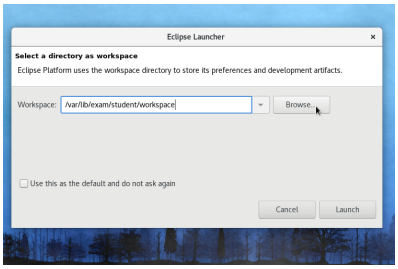
Anmelden und Eclipse starten

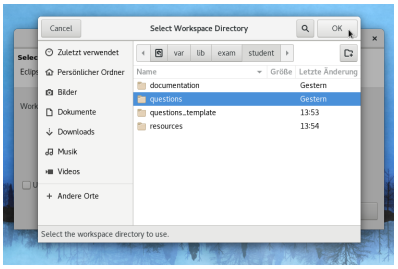
1. Sobald die Programmierprüfung startet, können Sie sich an Ihrem Computer anmelden. Geben Sie zuerst Ihren vollen Namen und im nächsten Schritt Ihren NETHZ-Namen und Ihre Legi-Nummer ein.
2. Starten Sie Eclipse, indem Sie oben links auf "Aktivitäten" klicken und dann im Suchfeld "Eclipse" eingeben. Wählen Sie "Eclipse" (*nicht* "Eclipse C/C++"). Warten Sie, bis Eclipse gestartet ist. Dies kann einige Minuten in Anspruch nehmen.

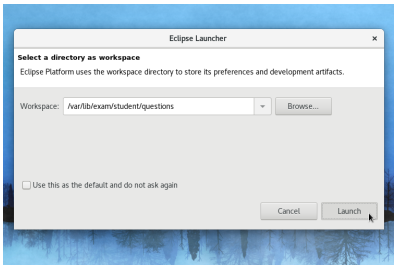
3. Wenn sich das Fenster "Eclipse Launcher" öffnet, stellen Sie sicher, dass der richtige Prüfungs-Workspace ausgewählt ist. Im Feld "Workspace" sollte folgender Pfad stehen, bevor Sie auf "Launch" klicken:

   /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".

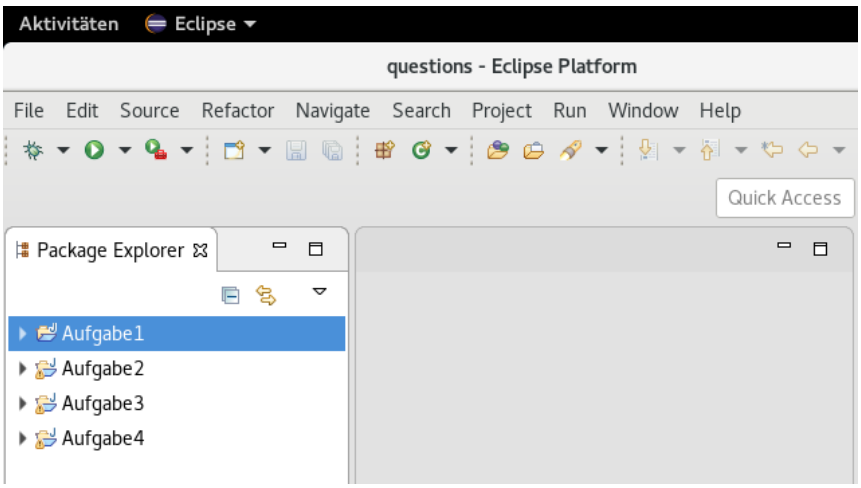






4. Wenn Eclipse fertig gestartet ist, sehen Sie den Willkommens-Bildschirm. Klicken Sie wenn nötig oben rechts auf "Workbench". Nun sollten Sie links die vier Projekte "Aufgabe 1" bis "Aufgabe 4" sehen.

   Erst wenn alle Studierenden diesen Punkt erreicht haben, beginnt die Prüfungszeit. Warten Sie mit dem Öffnen der Projekte, bis die Prüfungzeit offiziell gestartet wird.



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. Öffnen Sie die Projekte der Programmierprüfung erst, nachdem die Aufsicht den Beginn der Prüfung bekannt gegeben hat.
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 Aufsichtsperson. Sollten Sie durch die Behandlung eines technischen Problems Zeit verlieren, so werden Sie die verlorene Zeit nachholen können.
3. Die Prüfung hat 8 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. Es darf zur gleichen Zeit immer nur eine Studentin oder ein Student zur Toilette.
6. Wir beantworten keine inhaltlichen Fragen während der Prüfung.
7. 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.
8. Eine gut gelöste Aufgabe gibt mehr Punkte als zwei halb gelöste Aufgaben.
9. 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.
10. Für jede Aufgabe gibt es ein separates Java-Projekt in Ihrem Eclipse-Workspace.
11. 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.
12. 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.
13. Sollten Sie eine Ihrer Lösungsdateien überschreiben, so kann die Aufsicht Ihnen helfen! Melden Sie sich sofort.
14. Ändern Sie unter keinen Umständen die Signaturen der im Aufgabentext erwähnten Methoden (Name, Parameter oder Rückgabetyp) oder die Namen der erwähnten Klassen. 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.
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, und zwar mit dem “Terminate”-Knopf in der “Console View”; ansonsten kann Ihr System einfrieren, was Zeit kostet.
17. Die Prüfungscomputer haben keinen Internet-Zugang. Dadurch kann es in Eclipse zu Fehlermeldungen kommen. Meldungen im Zusammenhang mit fehlendem Internet-Zugang können Sie ignorieren.
18. 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.
19. Als zusätzliche Sicherheitsmassnahme wird Ihr Bildschirm während der Prüfung aufgezeichnet.
20. Im Prüfungsraum bitte keine Gespräche oder Lärm. 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.

Aufgabe 1 (★)

Schreiben Sie ein Programm, welches den Median einer Folge von `int`-Werten $(x_1, x_2, ..., x_n)$ berechnet und zurückgibt. Der Median ist definiert als der Wert, der sich in der Mitte der sortierten Liste dieser Zahlen befindet. Falls die Anzahl Werte gerade ist, entspricht der Median dem arithmetischen Mittel der beiden am nächsten bei der Mitte liegenden Zahlen. In diesem Fall ist der Median nicht unbedingt eine ganze Zahl.

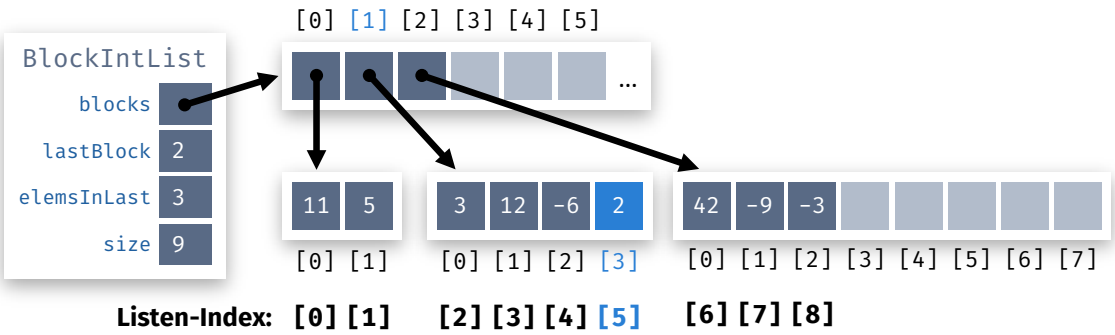
Beispiele:
Der Median von $(1, 5, 4, 3, 0)$ ist 3.
Der Median von $(1000, -100, 0)$ ist 0.
Der Median von $(4, 17, 5, 1)$ ist 4.5.

Implementieren Sie die Berechnung in der Methode `median`, welche sich in der Klasse `Median` befindet. Die Deklaration der Methode ist bereits vorgegeben; sie verwendet einen `Scanner` um die Werte zu übergeben. Sie können davon ausgehen, dass nur `Scanner`-Objekte übergeben werden, die ausschliesslich `int`-Werte enthalten (und mindestens einen davon).

Im "test"-Order finden Sie einen kleinen Unit-Test, der einen Beispiel-Aufruf zur `median`-Methode macht und denn Sie als Grundlage für weitere Tests verwenden können.

Aufgabe 2 (★★★)

In dieser Aufgabe erweitern Sie eine Collection-Klasse namens `BlockIntList`. Diese Klasse verwendet intern mehrere Arrays, sogenannte "Blöcke", um `int`-Werte in der Liste zu speichern. Die folgende Grafik illustriert den Aufbau einer `BlockIntList`, welche neun Elemente in drei Blöcken speichert:



In der Vorlage dieser Aufgabe finden Sie eine funktionierende Implementation von `BlockIntList`, mit den Methoden `add(int value)`, `size()` und `toString()`. Die add-Methode fügt einen neuen Wert nach dem bisher letzten Wert im letzten Block hinzu; falls dieser Block bereits voll ist, wird ein neuer Block hinzugefügt, mit der doppelten Grösse des bisher letzten Blocks. Der erste Block wird im Konstruktor der Liste erstellt und hat immer Grösse 2. Das Array, welches die Blöcke speichert, wird ebenfalls im Konstruktor erstellt und hat eine feste Grösse; wir nehmen an, dass nie mehr Blöcke benötigt werden, als in diesem Array Platz haben.

a) **(★)** Implementieren Sie eine Methode `get(int index)`, welche den Wert beim gegebenen Listen-Index zurückgibt. Die Methode soll für den gegebenen Listen-Index den Block und den darin gültigen Block-Index berechnen und dann den entsprechenden Wert in der Liste zurückgeben. Es ist bereits eine Deklaration für die `get`-Methode vorhanden.

   Beispiel: `list.get(5)` sollte den Block 1 und den Block-Index 3 berechnen und für die oben angebildete Liste den Wert 2 zurückgeben.

   Falls der gegebenene Listen-Index ungültig ist, d.h. falls es keinen Wert mit diesem Index in der Liste gibt, soll die Methode eine Exception vom Typ `IndexOutOfBoundsException` werfen.

b) **(★★)** Implementieren Sie eine Methode `addFirst(int value)`, welche einen Wert am Anfang der Liste (statt wie `add` am Ende) einfügt. Die Methode soll zuerst alle vorhandenen Werte um eine Position weiter in Richtung des Endes der Liste schieben und dann den neuen Wert beim Index 0 einfügen. Für `addFirst` ist ebenfalls bereits eine Deklaration vorhanden.

Sie finden für beide Teilaufgaben (und auch für die bereits implementierten Methoden) einige Unit-Tests in der Vorlage, welche zusätzlich zum Verständnis der Aufgabe beitragen sollten. Zögern Sie nicht, weitere Tests hinzuzufügen.

Achtung: Lösen Sie diese Aufgabe ohne neue Attribute hinzuzufügen und ohne die Typen oder Namen der vorhandenen Attribute zu ändern; andernfalls erhalten Sie keine Punkte. Verändern Sie auch nicht die Deklarationen der vorgegebenen Methoden. Die Implementation der vorhandenenen Methoden `add`, `size` und `toString` dürfen Sie im Prinzip ändern, aber das sollte nicht nötig sein und die Methoden müssen am Schluss immer noch gleich funktionieren.

Aufgabe 3 (★★★)

In dieser Aufgabe implementieren Sie eine Lotterie, welche zuerst Lose verkauft, dann 6 Gewinnzahlen zieht und schliesslich Gewinne ausbezahlt. Die Lotterie besteht aus zwei Klassen, `Lottery` und `Ticket`. Die beiden Klassen können folgendermassen verwendet werden:

```
Lottery lottery = new Lottery(42); // Zahlen 1 bis 42
Ticket tic1 = lottery.buyTicket(new int[] {1, 42, 33, 7, 21, 12});
Ticket tic2 = lottery.buyTicket(new int[] {5, 10, 15, 20, 25, 30});

// jetzt wird gezogen:
lottery.draw();
System.out.println("Gewinnzahlen: " + lottery.getWinning());

// Lose "wissen" jetzt den Gewinn und die korrekten Zahlen:
System.out.println("Los " + tic1.getNumber() + ":");
System.out.println("Richtige: " + tic1.getCorrectNumbers());
System.out.println(tic1.getPrize() + " CHF gewonnen!");
System.out.println("Los " + tic2.getNumber() + ":");
System.out.println("Richtige: " + tic2.getCorrectNumbers());
System.out.println(tic2.getPrize() + " CHF gewonnen!");
```

In der Vorlage finden Sie das Skelett für die beiden Klassen sowie eine Test-Klasse `LotteryTest`, welche Sie als Starthilfe für das Testen Ihrer Lösung verwenden können.

a) **(★)** Implementieren Sie den `Lottery`-Konstruktor, welcher die grösste mögliche Zahl für diese Lotterie als Argument nimmt. Gehen Sie davon aus, dass diese Zahl $\geq$ 6 ist; andernfalls ist das Verhalten der Lotterie undefiniert. Implementieren Sie dann die Methode `buyTicket`, welche ein Array von Zahlen nimmt und ein Los zurück gibt. Das Los soll diese Zahlen in einem `Set` speichern; man kann sie mittels `Ticket.getNumbers` abfragen. Jedes Los hat zudem eine (für dieses `Lottery`-Objekt) eindeutige Losnummer (das erste Los ist Nummer 1, das zweite Nummer 2, usw.), welche mit `Ticket.getNumber` abgefragt werden kann. Implementieren Sie schliesslich die Methode `Lottery.soldTickets`, welche die Anzahl der bisher verkauften Lose zurückgibt. Beachten Sie den Unit-Test `testBuyTicket`.

   `buyTicket` soll ausserdem prüfen, dass das gebenene Array mit den Lotto-Zahlen gültig ist, d.h. genau 6 verschiedene Zahlen enthält, welche nicht kleiner als 1 und nicht grösser als die grösstmögliche Zahl dieser Lotterie sind. Andernfalls soll die Methode eine `IllegalArgumentException` werfen. Siehe Unit-Test `testIllegalNumbers`.

b) **(★½)** Implementieren Sie `Lottery.draw`, welche "zufällig" 6 verschiedene (!) Zahlen zwischen 1 und der maximalen Zahl (inklusive) wählt. Verwenden Sie `Math.random` oder die `Random`-Klasse um die Zufallszahlen zu generieren. Die Gewinnzahlen soll man danach mit der `getWinning`-Methode abrufen können. Weiter soll man über die `Ticket.getCorrectNumbers`-Methode für jedes verkaufte Ticket abfragen können, welche Zahlen dieses Loses korrekt waren. Die `Ticket.getPrize`-Methode berechnet für ein Los zudem den Gewinn, welcher wie folgt definiert ist: für genau eine richtige Zahl beträgt der Gewinn 5 CHF; für jede weitere richtige Zahl vervielfacht sich der Gewinn 20×. (Wenn gar keine Zahl richtig war, gibt es 0 CHF.) Siehe Unit-Tests `testDraw`, `testCorrectNumbers` und `testPrize`.

c) **(½)** Als letztes sollen Sie noch sicherstellen, dass die `Lottery`-Klasse nicht missbraucht wird. Es ist z.B. verboten, Lose zu kaufen, *nachdem* die Ziehung bereits stattgefunden hat; in einem solchen Fall soll die `buyTicket`-Methode eine `IllegalStateException` werfen. Andere Methoden-Sequenzen sind ebenfalls nicht erlaubt, z.B. das Abfragen der Gewinnzahlen oder des Gewinns eines Loses *bevor* gezogen wurde. Verwenden Sie den Unit-Test `testIllegalState` um solche verbotenen Aufrufe zu identifizieren. Dieser Test ist vollständig, d.h. es gibt keine weiteren illegalen Sequenzen als die, welche dort getestet werden.

Aufgabe 4 (★★★★)

Sie wurden eingestellt als Inspektor für Labyrinthe und sollen eine Reihe von Aufgaben lösen.

Ein Labyrinth besteht aus einer Menge von Räumen, welche durch die Klasse `Room` dargestellt werden. Die Klasse hat zwei Eigenschaften: Der Integer `color` (zwischen 0 und 9 einschliesslich) beschreibt die Farbe des Raums und die Liste `doorsTo` beschreibt die Türen von diesem Raum zu anderen Räumen. Alle Türen sind Falltüren, d.h. sie funktionieren nur in eine Richtung. (Es ist jedoch möglich, dass zwei Räume je eine Falltür haben, die zum jeweils anderen Raum führt. Auch kann ein Raum eine Falltüre zu sich selber haben.) Ein Raum ist ein Ausgang aus dem Labyrinth, wenn keine Türen von dem Raum wegführen, das heisst, wenn `doorsTo` leer ist.

Für alle Aufgaben werden Sie in einen zufälligen Raum geworfen, welcher als Argument gegeben wird und von welchem aus Sie die Aufgabe lösen müssen.

a) **(★)** Ihre Chefin ist besorgt, dass Besucher nicht Räume aller 10 Farben besichtigen können. Daher ist Ihre erste Aufgabe zu prüfen, ob Sie von einem gegebenen Raum (auf irgendeinem Weg) einen Ausgang erreichen können, sodass ein Raum jeder Farbe exakt einmal durchschritten wird (inklusive dem Anfangs- und Ausgangs-Raum). Implementieren Sie dafür die Methode `colorExactlyOnce` in der Datei "Labyrinth.java".

b) **(★)** Ihre Chefin ist mit dem vorherigen Konzept unzufrieden. Ihre neue Idee ist, dass ein Labyrinth nur dann gut ist, wenn man zu einem Ausgang gelangen kann, ohne zwei Räume der gleichen Farbe direkt hintereinander zu verwenden. Ihre zweite Aufgabe ist es, diese Eigenschaft für einen gegebenen Raum zu überprüfen. Implementieren Sie dafür die Methode `colorNotSuccessively`.

c) **(★★)** Bei all den Kriterien Ihrer Chefin haben Sie Ihre ursprüngliche Aufgabe vernachlässigt: Die Sicherheit der Besucher. Manche Besucher haben das Labyrinth seit Tagen nicht verlassen, weil das Labyrinth eine Schleife hat. Das heisst, es gibt einen Sequenz von verschiedenen Räumen $s_1, s_2, \ldots, s_N$ (mindestens zwei), sodass jeder Raum eine Tür zum nächsten Raum in der Sequenz hat und der letzte Raum eine Tür zum ersten Raum der Sequenz besitzt. Eine Schleife besteht bereits, wenn es zwei Räume gibt, welche beide eine Tür zum jeweils anderen Raum haben.

   Prüfen Sie für einen gegebenen Raum, ob Sie von diesem Raum eine Schleife erreichen können, und falls das der Fall ist, dann entfernen Sie genau jede Tür, welche für diese Schleife benötigt wird. Dieser Ablauf kann zu neuen Ausgängen führen. Entfernen Sie keine Türen, welche nicht zu einer Schleife gehören. Beachten Sie des Weiteren, nur Türen aber keine Räume zu entfernen. Falls mehrere Schleifen erreicht werden können, dann entfernen Sie eine beliebige Schleife. Implementieren Sie dafür die Methode `removeCycle`.

Sie dürfen Methoden und Felder der Klasse `Room` hinzufügen. Beachten Sie aber, dass ein Labyrinth möglicherweise mehrmals "inspiziert" wird. Tests finden Sie in der Datei "LabyrinthTest.java".

Notizen

Notizen