Übungsblatt 5:

Aufgabe 1: Rekursion

(a) Ergänzen Sie die beiden Methoden `fakultaet` und `fibonacci` in der Klasse `Rekursiv`, so dass diese für einen als Parameter übergebenen Wert n

- die Fakultät $n!$ bzw.
- die n.te Fibonaccizahl berechnen.

Die Methode zur Berechnung der Fakultät soll dabei einen double Wert berechnen und die Methode für die n.te Fibonaccizahl einen `long`-Wert. Implementieren Sie die Lösungen mit einem rekursiven Ansatz.

Testen Sie Ihre Methoden mit Hilfe des vorgegebenen Codes in der `main`-Methode.

Mit dem Aufruf
`System.currentTimeMillis()` bzw. `System.nanoTime`
können Sie die aktuelle Systemzeit (gemessen in Millisekunden seit dem 1.1.1970) bzw. die Zeit in Nanosekunden (in Bezug auf einen festen Zeitpunkt) als `long`-Wert abrufen. Verwenden Sie Aufrufe einer der beiden Methoden jeweils vor und nach den Methodenaufrufen zur Berechnung der Fakultät und der n.ten Fibonaccizahl. Die Differenz der beiden Zeiten ermöglicht Ihnen, die Zeit zu bestimmen, die für die Berechnungen benötigt wurde. Geben Sie die Zeit in Sekunden bzw. Millisekunden aus. Um welchen Faktor verändert sich die Laufzeit, wenn n um den Faktor 2 verändert wird?

(b) Implementieren Sie eine zweite Version der Methode, die die n.te Fibonaccizahl berechnet. Diese soll bereits berechnete (Teil-) Lösungen speichern und wiederverwenden. Legen Sie dazu in der Klasse ein statisches Attribut an, welches ein Feld der Länge n repräsentiert, dessen Elemente zu Beginn (in der main-Methode) mit -1 initialisiert werden. Jede berechnete Fibonaccizahl soll dann dort eingetragen werden. Darüberhinaus muss vor jedem rekursiven Aufruf überprüft werden, ob der Wert für das jeweilige n bereits berechnet wurde. Wenn ja, sollte der Wert aus dem Feld direkt zurückgegeben werden und kein rekursiver Aufruf erfolgen.

Messen Sie nun die Laufzeiten der Methode für verschiedene n.

Protokollieren Sie die schließlich noch die Anzahl der Aufrufe der zweiten Methode für ein frei wählbares n mit Hilfe einer statischen Zählervariable mit.

Aufgabe 2: Rekursive Methoden: Minimum in einem Feld

Ergänzen Sie die Klasse `RekMin` durch eine statische Methode `minimum`, die den minimalen Wert in einem Feld von ganzen Zahlen bestimmt. Die Methode soll dabei nach folgendem rekursiven Schema vorgehen:

$$\min\{a_0, a_1, a_2, a_3, \cdots, a_n\} = \min\{\cdots \min\{\min\{\min\{a_0, a_1\}, a_2\}, a_3\} \cdots, a_n\}$$

Das Minimum zweier Zahlen (Trivialfall) soll dabei mit Hilfe des ternären ?-Operators bestimmt werden. Testen Sie die Methode mit einem Feld von Zufallszahlen zwischen 1 und 1000.

Aufgabe 3: Rekursive Methoden: Binärsuche

Sucht man in einem Feld der Länge n nach einem einzelnen Element e, ist der Aufwand linear: Ist das gesuchte Element in dem Feld stets vorhanden, benötigt man im Durchschnitt $\frac{n}{2}$ Zugriffe auf das Feld. Maximal sind jeweils n Zugriffe erforderlich. Sind die Elemente in dem Feld allerdings in Bezug auf ein Ordnungskriterium (aufsteigend) sortiert, kann man die maximale Anzahl von Zugriffen auf das Feld auf $\log_2(n)$ reduzieren, indem man (wiederholt) folgenden Schritt durchführt:
Aus einem zu überprüfenden Bereich (der am Anfang das vollständige Feld umfasst) wird das mittlere Element des Bereichs mit dem Element e verglichen:

- Sind die beiden Elemente gleich, ist die Suche (erfolgreich) beendet.
- Ist das Element e kleiner als das mittlere Element, überprüft man als nächstes die linke Hälfte des Bereichs auf die gleiche Weise.
- Ist das Element e größer als das mittlere Element, überprüft man als nächstes die rechte Hälfte des Bereichs auf die gleiche Weise.

Dieser Schritt wird solange wiederholt, bis man das gesuchte Element gefunden hat oder der zu überprüfende Bereich leer ist, d.h. bis die linke Grenze des Bereichs rechts von der rechten Grenze liegt.
Da bei dieser Vorgehensweise die Größe des zu überprüfenden Bereichs in jedem Schritt auf die Hälfte reduziert wird, ist die Suche nach spätestens $\lfloor \log_2(n) + 1 \rfloor$ Schritte beendet.
Diese Vorgehensweise nennt man *Binärsuche*.
Implementieren Sie nun die Klassenmethode `binSucheRek` in der Klasse `BinaerSuche`, so dass ein als Parameter übergebener Wert `wert` in dem ebenfalls als Parameter übergebenen Feld `feld` gesucht wird. Wenn das Element gefunden wird, soll die Methode den Wert `true` zurückgeben, ansonsten den Wert `false`. Die Methode soll rekursiv arbeiten. Hierzu dienen zwei Parameter `anf` und `ende`, die das von der Methode aktuell zu überprüfende Intervall darstellen. Die Methode soll dieses Intervall in jedem rekursiven Aufruf entsprechend dem oben beschriebenen Algorithmus weiter verkleinern, bis das gesuchte Element entweder gefunden wurde oder der Anfang des Intervalls hinter dem Ende liegt. Die Methode soll im Erfolgsfall den Index des gesuchten Elements in dem Feld zurückgeben und anderenfalls den Wert -1.
Implementieren Sie zum Vergleich eine iterative Version der Binärsuche in der Methode `binSucheIt`. Da hier der zu durchsuchende Bereich iterativ verkleinert wird, kann man die Bereichsgrenzen als lokale Variablen in der Methode definieren und benötigt dementsprechend keine zusätzlichen Parameter (im Funktionskopf).
Nutzen Sie die `main`-Methode der Klasse `BinaerSuche`, um Ihre Implementierungen zu testen. In der `main`-Methode wird hierzu zunächst ein aufsteigend sortiertes Feld mit Zufallszahlen generiert und ausgegeben. Anschließend kann der User ein Suchelement angeben, für das die beiden Methoden aufgerufen und das Suchergebnis dann ausgegeben wird.

Aufgabe 4: Verkettete Listen

In den Dateien `Inhalt.java` und `Liste.java` werden folgende Klassen definiert:

Inhalt Diese Klasse repräsentiert das in einem Listenelement gespeicherte Daten-Element, bestehend aus einem Namen und einer Nummer. Auf die privaten Attribute kann von außen über die `get`-Methoden lesend zugegriffen werden. Eine Instanz dieser Klasse kann zur Ausgabe (ei-

ner `String`-Repräsentation des jeweiligen Objekts) direkt der Methode `System.out.println` übergeben werden. Hierbei wird automatisch die Methode `toString` in dieser Klasse aufgerufen.

Listenelement Diese Klasse kapselt ein Daten-Element und eine Referenz auf das nächste Element in einer Liste in einem Objekt. Diese Klasse ist als innere Klasse der Klase Liste `definiert`.

Liste Diese Klasse repräsentiert eine Listenstruktur. Beim Aufruf des Default-Konstruktors der Klasse wird zunächst eine leere Liste angelegt. Dazu werden zwei Pseudolistenelemente (ohne Daten) für den Kopf und das Ende der Liste angelegt, wobei zunächst der Kopf der Liste direkt auf das Ende zeigt. Darüberhinaus sind einige weitere Methoden vorgegeben:

- Mit Hilfe der Methode `insert` kann ein neues Datenelement, in ein Listenelement gekapselt, in die Liste eingefügt werden. Hierbei steht das neue Element immer am Anfang der Liste.
- Mit der Methode `print` werden alle in der Liste gepeicherten (Daten-)Elemente ausgegeben.
- Die Methode `length` bestimmt die Anzahl der (echten) Listenelemente und gibt diese Zahl als Ergebnis zurück.
- Die Methode `lengthR` dient dazu, die Länge der Liste rekursiv zu bestimmen. Die Methode ist dazu überladen: Die öffentliche Methode ohne Parameter ruft die eigentliche Methode zur rekursiven Bestimmung der Länge mit dem ersten (echten) Element auf. Die private Methode bestimmt für einen Aufruf mit einem einzelnen Listenelement immer die Länge der (Rest-)Liste, die bei diesem Element beginnt und gibt diesen Wert jeweils zurück.

In der main-Methode wird eine Beispiel-Liste mit fünf Elementen generiert und ausgegeben.
Erweitern Sie die Klasse durch statische Methoden, die folgende Operationen auf Listen implementieren:

first Schreiben Sie eine Instanzenmethode `first`, die den Inhalt des ersten Listenelements zurückgibt.

last Schreiben Sie eine Instanzenmethode `last`, die den Inhalt des letzten Listenelements zurückgibt.

nth Schreiben Sie eine Instanzenmethode `nth`, die den Inhalt des n-ten Listenelements zurückgibt. Wenn die Liste weniger als n Elemente beinhaltet, soll die Methode eine Nullreferenz zurückgeben.

insertLast Schreiben Sie eine Instanzenmethode `insertLast`, die ein als Parameter übergebenes Datenelement in einem neuen Listenelement kapselt und am Ende der Liste einfügt.

insertAt Schreiben Sie eine Instanzenmethode `insertAt`, die ein Datenelement und einen Indexwert n als Parameter besitzt und die ein neues Listenelement mit diesem Datenelement erzeugt und es an n-ter Stelle in der Liste einfügt. Die Zählweise eines Indexwertes soll hier bei 1 beginnen (nicht bei 0). Besitzt die Liste weniger als n Elemente, soll das neue Lixstenelement am Ende der Liste eingefügt werden.

printRek Entwickeln Sie eine rekursiv arbeitende Methode `printRek` zur Ausgabe aller Elemente einer Liste. Dabei soll das letzte Element zuerst und das erste Element zuletzt ausgegeben werden (umgekehrte Reihenfolge). Orientieren Sie sich bei der Implementierung an der Methode `lengthR`. Schreiben Sie eine weitere Instanzenmethode `printRek2`, die ebenfalls rekursiv arbeitet, die Liste dabei aber von vorne nach hinten ausgibt.

toArray Schreiben Sie eine Instanzenmethode `toArray`, die ein Feld von Datenelementen erzeugt, das mit den Datenelementen aller Listenelemente gefüllt werden soll. Geben Sie das Feld am Ende als Ergebnis zurück.

reverse Schreiben Sie eine Instanzenmethode, die eine neue Liste generiert und mit dem Inhalt der aktuellen Listeninstanz füllt, wobei die Elemente in der neuen Liste in umgekehrter Reihenfolge stehen sollen.

Testen Sie die Methoden durch verschiedene Aufrufe in der `main`-Methode

Aufgabe 5: Referenzen und Rekursion

Ergänzen Sie die Datei `HuhnOderEi` um zwei nichtöffentliche Klassen `Ei` und `Huhn` mit folgenden Eigenschaften:

Ei Diese Klasse soll ein einziges privates Attribut mit dem Namen `gelegtVon` und dem Typ `Huhn` besitzen. Der Wert dieses Attributs sollvon einer Methode mit dem Namen `gelegtVonWelchemHuhn` zurückgegeben werden.

Statten Sie die Klasse mit einem Konstruktor aus, der als Parameter eine Referenz auf ein Huhn übergeben bekommt, welches dann der Variablen `gelegtVon` zugewiesen wird.

Dazu soll die Klasse eine Methode mit dem Namen `schluepfen` besitzen, die einen `String`-Parameter hat, mit der der Name des Huhns gesetzt wird, das in dieser Methode erzeugt und zurückgegeben werden soll. Die Methode soll dazu den Konstruktor der Klasse `Huhn` (siehe dort) mit diesem Parameter aufrufen.

Huhn Diese Klasse soll zwei private Attribute mit den Namen `geschluepftAus` (Typ `Ei`) und `name` (Typ `String`) besitzen. Der einzige Konstruktor, den diese Klasse besitzen soll, soll einen Parameter vom Typ `Ei` haben, mit dem das Attribut gesetzt wird, aus welchem Ei das Huhn geschlüpft ist sowie einen `String`-Parameter, mit dem der Name des Huhns gesetzt wird. Eine parameterlose Methode mit dem Namen `legEinEi` soll ein Ei erzeugen und es an den Aufrufer zurückgeben. Hierzu muß in der Methode der Konstruktor der Klasse Ei mit dem gerade vorliegenden Objekt als Parameter aufgerufen werden.

Statten Sie nun beide Klassen mit einer parameterlosen Methode `toString` aus, welche einen `String` zurückgibt. In dieser Methode soll jeweils die Information über das vorliegende Objekt in einem String wiedergegeben werden. Nutzen Sie hierzu die die Attributwerte und fügen einen erklärendem Text hinzu. (also z.B. für die Klasse Ei: " ein Ei gelegt von <Huhn>".
Schauen Sie sich nun die Testklasse `HuhnOderEi` an und versuchen Sie zunächst herauszufinden, wie die Ausgabe aussehen wird, ohne das Programm auf einem Rechner auszuführen.
Führen Sie das Programm anschließend aus und vergleichen Sie das Resultat mit Ihrer Vermutung.

Wenn Sie sich an die obigen Vorgaben gehalten haben, sollte diese Klasse mit Ihrem Code zusammen ein lauffähiges Programm ergeben.
Wie klärt sich damit die Frage, was zuerst da war: das Huhn oder das Ei?