

Herbst 2021/22 – A  
252-0027 – Einführung in die Programmierung

Departement Informatik  
ETH Zürich

27. Januar 2022 – Programmieren

Nachname: \_\_\_\_\_ Vorname: \_\_\_\_\_

Legi-Nummer: \_\_\_\_\_ – \_\_\_\_\_ – \_\_\_\_\_ Computer: slab \_\_\_\_\_

Sie dürfen diese Prüfung oder die schriftliche Prüfung erst öffnen nachdem die Aufsicht die Prüfung gestartet hat. **Wenn Sie diese Dokumente vorher öffnen gilt dies als Täuschungsversuch.**

Mit Ihrer Unterschrift bestätigen Sie, dass Sie die hier aufgeführte Person sind, Sie die Hinweise zur Kenntnis genommen haben, Sie die Aufgaben selbständig bearbeitet haben, Sie Ihre eigene Lösung abgeben, Sie keine Kopie der Prüfung mitnehmen, Sie alle technischen Probleme und etwaige störende äussere Einflüsse gemeldet haben bzw. wissen, dass Sie diese melden sollen, und dass Sie keine gesundheitlichen Probleme hatten, die Ihre Leistungen in dieser Prüfung beeinträchtigten.

Unterschrift: \_\_\_\_\_

## Hinweise

1. Bitte schreiben Sie Ihren Namen und Legi-Nummer sowie die Nummer ihres Computers (finden Sie auf dem Computer) auf diese Seite. Vergessen Sie nicht die Unterschrift am Ende der Prüfung.
2. Während der Programmierprüfung dürfen Sie nicht mehr an der schriftlichen Prüfung weiterarbeiten, auch wenn diese noch nicht eingezogen worden ist. **Dies gilt als Täuschungsversuch.**
3. Beachten Sie bitte während und *nach* der Prüfung unbedingt die Hygienevorschriften.
4. Die Prüfung hat 10 Seiten. Vergewissern Sie sich dass Ihr Exemplar vollständig ist. Die letzten zwei Seiten können Sie für Skizzen o.ä. benutzen, aber diese werden nicht für die Benotung hinzugezogen.
5. Die Programmierprüfung dauert 2 Stunden (120 Minuten). Falls Sie sich durch irgendjemanden oder irgendetwas gestört fühlen, oder technische Probleme an Ihrem Computer auftreten, so melden Sie dies sofort der Aufsicht. (Falls es unerwartete Fehlermeldungen gibt: Lassen Sie solche Fehlermeldungen oder PopUp Nachrichten auf dem Bildschirm und informieren Sie die Aufsicht. Nur so verhindern Sie, dass durch Systemfehler Ihre Programme verändert werden. ) Sollten Sie durch die Behandlung eines technischen Problems Zeit verlieren, so werden Sie die verlorene Zeit nachholen können.

6. Wir beantworten keine inhaltlichen Fragen während der Prüfung.
7. Lesen Sie die Aufgabenstellungen genau durch. Es ist wichtig, dass Ihre Antworten den Anforderungen der Aufgaben *genau* entsprechen. Wenn die Aufgabenstellung etwas nicht spezifiziert dann können Sie frei entscheiden (wir testen nur was wir spezifizieren). `Class.name()` heisst Methode `name()` in Klasse `Class`.
8. Benutzen Sie die Anzahl der Sterne in der Programmierprüfung als *Hinweis*, der ungefähr den Aufwand und die erreichbare Punktzahl der Aufgabe widerspiegelt. Je mehr Sterne, desto aufwändiger. Eine gut gelöste Aufgabe gibt mehr Punkte als zwei halb gelöste Aufgaben mit der selben Anzahl Sterne.
9. Für jede Aufgabe gibt es ein separates Java-Projekt in Ihrem Eclipse-Workspace.
10. Die Programmieraufgaben werden vorwiegend automatisch getestet und bewertet. Programme, welche nicht mindestens teilweise ein korrektes Resultat zurückgeben (oder gar nicht erst kompilieren), erhalten keine Punkte.
11. Stellen Sie regelmässig sicher, dass Ihre Dateien *im Workspace* gespeichert sind. Nur diese Dateien werden von einem Backup-Prozess während der Prüfung gespeichert. Was nicht gespeichert ist, kann nicht bewertet werden.
12. Sollten Sie eine Ihrer Lösungsdateien überschreiben, so kann die Aufsicht Ihnen helfen! Melden Sie sich sofort.
13. Ändern Sie unter keinen Umständen die Signaturen der im Aufgabentext erwähnten Methoden (Name, Typ und Reihenfolge der Parameter), ihren Rückgabetyt, Modifizierer wie `static`, `public` oder gegebenenfalls die Liste der geworfenen Exceptions. Das gleiche gilt für Konstruktoren und Attribute. Auch die Namen der erwähnten Klassen dürfen Sie nicht ändern und auch nicht Interfaces in Klassen umwandeln. Solche Änderungen können dazu führen, dass Sie keine Punkte für die Aufgabe erhalten. Wenn nicht anders vermerkt, dürfen Sie Methoden, Attribute, Interfaces oder Klassen zu den vorhandenen hinzufügen oder Klassen und Interfaces importieren. Die Verwendung von Java Reflection ist nicht erlaubt (und auch nicht von Vorteil).
14. Das Verwenden von `static`-Attributen ist grundsätzlich falsch. Rechnen Sie damit, dass wir das abgegebene Programm mehrfach ausführen (und ein Test selber aus mehreren Methodenaufrufen bestehen kann), ohne dass `static`-Attribute neu initialisiert werden. Lösungen, welche `static`-Attribute verwenden, und nur funktionieren, wenn ein Programm/eine Methode nur ein Mal ausgeführt wird, können potenziell 0 Punkte bekommen.
15. In jedem Projekt gibt es neben dem "src"-Ordner einen "test"-Ordner mit einigen JUnit-Tests. Wir empfehlen, diese mit ihren eigenen Tests zu erweitern. **Tests werden nicht bewertet.**
16. Falls gewisse Tests beim Ausführen scheinbar keine Resultate liefern, könnte es daran liegen, dass Ihre Lösung eine Endlosschleife enthält. Stoppen Sie in diesem Fall die Tests von Hand (siehe weitere Hinweise zu Eclipse weiter unten).
17. Als zusätzliche Sicherheitsmassnahme wird Ihr Bildschirm während der Prüfung aufgezeichnet.
18. Wenn Sie in der IDE Zeichen ersetzen statt einfügen, dann drücken Sie die Insert Taste (über der Delete Taste).
19. Auf einer Schweizer Tastatur schreiben Sie eckige und geschweifte Klammern durch `Alt + Ctrl` + die entsprechende Taste links neben der Enter Taste.
20. Wenn Sie zur Toilette müssen, melden Sie dies einer Aufsichtsperson durch Handzeichen. Es darf zur gleichen Zeit immer nur eine Person zur (Unisex)Toilette.
21. Wenn Sie früher abgeben wollen, sperren Sie bitte Ihren Computer und melden Sie sich bitte lautlos. Die Aufsicht wird Ihnen sagen, wann Sie Ihren Arbeitsplatz verlassen können. Vorzeitige Abgaben sind nur bis 20 Minuten vor Prüfungsende möglich.
22. Wenn die Aufsicht die Prüfung beendet, vergewissern Sie sich, dass alle Dateien gespeichert sind. Nach der Sperrung der Computer können Sie keine weiteren Änderungen mehr vornehmen. Befolgen Sie bitte die Anweisungen der Aufsicht (gestaffeltes Verlassen des Prüfungslokals).
23. Verlassen Sie bitte den Prüfungsraum *leise* nach der Prüfung. Es kann sein, dass andere Studierende noch weiterarbeiten da sie eine Zeitgutschrift bekommen haben. Auch diese Studierenden sollen in Ruhe arbeiten können. Bitte lassen Sie unbedingt die unterschriebene Aufgabenstellung auf Ihrem Tisch - wir sammeln diese später ein.

## Anmelden und Eclipse starten

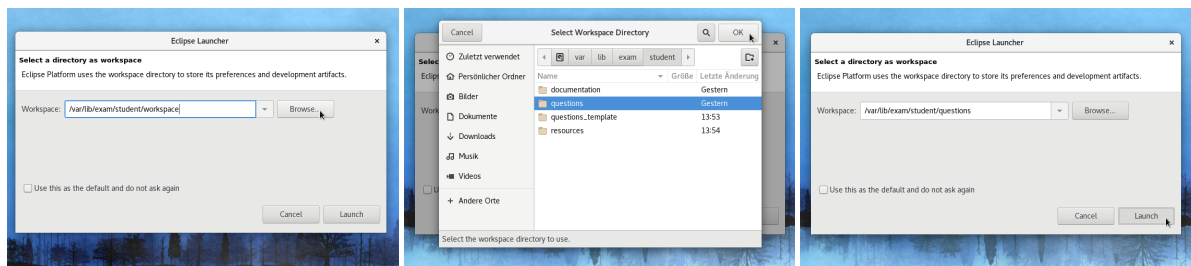
1. Sobald die Programmierprüfung startet, können Sie sich an Ihrem Computer anmelden. Geben Sie zuerst Ihren vollen Namen und im nächsten Schritt Ihren NETHZ-Namen und Ihre Legi-Nummer ein. (Sie brauchen *nicht* Ihr NETHZ-Passwort.) Sie werden auch in einem weiteren Fenster darauf hingewiesen, dass Ihr Computer aufgezeichnet wird, und dass Sie technische Probleme sofort melden müssen. Sobald Sie angemeldet sind, erscheint ein Browsertab mit allgemeinen Hinweisen zur Computer-Prüfung.
2. Starten Sie Eclipse, indem Sie oben links auf “Activities” (oder “Aktivitäten”) klicken und dann im Suchfeld “Eclipse” eingeben. Wählen Sie “Eclipse” (*nicht* “Eclipse C/C++”). Warten Sie, bis Eclipse gestartet ist. Dies kann einige Minuten in Anspruch nehmen.



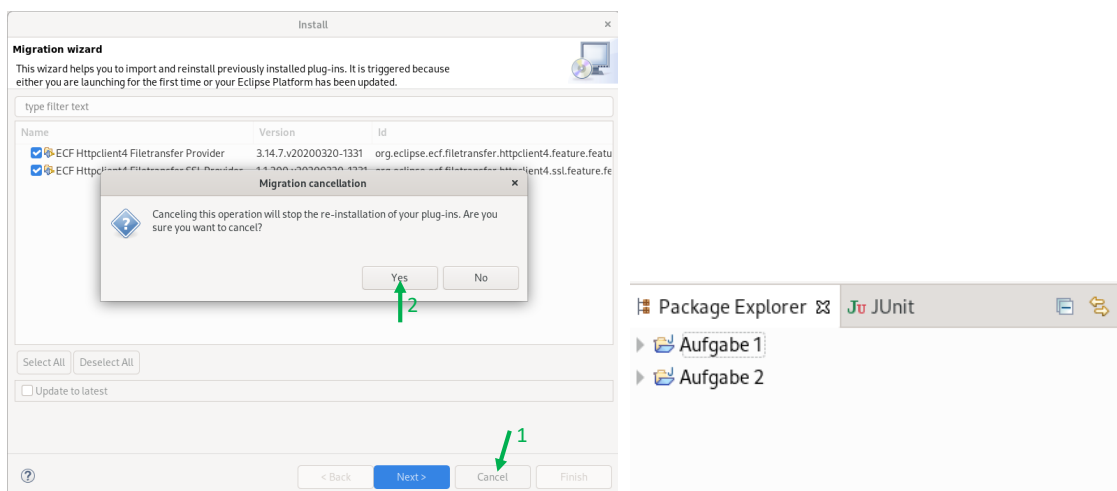
3. Wenn sich das Fenster “Eclipse Launcher” öffnet, stellen Sie sicher, dass der richtige Prüfungs-Workspace ausgewählt ist. Im Feld “Workspace” sollte folgender Pfad stehen, bevor Sie auf “Launch” klicken:

`/var/lib/exam/student/questions`

Falls dies nicht der Fall ist, klicken Sie auf “Browse...” und wählen Sie dann im Auswahldialog den “questions”-Ordner aus. Klicken Sie oben rechts auf “OK” und dann unten auf “Launch”.







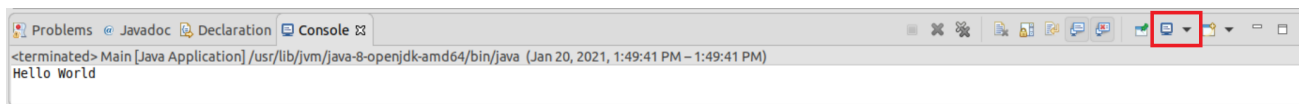
4. Nachdem der Workspace geöffnet wurde, erscheint ein Migration Wizard. Klicken Sie “Cancel” und bestätigen Sie dann mit “Yes”. Wenn Eclipse fertig gestartet ist, sehen Sie den Willkommens-Bildschirm. Klicken Sie wenn nötig oben rechts auf “Workbench”. Nun sollten Sie links die zwei Projekte “Aufgabe 1” und “Aufgabe 2” sehen. **Es kann einige Minuten dauern bis Eclipse alles geladen hat. Warten Sie bis die Ladenachricht “Initializing Java Tooling” (unten rechts in Eclipse) nicht mehr sichtbar ist.** Ausserdem können Sie allgemeine Hinweise zur Computer-Prüfung lesen, indem Sie oben links auf die “Activities” gehen und auf das Informationsicon klicken. Viel Spaß!




# Hinweise zu Eclipse






## Verhindern von Abstürzen

Bevor Sie ein Programm oder einen Test ausführen, achten Sie darauf, dass alle anderen Programme und Tests korrekt terminiert wurden. Wenn zu viele Programme gleichzeitig laufen, dann wird Ihr Computer langsamer, manchmal einfrieren, und im schlimmsten Fall abstürzen. Auch verhält sich dann manchmal Eclipse oder der Debugger unnatürlich. **Das geht von Ihrer Zeit ab.** Ein Problem ist, dass, auch wenn die aktive Konsole mit  terminiert wurde und somit ausgegraut ist () , es noch weitere Konsolen geben kann, welche immer noch laufen. Wenn das Icon  vorhanden und nicht ausgegraut ist () , dann gibt es mehr als eine Konsole, was auch heisst, dass mehrere Programme oder Tests noch nicht terminiert sein können:

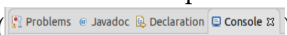



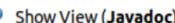


Durch einen Klick auf den Pfeil von , wird eine Liste aller vorhandenen Konsolen angezeigt:

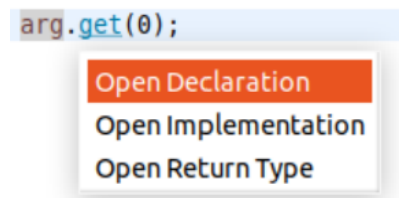


Durch klicken von  werden alle terminierten Konsolen geschlossen. Klicken Sie wiederholt  und  (oder ) bis  ausgegraut oder verschwunden ist, um alle Programme und Tests zu terminieren und alle Konsolen zu schliessen.




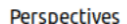


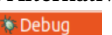
## Javadoc

Es gibt verschiedene Möglichkeiten die Java Dokumentation zu öffnen. Eine komfortable Option ist den Javadoc View zu verwenden. Dieser sollte in einer der Tabs bei der Konsole zu sehen sein (). Falls der Tab nicht vorhanden ist, oder falls Sie den Tab einfach nicht finden, dann können Sie durch das drücken von Alt + Shift + Q und dann J den Tab öffnen. Alternativ können Sie auch im Quick Access Fenster (ganz oben rechts ) "Javadoc" eingeben und dann   oder  drücken.

Wenn Sie den Javadoc View geöffnet haben, dann wird Ihnen die verfügbare Dokumentation von allem gezeigt das Sie anklicken. Zusätzlich, wenn Sie auf etwas zeigen, während Sie die Ctrl Taste gedrückt haben, dann können Sie sich die Declaration anzeigen lassen:



## Debugger

Ob Sie den Debugger verwenden ist Ihre Entscheidung. **Wir werden keine Fragen zum Debugger beantworten.** Die Aufgaben sind auch gut ohne Debugger lösbar. Um in den Debugging Modus zu wechseln, klicken Sie den Debugger Knopf  rechts neben dem Quick Access Fenster (ganz oben rechts ). Falls dieser Knopf nicht vorhanden ist, dann können Sie im Quick Access Fenster (ganz oben rechts ) "Debug" eingeben und dann   drücken. Alternativ können Sie auch den Knopf direkt neben dem Quick Access Fenster drücken  und dann "Debug" wählen .

[Diese Seite ist leer.]

## Aufgabe 1 (★★)

In dieser Aufgabe implementieren Sie zwei morphologische Operationen auf einem  $n \times n$  Grid von *Biomen* (Arten von Landschaften), wobei immer  $n \geq 2$  gilt. Es gibt zwei Arten von Biomen: Flachland und Wasser. Jedes Biom hat eine Floradiversität, welches ein Mass für die Pflanzendiversität ist und durch eine nicht-negative ganze Zahl ( $\geq 0$ ) gemessen wird. Zusätzlich haben Flachlandbiome eine Höhe, welche als eine strikt positive ganze Zahl ( $> 0$ ) angegeben wird.

Im vorgegeben Programmskelett ist eine Klasse *World* gegeben und ein Interface *Biom*. Die Klasse *World* soll eine Repräsentation eines  $n \times n$  Grid von Biomen speichern und darauf die morphologischen Operationen *World.stepDryUp()* und *World.stepDistribute(int p)* ausführen. Das Interface *Biom* repräsentiert Biome und schreibt drei Methoden vor: *Biom.getFlora()* (soll die Floradiversität zurückgeben), *Biom.getHeight()* (soll die Höhe bei Flachlandbiomen zurückgeben; beim Wasser muss immer 0 zurückgegeben werden), und *Biom.getBiomType()*, welche die Biomart als String zurückgibt. Die String-Repräsentation der Biomarten sind durch folgende Strings der Länge 1 definiert:

Biomart	Wasser	Flachland
<i>Biom.getBiomType()</i>	“W”	“F”

Konkrete Implementierungen von *Biom* für die unterschiedlichen Biomarten sind nicht gegeben und diese müssen Sie in den Unteraufgaben selber implementieren. Sie dürfen weitere Methoden zum *Biom*-Interface oder zur Klasse *World* hinzufügen (aber keine Methoden entfernen oder die Interfaces durch Klassen ersetzen). Der Entwurf der Klasse(n) und gegebenenfalls weiterer Interfaces ist Ihnen überlassen.

- (a) (★) Vervollständigen Sie den Konstruktor *World(String [][] biomGrid)*, welcher als Input ein 2D-Array *biomGrid* nimmt. *biomGrid* gibt das  $n \times n$  Startgrid von Biomen für das *World*-Objekt vor ( $n = \text{biomGrid.length}$  und  $n \geq 2$ ). *biomGrid[i][j]* gibt die String-Repräsentation der Biomart an Stelle  $(i, j)$  an ( $0 \leq i, j < n$ ). Die Floradiversität der Biome soll am Anfang für jedes Wasserbiom mit 15 und für jedes Flachlandbiom mit 12 initialisiert werden. Die Höhe der Flachlandbiome soll mit 3 initialisiert werden. Implementieren Sie zusätzlich die Methode *World.getBiom(int x, int y)*, welche das Biom an Stelle  $(x, y)$  vom Grid als Objekt vom Typ *Biom* zurückgibt.

Im nächsten Teil der Aufgabe geht es darum morphologische Operationen *stepDryUp()* und *stepDistribute(int p)* in *World* zu implementieren, welche das Grid von *World* potenziell ändern (die Grösse des Grids bleibt jedoch gleich). Morphologische Operationen wandeln ein Grid  $G$  in ein Grid  $G'$  um. Sei  $G_{i,j}$  (bzw.  $G'_{i,j}$ ) das Biom an Stelle  $(i, j)$  im Grid  $G$  (bzw.  $G'$ ). Dabei wird das neue Biom  $G'_{i,j}$  an Stelle  $(i, j)$  immer nur abhängig vom alten Biom an Stelle  $(i, j)$  (also  $G_{i,j}$ ) und den alten *Nachbar-Biomen* von  $G_{i,j}$  berechnet. Für *stepDryUp()* spielen die Nachbar-Biome keine Rolle. Für *stepDistribute(int p)* sind die Nachbar-Biome von  $G_{i,j}$  die Biome  $G_{i,j-k}$ ,  $G_{i,j+k}$ , bzw.  $G_{i-k,j}$  und  $G_{i+k,j}$  für alle  $k$  mit  $0 < k \leq p$ . Das heisst, für  $p = 1$  sind die Nachbar-Biome gegeben durch  $G_{i,j-1}$  und  $G_{i,j+1}$  [links und rechts daneben] und die Biome  $G_{i-1,j}$  und  $G_{i+1,j}$  [oben und unten]. Sie dürfen davon ausgehen, dass  $p \in \{1, 2\}$  gilt.

Nur die Elemente des Grids, die wirklich existieren, spielen eine Rolle. Zum Beispiel sind für *stepDistribute(1)* die Nachbar-Biome von  $G_{0,0}$  immer nur  $G_{0,1}$  und  $G_{1,0}$  (da die Positionen  $(0, -1)$ ,  $(-1, 0)$  nicht existieren). Für *stepDistribute(2)* bei einem  $n \times n$  Grid mit  $n \geq 3$  sind die Nachbar-Biome von  $G_{0,0}$  immer nur  $G_{0,1}$ ,  $G_{1,0}$ ,  $G_{0,2}$ ,  $G_{2,0}$ . Bei *stepDistribute(int p)* hat ein Biom maximal  $4p$  Nachbar-Biome.

Tabelle 1 (unten) beschreibt für Wasserbiome wie man für die beiden Operationen die neuen Biome von den alten Biomen berechnet; Tabelle 2 (unten) beschreibt dies für Flachlandbiome. Bei *stepDryUp* berechnet man die neue Floradiversität eines Bioms ( $\text{Diversität}_{\text{neu}}$ ) aus der alten Floradiversität des Bioms an der gleichen Position. Bei *stepDistribute* berechnet man die neue Floradiversität eines Bioms aus den alten Floradiversitäten der Nachbarbiome. Die neue Höhe ( $\text{Höhe}_{\text{neu}}$ ) von Flachlandbiomen ist bei beiden Operation von der alten Höhe des Bioms an der gleichen Position abhängig (wobei bei *stepDryUp* ein Flachlandbiom in ein Wasserbiom umgewandelt werden kann).

Lösen Sie die Aufgaben b) und c) (★★).

- (b) Implementieren Sie die Methode `World.stepDryUp()` (siehe Tabellen).

*Beispiel:* Angenommen ein Flachlandbiom hat vorher Floradiversität 5 und Höhe  $h$ . Wenn  $h - 1 \leq 0$ , dann wandelt sich das Biom in ein Wasserbiom mit Diversität  $5 - 3 = 2$ . Wenn  $h - 1 > 0$ , dann bleibt das Biom ein Flachlandbiom, aber mit Diversität  $5 - 3 = 2$  und Höhe  $h - 1$ .

- (c) Implementieren Sie die Methode `World.stepDistribute(int p)` (siehe Tabellen), wobei  $p \in \{1, 2\}$ .

*Beispiel:* Angenommen ein Flachlandbiom an Stelle  $(i, j)$  hat vorher Höhe 8 und zusätzlich hat das Flachlandbiom zwei Flachlandnachbar-Biome und zwei Wassernachbar-Biome. Die Flachlandnachbarn haben Diversität 4 bzw. 5 und die Wassernachbarn haben Diversität 6 bzw. 7. Dann bleibt das Biom an Stelle  $(i, j)$  ein Flachlandbiom, aber mit Diversität  $4 + 5 + 6 + 7 = 22$  und Höhe  $8 + 2 = 10$ .

Tests finden Sie in der Datei "BiomTest.java". Beachten Sie, dass die "BiomTest.java" die Methode `World.stepDistribute(int p)` nur für  $p = 1$  testet. Für die volle Punktzahl muss Ihre Implementierung auch für  $p = 2$  funktionieren.

Methode	Floradiversität
<code>stepDryUp</code>	$\text{Diversität}_{\text{neu}} = \text{Max}(\text{Diversität}_{\text{alt}} - 5, 0)$
<code>stepDistribute</code>	$\text{Diversität}_{\text{neu}} = \text{Summe der Diversität}_{\text{alt}}$ aller Nachbar-Biome

Tabelle 1: Operationen für Wasserbiome. Ein Wasserbiom bleibt nach einer Operation immer ein Wasserbiom.

Methode	Floradiversität	Höhe
<code>stepDryUp</code>	$\text{Diversität}_{\text{neu}} = \text{Max}(\text{Diversität}_{\text{alt}} - 3, 0)$	$\text{Höhe}_{\text{neu}} = \text{Höhe}_{\text{alt}} - 1$ Wenn $\text{Höhe}_{\text{neu}} = 0$ dann verwandelt sich das Biom in ein Wasser Biom (mit $\text{Diversität}_{\text{neu}}$ )
<code>stepDistribute</code>	$\text{Diversität}_{\text{neu}} = \text{Summe der Diversität}_{\text{alt}}$ aller Nachbar-Biome	$\text{Höhe}_{\text{neu}} = \text{Höhe}_{\text{alt}} + (\text{Anzahl Nachbar-Biome die Flachland sind})$

Tabelle 2: Operationen für Flachlandbiome. Nach einer `stepDryUp` Operation wandelt sich ein Flachlandbiom in ein Wasserbiom, wenn die alte Höhe 1 ist. In allen anderen Fällen bleibt ein Flachlandbiom nach einer Operation ein Flachlandbiom.

## Aufgabe 2 (★★★)

Ein Labyrinth besteht aus einer Menge von Räumen, welche durch die Klasse `Room` dargestellt werden. Die Klasse hat zwei Attribute: Der String `name` (nie null) beschreibt den Namen des Raums und der Array `doorsTo` (nie null) beschreibt die Türen von diesem Raum zu anderen Räumen. Alle Türen sind Falltüren, d.h. sie funktionieren nur in eine Richtung. Ein Raum ist ein Ausgang aus dem Labyrinth, wenn keine Türen von dem Raum wegführen, das heisst, wenn `doorsTo` eine Länge von 0 hat. Sie dürfen für alle Aufgaben annehmen, dass es im Labyrinth keinen Zyklus gibt. Das heisst, dass man einen Raum, welchen man durch eine Tür verlassen hat, nie wieder erreichen kann indem man weiteren Türen folgt. Ein Raum hat zu dem gleichen Raum immer nur eine oder keine Tür, nie zwei.

In einem Labyrinth nennen wir eine Sequenz von  $N$  Räumen  $r_1, \dots, r_N$  einen *Lösungspfad* für einen Raum `room` genau dann wenn: (1) Der erste Raum  $r_1$  ist der Raum `room`, (2) der letzte Raum  $r_N$  ist ein Ausgang, und (3) jeder Raum  $r_i$  mit  $1 \leq i < N$  hat eine Tür zum nächsten Raum in der Sequenz  $r_{i+1}$ .

- (a) (★) Implementieren Sie die Methode `Labyrinth.exits(Room room, List<Room> prefix)`, welche eine Liste von Räumen zurückgibt. Die zurückgegebene Liste  $l$  sollte alle Ausgänge  $r_e$  enthalten, für welche gilt, dass es einen Lösungspfad  $r_1, r_2, \dots, r_n$  für `room` gibt, wobei  $r_n = r_e$  gilt und `prefix` ein Präfix von  $r_1, r_2, \dots, r_n$  ist. `prefix` ist ein Präfix eines Lösungspfades  $r_1, r_2, \dots, r_n$  genau dann, wenn  $r_i$  dem  $i$ -ten Raum von `prefix` entspricht für  $1 \leq i \leq \text{prefix.size}()$  und  $n \geq \text{prefix.size}()$ . Jeder Ausgang, welcher die beschriebene Bedingung erfüllt, muss genau einmal in  $l$  enthalten sein und  $l$  darf keine anderen Räume enthalten.

- (b) (★) Sei ein Raum  $r_e$  ein *Ausgang für Raum  $r$*  genau dann, wenn  $r_e$  ein Ausgang ist und von  $r$  erreichbar sind (das heisst, es gibt einen Lösungspfad für  $r$ , der in  $r_e$  endet).

Implementieren Sie die Methode `Labyrinth.sortRooms(List<Room> a)`, welche die Liste  $a$  wie folgt sortieren soll: Ein Raum  $r_1$  muss vor einem Raum  $r_2$  vorkommen, wenn eine der folgenden Bedingungen gilt:

- Die Anzahl Ausgänge für  $r_1$  sind strikt kleiner als die Anzahl Ausgänge für  $r_2$ .
- Die Anzahl Ausgänge für  $r_1$  sind gleich gross wie die Anzahl Ausgänge für  $r_2$ , und es gibt einen Ausgang für  $r_1$  dessen Name aus strikt mehr Zeichen besteht als jeder Name der Ausgänge für  $r_2$ . Die Anzahl Zeichen eines Strings kann man mit `String.length()` berechnen.

Wenn gemäss den Bedingungen  $r_1$  nicht vor  $r_2$  und  $r_2$  nicht vor  $r_1$  vorkommen muss, dann können  $r_1$  und  $r_2$  relativ zueinander in einer beliebigen Reihenfolge auftreten.

- (c) (★★) Zwei Sequenzen  $a$  und  $b$  von Räumen sind *namensgleich* genau dann, wenn  $a$  und  $b$  gleich lang sind und die Namen der Räume aus  $a$  exakt gleich oft bei den Namen der Räume aus  $b$  vorkommen. Zum Beispiel, wenn die Räume in  $a$  die Namen `["A", "A", "B"]` und die Räume in  $b$  die Namen `["A", "B", "A"]` haben, dann sind  $a$  und  $b$  namensgleich (`"A"` kommt bei beiden genau zweimal vor, `"B"` kommt bei beiden genau einmal vor). Wenn die Räume in  $a$  die Namen `["A", "A", "B"]` und die Räume in  $b$  die Namen `["A", "B", "B"]` haben, dann sind  $a$  und  $b$  nicht namensgleich. Die Methode `Labyrinth.sameNames(List<Room> a, List<Room> b)` enthält schon eine korrekte Implementierung von Namensgleichheit. Die Methode gibt `true` zurück genau dann, wenn  $a$  und  $b$  namensgleich sind.

Implementieren Sie die Methode `Labyrinth.pathsWithSameNames(Room room, int n)`. Die Methode soll `true` zurückgeben genau dann, wenn es mindestens  $n$  viele **unterschiedliche** Lösungspfade für das Argument `room` gibt, die *namensgleich* sind. Zwei Lösungspfade  $L_1$  und  $L_2$  sind unterschiedlich, wenn  $L_1$  und  $L_2$  eine unterschiedliche Länge haben oder wenn für mindestens einen Index  $i$  die  $i$ -ten Räume von  $L_1$  und  $L_2$  unterschiedlich sind (also  $L_1$  und  $L_2$  für Index  $i$  unterschiedliche Referenzen enthalten).

Sie dürfen Methoden und Felder der Klasse `Room` hinzufügen. Tests finden Sie in der Datei `"LabyrinthTest.java"`.



**Notizen**

## Notizen