
Übungsblatt 4

Ausgabe: 13.12.2016 – 17:00
Abgabe: 12.01.2017 – 07:00

- Achten Sie darauf nicht zu lange Zeilen, Methoden und Dateien zu erstellen.¹
- Programmcode muss in englischer Sprache verfasst sein.
- Kommentieren Sie Ihren Code angemessen: So viel wie nötig, so wenig wie möglich.
- Wählen Sie geeignete Sichtbarkeiten für Ihre Klassen, Methoden und Attribute.
- Verwenden Sie keine Klassen der Java-Bibliotheken ausgenommen Klassen der Pakete `java.lang` und `java.io`, es sei denn die Aufgabenstellung erlaubt ausdrücklich weitere Pakete. Verwenden Sie also insbesondere keine Klassen aus `java.util` und Unterpaketen `java.util.[...]`.¹
- Achten Sie auf fehlerfrei kompilierenden Programmcode.¹
- Halten Sie alle Whitespace-Regeln ein.¹
- Halten Sie die Regeln zu Variablen-, Methoden und Paketbenennung ein und wählen Sie aussagekräftige Namen.¹
- Halten Sie die Regeln zur Javadoc-Dokumentation ein.
- Nutzen Sie nicht das default-Package.¹ Legen Sie die von Ihnen definierten Klassen in Unterpaketen `edu.[...]`, `com.[...]`, `org.[...]`, `net.[...]` oder `de.[...]` ab, beispielsweise in `edu.kit.informatik`.
- Halten Sie auch alle anderen Checkstyle-Regeln ein.

Abgabemodalitäten

Die Praktomat-Abgabe wird am **Mittwoch, den 21. Dezember 2016, um 13:00 Uhr**, freigeschaltet.

- Geben Sie Ihre Antworten zu den Aufgaben A, B, C und D jeweils als `.java`-Dateien ab.
- Die Beispielaufgaben für die Präsenzübung werden **nicht** abgegeben oder korrigiert.

Bitte beachten Sie, dass das erfolgreiche Bestehen der öffentlichen Tests für eine erfolgreiche Abgabe nötig ist. Planen Sie entsprechend Zeit für Ihren ersten Abgabeversuch ein.

Allgemeine Hinweise

- Übernehmen Sie die Methodensignaturen und Klassennamen genau wie in der Aufgabenstellung angegeben, falls diese vorgegeben sind. Sie können weitere Klassen und öffentliche Methoden hinzufügen. Es empfiehlt sich, private Hilfsmethoden und Konstanten hinzuzufügen um gegebenenfalls Code-Duplikate zu vermeiden und Ihren Code besser zu strukturieren.
- Sie können beim Lösen *dieses Übungsblattes* davon ausgehen, dass die Argumente für die beschriebenen Methoden immer valide sind, es sei denn, dies ist anders spezifiziert. Das bedeutet beispielsweise, dass nie `null` übergeben wird, es sei denn, dies ist explizit in der Methodenbeschreibung erwähnt. Weiterhin haben

¹Der Praktomat wird die Abgabe zurückweisen, falls diese Regel verletzt ist.

übergebene Felder bspw. immer korrekte Länge und Aufbau. Wenn Sie die Methoden aus Ihrem eigenen Code aufrufen müssen Sie natürlich trotzdem dafür sorgen, dass Sie nur valide Argumente übergeben.

A Sortierte Liste (7 Punkte)

Hinweis: Sie können im Folgenden immer davon ausgehen, dass die Argumente für die beschriebenen Methoden nicht den Wert `null` haben.

A.1 `interface SortedAppendList<T extends Comparable<T>>`

Erstellen Sie ein Interface `SortedAppendList<T extends Comparable<T>>`. `Comparable<T>` ist hierbei das Interface `java.lang.Comparable<T>`. Die angegebene Notation bedeutet, dass der Typ-Parameter `T` nur mit Typen belegt werden darf, die `java.lang.Comparable<T>` implementieren, also beispielsweise `String`, welcher `Comparable<String>` implementiert (und daher eine Methode `public int compareTo(String anotherString)` anbietet). Durch eine solche Einschränkung des Typ-Parameters ist es möglich, auf Objekten des Typs `T` die in `Comparable<T>` definierte Methode `compareTo(T o)` mit einem Argument vom selben Typ `T` aufzurufen. Das Interface soll die folgenden Methoden enthalten. [Die Beschreibung der Methoden drückt sich nicht durch Implementierung im Code der Schnittstelle aus, sondern beschreibt das erwartete Verhalten für Klassen, die die Schnittstelle implementieren. Eine Schnittstelle enthält i. d. R. nur Deklarationen](#)

A.1.1 `void addSorted(T element)`

Fügt der Liste ein neues Element hinzu. Das Element wird so in der Liste eingeordnet, dass alle Elemente weiterhin korrekt aufsteigend sortiert sind. Die Reihenfolge von gleichen Elementen, für die `compareTo` den Wert 0 zurückgibt, ist nicht relevant. [\(Hinweis: Sichtbarkeits-Modifier `public` entfernt.\)](#)

A.1.2 `SortedIterator<T> iterator()`

Gibt einen neuen Iterator für die Liste zurück. Der Typ `SortedIterator<T>` wird im Folgenden weiter beschrieben. [\(Hinweis: Sichtbarkeits-Modifier `public` entfernt.\)](#)

A.2 `interface SortedIterator<T extends Comparable<T>>`

Erstellen Sie eine Schnittstelle `SortedIterator<T extends Comparable<T>>`. Diese enthält die beiden in der Vorlesung für die Schnittstelle `Iterator` beschriebenen Methoden `boolean hasNext()` und `T next()`. Diese Schnittstelle hat zusätzlich die Bedeutung, dass für zwei nacheinander von der Methode `next()` zurückgegebene Werte `t0` und `t1` immer gilt `t0.compareTo(t1) <= 0`.

Diese Zusicherung (Sortierung) müssen Sie nicht im Code *der Schnittstelle* ausdrücken, aber Sie muss von den Klassen, die `SortedIterator` implementieren eingehalten werden.

A.3 `LinkedSortedAppendList<T extends Comparable<T>>`

[\(Hinweis: `extends Comparable<T>` hinzugefügt.\)](#)

Erstellen Sie eine Klasse `LinkedSortedAppendList<T extends Comparable<T>>`, die die in Unterabschnitt A.1 beschriebene Schnittstelle `SortedAppendList<T>` als **verkettete Liste** implementiert. Sie müssen nur die dort beschriebenen Methoden (`addSorted(T)` und `iterator()`) implementieren und **nicht** die vollständige Funktionalität einer Liste, wie Sie in der Vorlesung vorgestellt wurde.

Konkret bedeutet dies, dass (in der Notation aus der Vorlesung) für jede von der ersten Listenzelle aus erreichbare Zelle `cell` gilt: Wenn `cell.next != null`, dann

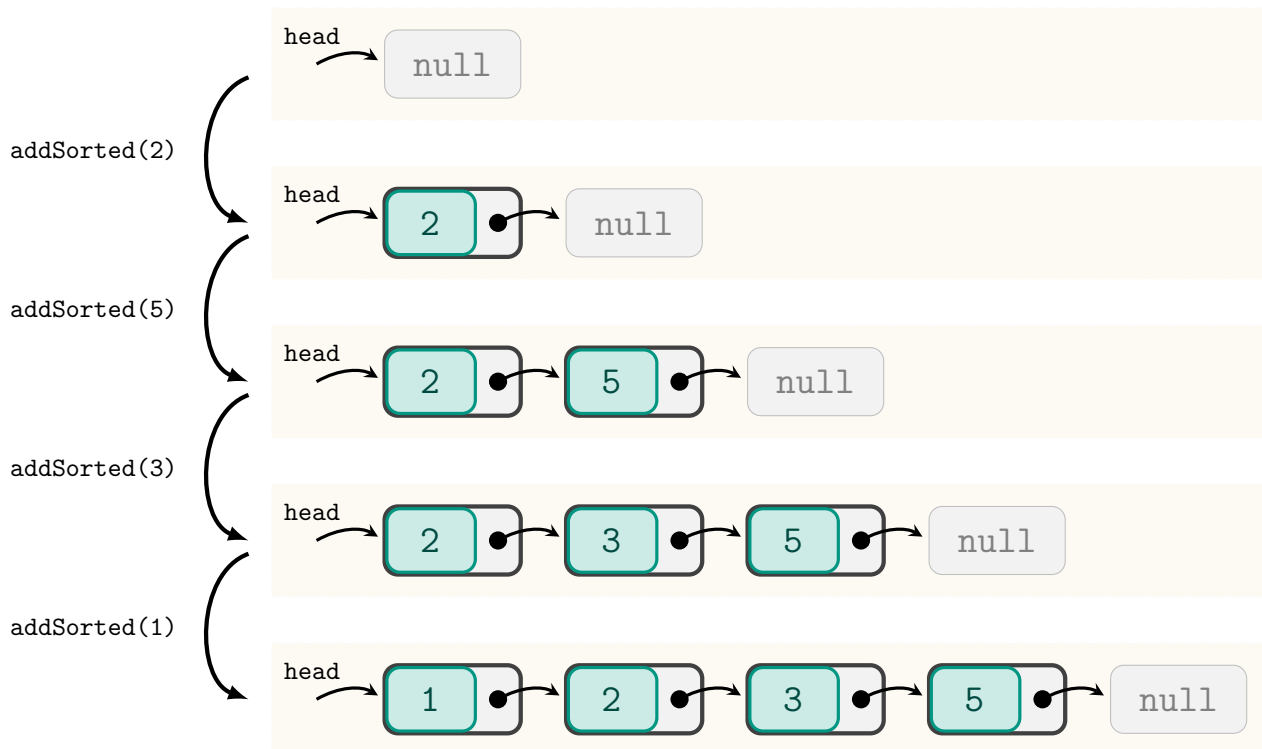


Abbildung 1: Verhalten einer `LinkedSortedAppendList<Integer>` für das aufeinanderfolgende Einfügen von 2, 5, 3 und 1 mit der Methode `addSorted(Integer)`.

```
cell.content.compareTo(cell.next.content) <= 0 .
```

Sie können hierzu den passenden Code für einfach verlinkte Listen aus der Vorlesung geeignet wiederverwenden. Im Folgenden ist beschrieben, wie Sie die in der Vorlesung vorgestellte einfach verkettete Liste anpassen müssen, um die Schnittstelle `SortedAppendList<T>` zu erfüllen. Es steht Ihnen frei, eine andere Implementierung einer verketteten Liste (wie bspw. eine doppelt verkettete Liste oder eine Liste mit Sentinel-Element²) zu wählen, falls dies für Sie einfacher ist. Dies fließt nicht in die Bewertung ein. Sie können bei Ihrer Implementierung die benötigten Klassen für Zellen und Iterator in die Klasse `LinkedSortedAppendList<T>` einschachteln (als *inner classes*³), können dies aber auch durch ganz normale Klassen, wie in der Vorlesung vorgestellt, tun. Geben Sie im letzteren Fall ggf. Interna der Implementierung der Zelle, des Iterators, oder der Liste durch entsprechende *getter*-Methoden nach außen (entgegen dem Geheimnisprinzip). Auch diese Entscheidung fließt nicht in die Bewertung ein.

Falls Sie die einfach verkettete Liste aus der Vorlesung verwenden möchten, passen Sie deren Code so an, dass ein Element immer so in der Liste eingefügt wird, dass die Ordnung erhalten bleibt. Erstellen Sie hierzu (wie in der Vorlesung beschrieben) eine neue Listenzelle und laufen Sie durch die vom Kopfzeiger (`head`) erreichbaren Zellen ab, bis Sie die korrekte Position erreichen. Fügen Sie dort die neu erstellte Listenzelle ein.

Das Verhalten ist in Abbildung 1 beispielhaft für eine `LinkedSortedAppendList<Integer>` für das aufeinanderfolgende Einfügen von 2, 5, 3 und 1 dargestellt. Die Darstellung geht hierbei von einer Implementierung einer einfach verketteten Liste wie in der Vorlesung vorgestellt aus.

Hinweis: Halten Sie falls nötig beim Durchlaufen der Elemente sowohl eine Referenz auf das aktuelle Element, als auch auf das dem aktuellen vorhergehende Element. Durch die obige Einschränkung des Typ-Parameters `T` können Sie in dieser Methode `t.compareTo(...)` aufrufen.

²https://en.wikipedia.org/wiki/Sentinel_value

³<https://docs.oracle.com/javase/tutorial/java/java00/innerclasses.html>

B Terminerweiterung (3 Punkte)

In dieser Aufgabe soll der Kalender, der auf den vorherigen Übungsblättern implementiert wurde, erweitert werden. Laden Sie sich hierzu zunächst die Musterlösung von Übungsblatt 2 herunter oder verwenden Sie Ihre eigene Lösung weiter.

Hinweis: Sie können im Folgenden immer davon ausgehen, dass die Argumente für die beschriebenen Methoden nicht den Wert `null` haben. Weiterhin können Sie davon ausgehen, dass die gegebenen **Appointments** immer einen Startzeitpunkt haben, der echt vor dem Endzeitpunkt liegt.

B.1 Appointment implements Comparable<Appointment>

Ändern Sie die Klasse `Appointment` so ab, dass Sie die Schnittstelle `Comparable<Appointment>` implementiert. Implementieren Sie die in der Schnittstelle beschriebene Methode `compareTo(Appointment o)` folgendermaßen:

- Gib einen negativen Wert zurück, wenn der Startzeitpunkt von `this` vor dem Startzeitpunkt von `o` liegt.
- Sind die Startzeitpunkt von `this` und `o` gleich, dann:
 - Gib einen negativen Wert zurück, wenn der Endzeitpunkt von `this` vor dem Endzeitpunkt von `o` liegt.
 - Gib einen positiven Wert zurück, wenn der Endzeitpunkt von `this` nach dem Endzeitpunkt von `o` liegt.
- Sind sowohl Start- als auch Endzeitpunkt der beiden Termine gleich, dann:
 - Gib einen negativen Wert zurück, wenn der Name von `this` lexikalisch vor dem Namen von `o` liegt. Dies kann über `String#compareTo(String)` geprüft werden.
 - Gib einen positiven Wert zurück, wenn der Name von `this` lexikalisch hinter dem Namen von `o` liegt.
- Gib 0 zurück, wenn sowohl Start- und Endzeitpunkt, als auch Namen von `this` und `o` genau gleich sind.

Die daraus resultierende Sortierung ist beispielhaft in Abbildung 2 dargestellt. Hierbei gilt für einen Termin `t0`, der in der Grafik über einem Termin `t1` steht immer: `t0.compareTo(t1) <= 0`. Der Name des Termins ist ausgespart. Nur für die Termine `f` und `g` kann bei Namensgleichheit `f.compareTo(g) == 0` gelten. Für alle anderen Paare ist das Ergebnis der `compareTo`-Methode unabhängig vom Namen immer ungleich 0.

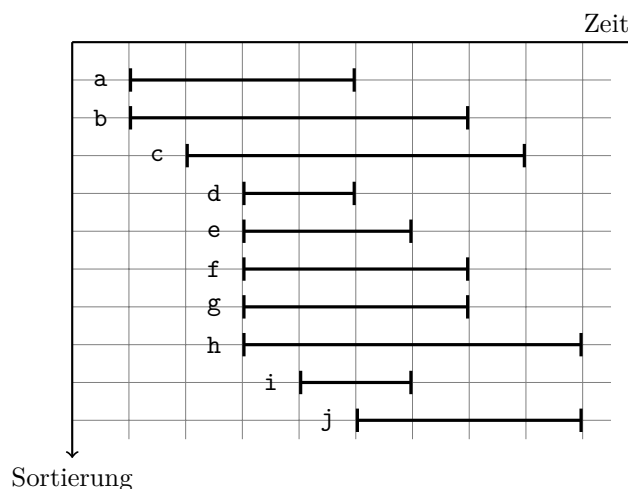


Abbildung 2: Beispielhafte Sortierung von Terminen. Namen sind nicht dargestellt.

C Kalenderverwaltung (8 Punkte)

Terminal-Klasse

Laden Sie für diese Aufgabe die **Terminal-Klasse**^a von unserer Homepage herunter und platzieren Sie diese unbedingt im Paket `edu.kit.informatik`. Die Methode `Terminal.readLine()` liest eine Benutzereingabe von der Konsole und ersetzt `System.in`. Die Methode `Terminal.println()` schreibt eine Ausgabe auf die Konsole und ersetzt `System.out`. Verwenden Sie für jegliche Konsoleneingabe oder Konsolenausgabe die **Terminal-Klasse**. Verwenden Sie in keinem Fall `System.in` oder `System.out`. Laden Sie die Terminal-Klasse niemals zusammen mit Ihrer Abgabe hoch.

^a<https://sdqweb.ipd.kit.edu/lehre/WS1617-Programmieren/Terminal.java>

In dieser Aufgabe entwickeln Sie zur Verwaltung der Termine ein Kalenderverwaltungssystem.

C.1 Interaktive Benutzerschnittstelle

Entwickeln Sie eine Klasse **CalendarManagement**, die die Logik für die *Benutzerschnittstelle* enthält. Dies ist die einzige Klasse mit einer `main`-Methode.

Entwickeln Sie eine **weitere** Klasse **Calendar**. Der Kalender verwaltet eine Liste von Terminen (**Appointments**) unter Benutzung der von Ihnen in den vorherigen Aufgaben definierten Typen. Bei Programmstart wird ein neuer Kalender erzeugt, der eine leere Terminliste hat. Das **CalendarManagement** kennt einen **Calendar** und verwaltet diesen anhand der Eingaben des Benutzers.

Nach dem Start nimmt Ihr Programm über die Konsole mittels `Terminal.readLine()` die im Folgenden genannten Befehlen entgegen. Nach Abarbeitung eines Befehls wartet das Programm auf weitere Befehle, bis das Programm irgendwann durch das Kommando `quit` beendet wird. Alle Befehle werden auf dem aktuellen Zustand des Kalenders ausgeführt.

Sie können im Folgenden davon ausgehen, dass die Kommandozeileingabe immer ein valides Kommando mit validen Parametern ist. Die Eingabe ist immer genau so wie im Kommando definiert. Insbesondere sind keine zusätzlichen Leerzeichen vor, nach oder zwischen den Wörtern eines Kommandos eingefügt. Die Kommandos selbst werden immer in Kleinbuchstaben eingegeben.

Hinweis: Sie können zur Verarbeitung der Eingabe die (statischen) Methoden der Klasse `String`^a verwenden. Beispielsweise könnten die Methoden `#split(String)`, `#join(CharSequence, CharSequence...)`, `#startsWith(String)` oder `#substring(int, int)` für Sie interessant sein.

Wenn Sie möchten können Sie auch die Methoden der Java-API für Reguläre Ausdrücke verwenden. Verwenden Sie hierzu entweder die entsprechenden Methoden der Klasse `String` (`#matches(String)`, `#replaceAll(String, String)`, ...) oder Klassen aus dem Paket `java.util.regex`^{b,c}.

Verwenden Sie keine anderen Unterpakete aus `java.util`.

^a<http://docs.oracle.com/javase/8/docs/api/java/lang/String.html>

^b<http://docs.oracle.com/javase/8/docs/api/java/util/regex/package-summary.html>

^c<http://docs.oracle.com/javase/8/docs/api/java/util/regex/Pattern.html>

Argumente für die Kommandos sind als `<argument>` angegeben. Die spitzen Klammern (`<` und `>`) sind *nicht* Teil der Eingabe durch den Benutzer. Beispielsweise ist für das Kommando `add appointment <appointment>` eine valide Eingabe durch den Benutzer:

```
add appointment 13-12-2016T17:00:00 11-01-2017T13:00:00 Blatt 4
```

C.1.1 Kommando: `add appointment <appointment>`

Erzeugt einen Termin aus dem übergebenen Argument. Das Format, in dem ein Termin **eingegeben** wird ist anders, als das in `Appointment#toString()` zurückgegebene. Die Eingabe besteht aus der textuellen Repräsentation des Anfangszeitpunktes, einem Leerzeichen, der textuellen Repräsentation des Endzeitpunktes, einem weiteren Leerzeichen und dem Namen des Termins. Sie müssen nicht prüfen, ob bereits ein Termin mit dem selben Namen im Kalender existiert.

Hinweis: Sie können zum Einlesen und Erzeugen des `Appointments` die auf der Webseite angebotene Klasse `DateUtila` und deren Methode `parseAppointment(String)` verwenden. Platzieren Sie hierzu die angebotene Klasse im Paket der von Ihnen verwendeten Lösung. Passen Sie gegebenenfalls die Packagedeklaration in den heruntergeladenen Dateien auf das von Ihnen verwendete Package an.

^a<https://sdqweb.ipd.kit.edu/lehre/WS1617-Programmieren/DateUtil.java>

C.1.2 Kommando: `print appointments`

Gibt alle Termine in der natürlichen Ordnung zurück, wie sie von `Appointment#compareTo(Appointment)` definiert ist. Die textuelle Repräsentation eines Ereignisses erhalten Sie mit der Methode `Appointment#toString`.

C.1.3 Kommando: `print appointments that start before <pointOfTime>`

Gibt alle Termine aus, deren **Startzeitpunkt** vor dem Zeitpunkt (`DateTime`) `pointOfTime` liegt. Sie können zum Umwandeln der Eingabe wie für `Appointments` beschrieben die Methode `DateUtil#parseDateTime(String)` verwenden.

C.1.4 Kommando: `print appointments on <date>`

Gibt alle Termine aus, die komplett im Tag (`Date`) `date` enthalten sind. Das bedeutet explizit, dass der Startzeitpunkt nach oder gleich dem Zeitpunkt 00:00:00 am Datum `date` und der Endzeitpunkt vor oder gleich dem Zeitpunkt 23:59:59 am Datum `date` ist. Sie können zum Umwandeln der Eingabe die Methode `DateUtil#parseDate(String)` verwenden.

C.1.5 Kommando: `print appointments in interval <startDate> <endDate>`

Gibt alle Termine aus, die komplett im Zeitintervall, das mit `startDate` beginnt und mit `endDate` endet, enthalten sind. Dies bedeutet, dass der Startzeitpunkt des Termins nach oder gleich dem Zeitpunkt (`DateTime`) `startDate` und der Endzeitpunkt des Termins vor oder gleich dem Zeitpunkt (`DateTime`) `endDate` ist. Sie können davon ausgehen, dass `startDate` echt vor `endDate` liegt. Sie können zum Umwandeln der Eingabe jeweils die Methode `DateUtil#parseDateTime(String)` verwenden.

C.1.6 Kommando: `print appointments that conflict with <appointmentTitle>`

Enthält der Kalender keinen Termin mit diesem Namen, gibt das Programm `Appointment not found.` aus und wartet auf den nächsten Befehl.

Sonst werden alle Termine im Kalender (bis auf den eingegebenen Termin selbst) ausgegeben, die mit dem Termin mit diesem Namen *in Konflikt stehen*. Zwei Ereignisse stehen in Konflikt, wenn sich die Termine überschneiden. Zwei Termine überschneiden sich, wenn der Start- oder Endzeitpunkt von einem der beiden Termine echt im anderen Termin enthalten ist.

C.1.7 Kommando: `quit`

Beendet das Programm. Verwenden Sie hierfür **nicht** `java.lang.System#exit(int)` oder Methoden der Klasse `java.lang.Runtime`.

Eine beispielhafte Interaktion mit dem Programm ist in Codeauflistung 1 dargestellt. Die grün eingefärbten mit einem Pfeil nach links (←) beginnenden Zeilen sind hierbei Eingaben durch den Benutzer, die restlichen Zeilen Ausgaben des Programms.

```

1  ← add appointment 29-11-2016T17:00:00 14-12-2016T13:00:00 Blatt 3
2  ← add appointment 13-12-2016T17:00:00 11-01-2017T13:00:00 Blatt 4
3  ← add appointment 05-11-2016T13:00:00 05-11-2016T14:00:00 Kaffee
4  ← print appointments that start before 01-12-2016T00:00:00
5  Kaffee 05-11-2016T13:00:00 05-11-2016T14:00:00
6  Blatt 3 29-11-2016T17:00:00 14-12-2016T13:00:00
7  ← print appointments on 13-12-2016
8  ← print appointments on 05-11-2016
9  Kaffee 05-11-2016T13:00:00 05-11-2016T14:00:00
10 ← print appointments in interval 15-11-2016T13:00:00 01-01-2017T00:00:00
11 Blatt 3 29-11-2016T17:00:00 14-12-2016T13:00:00
12 ← add appointment 24-12-2016T00:00:00 26-12-2016T23:59:59 Weihnachten
13 ← print appointments that conflict with Blatt 4
14 Blatt 3 29-11-2016T17:00:00 14-12-2016T13:00:00
15 Weihnachten 24-12-2016T00:00:00 26-12-2016T23:59:59
16 ← print appointments that conflict with Blatt 6
17 Appointment not found.
18 ← quit
    
```

Codeauflistung 1: Beispielinteraktion mit dem Kalenderverwaltungssystem

D Datums-Iteratoren (2 Punkte)

In dieser Aufgabe entwickeln Sie mehrere Iteratoren für wiederkehrende Datumsangaben.

In der Vorlesung haben Sie Iteratoren in Verbindung mit Listen kennengelernt. Diese iterieren schrittweise über eine Sammlung von Objekten. Diese muss nicht wie in den bisher vorgestellten Beispielen schon zu Beginn der Iteration komplett durch eine Objektstruktur, beispielsweise in Form eines Felds oder einer Liste, manifestiert sein. Stattdessen kann Sie auch während der Iteration dynamisch erstellt werden. Im Folgenden sollen Sie dieses Prinzip zunächst auf Datumsfolgen übertragen.

In Codeauflistung 2 ist beispielhafter Code für einen Iterator dargestellt, der ausgehend von einem Startwert und einem Inkrement bei jedem Aufruf von `next()` den nächstgrößeren Wert der Menge

$$\{ \text{startValue} + n \cdot \text{delta} \mid n \in \mathbb{N}_0 \}$$

zurückgibt, beispielsweise für `startValue` 5 und `delta` 3 die Zahlen 5, 8, 11, 14, 17, ...

D.1 FixedDeltaDateIterator

Implementieren Sie eine Klasse `FixedDeltaDateIterator`, die die Schnittstelle `SortedIterator<Date>` implementiert.

Hinweis: Die Klasse `Date` aus der Musterlösung implementiert bereits die Schnittstelle `Comparable<Date>`.

Die von Ihnen neu erstellte Klasse enthält neben den in der Schnittstelle beschriebenen Methoden einen Konstruktor mit der folgenden Signatur.

```

D.1.1 public FixedDeltaDateIterator(Date startDate, Date endDate,
    int deltaYear, int deltaMonth, int deltaDay)
    
```

Ein so erstellter Iterator gibt nacheinander beginnend mit `startDate` bei Aufruf von `next()` jedes Datum zurück, das durch wiederholtes ausführen der folgenden Operation erreichbar ist: Füge zum Datum `deltaDay` Tage, `deltaMonth` Monate und `deltaYear` Jahre hinzu (in dieser Reihenfolge, genau so wie bei der Implementierung von `Date#plus(Date)`)


```

1 public class IntegerIterator implements SortedIterator<Integer> {
2     private int currentValue;
3     private final int delta;
4
5     /**
6      * Creates a new {@link IntegerIterator} that will return all elements of
7      * the (infinite) set of all integers startValue + n * delta
8      * for any n greater than or equal to zero.
9      * The numbers are returned in the natural ordering,
10     * starting with startValue
11     * <p>
12     * Assumes that delta is always greater than or equal to zero.
13     * @param startValue
14     *     The value from which the iteration is started.
15     *     Also the first value returned by the iterator.
16     * @param delta
17     *     The delta between subsequent values returned by the iterator.
18     *     Must always be greater than zero.
19     */
20     public IntegerIterator(int startValue, int delta) {
21         this.currentValue = startValue;
22         this.delta = delta;
23     }
24
25     public Integer next() {
26         // Warning: overflows are deliberately ignored
27         int result = currentValue;
28         currentValue = currentValue + delta;
29         return result;
30     }
31
32     public boolean hasNext() {
33         // Warning: overflows are deliberately ignored
34         return true;
35     }
36 }
    
```

Codeauflistung 2: Beispieliterator für Zahlen

Implementieren Sie `hasNext()` und `next()` entsprechend.

Es werden nur Daten zurückgegeben, die vor oder gleich `endDate` sind. Jedes solches Datum wird genau einmal ausgegeben. Sie können davon ausgehen, dass immer mindestens eines der drei Argumente `deltaYear`, `deltaMonth` oder `deltaDay` ungleich 0 ist und alle drei Argumente größer oder gleich 0 sind. `hasNext()` gibt entsprechend den Wert `true` oder `false` zurück.

Ist `startDate` gleich `endDate` gibt der Iterator nur diesen Wert zurück. Vor dem ersten Aufruf von `next()` gibt `hasNext()` in diesem Fall `true` zurück, danach `false`. Liegt `endDate` echt vor `startDate` gibt `hasNext()` immer `false` zurück.

`endDate` kann auch den Wert `null` annehmen. Dann gibt `hasNext()` stets `true` zurück und es werden fortlaufend neue Werte zurückgegeben. Sie können hierbei einen eventuellen Überlauf bei der Integerarithmetik ignorieren.

Hinweis: Sie müssen bei der Implementierung den Fall, dass `next()` aufgerufen wird, obwohl `hasNext()` zuvor `false` zurückgegeben hat, nicht beachten.

D.2 UnionSortedIterator<T extends Comparable<T>>

Implementieren Sie eine Klasse `UnionSortedIterator<T extends Comparable<T>>`, die ebenfalls die Schnittstelle `SortedIterator<T>` implementiert.

Die Klasse hat neben den Methoden `next()` und `hasNext()` den folgenden Konstruktor:

D.2.1 `public UnionSortedIterator(SortedIterator<T> iteratorA, SortedIterator<T> iteratorB)`

Der Iterator gibt Elemente zurück, solange einer der beiden Iteratoren `iteratorA` oder `iteratorB` ein Element zurückgeben kann. Falls beide Iteratoren ein nächstes Element haben, wird immer das kleinere von beiden zurückgegeben.

Implementieren Sie die Methoden `next()` und `hasNext()` entsprechend.

In Codeauflistung 3 finden Sie Beispiele für die Benutzung der beiden implementierten Klassen. Die Terminalausgabe ist jeweils als Kommentar in den Code eingefügt.

```

1 FixedDeltaDateIterator exampleIterator1 = new FixedDeltaDateIterator(new Date(2017,1,1),
2     null, 3, 4, 10);
3 for (int i = 0; i < 5; i++) {
4     Date date = exampleIterator1.next();
5     Terminal.println(date.toString() + " " + date.getDayOfWeek());
6 }
7
8 // 01-01-2017 SUNDAY
9 // 11-05-2020 MONDAY
10 // 21-09-2023 THURSDAY
11 // 01-02-2027 MONDAY
12 // 11-06-2030 TUESDAY
13
14 FixedDeltaDateIterator dateIterator1 = new FixedDeltaDateIterator(new Date(2016,11,15),
15     new Date(2017,1,25), 0, 0, 14);
16 FixedDeltaDateIterator dateIterator2 = new FixedDeltaDateIterator(new Date(2016,10,26),
17     new Date(2016,12,28), 0, 0, 7);
18 UnionSortedIterator<Date> unionIterator = new UnionSortedIterator<Date>(dateIterator1,
19     dateIterator2);
20 while (unionIterator.hasNext()) {
21     Date date = unionIterator.next();
22     Terminal.println(date.toString() + " " + date.getDayOfWeek());
23 }
24
25 // 26-10-2016 WEDNESDAY
26 // 02-11-2016 WEDNESDAY
27 // 09-11-2016 WEDNESDAY
28 // 15-11-2016 TUESDAY
29 // 16-11-2016 WEDNESDAY
30 // 23-11-2016 WEDNESDAY
31 // 29-11-2016 TUESDAY
32 // 30-11-2016 WEDNESDAY
33 // 07-12-2016 WEDNESDAY
34 // 13-12-2016 TUESDAY
35 // 14-12-2016 WEDNESDAY
36 // 21-12-2016 WEDNESDAY
37 // 27-12-2016 TUESDAY
38 // 28-12-2016 WEDNESDAY
39 // 10-01-2017 TUESDAY
40 // 24-01-2017 TUESDAY
    
```

Codeauflistung 3: Beispiele für das Verhalten von `FixedDeltaDateIterator` und `UnionSortedIterator<Date>`

Beispielaufgaben Präsenzübung

Am 25.01.2017 findet die Präsenzübung statt. Informationen zur Hörsaaleinteilung und zum Zeitplan werden rechtzeitig auf der Vorlesungshomepage und in der Vorlesung bekanntgegeben. Im Folgenden finden Sie Aufgaben, die beispielhaft für die Aufgaben in der Präsenzübung sind. Weitere Beispielaufgaben werden auf dem 5. Übungsblatt vorgestellt.

Hinweis: Die Aufgaben werden in der Präsenzübung nur auf Deutsch gestellt. Falls Sie ein Wörterbuch benötigen, geben Sie dieses **bis spätestens 18.01.2017** im Sekretariat von Prof. Reussner (Gebäude 50.34, Raum 328. Öffnungszeiten Montag bis Freitag, 9:00 Uhr bis 13:00 Uhr) zur Überprüfung ab. **Nicht durch uns geprüfte Wörterbücher können nicht verwendet werden.** Sie erhalten das Wörterbuch in der Präsenzübung von uns zurück. Falls Sie nicht an der Präsenzübung teilnehmen, können Sie es wieder im Sekretariat abholen.

All questions in the written test („Präsenzübung“) will be in German. If you need a dictionary, hand it in for checking at the secretary's office of Professor Reussner (building 50.34, room 328. Opening hours: Monday to Friday, 9:00 to 13:00) **at the latest on 18.01.2017. If we have not checked the dictionary, you will not be allowed to use it during the test.** We will return the dictionary to you immediately before the written test. If you don't participate in the written test you can pick the dictionary up at the secretary's office.

Die Beispielaufgaben werden nicht abgegeben oder korrigiert. Sie erhalten die Lösungen eine Woche nach Ausgabe dieses Blattes.

Hinweis: Für eine korrekte Lösung müssen nicht unbedingt alle Lücken ausgefüllt werden. Die Größe der Lücken steht weiterhin nicht unbedingt in Relation zur Länge des Texts, der eingefüllt werden muss.

Operanden und Operatoren

Hinweis: Diese Aufgabe ist gleich der ersten Teilaufgabe aus Aufgabe D des ersten Übungsblattes.

Vervollständigen Sie Tabelle 1 für den Zustand nach Ausführung des folgenden Codes (vgl. Übungsblatt 1, Aufgabe D):

```
1 boolean x = true;
2 boolean y = false;
3 boolean z = true;
4 boolean w = (!(x || y) & z) | ((z && !y) ^ !x);
```

Expression	true	false	Expression	true	false
x	X		!(x y) & z		
y		X	(z && !y)		
z	X		(z && !y) ^ !x		
!(x y)			w		

Tabelle 1: Wahrheitswerte

Vererbung

Ergänzen Sie den folgenden lückenhaften Code so, dass das Programm kompiliert und ein Aufruf der `main`-Methode den Text „correct message“ ausgibt.

```
public class ClassA {
    public String getMessage() {
        return "wrong message";
    }
}

public class ClassB {

}

public static void main(String[] args) {
    ClassA messageProvider = new {
    };
    System.out.println(messageProvider.getMessage());
}
```

Kontrollstrukturen

Ergänzen Sie die folgende Methode so, dass alle Werte des Felds `values` ausgegeben werden, die echt größer als `threshold` sind. Verwenden Sie zur Ausgabe die Methode `System.out.println`.

```
public static void printGreaterThan(int[] values, int threshold) {

}
}
```

Modellierung

Ergänzen Sie die folgenden drei Klassen `Person` (`Person`), `Studierende(r)` (`Student`) und `Universität` (`University`) um die folgende Beschreibung zu erfüllen. **Sie müssen keinen Code zum Setzen der Attribute schreiben (keine Setter oder Konstruktoren). Ergänzen Sie falls nötig Attribute und Getter-Methoden. Greifen Sie falls möglich ohne Getter-Methoden auf Attribute zu.**


Ein(e) Studierende(r) ist eine Person, die zu genau einer Universität gehört. Die Email-Adresse einer studierenden Person setzt sich zusammen aus

Vorname in Kleinbuchstaben . Nachname in Kleinbuchstaben @student. Universitätsdomäne

Diese wird von der Methode `getStudentMailAddress()` der Klasse `Student` **berechnet** und ist kein Attribut von `Student`. Das Ergebnis wäre beispielsweise für eine Person mit dem Vornamen „Efemena“ und dem Nachnamen „Girard“, die an einer Universität mit der Domäne „kit.edu“ studiert: `efemena.girard@student.kit.edu`. Verwenden Sie zum Konvertieren in Kleinbuchstaben die Instanz-Methode `toLowerCase()` der Klasse `String`.

```
public class Student {
    // ...

    public String getStudentMailAddress() {
        // ...
    }
}
```

```
public class Person {  
    protected String firstName;  
    protected String lastName;  
  
      
}
```

```
public class University {  
    private String domain;  
  
      
}
```