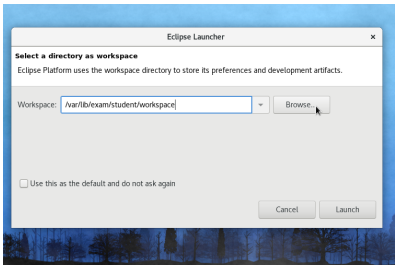
Anmelden und Eclipse starten

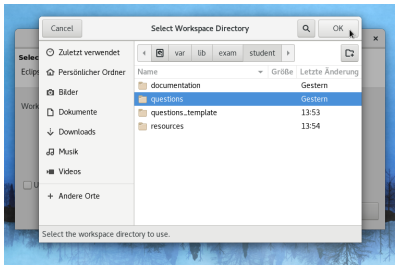
1. Sobald die Programmierprüfung startet, können Sie sich an Ihrem Computer anmelden. Geben Sie zuerst Ihren vollen Namen und im nächsten Schritt Ihren NETHZ-Namen und Ihre Legi-Nummer ein. (Sie brauchen *nicht* Ihr NETHZ-Passwort.) Sie werden auch in einem weiteren Fenster darauf hingewiesen, dass Ihr Computer aufgezeichnet wird, und dass Sie technische Probleme sofort melden müssen.
2. Starten Sie Eclipse, indem Sie oben links auf "Aktivitäten" (oder "Activities") 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.

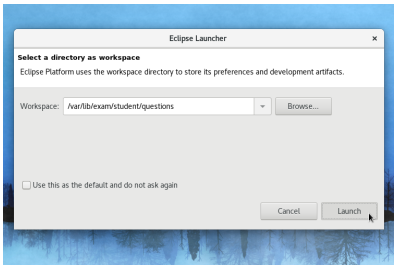
3. Wenn sich das Fenster "Eclipse Launcher" öffnet, stellen Sie sicher, dass der richtige Prüfungs-Workspace ausgewählt ist. Im Feld "Workspace" sollte folgender Pfad stehen, bevor Sie auf "Launch" klicken:

   /var/lib/exam/student/questions

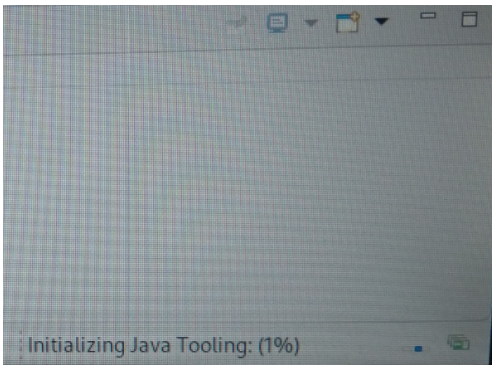
   Falls dies nicht der Fall ist, klicken Sie auf "Browse..." und wählen Sie dann im Auswahldialog den "questions"-Ordner aus. Klicken Sie oben rechts auf "OK" und dann unten auf "Launch".

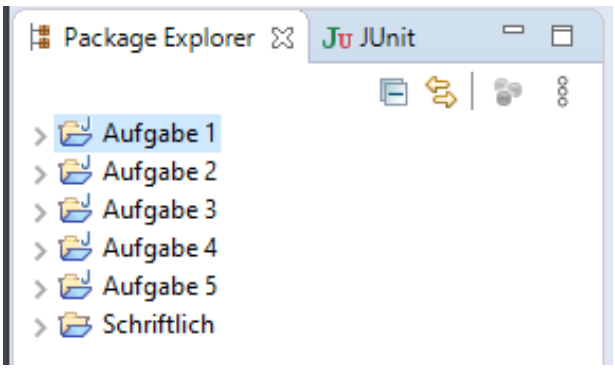






4. Wenn Eclipse fertig gestartet ist, sehen Sie den Willkommens-Bildschirm. Klicken Sie wenn nötig oben rechts auf "Workbench". Nun sollten Sie links die fünf Programmierprojekte "Aufgabe 1" bis "Aufgabe 5" sehen, sowie den Ordner "Schriftlich" für den schriftlichen Teil. Es kann kann einige Minuten dauern bis Eclipse alles geladen hat. Warten Sie bis die Ladenachricht "Initializing Java Tooling" (unten rechts) nicht mehr sichtbar ist. Ausserdem können Sie die rechtlichen Hinweise zur Computer-Prüfung lesen, indem Sie oben links auf die "Activities" gehen (ausserhalb von Eclipse) und auf das Informationsicon klicken. Viel Spass!





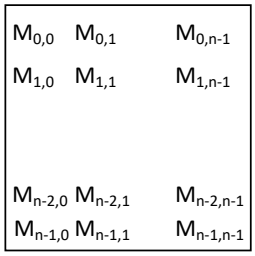
Hinweise

Während der Programmierprüfung dürfen Sie nicht mehr an der schriftlichen Prüfung weiterarbeiten, auch wenn diese noch nicht eingezogen worden ist. **Dies gilt als Täuschungsversuch.**

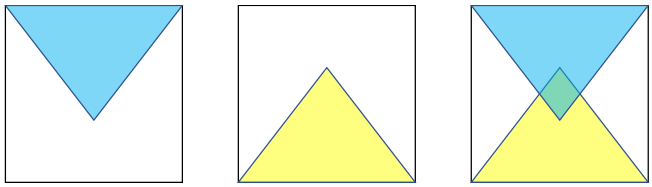
1. Im Prüfungsraum bitte keine Gespräche oder Lärm.
2. Die Programmierprüfung dauert 3 Stunden (180 Minuten). Falls Sie sich durch irgendjemanden oder irgendetwas gestört fühlen, oder technische Probleme an Ihrem Computer auftreten, so melden Sie dies sofort der Aufsichts. (Falls es unerwartete Fehlermeldungen gibt: Lassen Sie solche Fehlermeldungen oder PopUp Nachrichten auf dem Bildschirm und informieren Sie die Aufsicht.) Sollten Sie durch die Behandlung eines technischen Problems Zeit verlieren, so werden Sie die verlorene Zeit nachholen können.
3. 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.
4. Lesen Sie die Aufgabenstellungen genau durch. Es ist wichtig, dass Ihre Antworten den Anforderungen der Aufgaben *genau* entsprechen.
5. Wenn Sie zur Toilette müssen, melden Sie dies einer Aufsichtsperson durch Handzeichen und legen Sie Ihre Maske an. Es darf zur gleichen Zeit immer nur eine Studentin oder ein Student zur Toilette.
6. Wir beantworten keine inhaltlichen Fragen während der Prüfung.
7. Eine gut gelöste Aufgabe gibt mehr Punkte als zwei halb gelöste Aufgaben.
8. Jede Aufgabe ist mit einer Anzahl von Sternen **(★)** versehen, welche ungefähr den Aufwand und die erreichbare Punktzahl der Aufgabe widerspiegeln. Je mehr Sterne, desto aufwändiger.
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 Rückgabetyp, Modifizierer wie `static`, `public` oder gegebenfalls die Liste der geworfenen Exceptions. Das gleiche gilt für Signaturen von Konstruktoren. Auch die Namen der erwähnten Klassen dürfen Sie nicht ändern. Solche Änderungen können dazu führen, dass Sie keine Punkte für die Aufgabe erhalten. Wenn nicht anders vermerkt, dürfen Sie Methoden, Attribute oder Klassen zu den vorhandenen hinzufügen. Ebenso dürfen Sie, sofern keine Einschränkungen aufgeführt sind, Klassen importieren.
14. 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.**
15. 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, und zwar mit dem "Terminate"-Knopf in der "Console View"; ansonsten kann Ihr System einfrieren, was Zeit kostet.
16. Die Prüfungscomputer haben keinen Internet-Zugang. Dadurch kann es in Eclipse zu Fehlermeldungen kommen. Meldungen im Zusammenhang mit fehlendem Internet-Zugang können Sie ignorieren.
17. Die Dokumentation der Java-Klassen ist offline vorhanden. Sie können die "Javadoc"-Ansicht im unteren Teil des Eclipse-Fensters verwenden, um Informationen zu Klassen und Methoden zu erhalten.
18. Als zusätzliche Sicherheitsmassnahme wird Ihr Bildschirm während der Prüfung aufgezeichnet.
19. Wenn Sie in der IDE Zeichen ersetzen statt einfügen, dann drücken Sie die Insert Taste (über der Delete Taste).
20. Auf einer Schweizer Tastatur schreiben Sie eckige und geschweifte Klammern durch `Alt` + `Ctrl` + die entsprechende Taste links neber der Enter Taste.
21. Wenn Sie früher abgeben wollen, melden Sie sich bitte lautlos, und wir holen die Prüfung ab. Vorzeitige Abgaben sind nur bis 20 Minuten vor Prüfungsende möglich.
22. Verlassen Sie bitte den Prüfungsraum *leise* nach der Prüfung. Es kann sein, dass andere noch weiterarbeiten da sie eine Zeitgutschrift bekommen haben. Auch diese Kandidaten sollen in Ruhe arbeiten können. Nehmen Sie bitte keine Aufgabenstellung mit – wir sammeln diese später ein.

Aufgabe 1 (★)

Gegeben sei eine $n \times n$ Matrix M von `int` Werten. n ist ungerade und strikt grösser als 1. Ihre Aufgabe ist es, eine Methode zu implementieren die prüft, ob die Werte der Matrix eine bestimmte Bedingung erfüllen.



Die Matrix M erfüllt die Bedingung wenn Zeilen der Matrix (ganz oder teilweise) übereinstimmen. Die erste Zeile muss die selben Werte haben wir die letzte, in der zweiten Zeile müssen alle Werte bis auf das erste und letzte Element mit der vorletzten Zeile übereinstimmen, und so weiter. In dieser Skizze



muss der blaue Teil mit dem gelben Teil übereinstimmen. Also die Bedingung ist für die Matrix M erfüllt, wenn $M_{0,0} = M_{n-1,0}, M_{0,1} = M_{n-1,1}, \ldots, M_{0,n-1} = M_{n-1,n-1}$ und $M_{1,1} = M_{n-2,1}, M_{1,2} = M_{n-2,2}, \ldots, M_{1,n-2} = M_{n-2,n-2}$, wohingegen $M_{1,0}$ und $M_{n-2,0}$ (und auch $M_{1,n-1}$ und $M_{n-2,n-1}$) unterschiedliche Werte haben können. In der $n/2$-ten Zeile gilt $M_{n/2,n/2} = M_{n/2,n/2}$ trivialerweise. Beachten Sie, dass die Division $n/2$ nach den Regeln der Arithmetik für ganze Zahlen ausgeführt wird, also $5/2 = 2$.

Die folgenden zwei Beispiele zeigen zwei Matrizen, die diese Eigenschaft haben:

```
1 2 3
0 4 5
1 2 3
```

```
1 2 3 4 5
9 3 8 2 8
7 2 1 9 8
6 3 8 2 0
1 2 3 4 5
```

Die naechsten beiden Matrizen haben diese Eigenschaft nicht:

```
1 2 3 4 5
9 3 8 2 8
7 2 1 9 8
6 3 8 2 0
1 2 3 4 0
```

```
1 2 5
0 4 5
1 2 3
```

Ihre Aufgabe ist es, die Methode `Square.checkProperty(int[][] x)` zu implementieren, die prüft ob die Matrix `x` die gewünschte Eigenschaft hat. In dem Fall (und nur in dem Fall) muss diese Methode `true` zurückgeben.

Aufgabe 2 (★★)

Die Klasse `Node` repräsentiert einen Knoten in einem gerichteten Graphen, wobei es für jeden Knoten g höchstens zwei gerichtete Kanten von g zu anderen Knoten f, h geben kann (f, g und h können gleich sein). Wir unterscheiden dabei zwischen der rechten und unteren Kante (und damit dem rechten und unteren Knoten). Die Methode `Node.getRight()` (bzw. `Node.getBottom()`) gibt den rechten Knoten (bzw. unteren Knoten) zurück (als `Node`-Objekt). Wenn der rechte Knoten von n_0 nicht existiert, dann gibt `Node.getRight()` `null` zurück (analog für den unteren Knoten). Die Methode `Node.setRight(Node r)` (bzw. `Node.setBottom(Node b)`) setzt den rechten (bzw. unteren Knoten).

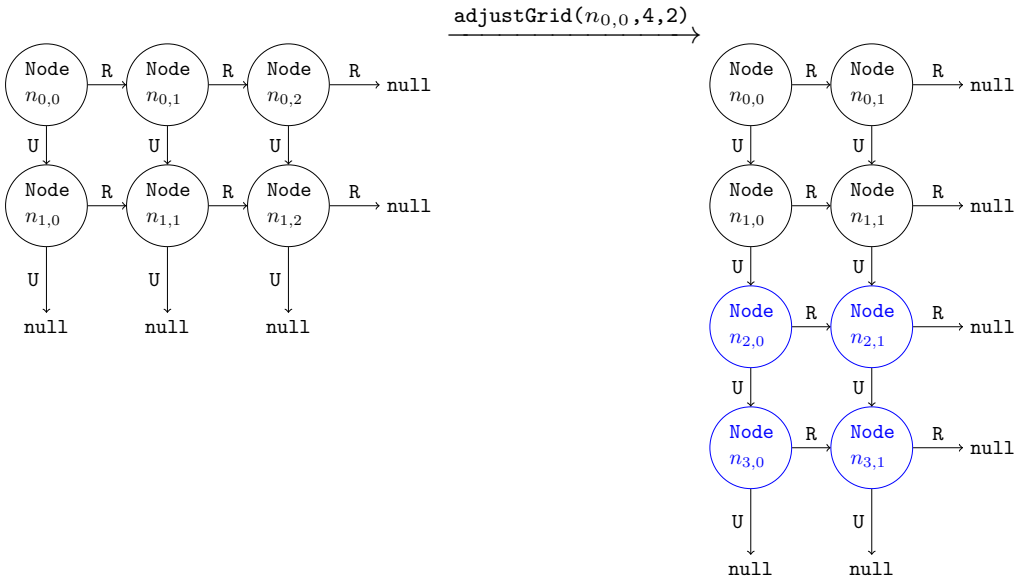
In dieser Aufgabe geht es um `Node`-Objekte, welche ein $N \times M$ Gitter G mit N Zeilen und M Spalten modellieren ($N, M > 0$). Die $N \cdot M$ unterschiedlichen Node-Objekte $\{n_{i,j} | 0 \leq i < N, 0 \leq j < M\}$ modellieren die Knoten $G_{i,j}$ auf Position (i, j) im Gitter genau dann, wenn folgendes gilt (wobei $0 \leq i < N$, $0 \leq j < M$):

- Der rechte Knoten (bzw. untere Knoten) von $n_{i,j}$ existiert genau dann, wenn $j < M - 1$ (bzw. $i < N - 1$).
- Der rechte Knoten von $n_{i,j}$ ist $n_{i,j+1}$ (wenn der rechte Knoten existiert) und der untere Knoten von $n_{i,j}$ ist $n_{i+1,j}$ (wenn der untere Knoten existiert).

Implementieren Sie die Methode `Grid.adjustGrid(Node origin,int A,int B)`. Der erste Parameter ist ein `Node`-Objekt `origin`, welches den Ursprungsknoten $G_{0,0}$ auf Position (0,0) eines $N \times M$ Gitters G ($N, M > 0$) modelliert (`origin` ist nie `null`). Nach dem Aufruf von `adjustGrid` muss `origin` den Ursprungsknoten $H_{0,0}$ eines $A \times B$ Gitters H modellieren (A und B sind die Parameter von `adjustGrid` und es gilt immer $A, B > 0$). Für die Knoten von H muss zusätzlich gelten:

- Für alle i, j mit $0 \leq i < \min(A, N)$ und $0 \leq j < \min(B, M)$ muss der Knoten $H_{i,j}$ durch das bestehende Node-Objekt $n_{i,j}$, welches vor dem Aufruf von `adjustGrid` $G_{i,j}$ modelliert, modelliert werden (mit einem potenziell anderen rechten bzw. unteren Knoten als vor dem Aufruf von `adjustGrid`).
- Für Knoten in H, die nicht aus G übernommen werden, gelten nur die oben aufgeführten Bedingungen für die Modellierung eines $A \times B$ Gitters.

Im folgenden Beispiel sehen Sie den Effekt des Aufrufs `adjustGrid(`$n_{0,0}$`,4,2)` für ein 2×3 Gitter. Das Bild links zeigt die Verlinkung vor dem Aufruf und das Bild rechts zeigt die Verlinking nach dem Aufruf. Die blauen `Node`-Objekte können neu erstellt werden, oder Sie können nicht mehr benötigte `Node`-Objekte verwenden. `R` und `U` verweisen auf den rechten bzw. unteren Knoten.



Beachten Sie dass in diesem Beispiel $n_{1,0}$ sowie vor als auch nach dem Aufruf den Knoten auf Position $(1, 0)$ modelliert. Vor dem Aufruf ist der Verweis auf den unteren Knoten von $n_{1,0}$ `null`, nach dem Aufruf ist der untere Knoten $n_{2,0}$.

Aufgabe 3 (★★)

In dieser Aufgabe implementieren Sie Analysen von Zahlen in einer Zahlenfolge. Eine Zahl in einer Zahlenfolge wird von der Klasse `SeqNumber` repräsentiert. Diese Klasse speichert sowohl die Zahl als Integer, als auch alle Positionen dieser Zahl in der Zahlenfolge als `Set`. Zum Beispiel in der Zahlenfolge "1 3 2 9 2 45" halten wir für die Zahl 1 die Position $\{0\}$ und für die Zahl 2 die Positionen $\{2, 4\}$ fest.

Ihre Aufgabe ist die Klasse `SeqAnalyzer` und ihre Analysen zu implementieren. Die Klasse erfordert einen Konstruktor, welche eine Zahlenfolge aus einem `Scanner` ausliest und damit das Objekt initialisiert.

(a) **(★)** Implementieren Sie den `SeqAnalyzer`-Konstruktor und die Methode `SeqAnalyzer.getNumbers()`. `getNumbers()` muss alle Zahlen in der Zahlensequenz, mit welcher der Konstruktor initialisiert worden ist, als eine Liste von `SeqNumber`-Objekten zurückgeben. Dabei muss jede Zahl, welche in der Zahlensequenz vorkommt, genau ein Mal in der Liste enthalten sein. Die Reihenfolge der zurückgegebenen Liste spielt keine Rolle.

(b) **(★)** Wir definieren nun eine Ordnung *MinDistance* auf Zahlen in einer Zahlenfolge: Eine Zahl x ist kleiner als eine Zahl y in einer Zahlenfolge A, wenn der kleinste Abstand zwischen zwei verschiedenen Positionen von x in A kleiner ist als der kleinste Abstand zwischen zwei verschiedenen Positionen von y in A. Eine Zahl mit nur einer Position in einer Zahlenfolge B ist nie strikt grösser als eine andere Zahl in B und ist immer strikt kleiner als jede Zahl mit mehr als einer Position in B.

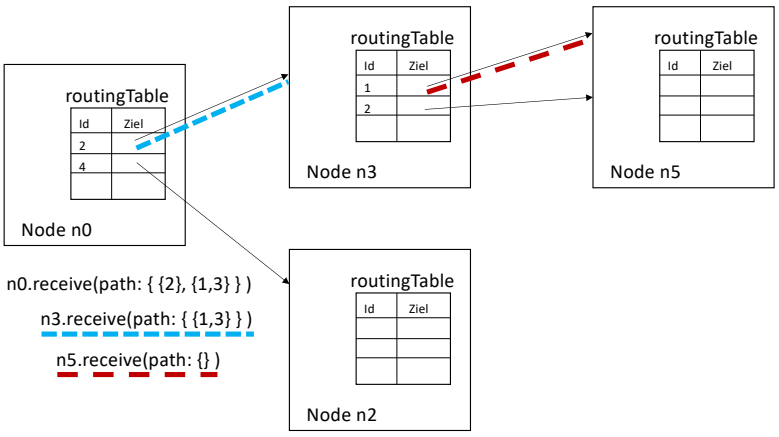
Implementieren Sie die Methode `SeqAnalyzer.getRanking(int n)`, welche eine positive ganze Zahl `n` als Parameter akzeptiert und die `n` grössten Zahlen (gemäss der vorher definierten *MinDistance* Ordnung in einer Zahlenfolge) zurückgibt – als eine Liste von `SeqNumber`-Objekten. Die zurückgegebene Liste muss nach der *MinDistance* Ordnung absteigend sortiert sein, wobei die Reihenfolge von gleich grossen Zahlen in der zurückgegebenen Liste keine Rolle spielt. (Wenn z die letzte Zahl der zurückgegebenen Liste ist, und es eine weitere Zahl w gibt, die nach der *MinDistance* Ordnung genauso gross ist, dann kann w oder z in der Liste zurückgegeben werden.) Wenn es weniger als `n` unterschiedliche Zahlen in der Zahlenfolge gibt, dann muss eine `IllegalArgumentException` geworfen werden. Zum Beispiel muss für die Zahlenfolge "1 2 1" eine `IllegalArgumentException` geworfen werden, wenn `n` grösser als zwei ist (da die Folge nur die zwei Zahlen 1 und 2 hat).

Aufgabe 4 (★★★★)

In dieser Aufgabe implementieren Sie ein Routing Netzwerk, welches Nachrichten durch ein Netz von Knoten leitet. (Sie brauchen keine Netzwerk Kenntnisse, die Aufgabenstellung beschreibt alle Aspekte.) Das Netzwerk besteht aus Knoten, welche das Interface `Node` implementieren. Der erste Teil der Aufgabe ist es, eine Implementation dieses Interfaces (die Klasse `RoutingNode`) zu vervollständigen. Im zweiten Teil werden Sie eine zweite Art von Knoten in einer weiteren Klasse implementieren. Das Interface `Node` erfordert eine Methode `Node.receive(Message msg)`, durch welche ein Knoten k die Nachricht `msg` empfängt (also `k.receive(msg)` lässt k die Nachricht `msg` empfangen). Was ein Knoten macht, wenn er eine Nachricht empfängt, wird durch die Implementation von `Node` (d.h., der Klasse) bestimmt.

Exemplare der Klasse `RoutingNode` leiten eine empfangene Nachricht an einen Nachbarknoten n weiter (indem sie `n.receive(..)` ausführen). Dazu muss der Knoten k den Zielknoten n finden, an den die Nachricht weitergeleitet wird. Jeder Knoten unterhält Verbindungen zu anderen Knoten. Jede Verbindung besteht aus einer ganzen positiven (`int`) Zahl (der VerbindungsId) und der Referenz des Nachbarknotens, der über diese Verbindung erreichbar ist. Die Verbindungen eines Knotens werden in der Routing Table (dem Attribut `routingTable` der Klasse `RoutingNode`) gespeichert. `routingTable` ist vom Typ `Map<Integer, Node>` (und somit kann es für eine Zahl nur einen Eintrag geben).

Jede Nachricht `msg` enthält den Pfad, den die Nachricht durch das Netzwerk nehmen soll. Ein Pfad ist eine Liste; jedes Element der Liste ist eine Menge von VerbindungsIds, über die eine Nachricht weitergeleitet werden kann. Ein Knoten k benutzt das erste Element der Liste von `msg` um den nächsten Knoten n zu finden und ruft dann die Methode `n.receive(aMsg)` auf. Der Pfad der neuen Nachricht `aMsg` besteht aus dem Pfad der von k empfangenen Nachricht `msg` ohne dem ersten Element (also dem Rest der Liste). Wenn die Liste leer ist, dann hat die Nachricht ihr Ziel erreicht. Die Skizze zeigt den Weg einer Nachricht durch ein (einfaches) Netzwerk (von $n0$ über $n3$ nach $n5$), bei welchem der Knoten $n0$ eine Nachricht mit dem Pfad $[\{2\}, \{1, 3\}]$ empfängt.



Jede Nachricht ist ein Exemplar einer Klasse, die das Interface `Message` implementiert, und enthält zusätzlich zum Pfad evtl. noch weitere Attribute. Beachten Sie, dass der Identifier einer Verbindung lokal ist, d.h. dass verschiedene Knoten die gleichen VerbindungsIds haben können (die dann zu irgendwelchen anderen Knoten führen können). So haben sowohl $n0$ als auch $n3$ eine Verbindung mit VerbindungsId 2. Es können mehrere Verbindungen zum selben Knoten führen (siehe Verbindungen 1 und 2 in $n3$), aber für das Routing hier spielt das keine Rolle und erfordert keine Sonderbehandlung.

Für Nachrichten gibt es die `Message.getPath()` Methode, welche den Pfad (eine Liste von Integer Sets) zurückgibt. Die Elemente des ersten Sets geben an, über welche Verbindungen die Nachricht in einem Knoten weitergeleitet werden *könnte*. Es ist möglich, dass für eine Verbindung in dieser Menge (d.h., einer Zahl) kein Eintrag im `routingTable` vorhanden ist. Z.B. ist oben im Knoten $n3$ kein Eintrag mit VerbindungsId 3 im `routingTable`, obwohl die Zahl 3 im ersten Set in der Liste auftritt. Solche VerbindungsIds werden ignoriert. Sie dürfen aber annehmen, dass immer mindestens eine gültige Verbindung im Set ist. Die weiteren Elemente des Pfades bestimmen, wie die Nachricht vom nächsten (und folgenden) Knoten weitergeleitet wird. Daher wird vor dem Weiterleiten der Nachricht das erste Set des Pfades entfernt und eine neue Nachricht mittels `Message.withPath(List<Set<Integer» newPath)` erstellt.

Die Methode `RoutingNode.receive(Message msg)` ist bereits teilweise implementiert. In den folgenden Tasks implementieren Sie Hilfsmethoden, welche dafür verwendet wurden:

(a) **(★)** Implementieren Sie die Methode `RoutingNode.candidates(List<Set<Integer>> path)`, welche den Pfad einer Nachricht nimmt und die Menge der gültigen VerbindungsIds bestimmt. Eine Verbindung ist gültig wenn die VerbindungsId im ersten Set von `path` enthalten ist *und* es einen Eintrag in `routingTable` gibt. Im Knoten $n3$ im obigen Beispiel gibt diese Methode das Set $\{1\}$ zurück.

(b) **(★)** Implementieren Sie die Methode `RoutingNode.selectConnection(Set<Integer> candidates)`, welche ein Set von gültigen VerbindungsIds nimmt und den Identifier mit der höchsten Priorität zurückgibt.

- Für einen Routing Knoten n hat ein Identifier `a` eine höhere Priorität als ein Identifier `b`, wenn über die Verbindung von `a` weniger Nachrichten weitergeleitet wurden als über die Verbindung von `b`.
- Wenn beide Verbindungen gleich oft verwendet wurden, dann hat der kleinere Identifier die höhere Priorität.

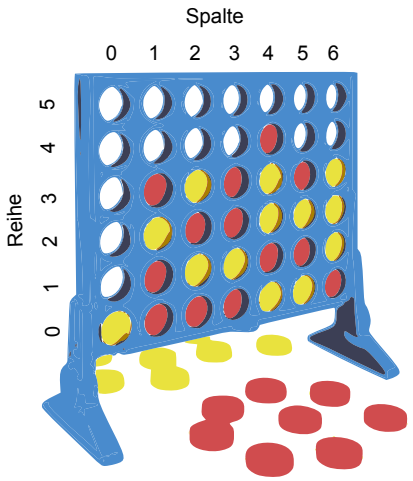
Zur Erinnerung: `candidates` ist nicht leer und enthält nur Identifier, die in der Routing Table von `n` vorkommen.
Implementieren Sie auch die Methode `RoutingNode.incrementCount(int id)`, welche für eine Verbindung (bestimmt durch den Parameter `id`) die Anzahl der weitergeleiteten Nachrichten um 1 erhöht. Sie dürfen annehmen, dass `id` in der Routing Table vorkommt. In der vorgegebenen Implementation von `receive` wird `incrementCount` aufgerufen bevor die Nachricht weitergeleitet wird (d.h. wenn die gewählte Verbindung den Identifier *id* und den Nachbarknoten *n* hat, dann wird zuerst `incrementCount(`*id*`)` und dann `n.receive(...)` aufgerufen).

(c) **(★)** Implementieren Sie die Methode `RoutingNode.process(Message msg)`, welche *zugestellte* Nachrichten verarbeitet. Wie in der Methode `RoutingNode.receive(Message msg)` implementiert, gilt eine empfangene Nachricht als zugestellt, wenn der Pfad der Nachricht leer ist. Wenn einem `RoutingNode` ein Exemplar der Klasse `UpdateMessage` zugestellt wird, dann wird in der Routing Table dieses Knotens ein Eintrag hinzugefügt. Die Attribute `UpdateMessage.newId` und `UpdateMessage.newNode` enthalten dafür die VerbindungsId und die Referenz des Nachbarknotens. Wenn der Identifier in der Routing Table bereits vorhanden ist, dann wird der Eintrag mit dem neuen Knoten überschrieben und die Anzahl der über diese Verbindung weitergeleiteten Nachrichten wird auf 0 gesetzt.
Bei allen anderen Arten von Nachrichten (die natürlich auch `Message` implementieren) passiert nichts.

(d) **(★)** Implementieren Sie zusätzlich die Klasse `CountingNode`. Die Exemplare dieser Klasse sollen genau wie `RoutingNode` Nachrichten weiterleiten und verarbeiten. Aber anders als `RoutingNode` ignoriert `CountingNode` zugestellte Exemplare der Klasse `UpdateMessage`. Wird einem `CountingNode` hingegen ein Exemplar der Klasse `IntMessage` zugestellt, dann soll der Wert des `CountingNode.sum` Attributes dieses Knotens um den Wert des `IntMessage.payload` Attributes erhöht werden. (Sie können davon ausgehen, dass `payLoad` > 0 ist.)

Aufgabe 5 (★★★)

"Vier Gewinnt" (englisch: "Connect Four" oder "Four in a Row") ist ein Zweipersonen-Strategiespiel. Es wird auf einem senkrecht stehenden hohlen Spielbrett gespielt, in das die Spieler **abwechselnd** ihre Spielsteine fallen lassen. Das Spielbrett besteht aus sieben Spalten und sechs Reihen. Jeder Spieler besitzt 21 gleichfarbige (rote oder gelbe) Spielsteine. Wenn ein Spieler einen Stein in eine Spalte fallen lässt, besetzt dieser den untersten freien Platz der Spalte. Gewinner ist der Spieler, der es als erster schafft, vier seiner Steine waagerecht, senkrecht oder diagonal in eine Linie zu bringen. Das Spiel endet unentschieden, wenn das Brett komplett gefüllt ist, ohne dass ein Spieler eine Viererlinie gebildet hat.



(a) **(★)** Implementieren Sie in der Datei `Spielzustand.java` in der Klasse `Spielzustand` die Funktionen `setzeStein(boolean roterSpieler, int spalte)` und `moeglicheSpalten()`, wobei `setzeStein` für den gegebenen Spieler einen Stein in die entsprechende Spalte fallen lässt. Das Argument `roterSpieler` gibt an, ob der Stein für den roten oder gelben Spieler eingeworfen wird. `moeglicheSpalten` berechnet die freien Spalten. Das heisst, es wird ein Set vom Typ `Set<Integer>` zurückgegeben, welches die Nummern der Spalten enthält, die noch Platz haben.

Verwenden sie die zwei-dimensionalen boolean-Arrays `gelbeSteine[spalten][reihen]` und `roteSteine[spalten][reihen]` in der Klasse `Spielzustand`, welche die Positionen der gelben, respektive roten, Steine speichern sollten. Zum Beispiel für den Spielzustand im Bild oben rechts gilt `gelbeSteine[1][3]` = *false* und `roteSteine[1][3]` = *true*.

(b) **(★)** Schreiben Sie in der Datei `Fourinarow.java` in der Klasse `Fourinarow` einen Algorithmus `kannGewinnen(boolean roterSpieler, Spielzustand brett, int n)`, welcher ermittelt, ob es für den Spieler am Zug, möglich ist, eine bestehende "Vier Gewinnt" Partie mit höchstens *n* weiteren Steinen als Sieger zu beenden. Das heisst, falls `kannGewinnen` *true* zurückgibt, gibt es eine Reihenfolge von Spielzügen (für den Spieler am Zug und den Gegenspieler abwechselnd), die zum Sieg des Spielers am Zug führt, wobei der Spieler am Zug höchstens *n* Steine verwenden kann.

Der Spieler am Zug (d.h., der Spieler, welcher zuerst einen Stein einwirft) ist definiert durch das Argument `roterSpieler` (*true* für rot und *false* für gelb) und dieser Spieler ist auch immer der Spieler, für welchen evaluiert werden muss, ob er gewinnen kann. Wenn das Brett voll ist, also kein Stein mehr eingeworfen werden kann, soll `kannGewinnen` *false* zurückgeben.

`kannGewinnen` muss nur Spiele analysieren können, welche noch von keinem Spieler gewonnen wurden, und wirft eine Exception, falls das Spiel schon gewonnen ist.

(c) **(★)** Wir wollen nun `kannGewinnen` benutzen, um Spielstände zu analysieren. Die Methode `unentschieden(boolean roterSpieler, Spielzustand brett)` in der Klasse `Fourinarow` soll herausfinden, ob ein Spielzustand zu einem unausweichlichen Unentschieden führt. `unentschieden` akzeptiert einen Spielzustand und den Spieler, welcher am Zug ist (d.h. der, welcher zuerst einwirft). Wie bei `kannGewinnen` definiert das Argument `roterSpieler`, welcher Spieler am Zug ist.

`unentschieden` muss nur Spiele analysieren können, welche noch von keinem Spieler gewonnen wurden, und wirft eine Exception, falls das Spiel schon gewonnen ist.

Wichtig: Verwenden Sie das bereitgestellte Code-Skelett (ändern Sie nicht die Signaturen der bereits vorhandenen Funktionen/Attributen) und vervollständigen Sie die gegebenen Funktionen, welche sich untereinander aufrufen sollten. Sie können die Funktion `hatGewonnen(boolean roterSpieler)` in der Klasse `Spielzustand` verwenden, um herauszufinden, ob schon eine Viererline besteht und `roterSpieler` gewonnen hat.
Tipp: Es kann von Vorteil sein, gewisse Funktionen rekursiv zu implementieren.

Notizen

Notizen