hilfreiche Tutorials zu seiner Verwendung finden). Achten Sie darauf, das Fenster über die Methode **public void** *setVisible*(**boolean** *arg*) anzuzeigen. Dieser Aufruf soll im sogenannten Event Dispatch Thread ausgeführt werden (dies ist sinnvoll, um das parallele Verhalten von Grafik und restlichem Programm korrekt zu implementieren – andernfalls kann es leicht zu unerwartetem Verhalten oder gar Deadlocks kommen). Dazu können Sie die Methode *SwingUtilities.invokeLater*(*Runnable arg*) benutzen. Auch hier bietet die API hilfreiche Informationen.

Durch Klick auf eine der Schaltflächen zur Auswahl eines der beiden Verzeichnisse soll ein Dialog angezeigt werden, in welchem der Benutzer das entsprechende Verzeichnis auswählen kann. Nutzen Sie dazu die Klasse *JFileChooser* (wiederum bieten die API und Tutorials aus dem Internet Hilfestellungen). Der Benutzer soll nur Verzeichnisse auswählen können und auch nur eins auf einmal. Der folgende Screenshot zeigt einen solchen Dialog:



Durch Klick auf die Schaltfläche zum Starten des Backup-Prozesses soll dieser in einem neuen Thread gestartet werden. Erzeugen Sie dazu ein neues *Thread* Objekt und rufen Sie dessen *start* Methode auf. Auch hier helfen Ihnen die API und Tutorials aus dem Internet weiter.

Während dieses Backup-Prozesses soll in dem Textfeld angezeigt werden, welche Dateien und Verzeichnisse momentan verarbeitet werden. Sollten Fehler auftreten, sollen diese ebenfalls dort angezeigt werden. Das Textfeld lässt sich am besten über ein *JTextArea* Objekt realisieren. Um die Ausgabe in das Textfeld umzuleiten, bietet es sich an, eine Unterklasse von *OutputStream* mit dem JTextArea Objekt als Attribut zu erstellen und dort die Methode **public void** *write*(**int** *b*) zu überschreiben. Die Klasse *JTextArea* bietet die Methoden **public void** *append*(*String arg*) und **public void** *setCaretPosition*(**int** *arg*) an, welche hier nützlich sein könnten. Um statt des Ihnen bekannten *System.out.println*() die Methode *println*() auf Ihrem eigenen Ausgabestrom anzuwenden, erzeugen Sie einfach ein Objekt vom Typ *PrintStream* und übergeben dem Konstruktor ein Objekt Ihrer Unterklasse von OutputStream. Dann können Sie von dem *PrintStream* Objekt aus die Methode *println*() aufrufen und die Ausgabe wird in das Textfeld umgeleitet. Im folgenden Screenshot ist dies angedeutet:



Um mit Dateien und Verzeichnissen arbeiten zu können, sind die Klassen *File* und *Files* hilfreich. Zu all diesen Klassen und Methoden finden sich hilfreiche Informationen in der API und in Tutorials.

Abgabe zum 14.01.2014

[1]Bitte Quelltext für die Abgabe ausdrucken und zusätzlich per E-mail an den jeweiligen Tutor senden.