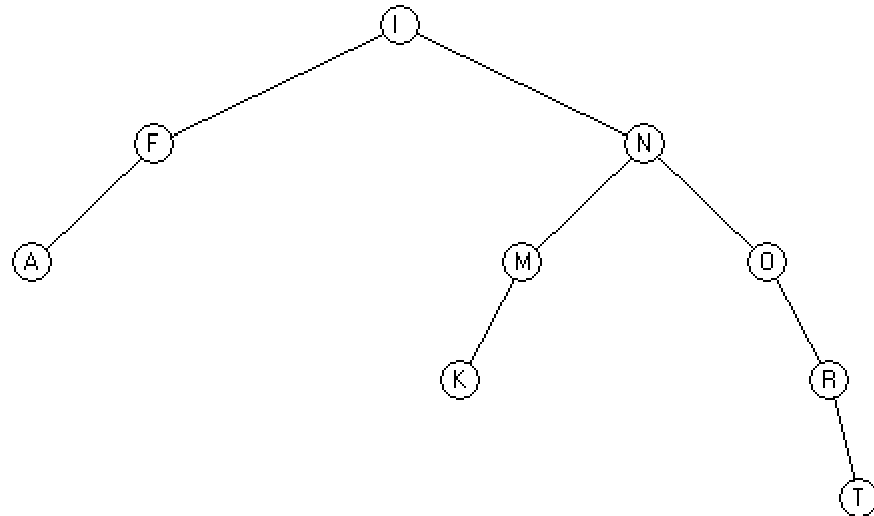


Dynamische

Strukturen

objektorientierte Programmierung
<u>in Java</u>

Version 2019.7

Autor: Dieter Lindenberg

Inhaltsverzeichnis

Datentypen und Variablen 3

Klassen und Objekte 6

Grundlagen 6

Schutz-Eigenschaften 10

Vererbung und Methoden 12

Polymorphie 17

Methoden überladen 19

Lineare Strukturen 20

Die Klasse *Queue* 20

Die Klasse *Queue*, Vorversion 22

Die Klasse *Queue*, generische Version 32

Die Klasse *Item* 39

Die Klasse *Stack,* generische Version 53

Rangieraufgabe 58

Die Klasse *Langzahl* 62

Die Klasse *VZLangzahl* 78

Keep or Throw 83

Umgekehrte polnische Notation 89

Die Klasse *List*, generische Version 93

Nicht-Lineare Strukturen 123

Die Klasse *BinaryTree,* generische Version 126

Morsecode-Aufgabe 147

Termbäume 152

Die Klasse *BinarySearchTree* 162

Datentypen und Variablen

Die Datentypen in Java zerfallen in zwei Kategorien: *primitive Typen* und *Referenztypen* (auch *Klassentypen*). Die einfachen Typen (Zahlen, Zeichen, Strings) sind die eingebauten Datentypen, die nicht als Objekte verwaltet werden.

Ein Grund für diese Teilung ist, dass häufig vorkommende, elementare Rechenoperationen schnell durchgeführt werden müssen und bei einem einfachen Typ leicht Optimierungen durchzuführen sind.

Das heißt insbesondere, dass an Objekte dieser *primitiven Typen* keine Nachrichten gesandt werden können, weil sie keine Methoden besitzen, um darauf reagieren zu können. Es existiert aber in Java für jeden primitiven Objekttyp eine sog. *Wrapper-Klasse*, die einige Methoden zum Umgang mit diesen Objekten zur Verfügung stellt.

primitive Java-Datentypen und deren Wertebereiche

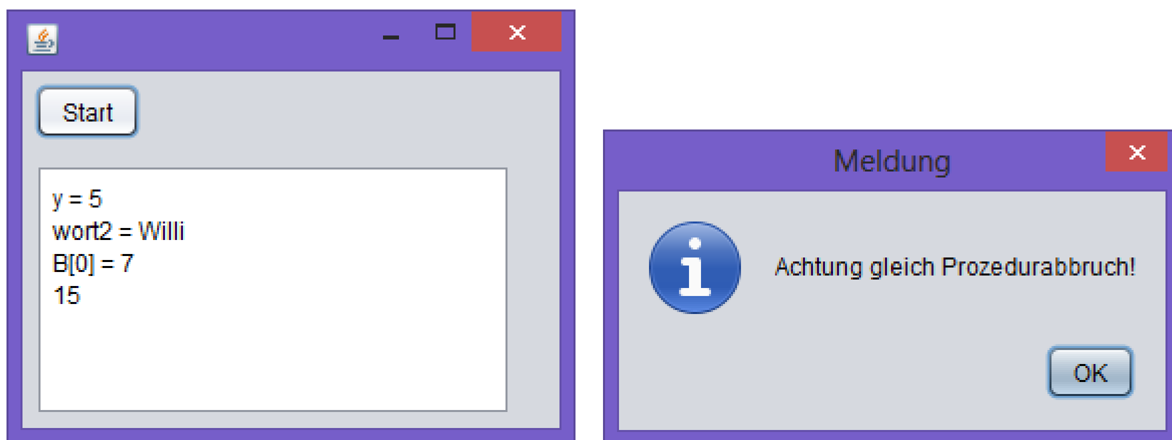
Schlüsselwort/Typ	Länge in Byte	Belegung (Wertebereich)
boolean	1	true oder false
char	2	16-Bit Unicode Zeichen
byte	1	-2^7 bis $2^7 - 1$ (–128...127)
short	2	-2^{15} bis $2^{15} - 1$ (–32768...32767)
int	4	-2^{31} bis $2^{31} - 1$ (–2147483648...2147483647)
long	8	-2^{63} bis $2^{63} - 1$ (–9223372036854775808...9223372036854775807)
float	4	$\pm 3{,}40282347 \cdot 10^{38}$
double	8	$\pm 1{,}79769131486231570 \cdot 10^{308}$

Alle *primitiven Datentypen* haben eine festgesetzte Länge, die sich unter keinen Umständen ändert.

Der Datentyp *String* nimmt in Java noch eine Sonderstellung ein.

Neben den acht *primitiven Datentypen* gibt es die *komplexen Datentypen* (*Referenz*- bzw. *Klassentypen*). Jede Klasse, die über mehrere Attribute verfügt, definiert einen *komplexen Datentypen*. Als komplex werden sie deshalb bezeichnet, weil sie sich im Gegensatz zu den *primitiven Datentypen* aus verschiedenen Elementen zusammensetzen.

Das folgende kleine Demoprogramm zeigt den Unterschied zwischen Variablen für *primitive Datentypen* und Referenzvariablen.




```
import javax.swing.*;  // wegen showMessageDialog

class Testklasse extends Object {
  public int inhalt;
}


public class Demoprogramm extends javax.swing.JFrame {

  public Demoprogramm()  { //Konstruktor
    initComponents();
  }

  Testklasse test1, test2;    //globale Variablen
  int x, y;
  String wort1, wort2;
  int[] A = new int[3];
  int[] B = new int[3];
```

```
  private void btStartMouseClicked(java……..) {
    x = 5;
    y = x;
    x = 1;
    taAusgabe.append("y = " + y + "\n"); // es wird y = 5 ausgegeben

    wort1 = "Willi";
    wort2 = wort1;
    wort1 = "Anton";
    taAusgabe.append("wort2 = " + wort2 + "\n");
    // es wird Willi ausgegeben

    A[0]=3; A[1]=3; A[2]=3;
    B = A;
    A[0]=7; A[1]=7; A[2]=7;
    taAusgabe.append("B[0] = " + B[0] + "\n"); // es wird 7 ausgegeben

    test1 = new Testklasse();
    test1.inhalt = 30;
    test2 = new Testklasse();
    test2 = test1;
    test1.inhalt = 15;
    taAusgabe.append(test2.inhalt+"\n");   // es wird 15 ausgegeben

    JOptionPane.showMessageDialog(this, "Achtung
                           gleich Prozedurabbruch!");
    test1 = null;
    taAusgabe.append("\n"+test1.inhalt);
    /* Leider wird hier die Prozedur btStartMouseClicked(…) nur abgebrochen, aber man
      kann (leider) den Startbutton neu anklicken. Die folgenden beiden Befehle werden
      nicht mehr ausgeführt. */

    taAusgabe.append("Hallo");
    JOptionPane.showMessageDialog(this, "Hallo");
  }
…………
}   // Ende der Klasse Demoprogramm
```

Bemerkung:
Es gibt in Java für den Programmierer keine (einfache) Möglichkeit, den von einer Variablen belegten Speicherplatz wieder freizugeben und damit die Existenz dieser Variablen zu löschen. Diese Aufgabe wird in Java automatisch vom sog. *Garbage-Collector* durchgeführt.

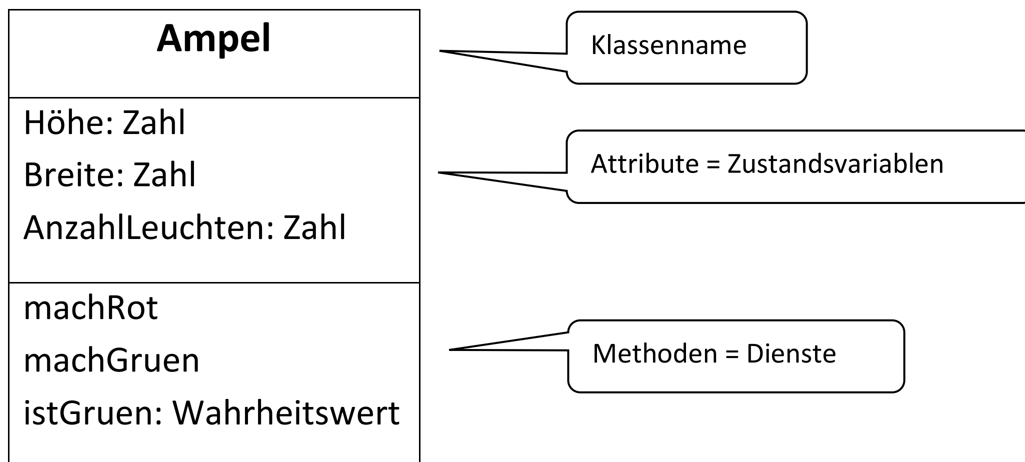
Klassen und Objekte

Grundlagen

Eine Klasse wird durch ein sog. UML-Klassendiagramm (***U****nified* ***M****odelling* ***L****anguage*) dargestellt. Dieses ist ein dreigeteiltes Rechteck und enthält entweder nur den Namen oder zusätzlich auch die Attribute und / oder die Methoden (= Dienste) der Klasse. Attribute und Methoden können zusätzliche Angaben zu Parametern und Sichtbarkeit *(public (+), private (-), protected (#))* besitzen.

Man unterscheidet zwischen *Entwurfsdiagrammen* und *Implementations-diagrammen*. Letztere enthalten programmiersprachenabhängige Angaben von Datentypen. Ein Implementationsdiagramm für Java sieht also (leicht) anders aus als ein entsprechendes Diagramm für z.B. Delphi.

UML-Entwurfsdiagramme enthalten nur allgemeine oder gar keine Typbezeichnungen (weil diese von der gewählten Programmiersprache abhängig wären).



Eine Klasse besitzt Attribute und Methoden (ein anderer Name für *Methode* ist *Dienst*). Ein Objekt einer Klasse wird auch Instanz genannt.

Attribute sind Eigenschaften, die zu jedem Objekt einer Klasse gehören. Verschiedene Objekte derselben Klasse haben üblicherweise unterschiedliche

Attributwerte. Aus diesem Grund nennt man die Attribute auch *Zustandsvariablen*.

Eine *Methode* ist eine Prozedur oder Funktion, die zu einer Klasse gehört und von allen Klassenobjekten gemeinsam genutzt werden kann.

Eine Methode wird Funktion oder Anfrage genannt, wenn sie einen Wert zurückgibt. Der Typ des Rückgabewertes muss im Methodenkopf angegeben werden. Beispiel:

```
public  int  gibFakultaetZurueck(int n)  {
   int hilf = 1;
   for (int i = 1; i <= n; i++)   hilf = hilf*i;
   return hilf;
}
```

Beispiel für einen Funktionsaufruf:

```
n = gibFakultaetZurueck(5);
taAusgabe.append(""+n);
```

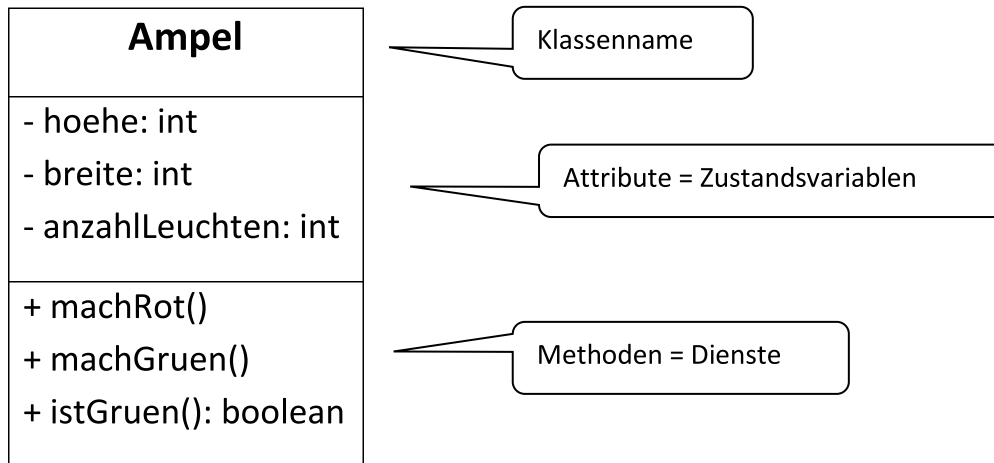
Eine Methode wird Prozedur oder Auftrag genannt, wenn sie keinen Wert, also nichts (=void) zurückgibt. Auch dies muss im Methodenkopf angegeben werden. Beispiel:

```
public  void  gibFakultaetAus(int n)  {
   int hilf = 1;
   for (int i = 1; i <= n; i++)   hilf = hilf*i;
   tfAusgabe.setText("" + hilf);
}
```

Beispiel für einen Prozeduraufruf: `gibFakultaetAus(5);`

Ein Implementationsdiagramm ergibt sich durch Präzisierung eines Entwurfs-diagramms und orientiert sich stärker an der verwendeten Programmiersprache. Ein Implementationsdiagramm für Java sieht also (leicht) anders aus als ein entsprechendes Diagramm für z.B. Delphi. Für die im Entwurfsdiagramm angegebenen Datentypen werden konkrete Datenstrukturen gewählt. Die Attribute werden mit den in der Programmiersprache (hier Java) verfügbaren Datentypen versehen und die Methoden mit Parametern incl. ihrer Datentypen.

Das entsprechende Implementationsdiagramm für Java sähe so aus:



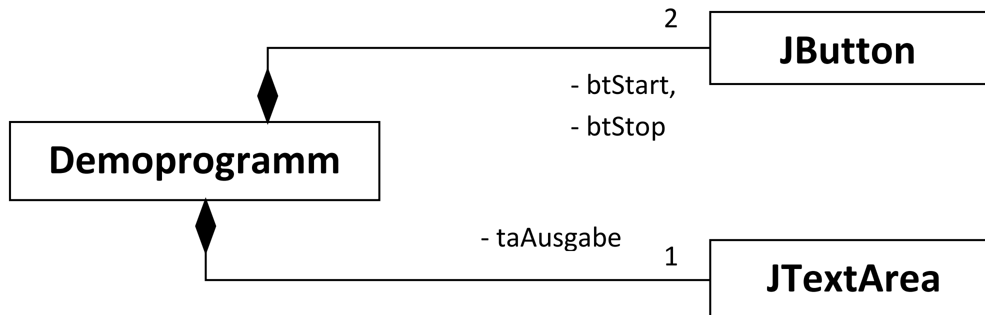
In Java gibt es sehr viele, schon vordefinierte Klassen. Alle Klassen in Java erben direkt oder indirekt von der Java Basisklasse *Object*. Wird bei einer Klassendeklaration keine *extends* Klausel angegeben so wird die Klasse automatisch von der Klasse *Object* abgeleitet.
In diesem Kurs werden wir eigene neue Klassen definieren. Alle Klassennamen sollen mit einem Großbuchstaben beginnen.

Die (frei wählbaren) Namen der von Java zur Verfügung gestellten Objekte sollen alle mit einem oder mehreren kennzeichnenden Buchstaben beginnen: *tfEingabe, taAusgabe, btStart, rbGeschlecht, labEingabe,* usw.
Alle gewählten Namen müssen sinnvoll und aussagekräftig sein! Natürlich dürfen die gewählten Klassennamen nicht mit dem Namen eines Java-Begriffs identisch sein. Beispiel: Keine Klasse darf *while* heißen.

Durch die Existenz des Namens einer Objektvariablen gibt es aber das Objekt selbst noch nicht. Dieses wird entweder durch die Java-Programmierumgebung schon bei der Gestaltung des Formblattes automatisch erzeugt, oder aber erst im laufenden Programm durch Aufruf eines entsprechenden Konstruktorbefehls:

```
Buchklasse buch1;
private void btStartMouseClicked(java……) {
  buch1 = new Buchklasse();
  buch1.setTitle("Informatik Band 3");
  …………………
}
```

Sehr oft „haben“ Objekte einer Klasse wiederum andere Objekte. Zum Beispiel besitzt üblicherweise unser Hauptformblatt, welches ein Objekt der Klasse *JForm* ist, ein oder mehrere Objekte vom Typ *JButton, JTextarea, JLabel* usw. Diese sog. ***Hat-Beziehung*** wird dann so dargestellt:



Bezeichner der Attribute, durch die die Beziehungen realisiert werden, stehen oft an den Pfeilen.

Schutz-Eigenschaften

In größeren Programmen werden Teilmodule (damit sind eine oder mehrere Klassen mit all ihren Attributen und Methoden gemeint) üblicherweise von vielen unterschiedlichen Programmierern erstellt. Ein fertig gestelltes Modul wird anschließend allen anderen Programmierern, die es gebrauchen könnten, zur Verfügung gestellt. Dabei darf es natürlich nicht erlaubt sein, dass jeder Programmierer für seine Zwecke kleine Änderungen an diesem Modul vornimmt. Ansonsten gäbe es in kürzester Zeit viele unterschiedliche Versionen dieses Moduls und niemand wüsste mehr, was dieses Modul denn nun wirklich macht.

Der Ersteller des Moduls (bzw. einer oder mehrerer Klassen) bestimmt nun, welche Methoden des Moduls benutzt werden dürfen. Verändert werden darf und kann überhaupt nichts in diesem Modul, weil nur dessen Ersteller üblicherweise den Quelltext dieses Moduls kennt.

Diese benutzbaren Methoden werden unter der Schutzklasse ***public*** eingetragen. Ein Benutzer dieses Moduls (üblicherweise ein anderer Programmierer) kann nur diese Methoden aufrufen. An deren Programmierung kann er auch nichts ändern, weil er natürlich nur (im Gegensatz zu unserem Unterricht!) den bereits compilierten Code und selbstverständlich nicht den Programm-Quelltext erhält.

Davon abgesehen benötigt das Modul auch Methoden, die von einem anderen Programmierer nicht benutzt werden dürfen, bzw. von deren Existenz ein anderer Programmierer überhaupt nichts wissen muss (weil es ihn normalerweise auch nicht interessiert, wie das Modul intern funktioniert). Derartige Attribute und Methoden werden unter der Schutzklasse ***private*** eingetragen.

Im Prinzip **sollen alle Attribute** der Schutzklasse *private* (oder der Schutzklasse *protected*) angehören.

Beispiel: Angenommen, man hätte eine Klasse *Rechteck* mit den Attributen *Länge, Breite* und *Flächeninhalt*. Wenn jemand willkürlich nur das Attribut *Länge* ändern würde, so hätte dies Auswirkungen auf das Attribut *Flächeninhalt*. Eine mögliche Änderung des Attributes *Länge* darf deshalb nur über eine öffentlich zur Verfügung gestellte Methode *setzeLaenge(int n)* vorgenommen

werden. Diese vom Ersteller der Klasse *Rechteck* implementierte Methode berücksichtigt gleichzeitige Änderungen des Attributes *Flächeninhalt*.

Oft möchte man von gegebenen Klassen weitere Unterklassen erstellen, welche größtenteils der Originalklasse ähneln, aber noch zusätzliche oder auch weniger Attribute und Methoden haben als die Oberklasse. Für diesen Fall gibt es noch die Schutzklasse ***protected***. Alle hier deklarierten Attribute und Methoden sind auch (nur) in allen abgeleiteten Unterklassen sichtbar.

Unter *private* deklarierte Attribute und Methoden werden nicht an Unterklassen vererbt.

Im Klassendiagramm wird die Schutzklasse hinzugefügt. Dabei gilt:
„+“ = *public*, “–“ = *private*, „#“=*protected*

Zusätzlich unterscheidet man bei den Diensten zwischen Aufträgen (Prozeduren) und Anfragen (Funktionen). In den Implementationsdiagrammen für Java erkennt man den Unterschied daran, dass bei Funktionen immer ein Ergebnistyp mit angegeben wird.

Wenn man Klassen benutzen will, die das Java-System zur Verfügung stellt, so muß man dafür sorgen, dass diejenigen Module (*packets*), in denen die zugehörigen Klassen definiert sind, mithilfe der *import*-Anweisung eingebunden werden.

Vererbung und Methoden

Private Methoden werden nicht vererbt. Sämtliche Methoden vom Typ *public* und *protected* werden von Unterklassen völlig identisch übernommen. Allerdings können Methoden in Unterklassen überschrieben werden.

```
class  Figur extends Object {
  public void schreibeName() {
    System.out.println("Figur");
  }
}



class  Rechteck extends Figur {
  /* Mit der folgenden, sog. Annotation kann bzw. muss eine Methode gekennzeichnet
     werden, die die Methode ihrer Oberklasse überschreibt. Der Compiler überprüft dann
     ob die Oberklasse diese Methode (mit übereinstimmender Parameterliste) enthält und
     gibt einen Fehler aus, wenn dies nicht der Fall ist. */
  @Override
  public void schreibeName()  {
    System.out.println("Rechteck");
    super.schreibeName();   /*man kann also – muss aber nicht – auch die
                              gleichnamige Methode der Oberklasse aufrufen */
  }
}
```

Falls keine besonderen Initialisierungen vorgenommen werden müssen, wird der jeweilige Konstruktor vom Java-System selbständig realisiert.

Hinweis: Man kann einer Variablen vom Typ einer bestimmten Klasse beliebige Objekte von abgeleiteten Klassen zuordnen. Das Umgekehrte geht nicht!

Das nachfolgende Programm ergibt folgende Konsolenausgabe:
Figur

Rechteck
Figur

Rechteck
Figur

```
Figur OKVariable;
Rechteck UKVariable;

private void btStartMouseClicked(java…….) {
  OKVariable = new Figur();
  OKVariable.schreibeName();   // ruft Oberklassenmethode auf
  System.out.println();

  OKVariable = new Rechteck();   // damit ist OKVariable jetzt ein Objekt der
                                    Klasse Rechteck
  OKVariable.schreibeName();   // ruft Unterklassenmethode auf
  System.out.println();

  UKVariable = new Rechteck();
  UKVariable.schreibeName();   // ruft Unterklassenmethode auf
}
```

Bemerkung:
Der *super*-Aufruf hat immer folgende Form:
super.Methodenname(Parameterliste);
Der Methodenname muss also immer explizit genannt werden.

Im Gegensatz zu Aufrufen in Konstruktoren kann der *super*-Aufruf in einer Methode an jeder beliebigen Stelle der Methode erfolgen.

In Konstruktoren muss ein eventueller *super*-Aufruf immer an erster Stelle stehen. Die Syntax lautet dann: **`super();`**

Falls als erste Anweisung im Konstruktor kein Aufruf *super()* steht, setzt der Compiler an dieser Stelle einen impliziten Aufruf *super();* ein und ruft damit den parameterlosen Konstruktor der Oberklasse auf. Falls ein solcher

Konstruktor in der Oberklasse nicht existiert, weil dort nur ein Konstruktor mit Parametern erstellt wurde, gibt es einen Compiler-Fehler. Das ist genau dann der Fall, wenn in der Oberklassendeklaration lediglich parametrisierte Konstruktoren angegeben wurden und daher ein parameterloser *default*-Konstruktor nicht automatisch erzeugt wurde.

Das Anlegen von Konstruktoren in einer Klasse ist optional. Falls in einer Klasse überhaupt kein Konstruktor definiert wurde, erzeugt der Compiler beim Übersetzen der Klasse automatisch einen parameterlosen *default*-Konstruktor. Dieser enthält lediglich einen Aufruf des parameterlosen Superklassen-konstruktors.

Konstruktoren werden nicht vererbt (obwohl sie public sind). Alle Konstruktoren, die in einer abgeleiteten Klasse benötigt werden, müssen entweder vom Programmierer neu geschrieben werden, selbst wenn sie nur aus einem Aufruf des Oberklassenkonstruktors bestehen, oder sie werden automatisch vom Compiler neu mit erzeugt, wobei dieser neu erzeugte Konstruktor dann auch nur den Aufruf des Oberklassenkonstruktors beinhaltet.

Durch diese Regel wird bei jedem Neuanlegen eines Objekts eine ganze Kette von Konstruktoren aufgerufen. Da nach den obigen Regeln jeder Konstruktor zuerst den Oberklassenkonstruktor aufruft, wird die Initialisierung von oben nach unten in der Vererbungshierarchie durchgeführt: zuerst wird der Konstruktor der Klasse *Object* ausgeführt, dann der der ersten Unterklasse usw., bis zuletzt der Konstruktor der zu instanzierenden Klasse ausgeführt wird.

Alle Attribute und Methoden einer Oberklasse, welche die Schutzklasse *public* oder *protected* haben, können in der Unterklasse genutzt werden als ob es eigene/lokale sind. Man kann also einem Objekt der Unterklasse natürlich Werte für die (*public* oder *protected*) Attribute der Oberklasse zuweisen.

Die Vererbung funktioniert jedoch nicht rückwärts. Man kann einem Objekt der Oberklasse keinen Wert für ein Attribut der Unterklasse zuweisen, welches nur in der Unterklasse existiert.
Begründung: Dieses Attribut existiert in der Oberklasse nicht und kann damit auch keinem Objekt der Oberklasse zugewiesen werden. Die Oberklasse „weiß“ überhaupt nicht, dass es von ihr Unterklassen gibt, aber jede Unterklasse „weiß“, von welcher Oberklasse sie abgeleitet wurde.

Es gibt einige wenige sog. *Klassenmethoden*, die aufgerufen werden können, ohne dass ein Objekt der Klasse existiert. Zum Beispiel enthält die Klasse *Math* ausschließlich derartige *Klassenmethoden*.

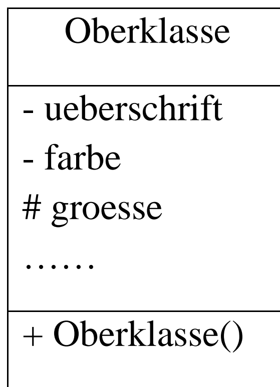
Ebenfalls eine derartige *Klassenmethode* ist natürlich die Konstruktormethode. Mit ihr wird ein Objekt erzeugt. Logischerweise muss man sie also schon vor der Existenz eines Objektes aufrufen können.

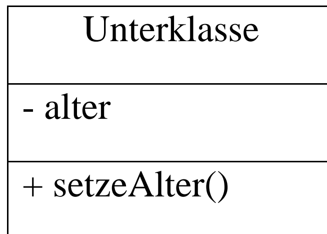
Alle Klassen sind Unterklassen der obersten Javaklasse namens *Object*. Obwohl die Deklaration der Konstruktormethode keinen Rückgabewert enthält (es wird noch nicht einmal die Angabe *void* benutzt), gibt ein Konstruktor immer einen Verweis (einen sog. Zeiger) auf das Objekt, das er erstellt, zurück.

Der Konstruktor reserviert zunächst Speicherplatz für das neue Objekt. Anschließend werden alle existierenden Attribute initialisiert. Zahlen werden in allen Methoden automatisch mit 0, Strings mit einem Leerstring und Referenzvariable mit *null* initialisiert. Aus diesem Grund braucht der Programmierer in dieser Konstruktor-Methode nur noch denjenigen Attributen einen Wert zuweisen, die einen bestimmten Anfangswert haben sollen.

Wenn man in der Unterklasse eine Methode (neu) definiert, die denselben Namen wie eine Methode der Oberklasse besitzt, so „verdeckt" die neue Methode in der Unterklasse die alte Methode aus der Oberklasse (ähnlich verhält es sich z.B. mit lokalen und globalen Variablen gleichen Namens in Methoden).
Auf die Methode der Oberklasse lässt sich allerdings immer noch mit dem Stichwort ***super*** zugreifen (zum Beispiel mit *super.schreibeName();*).

Im Folgenden betrachten wir nun die beiden Klassen *Oberklasse* und *Unterklasse*. Die Unterklasse leitet sich von der Oberklasse ab. Sie beinhaltet noch eine (*private*) Eigenschaft *alter*, die sich also nicht noch weiter vererben lässt.





Private Eigenschaften und Methoden der Oberklasse werden **nicht** auf die Unterklasse vererbt. Jedes Objekt der Unterklasse besitzt also nur diejenigen Eigenschaften und Methoden der Oberklasse, die entweder *public* oder *protected* sind.
Zusätzlich gibt es aber noch weitere, nur in der Unterklasse definierte Eigenschaften und Methoden.

Falls die Objekte der Unterklasse nur **zusätzliche** Attribute (mit den Standardinitialisierungen) haben sollen (und nicht schon bei der Erzeugung andere Attributswerte als ein Objekt der Oberklasse), so genügt derselbe Konstruktor. Das Javasystem hängt die zusätzlichen Attribute und Methoden automatisch bei der Erzeugung mit an.

Polymorphie

Das nachfolgende Programm liefert folgende Konsolenausgabe:
Dieses Fahrzeug besitzt 5 Plätze
Dieses Fahrzeug besitzt 3 Plätze
Dieses Fahrzeug besitzt 3 Plätze

```
class Auto extends Object  {
  public void informationsBereitstellung()  {
   System.out.println("Dieses Fahrzeug besitzt " +sitzplatzAnzahl()+"Plätze");
  }

  public int sitzplatzAnzahl()  {
    return 5;
  }
}


class LKW extends Auto  {
  @Override
  public int sitzplatzAnzahl()  {
    return 3;
  }
}


public class Demoprogramm extends javax.swing.JFrame {
......
  Auto auto1;
  LKW lkw1;

  private void btStartMouseClicked(java......) {
    auto1 = new Auto();
    lkw1 = new LKW();

    auto1.informationsBereitstellung();
    lkw1.informationsBereitstellung();

    auto1 = new LKW();
    auto1.informationsBereitstellung();
  }
} //Ende des Demoprogramms
```

Beachte: Die Methode *informationsBereitstellung()* ist in der Unterklasse *LKW* identisch mit der gleichnamigen Methode *informationsBereitstellung()* in der Oberklasse *Auto*.
Wenn allerdings innerhalb dieser Methode *informationsBereitstellung()* die Hilfsmethode *sitzplatzAnzahl()* aufgerufen wird, dann wird erst geprüft, welcher Objekttyp gerade vorliegt. In Abhängigkeit vom Ergebnis dieser Prüfung wird entschieden, ob die Hilfsmethode *sitzplatzAnzahl()* aus der Ober- oder aus der Unterklasse verwendet wird.

Der Java-Quelltext wird also so compiliert, dass erst zur Laufzeit des Programms geprüft wird, welcher Objekttyp vorliegt und in Abhängigkeit von dieser Prüfung wird entschieden, welche der beiden gleichnamigen Methoden ausgeführt wird. Diese Programm- oder Compilereigenschaft nennt man ***dynamische*** oder ***späte Bindung***.

Das Ergebnis dieser Prüfung ist offensichtlich nicht abhängig von der Deklaration der Typvariablen (bekanntlich kann eine Variable der Oberklasse auch ein Objekt einer Unterklasse beinhalten).

Aus der Sicht des Programmbenutzers zeigen die verschiedenen Objekte ein ***polymorphes*** (=vielgestaltiges) Verhalten: Objekte verschiedener Klassen reagieren unterschiedlich auf identische Befehle *InformationsBereitstellung*).
Man spricht auch von ***Polymorphie.***

Hinweis: Man kann einer Variablen vom Typ einer bestimmten Klasse beliebige Objekte von abgeleiteten Klassen zuordnen. Das Umgekehrte geht nicht!

Methoden überladen

Man kann auch Methoden mit identischem Namen mehrfach deklarieren mit jeweils unterschiedlichen Parametern. Öfter benutzt wird dies bei der Konstruktor-Methode, aber es funktioniert generell bei allen Methoden:

```
public int summe(int a, int b)  {
  return a + b;
}

public int summe(int a, int b, int c)  {
  return a + b + c;
}
```

Der Compiler benutzt den Typ und die Anzahl der Parameter, um zu entscheiden, welche überladene Methode aufgerufen werden soll.

Lineare Strukturen

Die Klasse *Queue*

Objekte der Klasse *Queue* (Schlange) verwalten beliebige Objekte nach dem ***FIFO***-Prinzip (First-In-First-Out), d.h. das zuerst abgelegte Element wird als erstes wieder entnommen.

Der Typ *Queue* ist im Prinzip eine Liste von Objekten. Es können zum Beispiel mehrere Listen existieren, die teilweise dieselben Objekte enthalten. Beispiel: Eine erste Liste der noch nicht volljährigen, und eine zweite Liste aller männlichen Schüler in der JgSt. 12. Wenn man hier ein Objekt aus der ersten Liste entfernt, darf das natürlich keinen Einfluss auf die zweite Liste haben. Aus diesem Grund sollte beim Löschen eines Listenelementes zwar aus der Liste, aber nicht aus dem Speicher gelöscht werden.

Die Syntax der Methoden der Klasse *Queue* wurde NRW-weit für die Abiturjahrgänge ab 2017 gemeinsam folgendermaßen festgelegt.

Dokumentation der Klasse Queue

Konstruktor **Queue()**
Eine leere Schlange wird erzeugt.

Anfrage **boolean isEmpty()**
Die Anfrage liefert den Wert *true*, wenn die Schlange keine Objekte enthält, sonst liefert sie den Wert *false*.

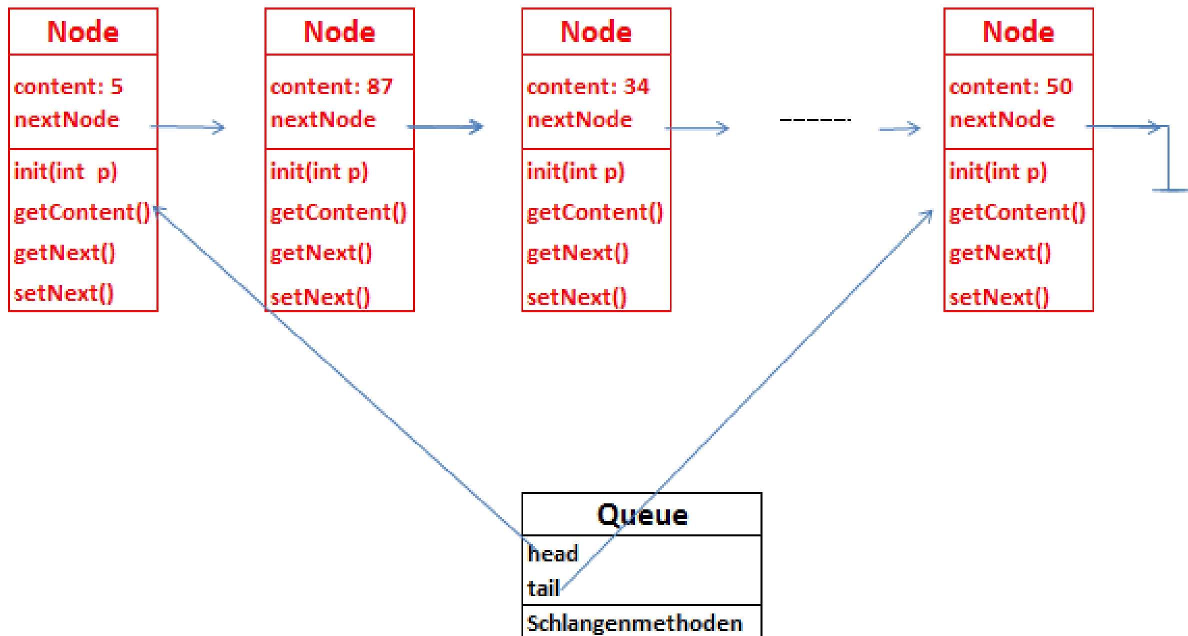
Auftrag **void enqueue(ContentType pContent)**
Das Objekt pContent wird an die Schlange angehängt. Falls pContent gleich *null* ist, bleibt die Schlange unverändert.

Auftrag **void dequeue()**
Falls die Schlange leer ist, wird sie nicht verändert. Ansonsten wird das erste Objekt aus der Schlange entfernt.

Anfrage **ContentType front()**
Die Anfrage liefert das erste Objekt (vom Typ ContentType) der Schlange. Die Schlange bleibt unverändert. Falls die Schlange leer ist, wird *null* zurückgegeben.

Die Klasse *Queue,* Vorversion

Obwohl eine Schlange völlig beliebige Elemente enthalten kann (z.B. Bilder, Musikstücke, Texte), arbeiten wir zunächst mit einfachen Elementen, deren eigentlicher Inhalt nur eine Integer-Zahl ist. Allerdings kennt jedes Element seinen Nachfolger in der Schlange. Und weil es nur einen Nachfolger kennt, kann dieses Element auch nur in einer einzigen Schlange enthalten sein.



Nur das in der obigen Skizze schwarz Gezeichnete ist ein Objekt der Klasse *Schlange* bzw. *Queue*. Jede Schlange belegt also nur sehr wenig Speicherplatz bzw. alle Schlangen belegen einen Speicherplatz gleicher Größe.
Man benötigt jeweils nur Speicherplatz für die Methoden der Schlange sowie für die zwei Variablennamen *head* und *tail*.
Allerdings benötigt man für jedes einzelne Element, welches zur Schlange gehört, entsprechend viel Speicherplatz.
Die Elemente der Schlange müssen untereinander selbst verkettet sein.
Die Schlange selbst hat nur Zugriff auf das erste und letzte Element, welches zur Schlange gehört.

Erzeuge die beiden Klassen *Node* und *Queue,* deren Quelltext gleich dargestellt werden wird, folgendermaßen:
NetBeans/ File/ New File/ Kategorie Java/ Type Java Class

Bemerkungen:
Wir werden zunächst nur eine vereinfachte Schlangenversion besprechen. Bei dieser nun folgenden ersten Version wird die Schlange aus Vereinfachungsgründen noch keine beliebigen, allgemeinen Objekte, sondern spezielle Objekte (welche Integer-Zahlen enthalten) verwalten.
Außerdem können diese speziellen Objekte noch nicht in mehreren Schlangen enthalten sein. Beide Nachteile werden in der später folgenden, zweiten Version behoben.

Beim Erzeugen eines Klassenobjektes belegt die Java-Programmierumgebung automatisch die jeweils vorhandenen Attribute mit folgenden Startwerten:

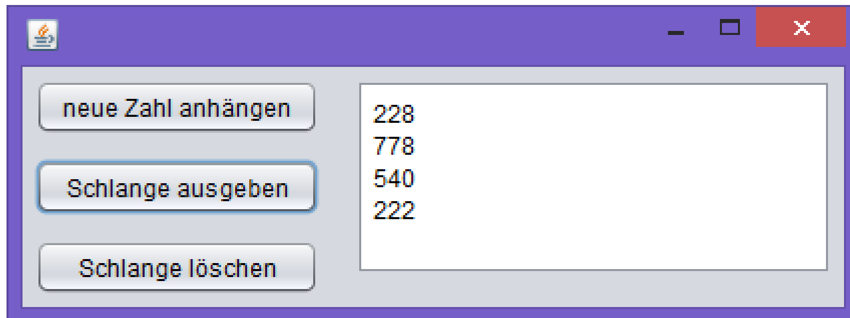
- Alle Datenfelder mit einem ganzzahligen Datentyp (z.B. Integer) werden mit 0 initialisiert.
- Alle Datenfelder mit einem String-Typ werden durch eine leere Zeichenkette initialisiert.
- Alle Datenfelder mit einem Zeigertyp werden mit dem Wert *null* initialisiert.
- Alle anderen Datenfelder bleiben undefiniert!

```
public class Node {      // Vorversion
  private int content;
  private Node nextNode;

  public Node(int zahl)  {
    content = zahl;
    nextNode = null;   //eigentlich überflüssig
  }

  public int getContent()  {
    return content;
  }

  public void setNext(Node pNext) {
    nextNode = pNext;
  }

  public Node getNext()  {
    return nextNode;
  }
}
```

Beachte, dass in der nun folgenden, noch vorläufigen Vorversion der Klasse *Queue* die Methode *enqueue* als Parameter und die Methode *front* als Ergebnis noch keine <u>allgemeinen</u> Objekte enthalten!

```
public class Queue {
  private Node head, tail;

  public Queue()   {
    head = null; //eigentlich überflüssig
    tail = null;
  }

  public boolean isEmpty(){
    return head == null;
  }

  public void enqueue(int zahl)   {
    Node lNewNode = new Node(zahl);
    if (this.isEmpty()){
       head = lNewNode;
       tail = lNewNode;
    }
    else {
       tail.setNext(lNewNode);
       tail = lNewNode;
    }
  } //Ende von enqueue

  public void dequeue()   {
    if (!this.isEmpty()) {
       head = head.getNext();
       if (this.isEmpty()) tail = null;
    }
  }

  public int front(){
    if (this.isEmpty()) return Integer.MIN_VALUE;
    // das ist in dieser Version leider nicht schön! Es entspricht auch nicht der Klassen
    // dokumentation von Queue
    else return head.getContent();
  } //Es wird also nicht der Knoten, sondern dessen Inhalt geliefert
}
```

Das folgende Hauptprogramm dient nur dazu, mit der Klasse *Queue* etwas zu arbeiten.
Der oberste Button hängt eine neue Zufallszahl an die Schlange an, gibt diese Zahl aber noch nicht in der Textarea aus. Der mittlere Button gibt die Schlange aus (beachte, dass dabei die Schlange nicht gelöscht wird!). Der unterste Button löscht den gesamten Inhalt der Schlange.




```
private Queue schlange = new Queue();

private void btNeueZahlAnhaengenMouseClicked(java…) {
  int zahl = (int) (1000*Math.random());
  schlange.enqueue(zahl);
}


private void btAusgabeMouseClicked(java……) {
  taAusgabe.setText("");
  int zahl;
  Queue hilfsschlange = new Queue();
  while (! schlange.isEmpty()) {
     zahl = schlange.front();
     hilfsschlange.enqueue(zahl);
     schlange.dequeue();
     taAusgabe.append(zahl + "\n");
  }
  schlange = hilfsschlange;


private void btLoeschenMouseClicked(java……) {
   while (! schlange.isEmpty()) schlange.dequeue();
   taAusgabe.setText("");
}
```

Aufgaben zur Struktur der Schlange

1. Erweitere die Klasse *Queue* durch ein weiteres Attribut namens *anzahl* und einer entsprechenden Methode *getAnzahl*, welche die Anzahl der Elemente der Schlange angibt. Natürlich müssen auch die Einfüge- und Lösch-methoden angepasst werden.

2. Erweitere die Klasse *Queue* durch eine Methode namens *append(Queue s)*, welche an die bereits vorhandene Schlange die Schlange ***s*** anhängt.

Aufgaben zur Anwendung der Schlange

3. Simuliere folgendes Spiel: Ein Kartenspiel mit 32 Karten soll gemischt werden. Erzeuge dafür eine Originalschlange, welche als Elemente alle natürlichen Zahlen von 1 bis 32 enthält. Die Inhalte dieser Originalschlange werden nun nach dem Zufallsprinzip auf vier Hilfsschlangen verteilt. Die Originalschlange ist danach leer. Dann werden die vier Hilfsschlangen aneinander gehängt zu einer neuen Originalschlange. Das Ganze wird mehrmals gemacht. Gib anschließend die Originalschlange aus!

4. Zwei bereits sortierte Zahlenschlangen A und B sollen zu einer einzigen sortierten Schlange C verschmolzen werden. Wähle zum Testen deines Programms für A die ersten 10 Quadratzahlen und für B die ersten 5 Kubikzahlen und umgekehrt!

5. Eine Schlange enthält zufällige, ungeordnete, natürliche Zahlen. Mit Hilfe zweier Hilfsschlangen sollen die Zahlen so geordnet werden, dass anschließend wieder alle Zahlen in der Originalschlange stehen, aber alle geraden Zahlen (durchaus ungeordnet) am Anfang und alle ungeraden Zahlen am Schluß der Schlange.

6. Eine Schlange enthält 100 zufällige, ungeordnete, natürliche Zahlen, die alle kleiner als 10 000 sind. Mit Hilfe von zehn Hilfsschlangen Hilf0, Hilf1, Hilf2, ... , Hilf9 sollen die Zahlen folgendermaßen geordnet werden: Zuerst

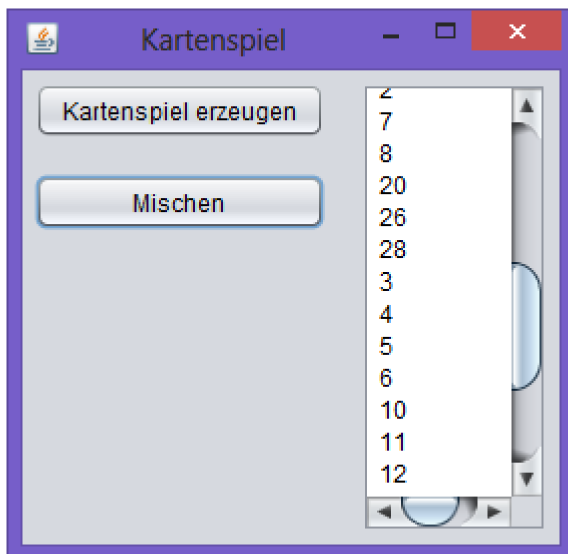
ordnet man alle Zahlen aufgrund ihrer Einerziffer in die zehn Hilfsschlangen ein. Danach werden alle Hilfsschlangen nacheinander wieder an die nun leere Originalschlange angehängt. Anschließend ordnet man aufgrund der Zehnerziffern usw.

Lösungen

Aufgabe 2

```
public void append(Queue s)  {
  if (s != null)  {
    while (! s.isEmpty())  {
       this.enqueue(s.front());
       s.dequeue();
    }
  }
}
```

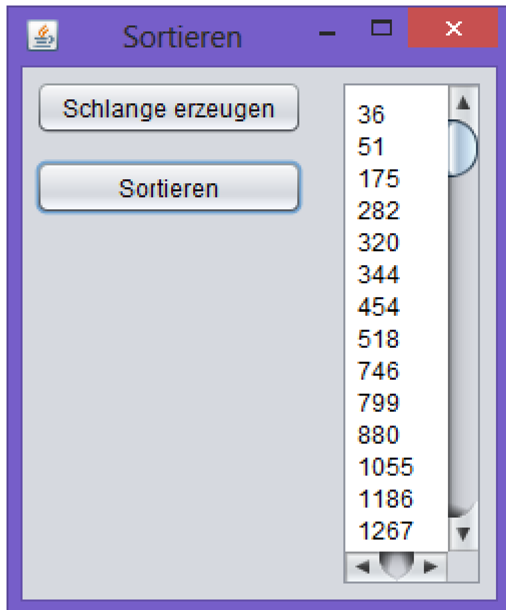
Aufgabe 3




```
Queue spiel, hilfA, hilfB, hilfC, hilfD;

private void btErzeugenMouseClicked(java......) {
  spiel = new Queue();
  for (int i = 1; i <= 32; i++) spiel.enqueue(i);
  gibAus();
}
```

```
private void btMischenMouseClicked(java……) {
  hilfA = new Queue();
  hilfB = new Queue();
  hilfC = new Queue();
  hilfD = new Queue();
  int zahl, zufallszahl;
  while (! spiel.isEmpty())  {
      zahl = spiel.front();
      spiel.dequeue();
      zufallszahl = (int) (1 + 4*Math.random());
      switch (zufallszahl)  {
         case 1: hilfA.enqueue(zahl); break;
         case 2: hilfB.enqueue(zahl); break;
         case 3: hilfC.enqueue(zahl); break;
         case 4: hilfD.enqueue(zahl);
      }
  }
  spiel.append(hilfA);
  spiel.append(hilfB);
  spiel.append(hilfC);
  spiel.append(hilfD);
  gibAus();
}


private void gibAus()  {
  taAusgabe.setText("");
  Queue hilf = new Queue();
  int blatt;
  while (! spiel.isEmpty())  {
     blatt = spiel.front();
     taAusgabe.append(blatt+"\n");
     spiel.dequeue();
     hilf.enqueue(blatt);
  }
  spiel = hilf;
  hilf = null;
}
```

Aufgabe 6




```
Queue schlange, hilf0, hilf1, hilf2, hilf3, hilf4,
      hilf5, hilf6, hilf7, hilf8, hilf9;


private void gibAus()  {
   …….
}


private void btErzeugenMouseClicked(java……) {
  schlange = new Queue();
  for (int i = 1; i <= 100; i++)
      schlange.enqueue((int) (1+9999*Math.random()));
  gibAus();
}


private int gibStelle(int n, int stelle)  {
  int divisor = 1;
  for (int i= 1; i< stelle; i++) divisor = divisor*10;
  return (n/divisor) % 10;
}
```

```
private void btSortierenMouseClicked(java……) {
  hilf0 = new Queue();
  ……………
  hilf9 = new Queue();
  int zahl, ziffer;
  for (int stelle=1; stelle<=4; stelle++)  {
     while (! schlange.isEmpty())  {
        zahl = schlange.front();
        schlange.dequeue();
        ziffer = gibStelle(zahl, stelle);
        switch (ziffer)  {
           case 0: hilf0.enqueue(zahl); break;
           ………
           case 9: hilf9.enqueue(zahl);
        }
     }
     schlange.append(hilf0);
     …….
     schlange.append(hilf9);
  }

  gibAus();
}
```

Die Klasse *Queue*, generische Version

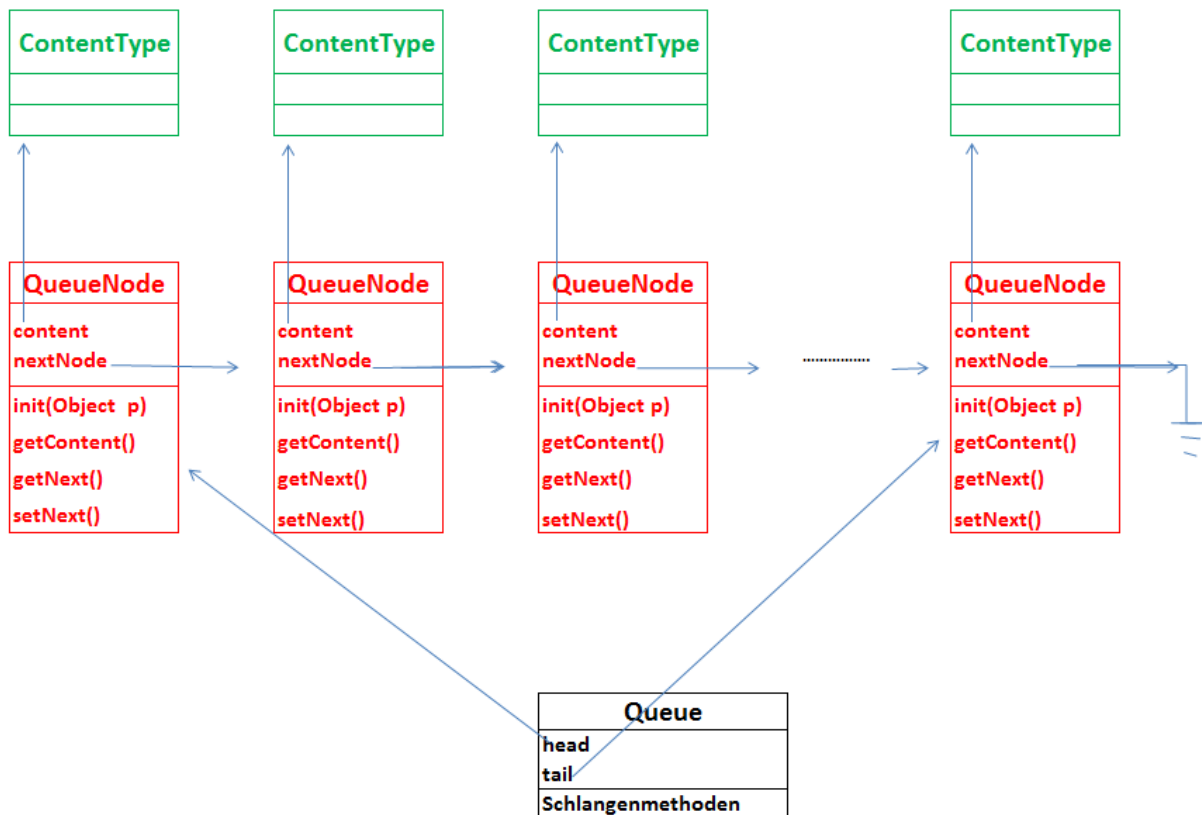
Nachteilig an der vorherigen Version war, dass die Schlange nur ganz spezielle Elemente enthalten konnte. Außerdem war die Information darüber, welches das nächste Element war, in jedem Element selbst gespeichert. Das hatte den weiteren Nachteil, dass jedes Element nur in einer einzigen Schlange enthalten sein konnte. Im Folgenden soll die Klasse *Queue* so verfeinert werden, dass man Objekte einer beliebigen Klasse, die wir im Folgenden immer *ContentType* nennen, in beliebig vielen Schlangen speichern kann.

Aus diesem Grund muss die Information über das jeweils nächste Objekt *„in der Schlange selbst“* und nicht mehr in den Objekten gespeichert werden. Man denke etwa an eine Dia-Show oder eine Abfolge von Musikstücken. Dabei ist die Information über das nächste Bild bzw. Musikstück auch nicht in den Bildern oder Musikstücken gespeichert.

Das ganze Problem bzw. die Verbesserung der alten Version läßt sich im Prinzip durch eine einzige Änderung lösen: In der Skizze auf Seite 22 dieses Skriptes wird in den dort gezeichneten Elementen die Information *content* ersetzt durch einen Zeiger auf den eigentlichen Inhalt. Damit ergibt sich die nachfolgend gezeichnete Struktur.

Jeder Knoten enthält nun zwei Verweise: einen auf das eigentliche Objekt, und einen auf den nächsten Knoten.

Mit der nun folgenden Version lassen sich Objekte beliebiger Klassen in der Schlange verwalten.



Der Unterschied zur Vorversion besteht darin, dass der *content* eines Knotens jetzt ein Zeiger auf ein Objekt der beliebigen Klasse *ContentType*, also eine Objektvariable ist. Außerdem liefert die Funktion *getContent()* nun eben diesen Zeiger (*content*) zurück.
Auch die Schlange selbst liefert nun mit *front()* nur den Objektzeiger *content* zurück. Analog enthält die Schlangenmethode *enqueue(ContentType p)* jetzt eine Objektvariable als Parameter.

Es werden nun die Dokumentationen für die generische Schlange und die zugehörige Klasse QueueNode angegeben:

Dokumentation der Klasse Queue<ContentType>

Konstruktor **Queue<ContentType>()** bzw. **Queue<>()**
Eine leere Schlange wird erzeugt.

Anfrage **boolean isEmpty()**
Die Anfrage liefert den Wert *true*, wenn die Schlange keine

Objekte enthält, sonst liefert sie den Wert *false*.

Auftrag **void enqueue(ContentType pContent)**
Das Objekt pContent wird an die Schlange angehängt. Falls pContent gleich *null* ist, bleibt die Schlange unverändert.

Auftrag **void dequeue()**
Falls die Schlange leer ist, wird sie nicht verändert. Ansonsten wird das erste Objekt aus der Schlange entfernt.

Anfrage **ContentType front()**
Die Anfrage liefert das erste Objekt der Schlange. Die Schlange bleibt unverändert. Falls die Schlange leer ist, wird *null* zurückgegeben.

Dokumentation der privaten Klasse QueueNode

Konstruktor **QueueNode(ContentType pContent)**
Ein neues Objekt vom Typ *QueueNode<ContentType>* wird erschaffen. Der Inhalt wird per Parameter gesetzt. Der Verweis auf den nächsten Knoten ist leer.

Auftrag **void setNext(QueueNode pNext)**
Der Verweis wird auf den nächsten Knoten, das als Parameter übergeben wird, gesetzt.

Anfrage **QueueNode getNext()**
Liefert das nächste Element des aktuellen Knotens.

Anfrage **ContentType getContent()**
Liefert das Inhaltsobjekt des Knotens vom Typ ContentType.

Im Folgenden werden die Implementationen der beiden Klassen *QueueNode* und *Queue<ContentType>* aufgeführt. Dabei wird die Klasse *QueueNode* als private Klasse innerhalb der Klasse *Queue<ContentType>* implementiert. Neu ist auch, dass die Klasse *Queue<ContentType>* als *generische Klasse* realisiert wird, d.h. einem Objekt der Klasse *Queue<ContentType>*, also einer konkreten Schlange, wird erst bei seiner Erzeugung mitgeteilt, welche Objekttypen die Warteschlange eigentlich verwalten soll.

Materialien zu den zentralen NRW-Abiturprüfungen im Fach Informatik ab 2018.
Objekte der generischen Klasse Queue (Warteschlange) verwalten beliebige Objekte vom Typ ContentType nach dem First-In-First-Out-Prinzip, d.h., das zuerst abgelegte Objekt wird als erstes wieder entnommen.

Autor: Qualitäts- und UnterstützungsAgentur - Landesinstitut für Schule

Erzeuge die Klasse *Queue<ContentType>,* deren Quelltext gleich dargestellt werden wird, folgendermaßen:
NetBeans/ File/ New File/ Kategorie Java/ Type Java Class
Verwende dabei als Dateinamen für den Typ *Java Class* einfach den Kurznamen *Queue*!

```
public class Queue<ContentType> {

// --------- Anfang der privaten inneren Klasse --------------
private class QueueNode {
  private ContentType content = null;
  private QueueNode nextNode = null;

 // Ein neues Objekt vom Typ QueueNode<ContentType> wird erschaffen.
 // Der Inhalt wird per Parameter gesetzt. Der Verweis auf den nächsten Knoten ist leer.
  public QueueNode(ContentType pContent) {
    content = pContent;
    nextNode = null;
  }
```

```
    // Der Verweis wird auf den nächsten Knoten, der als Parameter übergeben wird, gesetzt.
    public void setNext(QueueNode pNext) {
      nextNode = pNext;
    }

    // Liefert das nächste Element des aktuellen Knotens.
    public QueueNode getNext() {
      return nextNode;
    }

    // Liefert das Inhaltsobjekt des Knotens vom Typ ContentType.
    public ContentType getContent() {
      return content;
    }

  }// ----------- Ende der privaten inneren Klasse-------------------

  // Nun folgen die Attribute und Methoden der Klasse Queue<ContentType>

  private QueueNode head;
  private QueueNode tail;

  // Eine leere Schlange wird erzeugt.
  // Objekte, die in dieser Schlange verwaltet werden, müssen vom Typ ContentType sein.
  public Queue() {
    head = null;
    tail = null;
  }


  // Die Anfrage liefert den Wert true, wenn die Schlange keine Objekte enthält, sonst liefert
  // sie den Wert false.
  public boolean isEmpty() {
    return head == null;
  }
```

```
// Das Objekt pContent wird an die Schlange angehängt. Falls pContent gleich null ist,
// bleibt die Schlange unverändert.
public void enqueue(ContentType pContent) {
  if (pContent != null) {
    QueueNode newNode = new QueueNode(pContent);
    if (this.isEmpty()) {
      head = newNode;
      tail = newNode;
    }
    else {
      tail.setNext(newNode);
      tail = newNode;
    }
  }
}


// Das erste Objekt wird aus der Schlange entfernt.
// Falls die Schlange leer ist, wird sie nicht veraendert.
  public void dequeue() {
    if (!this.isEmpty()) {
      head = head.getNext();
      if (this.isEmpty()) {
        head = null;
        tail = null;
      }
    }
  }


// Die Anfrage liefert das erste Objekt der Schlange. Die Schlange bleibt unverändert.
// Falls die Schlange leer ist, wird null zurückgegeben.
  public ContentType front() {
    if (this.isEmpty()) return null;
    else return head.getContent();
  }

}   // Ende der Klasse Queue<ContentType>
```

Diese neue Datenstruktur hat den entscheidenden Vorteil, dass man alle möglichen Datentypen mit ihnen verwalten kann, **vorausgesetzt, es sind sog. Referenztypen, also Objekte einer Klasse.**

Wir werden in Zukunft aber immer wieder auch mit *primitiven* Datentypen (Zahlen, Zeichen, Strings) arbeiten, insbesondere deshalb, weil die entsprechenden Beispiele damit relativ einfach sind. Leider kann man *primitive* Datentypen nicht mit obiger Klasse *Queue<ContentType>* verwalten. Um trotzdem damit arbeiten zu können, muss man Klassen erschaffen, deren einziger Inhalt im Prinzip nur diese primitive Datentypen sind. Diese Klassen bezeichnet man auch als *Ummantelungsklassen* , *Envelopklassen* oder *Wrapperklassen*.

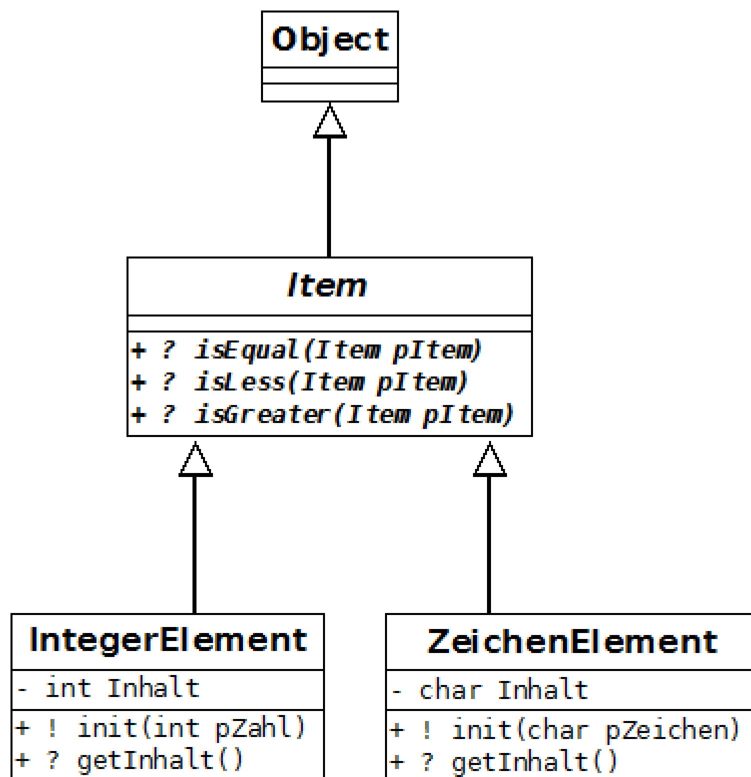
In Java werden für alle *primitiven* Datentypen passende *Wrapperklassen* zur Verfügung gestellt. Diese Klassen enthalten zusätzlich noch sehr viele Methoden z.B. für die Umwandlung von Zahlen in Strings, für die wissenschaftliche Darstellung von reellen Zahlen (Exponentialschreibweise), für die kaufmännische Darstellung von Zahlen und vieles mehr.

Wir werden später auch Zahlen, Zeichen und Strings der Größe nach miteinander vergleichen müssen. Die Objekte der entsprechenden Wrapperklassen müssen sich also auch größenmäßig miteinander vergleichen lassen. Dazu muss es entsprechende Methoden geben.

Wir werden nun folgende Wrapper-Klassen einführen: *IntegerElement, RealElement, ZeichenElement, StringElement*. Weil alle diese Klassen ähnliche Methoden zum Größenvergleich haben, werden wir diese Klassen als Unterklasse einer gemeinsamen Oberklasse namens *Item* realisieren.

Die Klasse *Item*

Die Klasse *Item* ist Oberklasse aller Klassen, deren Objekte von uns der Größe nach irgendwie sortiert werden müssen.
Die Ordnungsrelation wird in den Unterklassen von *Item* durch Überschreiben der drei abstrakten Methoden *isEqual*, *isGreater* und *isLess* festgelegt.
In UML-Diagrammen werden die Namen von abstrakten Klassen und abstrakten Methoden kursiv geschrieben.




```
public abstract class Item {

  public abstract boolean isGreater (Item pItem);

  public abstract boolean isLess (Item pItem);

  public abstract boolean isEqual (Item pItem);
}
```

Die Definition einer abstrakten Methode wird mit einem Semikolon abgeschlossen!

Das Schlüsselwort *abstract* leitet die Definition einer abstrakten Klasse ein. Eine Klasse ist abstrakt, wenn sie als abstrakt deklariert wird. Von einer abstrakten Klasse können keine Objekte erzeugt werden. Eine abstrakte Klasse hat üblicherweise mindestens eine abstrakte Methode; es ist allerdings auch erlaubt, dass sie keine abstrakte Methode besitzt. Im letzteren Fall ist es dann aus anderen Gründen nicht sinnvoll, Objekte dieser abstrakten Klasse zu erzeugen.

Eine abstrakte Methode definiert lediglich den Namen und die Parameter einer Methode, und eine Unterklasse implementiert dann irgendwann diese Methode. Durch abstrakte Methoden wird ausgedrückt, dass die Oberklasse keine Ahnung von der Implementierung hat und dass sich die Unterklassen darum kümmern müssen.

Von einer abstrakten Klasse kann man direkt keine Objekte bilden, weil die abstrakten Methoden noch gar nicht definiert sind. Erst von den Unterklassen lassen sich Objekte bilden.

Als nächstes folgen die Implementationen der von *Item* abgeleiteten Klassen.

```
class IntegerElement extends Item {
  private int inhalt;


  public IntegerElement(int n)  {   //Konstruktor
    inhalt = n;
  }


  public int getInhalt()  {
    return inhalt;
  }


  @Override   /* Dies ist eine Anweisung für den Compiler. Der Compiler
  soll darauf achten, dass in der zugehörigen Oberklasse wirklich eine exakt
  gleichnamige Methode mit exakt denselben Parametern existiert.  */
  public boolean isEqual(Item pItem)  {
    return (this.inhalt ==
                ((IntegerElement)pItem).getInhalt());
  }


  @Override
  public boolean isLess(Item pItem)  {
    return (this.inhalt <
                ((IntegerElement)pItem).getInhalt());
  }


  @Override
  public boolean isGreater(Item pItem)  {
    return (this.inhalt >
                ((IntegerElement)pItem).getInhalt());
  }
} // Ende von class IntegerElement
```

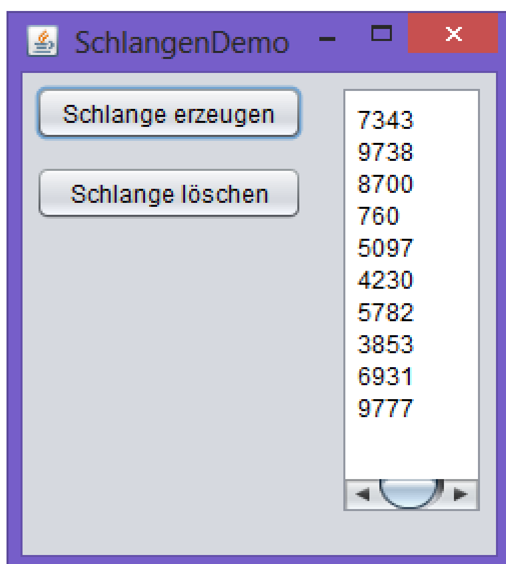
```
class ZeichenElement extends Item {
  private char inhalt;


  public ZeichenElement(char ch)  {   //Konstruktor
    inhalt = ch;
  }


  public char getInhalt()  {
    return inhalt;
  }


  @Override
  public boolean isEqual(Item pItem)  {
    return (this.inhalt ==
              ((ZeichenElement)pItem).getInhalt());
  }


  @Override
  public boolean isLess(Item pItem)  {
    return (this.inhalt <
              ((ZeichenElement)pItem).getInhalt());
  }


  @Override
  public boolean isGreater(Item pItem)  {
    return (this.inhalt >
              ((ZeichenElement)pItem).getInhalt());
  }
} // Ende von class ZeichenElement
```

```
class RealElement extends Item {
  private double inhalt;


  public RealElement(double x)  {   //Konstruktor
    inhalt = x;
  }


  public double getInhalt()  {
    return inhalt;
  }


  @Override
  public boolean isEqual(Item pItem)  {
    return (this.inhalt ==
                ((RealElement)pItem).getInhalt());
  }


  @Override
  public boolean isLess(Item pItem)  {
    return (this.inhalt <
                ((RealElement)pItem).getInhalt());
  }


  @Override
  public boolean isGreater(Item pItem)  {
    return (this.inhalt >
                ((RealElement)pItem).getInhalt());
  }
} // Ende von class RealElement
```

```
class StringElement extends Item {
  private String inhalt;


  public StringElement(String s)  {   //Konstruktor
    inhalt = s;
  }


  public String getInhalt()  {
    return inhalt;
  }


  @Override
  public boolean isEqual(Item pItem)  {
    return
    (this.inhalt.compareTo(((StringElement)pItem).getInhalt())
                                                          == 0);
  }


  @Override
  public boolean isLess(Item pItem)  {
    return
    (this.inhalt.compareTo(((StringElement)pItem).getInhalt())
                                                           < 0);
  }


  @Override
  public boolean isGreater(Item pItem)  {
    return
    (this.inhalt.compareTo(((StringElement)pItem).getInhalt())
                                                           > 0);
  }
}  // Ende von class StringElement
```

Die String-Methode *compareTo(String s)* führt einen *lexikalischen* Vergleich beider Strings durch. Bei einem lexikalischen Vergleich werden die Zeichen paarweise von links nach rechts verglichen. Tritt ein Unterschied auf oder ist einer der Strings beendet, wird das Ergebnis ermittelt. Ist das aktuelle String-Objekt dabei lexikalisch kleiner als s, wird ein negativer Integer-Wert zurück-gegeben. Ist es größer, wird ein positiver Integer-Wert zurückgegeben. Bei Gleichheit liefert die Methode den Rückgabewert 0. Der Rückgabewert entspricht immer der Differenz der ASCII-Code-Werte der beiden ersten ungleichen Zeichen (bzw. 0 bei völliger Gleichheit).

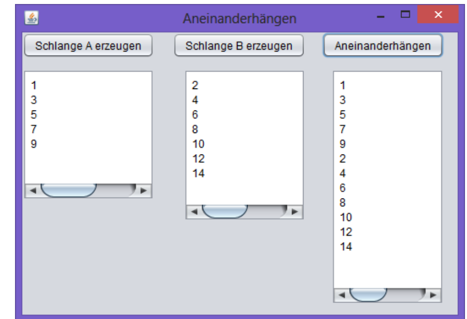
Im folgenden kleinen Demo-Programm sollen Zahlen mit einer Schlange verwaltet werden.




```
public class SchlangenDemo extends javax.swing.JFrame {
  Queue<IntegerElement> schlange;

  private void gibAus()  {
    taAusgabe.setText("");
    Queue<IntegerElement> hilf =
                                    new Queue<IntegerElement> ();
// alternativ kann man ab der Version Java 7 auch den sog. Diamond-Operator <> benutzen,
// allerdings nur im Zusammenhang mit der Anweisung new
// Queue<IntegerElement> hilf = new Queue<> ();
```

```
    IntegerElement e;
    while (! schlange.isEmpty())  {
      e = schlange.front();
      taAusgabe.append(e.getInhalt()+"\n");
      schlange.dequeue();
      hilf.enqueue(e);
    }
    schlange = hilf;
    hilf = null;
  }


  private void btErzeugenMouseClicked(java……) {
    if (schlange == null) schlange = new Queue<>();
    for (int i = 1; i <= 10; i++) {
      IntegerElement e = new IntegerElement(
                         (int) (1+9999*Math.random()));
      schlange.enqueue(e);
    }
    gibAus();
  }


  private void btLoeschenMouseClicked(java……) {
    schlange = null;
    taAusgabe.setText("");
  }
}
```

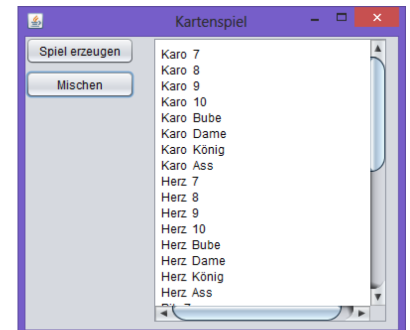
Aufgaben



1. Erweitere die generische Klasse Queue<ContentType> um eine weitere Methode namens **`void append(Queue<ContentType> s)`**, so dass man eine zweite Schlange an die aktuelle Schlange anhängen kann.

2. Das Hauptprogramm soll eine Schlange B an eine schon bestehende Schlange A anhängen. Die Schlange A wird dadurch länger. Die Schlange B verschwindet.

3. Schreibe ein Programm zur Simulation einer Tanzschule. In der Tanzschule treffen neue Personen ein, die entweder in die Schlange der Damen oder in die Schlange der Herren eingereiht werden. Anschließend werden Paare gebildet, die in eine dritte Schlange, die Tanzschlange, eingereiht werden. Paare verlassen die Tanzschlange in der Reihenfolge ihrer Ankunft.



4. Simulation eines Skat-Kartenspiels.
Benutze für das folgende Programm die Aufzählungstypen
public enum Farben {KARO, HERZ, PIK, KREUZ };
public enum Werte {SIEBEN, ACHT, NEUN, ZEHN, BUBE, DAME, KOENIG, ASS};

a) Erzeuge eine Klasse *Spielkarte* mit den Attributen *farbe* und *wert*! Für die Ausgabe werden u.a. Methoden zur Umwandlung der Attribute in Strings benötigt, also etwa *wert2String()* und *farbe2String()*.

b) Erzeuge eine Schlange namens *spiel*, welche alle 32 Skat-Karten speichert.

c) Alle Karten der Schlange *spiel* sollen untereinander in einer Textarea ausgegeben werden.

d) Die Karten sollen gemischt werden. Dazu werden alle Karten nach dem Zufallsprinzip auf vier unterschiedliche Schlangen verteilt. Anschließend werden sie wieder in die Originalschlange *spiel* zurückgeschrieben. Die Karten der Schlange werden anschließend in der Textarea ausgegeben.

e) Das *spiel* soll nach Farben sortiert werden. Die Karten der Schlange werden anschließend in der Textarea ausgegeben.

Lösungen

Aufgabe 1 (Methode append)

```
public void append(Queue<ContentType> s)  {
  ContentType e;
  while (! s.isEmpty()) {
     e = s.front();
     this.enqueue(e);
     s.dequeue();
  }
}
```

Aufgabe 2 (Schlange anhängen, ohne Benutzung von append)

```
Queue<IntegerElement> schlangeA, schlangeB;

private void btAErzeugenMouseClicked(java......) {
  schlangeA = new Queue<>();
  taAusgabeA.setText("");
  for (int i=1; i <= 5; i++)  {
    IntegerElement e = new IntegerElement(2*i-1);
    schlangeA.enqueue(e);
    taAusgabeA.append(e.getInhalt() + "\n");
  }
}

private void btBErzeugenMouseClicked(java......) {
  schlangeB = new Queue<>();
  taAusgabeB.setText("");
  for (int i=1; i <= 7; i++)  {
    IntegerElement e = new IntegerElement(2*i);
    schlangeB.enqueue(e);
    taAusgabeB.append(e.getInhalt() + "\n");
  }
}
```

```
private void btAnhaengenMouseClicked(java......) {
  if ((schlangeA != null) && (schlangeB != null))  {
     IntegerElement ob;
     while (! schlangeB.isEmpty()) {
        ob = schlangeB.front();
        schlangeA.enqueue(ob);
        schlangeB.dequeue();
     }

     taAusgabeC.setText("");
     Queue hilf = new Queue<>();
     IntegerElement e;
     while (! schlangeA.isEmpty())  {
        e = schlangeA.front();
        hilf.enqueue(e);
        taAusgabeC.append(e.getInhalt() + "\n");
        schlangeA.dequeue();
     }
     schlangeA = hilf;
  }
}
```

Aufgabe 4 (Skat-Kartenspiel)

```
public class Kartenspiel extends javax.swing.JFrame {

public enum Farben {KARO, HERZ, PIK, KREUZ};
public enum Werte {SIEBEN, ACHT, NEUN, ZEHN, BUBE,
                                  DAME, KOENIG, ASS};


class Spielkarte  {
  Farben farbe;
  Werte wert;

  public Spielkarte(Farben pf, Werte pw)  {
```

```
  farbe = pf;
  wert = pw;
}

public Farben getFarbe()  {
  return farbe;
}

public Werte getWert()  {
  return wert;
}

public String farbe2String() {
  String s="";
  switch (farbe)  {
    case KARO: s = "Karo"; break;
    case HERZ: s = "Herz"; break;
    case PIK: s = "Pik"; break;
    case KREUZ: s = "Kreuz";
  }
  return s;
}


public String wert2String() {
  String s="";
  switch (wert)  {
    case SIEBEN: s = "7"; break;
    case ACHT: s = "8"; break;
    case NEUN: s = "9"; break;
    case ZEHN: s = "10"; break;
    case BUBE: s = "Bube"; break;
    case DAME: s = "Dame"; break;
    case KOENIG: s = "König"; break;
    case ASS: s = "Ass";
  }
  return s;
}
```

```
} // Ende der Klasse Spielkarte


Queue<Spielkarte> spiel;


public Kartenspiel() {
        initComponents();
    }


private void btErzeugenMouseClicked(java……) {
  spiel = new Queue<>();
  Spielkarte e;
  for (Farben f: Farben.values())  {
     for (Werte w: Werte.values())  {
        e = new Spielkarte(f, w);
        spiel.enqueue(e);
     }
  }
  spielAusgabe();
}


private void btMischenMouseClicked(java……) {
  Queue<Spielkarte> hilfA = new Queue<>();
  Queue<Spielkarte> hilfB = new Queue<>();
  Queue<Spielkarte> hilfC = new Queue<>();
  Queue<Spielkarte> hilfD = new Queue<>();
  int zufallszahl;
  Spielkarte ob;
  while (! spiel.isEmpty())  {
    ob = spiel.front();
    spiel.dequeue();
    zufallszahl = (int) (1 + 4*Math.random());
    switch (zufallszahl)  {
```

```
      case 1: hilfA.enqueue(ob); break;
      case 2: hilfB.enqueue(ob); break;
      case 3: hilfC.enqueue(ob); break;
      case 4: hilfD.enqueue(ob);
    }
  }
  spiel.append(hilfA);
  spiel.append(hilfB);
  spiel.append(hilfC);
  spiel.append(hilfD);
  spielAusgabe();
}


void ausgabe(Spielkarte sp)  {
   taAusgabe.append(sp.farbe2String() + "  " +
                            sp.wert2String() + "\n");
}


void spielAusgabe()  {
   taAusgabe.setText("");
   Queue<Spielkarte> hilf = new Queue<>();
   Spielkarte e;
   while (! spiel.isEmpty())  {
     e = spiel.front();
     ausgabe(e);
     spiel.dequeue();
     hilf.enqueue(e);
   }
   spiel = hilf;
}

}// Ende von Kartenspiel
```

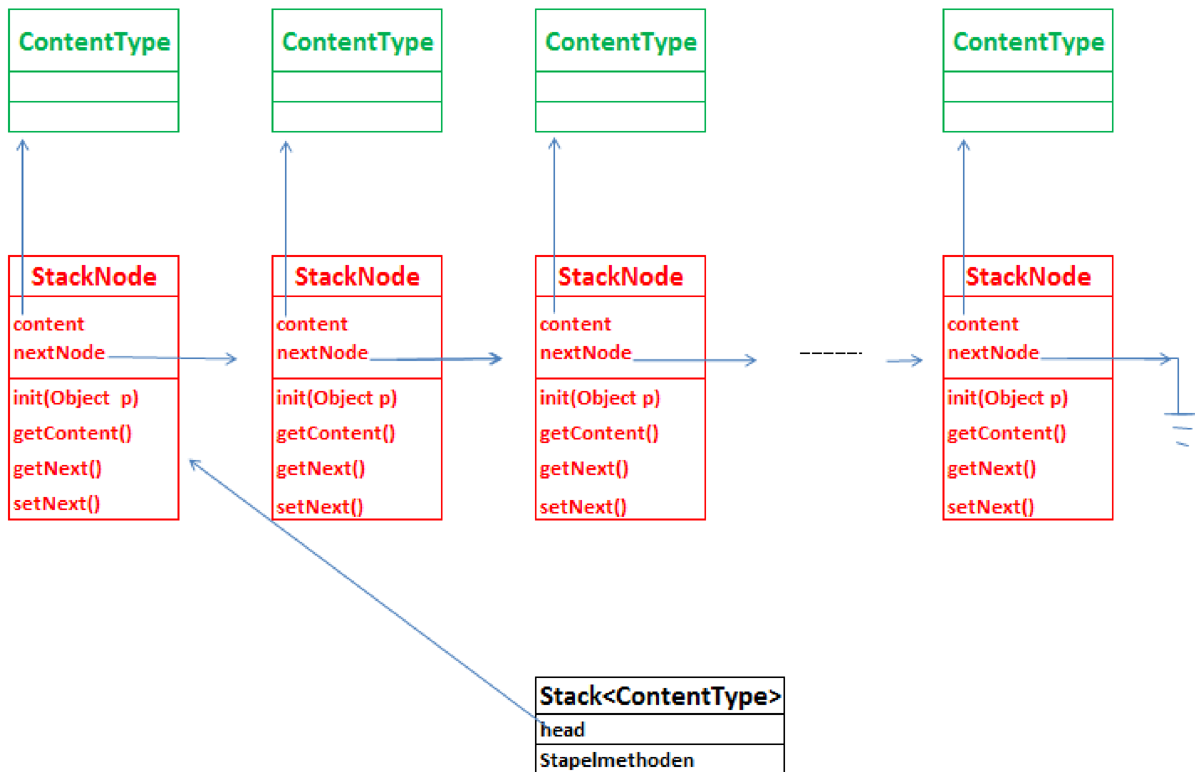
Die Klasse *Stack*, generische Version

Objekte der generischen Klasse *Stack<ContentType>* (Keller, Stapel) verwalten beliebige Objekte nach dem ***LIFO***-Prinzip (**L**ast-**In**-**F**irst-**O**ut), d.h. das zuletzt abgelegte Element wird als erstes wieder entnommen.
Auch der Typ *Stack<ContentType>* ist im Prinzip eine Liste von Objekten. Es können zum Beispiel mehrere Listen existieren, die teilweise dieselben Objekte enthalten. Aus diesem Grund soll bei der Methode *pop* das Objekt zwar aus der Liste, aber nicht aus dem Speicher gelöscht werden.

Die Syntax der Methoden der generischen Klasse *Stack<ContentType>* wurde für alle Abiturjahrgänge ab 2017 in NRW gemeinsam folgendermaßen festgelegt:

Dokumentation der Klasse Stack<ContentType>

Konstruktor	**Stack()** ein leerer Stapel wird erzeugt.
Anfrage	**boolean isEmpty()** Die Anfrage liefert den Wert *true*, wenn der Stapel keine Objekte enthält, sonst liefert sie den Wert *false*.
Auftrag	**void push(ContentType p)** Das Objekt p wird oben auf den Stapel gelegt. Falls p gleich *null* ist, bleibt der Stapel unverändert.
Auftrag	**void pop()** Das zuletzt eingefügte Objekt wird von dem Stapel entfernt (aber nicht aus dem Speicher gelöscht). Falls der Stapel leer ist, bleibt er unverändert.
Anfrage	**ContentType top()** Die Anfrage liefert das oberste Stapelobjekt. Der Stapel bleibt unverändert. Falls der Stapel leer ist, wird *null* zurückgegeben.



Materialien zu den zentralen NRW-Abiturpruefungen im Fach Informatik ab 2017

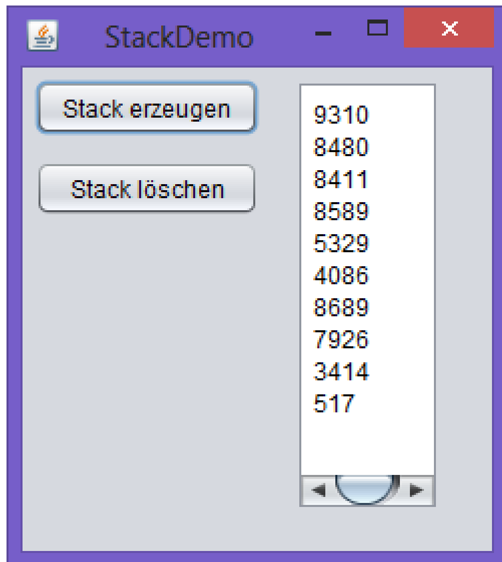
Generische Klasse Stack<ContentType>

Objekte der generischen Klasse Stack (Keller, Stapel) verwalten beliebige Objekte vom Typ ContentType nach dem Last-In-First-Out-Prinzip, d.h., das zuletzt abgelegte Objekt wird als erstes wieder entnommen.

@author Qualitaets- und UnterstuetzungsAgentur - Landesinstitut fuer Schule, Materialien zum schulinternen Lehrplan Informatik SII

```
public class Stack<ContentType> {

/* ----- Anfang der privaten inneren Klasse ----- */
  private class StackNode {

    private ContentType content = null;
    private StackNode nextNode = null;

    public StackNode(ContentType pContent) {
      content = pContent;
      nextNode = null;
    }


    public void setNext(StackNode pNext) {
      nextNode = pNext;
    }


    public StackNode getNext() {
      return nextNode;
    }


    public ContentType getContent() {
      return content;
    }
  }
 /* ------- Ende der privaten inneren Klasse -------- */


  private StackNode head;

  public Stack() {
    head = null;
  }
```

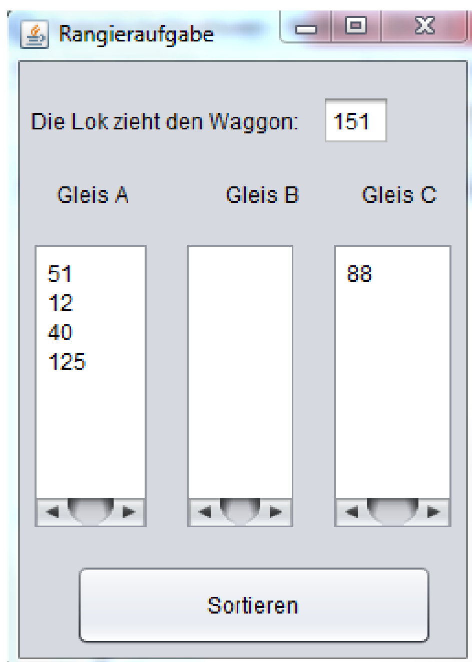
```
  public boolean isEmpty() {
    return (head == null);
  }


  public void push(ContentType pContent) {
    if (pContent != null) {
      StackNode node = new StackNode(pContent);
      node.setNext(head);
      head = node;
    }
  }


  public void pop() {
    if (!isEmpty()) {
      head = head.getNext();
    }
  }


  public ContentType top() {
    if (!this.isEmpty()) {
      return head.getContent();
    } else {
      return null;
    }
  }
}
```




```
public class StackDemo extends javax.swing.JFrame {

  Stack<IntegerElement> stapel;

  public StackDemo() {
     initComponents();
  }

  private void gibAus()  {
    taAusgabe.setText("");
    Stack<IntegerElement> hilf = new Stack<>();
    IntegerElement e;
    while (! stapel.isEmpty())  {
       e = stapel.top();
       taAusgabe.append(e.getInhalt()+"\n");
       stapel.pop();
       hilf.push(e);
    }
    while (! hilf.isEmpty())  {
       e = hilf.top();
       stapel.push(e);
       hilf.pop();
    }
    hilf = null;
```

```
  }

  private void btErzeugenMouseClicked(java……) {
    stapel = new Stack<>();
    for (int i = 1; i <= 10; i++) {
       IntegerElement e = new IntegerElement((int)
                                    (1+9999*Math.random()));
       stapel.push(e);
    }
    gibAus();
  }


  private void btLoeschenMouseClicked(java……) {
    stapel = null;
    taAusgabe.setText("");
  }
} // Ende des Programms StackDemo
```

Rangieraufgabe



Auf einem Bahnhof soll eine Lok die einzelnen Waggons eines Zuges sortieren. Die Waggons sind nummeriert und stehen auf dem Gleis A in ungeordneter Reihenfolge hintereinander. Nach getaner Arbeit sollen die einzelnen Waggons auf dem Gleis C in sortierter Reihenfolge stehen. Als Hilfsgleis steht das Gleis B zur Verfügung. Die Lok kann immer nur einen einzelnen Waggon verschieben. Die Lok „sieht“, welche Waggonnummer auf welchem Gleis vorne steht.

Lösung der Rangieraufgabe

Die *showMessageDialog*-Anweisung in der Methode Listenausgabe im folgenden Programm behindert zwar erheblich den flüssigen Programmablauf, aber sie verdeutlicht den Lösungsweg.

```
import javax.swing.*;   // wird für die Ausgabe von Dialogen benötigt
public class Rangieraufgabe extends javax.swing.JFrame {
 Stack<IntegerElement> StackA, StackB, StackC;
 IntegerElement e;

 public Rangieraufgabe() {
   initComponents();
   StackA = new Stack<>();
   StackB = new Stack<>();
   StackC = new Stack<>();

   for (int i = 1; i<=6; i++) {
      e = new IntegerElement((int) (Math.random()*200));
      StackA.push(e);
   }

   listenausgabe();
 }


 private void Ausgabe(Stack<IntegerElement> stapel, JTextArea
                                                    memo)   {
   IntegerElement e;
   Stack<IntegerElement> hilf;

   if (stapel != null)  {
      hilf = new Stack<>();
      while (! stapel.isEmpty())  {
        e = stapel.top();
        memo.append(e.getInhalt() + "\n");
        hilf.push(e);
        stapel.pop();
      }
      while (! hilf.isEmpty())   {
        e =  hilf.top();
        stapel.push(e);
        hilf.pop();
```

```
      }
    }     // of if
  }


private void listenausgabe()  {
  taA.setText("");
  taB.setText("");
  taC.setText("");

  if (! StackA.isEmpty()) Ausgabe(StackA, taA);
  if (! StackB.isEmpty()) Ausgabe(StackB, taB);
  if (! StackC.isEmpty()) Ausgabe(StackC, taC);
  JOptionPane.showMessageDialog(this, "Wartepause");
}


 private void btSortierenMouseClicked(java……..)    {
   IntegerElement eA, eB, eC;
   while (! StackA.isEmpty())   {
      eA = StackA.top();
      if (StackC.isEmpty())   {
         tfLok.setText("" + eA.getInhalt());
         StackA.pop();
         listenausgabe();
         StackC.push(eA);
         tfLok.setText("");
         listenausgabe();
      }
      else     {  //else Nummer 1
        eC = StackC.top();
        if (eC.isLess(eA))   {
           tfLok.setText("" + eA.getInhalt());
           StackA.pop();
           listenausgabe();
           StackC.push(eA);
           tfLok.setText("");
           listenausgabe();
        }
        else    {   //else Nummer 2
           do  {
             tfLok.setText("" + eC.getInhalt());
             StackC.pop();
             listenausgabe();
```

```
          tfLok.setText("");
          StackB.push(eC);
          listenausgabe();
          if(! StackC.isEmpty())
          eC = StackC.top();
        }   // von do
        while (! StackC.isEmpty() && eC.isGreater(eA));

        tfLok.setText("" + eA.getInhalt());
        StackA.pop();
        listenausgabe();
        StackC.push(eA);
        tfLok.setText("");
        listenausgabe();
        do  {
          eB = StackB.top();
          StackB.pop();
          tfLok.setText("" + eB.getInhalt());
          listenausgabe();
          tfLok.setText("");
          StackC.push(eB);
          listenausgabe();
        }
        while (!StackB.isEmpty());
      }   // von else Nummer 2
    }   // von else Nummer 1
  }   // von while
  JOptionPane.showMessageDialog(this, "Fertig");
}

}   // Ende der Klasse Rangieraufgabe
```

Die Klasse *Langzahl*

Dokumentation

Objekte der Klasse *Langzahl* stellen beliebig große, natürliche Zahlen einschließlich 0 dar. Ein Objekt wird durch einen Stack<IntegerElement> dargestellt. Jedes Stackelement enthält nur eine einzige Dezimalziffer. Oben auf dem Stack liegt die Einerstelle der Zahl.

Oberklasse:	**`Stack<IntegerElement>`**
Konstruktor	**`Langzahl()`**
nachher	Eine leeres Objekt ist erzeugt. Insbesondere besitzt diese Langzahl noch keinen Wert.
Konstruktor	**`Langzahl(String s)`**
vorher	s stellt eine gültige natürliche Zahl einschließlich 0 dar.
nachher	ein Langzahl-Objekt mit dem entsprechenden Wert wird erzeugt.
Anfrage	**`String toStr()`**
nachher	der Wert des Objektes wird als String geliefert. Das Objekt selbst wird nicht verändert.
Auftrag	**`eliminiereFuehrendeNullen()`**
nachher	das Objekt besitzt die kürzest mögliche Darstellung.
Anfrage	**`Langzahl plus(Langzahl lz)`**
nachher	es wird die Summe (Objekt + lz) geliefert. Das Objekt selbst und auch lz werden nicht verändert.
Anfrage	**`Langzahl minus(Langzahl lz)`**
vorher	das Objekt muss größer oder gleich LZ sein, so dass die Differenz nicht negativ wird.
nachher	es wird die Differenz (Objekt – lz) geliefert. Das Objekt selbst und auch lz werden nicht verändert.

Anfrage **`Langzahl mal(int faktor)`**
vorher faktor darf nicht negativ sein.
nachher das Produkt wird geliefert. Das Objekt selbst und auch faktor werden nicht verändert.

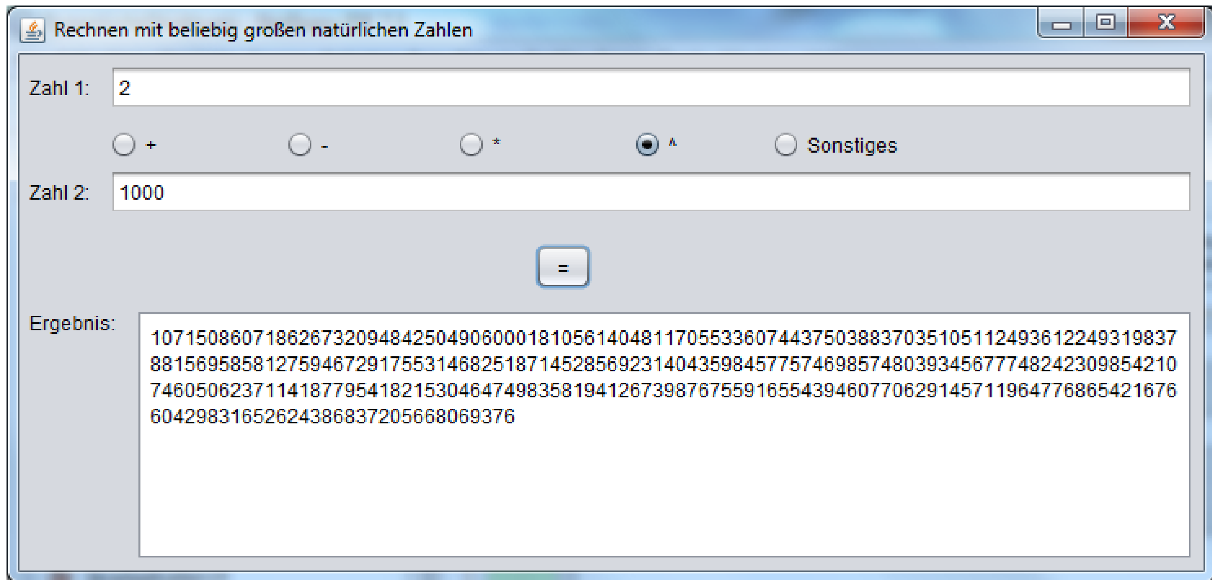
Anfrage **`Langzahl hoch(int exponent)`**
vorher exponent darf nicht negativ sein. Der Wert des Objektes (this) liegt noch im Integerbereich.
nachher die Potenz wird geliefert. Das Objekt (this) selbst und auch exponent werden nicht verändert.

Anfrage **`int stellenanzahl()`**
nachher die Anzahl der Dezimalstellen wird geliefert.

Anfrage **`boolean isGreater(Langzahl lz)`**
nachher liefert Wahrheitswert von (Objekt > lz).

Anfrage **`boolean isLess(Langzahl lz)`**
nachher liefert Wahrheitswert von (Objekt < lz).

Anfrage **`boolean isEqual(Langzahl lz)`**
nachher liefert den Wahrheitswert von (Objektwert = lz).



Hinweis: Die Textarea sollte die Eigenschaft *LineWrap* besitzen. Dann erfolgt der Zeilenumsprung automatisch.

```
public class Langzahl extends Stack<IntegerElement> {

 public Langzahl()   {
  // keine besondere Konstruktion, eine leere Langzahl wird erzeugt
 }


 public Langzahl(String s)  {
   IntegerElement e;
   char ch;
   int n;
   while ( (1<s.length()) && (s.charAt(0)=='0') )
       s = s.substring(1);
   for (int i=0; i<s.length(); i++)  {
      ch = s.charAt(i);
      n = (int) ch;
      e = new IntegerElement(n-48);
      this.push(e);
   }
 }



 public String toStr()  {
    String s = "";
    IntegerElement e;
    Langzahl hilf = new Langzahl();

    while (! this.isEmpty()) {
       e = this.top();
       s = e.getInhalt() + s;
       hilf.push(e);
       this.pop();
    }
    while (! hilf.isEmpty()) {
       this.push(hilf.top());
       hilf.pop();
    }
    return s;
 }



 public void eliminiereFuehrendeNullen()  {
   Langzahl hilf = new Langzahl();
   IntegerElement e;

   while (! this.isEmpty()) {
      hilf.push(this.top());
      this.pop();
```

```
    }
   while (! hilf.isEmpty() &&  (hilf.top().getInhalt() == 0))
      hilf.pop();
   if (hilf.isEmpty())   {
      e=new IntegerElement(0);
      hilf.push(e);
   }

   while (! hilf.isEmpty())  {
      this.push(hilf.top());
      hilf.pop();
   }
 }



 public int stellenanzahl()  {
    int zaehler = 0;
    Stack<IntegerElement> hilf = new Stack<>();
    this.eliminiereFuehrendeNullen();
    while (! this.isEmpty())  {
       hilf.push(this.top());
       zaehler++;
       this.pop();
    }
    while (! hilf.isEmpty())  {
       this.push(hilf.top());
       hilf.pop();
    }

    return zaehler;
 }



 public Langzahl plus(Langzahl lz)  {

   Langzahl hilf = new Langzahl();
   Langzahl zahl1 = new Langzahl();
   Langzahl zahl2 = new Langzahl();

   while (! this.isEmpty())  {
      hilf.push(this.top());
      this.pop();
   }
   while (! hilf.isEmpty())  {
      this.push(hilf.top());
      zahl1.push(hilf.top());
      hilf.pop();
   }
```

```
while (! lz.isEmpty())  {
   hilf.push(lz.top());
   lz.pop();
}
while (! hilf.isEmpty())   {
   lz.push(hilf.top());
   zahl2.push(hilf.top());
   hilf.pop();
}

IntegerElement e1, e2, e3;
int uebertrag = 0;
int ziffer;

while (! zahl1.isEmpty() && ! zahl2.isEmpty())   {
   e1 = zahl1.top();
   e2 = zahl2.top();
   ziffer = (e1.getInhalt() + e2.getInhalt() +uebertrag)%10;
   uebertrag = (e1.getInhalt() + e2.getInhalt() + uebertrag)/10;
   zahl1.pop();
   zahl2.pop();
   e3 = new IntegerElement(ziffer);
   hilf.push(e3);
}

if (zahl1.isEmpty())   {
   while (! zahl2.isEmpty())   {
      e2 = zahl2.top();
      ziffer = (e2.getInhalt() + uebertrag)% 10;
      uebertrag = (e2.getInhalt() + uebertrag)/ 10;
      e3 = new IntegerElement(ziffer);
      hilf.push(e3);
      zahl2.pop();
   }
}
else
   while (! zahl1.isEmpty())   {
      e1 = zahl1.top();
      ziffer = (e1.getInhalt() + uebertrag)% 10;
      uebertrag = (e1.getInhalt() + uebertrag)/ 10;
      e3 = new IntegerElement(ziffer);
      hilf.push(e3);
      zahl1.pop();
   }

if (uebertrag > 0)   {
   e3 = new IntegerElement(uebertrag);
   hilf.push(e3);
}
```

```
  Langzahl ergebnis = new Langzahl();
  while (! hilf.isEmpty())  {
     ergebnis.push(hilf.top());
     hilf.pop();
  }

  return ergebnis;
}



public Langzahl minus(Langzahl lz)  {

  Langzahl hilf = new Langzahl();
  Langzahl zahl1 = new Langzahl();
  Langzahl zahl2 = new Langzahl();

  while (! this.isEmpty())  {
     hilf.push(this.top());
     this.pop();
  }
  while (! hilf.isEmpty())  {
     this.push(hilf.top());
     zahl1.push(hilf.top());
     hilf.pop();
  }
  while (! lz.isEmpty())  {
     hilf.push(lz.top());
     lz.pop();
  }
  while (! hilf.isEmpty())   {
     lz.push(hilf.top());
     zahl2.push(hilf.top());
     hilf.pop();
  }

  int uebertrag = 0;
  int ziffer1, ziffer2, ziffer;
  IntegerElement e;

  while (! zahl2.isEmpty())   {
     ziffer1 = zahl1.top().getInhalt();
     ziffer2 = zahl2.top().getInhalt();
     if (ziffer1 >= ziffer2 + uebertrag)  {
        ziffer = ziffer1 - (ziffer2 + uebertrag);
        uebertrag = 0;
     }
     else  {
        ziffer = ziffer1 + 10 - (ziffer2 + uebertrag);
        uebertrag = 1;
```

```
      }
      zahl1.pop();
      zahl2.pop();
      e = new IntegerElement(ziffer);
      hilf.push(e);
   }

   while (! zahl1.isEmpty())   {
      ziffer1 = zahl1.top().getInhalt();
      if (ziffer1 >= uebertrag)  {
         ziffer = ziffer1 - uebertrag;
         uebertrag = 0;
      }
      else  {
          ziffer = ziffer1 + 10 - uebertrag;
          uebertrag = 1;
      }
      zahl1.pop();
      e = new IntegerElement(ziffer);
      hilf.push(e);
   }

   Langzahl ergebnis = new Langzahl();
   while (! hilf.isEmpty())  {
      ergebnis.push(hilf.top());
      hilf.pop();
   }
   ergebnis.eliminiereFuehrendeNullen();
   return ergebnis;
}  // Ende von Methode minus



private Langzahl multiplikation(int ziffer)   {

   Langzahl hilf = new Langzahl();
   Langzahl zahl1 = new Langzahl();

   while (! this.isEmpty())  {
      hilf.push(this.top());
      this.pop();
   }
   while (! hilf.isEmpty())  {
      this.push(hilf.top());
      zahl1.push(hilf.top());
      hilf.pop();
   }

   IntegerElement e, e2;
```

```
  int uebertrag = 0;
  int nummer;
  while (! zahl1.isEmpty())   {
     e = zahl1.top();
     nummer = (e.getInhalt() * ziffer + uebertrag) % 10;
     uebertrag = (e.getInhalt() * ziffer + uebertrag) / 10;
     zahl1.pop();

     e2 = new IntegerElement(nummer);
     hilf.push(e2);
  }

  if (uebertrag > 0)   {
     e2 = new IntegerElement(uebertrag);
     hilf.push(e2);
  }

  Langzahl ergebnis = new Langzahl();
  while (! hilf.isEmpty())  {
     ergebnis.push(hilf.top());
     hilf.pop();
  }

  return ergebnis;
 }   // Ende von Methode Multiplikation



public Langzahl mal(int faktor)   {

  Langzahl hilf = new Langzahl();
  Langzahl selfKopie = new Langzahl();
  Langzahl erg;
  while (! this.isEmpty())  {
     hilf.push(this.top());
     this.pop();
  }
  while (! hilf.isEmpty())  {
     this.push(hilf.top());
     selfKopie.push(hilf.top());
     hilf.pop();
  }

  erg = new Langzahl();
  IntegerElement e = new IntegerElement(0);
  erg.push(e);
  int rest;

  while (faktor > 0)   {
```

```
      rest = faktor % 10;
      faktor = faktor / 10;
      hilf = selfKopie.multiplikation(rest);
      erg = erg.plus(hilf);
      if (faktor != 0)   {
         e = new IntegerElement(0);
         selfKopie.push(e);
      }
   }

   return erg;
}   // Ende der Methode mal



public Langzahl hoch(int exponent)   {
   Langzahl hilf = new Langzahl();
   Langzahl selfKopie = new Langzahl();
   Langzahl erg;
   while (! this.isEmpty())  {
      hilf.push(this.top());
      this.pop();
   }
   while (! hilf.isEmpty())  {
      this.push(hilf.top());
      selfKopie.push(hilf.top());
      hilf.pop();
   }

   erg = new Langzahl();
   if (exponent == 0)   {
      IntegerElement e = new IntegerElement(1);
      erg.push(e);
   }
   else  {
      erg = selfKopie;
      int intZahl = Integer.parseInt(selfKopie.toStr());
      for (int i = 2; i <= exponent; i++) erg=erg.mal(intZahl);
   }

   return erg;

}   // Ende der Methode hoch
```

```java
private int vergleich(Langzahl y)  {
  int ergebnis;
  this.eliminiereFuehrendeNullen();
  y.eliminiereFuehrendeNullen();
  if (this.stellenanzahl() > y.stellenanzahl()) ergebnis = 1;
  else if (y.stellenanzahl() > this.stellenanzahl()) ergebnis = 2;
       else  {
          Langzahl hilf = new Langzahl();
          Langzahl ichrueckwaerts = new Langzahl();
          while (! this.isEmpty())  {
             hilf.push(this.top());
             ichrueckwaerts.push(this.top());
             this.pop();
          }
          while (! hilf.isEmpty())   {
             this.push(hilf.top());
             hilf.pop();
          }
          Langzahl zahlyrueckwaerts = new Langzahl();
          while (! y.isEmpty())  {
             hilf.push(y.top());
             zahlyrueckwaerts.push(y.top());
             y.pop();
          }
          while (! hilf.isEmpty())   {
             y.push(hilf.top());
             hilf.pop();
          }

          ergebnis = 0;
          int ziffer1, ziffer2;
          while (!ichrueckwaerts.isEmpty() && (ergebnis==0)) {
             ziffer1 = ichrueckwaerts.top().getInhalt();
             ziffer2 = zahlyrueckwaerts.top().getInhalt();
             if (ziffer1 > ziffer2) ergebnis = 1;
             else if (ziffer2 > ziffer1) ergebnis = 2;
             ichrueckwaerts.pop();
             zahlyrueckwaerts.pop();
          }
       }

  return ergebnis;
}



public boolean isGreater(Langzahl lz) {
   return (vergleich(lz) == 1);
}
```

```
public boolean isLess(Langzahl lz) {
   return (vergleich(lz) == 2);
}


public boolean isEqual(Langzahl lz) {
   return (vergleich(lz) == 0);
}
}
```

Nun folgt das Hauptprogramm zum Rechnen mit Langzahlen:

```
public class Langzahlprogramm extends javax.swing.JFrame {

  Langzahl zahl1, zahl2, ergebnis;

  public Langzahlprogramm() {
    initComponents();

    ergebnis = new Langzahl();
    tf1.setText("");
    tf2.setText("");
    taErgebnis.setText("");
  }


  private void btBerechneMouseClicked(java.awt…..) {
    int faktor, exponent;

    taErgebnis.setText("");
    zahl1 = new Langzahl(tf1.getText());

    if (rbPlus.isSelected())   {
      zahl2 = new Langzahl(tf2.getText());
      ergebnis = zahl1.plus(zahl2);
      taErgebnis.setText(ergebnis.toStr());
    }
    else if (rbMinus.isSelected())   {
            zahl2 = new Langzahl(tf2.getText());
            ergebnis = zahl1.minus(zahl2);
            taErgebnis.setText(ergebnis.toStr());
         }
         else if (rbMal.isSelected())   {
                faktor = Integer.parseInt(tf2.getText());
                ergebnis = zahl1.mal(faktor);
                taErgebnis.setText(ergebnis.toStr());
              }
              else if (rbHoch.isSelected())   {
                     exponent = Integer.parseInt(tf2.getText());
                     ergebnis = zahl1.hoch(exponent);
                     taErgebnis.setText(ergebnis.toStr());
                   }
```

```
        else if (rbSonstiges.isSelected())    {
                zahl2 = new Langzahl(tf2.getText());
                if (zahl1.isGreater(zahl2))
                   taErgebnis.setText("Zahl1 ist größer");
                else if (zahl1.isLess(zahl2))
                        taErgebnis.setText("Zahl1 ist kleiner");
                     else taErgebnis.setText("Zahl1 = Zahl2");
             }
  }
```

Aufgaben

1. Berechne die Fakultät einer natürlichen Zahl n als Langzahl! Die Zahl n sei vom Typ ***int***, deren Fakultät sei eine Langzahl.

2. Die **Fibonacci-Folge** ist eine unendliche Folge von Zahlen (den Fibonacci-Zahlen), bei der sich die jeweils folgende Zahl durch Addition ihrer beiden vorherigen Zahlen ergibt: 0, 1, 1, 2, 3, 5, 8, 13, … Benannt ist sie nach *Leonardo Fibonacci*, der damit 1202 das Wachstum einer Kaninchen-population beschrieb. Die Fibonacci-Folge $f_0, f_1, f_2, \ldots$ist durch das rekursive Bildungsgesetz $f_n = f_{n-1} + f_{n-2}$ für $n \geq 2$ mit den Anfangswerten $f_0 = 0$ und $f_1 = 1$ definiert.

Erstelle eine Liste der ersten 200 Fibonacci-Zahlen!

3. Erstelle eine Liste der ersten 300 Zweierpotenzen, also:
$2^0 = 1$
$2^1 = 2$
$2^2 = 4$
……
$2^{300} =$ …….

4. Der Rechner soll diejenige kleinste Zahl n ausgeben, für welche gilt: 2^n hat mehr als 1000 Dezimalstellen.
Hinweis: Achte darauf, dass dein Algorithmus nicht zu langsam ist! Es gibt hier schnelle und sehr langsame Lösungswege. Vielleicht ersetzt du die Zahl 1000 zu Testzwecken erst einmal durch die Zahl 100.

Lösungen

Aufgabe 1 (Fakultät)

```
...........
int hilf = Integer.parseInt(tf1.getText());
ergebnis = new Langzahl("1");
for (int i = 1; i <= hilf; i++) ergebnis = ergebnis.mal(i);
taErgebnis.setText(ergebnis.toStr());
```

Aufgabe 2 (Fibonacci-Folge)

```
..............
Langzahl f3;
int hilf = Integer.parseInt(tf1.getText());
Langzahl f1 = new Langzahl("0");
taErgebnis.setText("1: " + f1.toStr() + "\n");
Langzahl f2 = new Langzahl("1");
taErgebnis.append("2: " + f2.toStr() + "\n");

for (int i = 3; i <= hilf; i++)  {
   f3 = f2.plus(f1);
   taErgebnis.append(i + ": " + f3.toStr() + "\n");
   f1 = f2;
   f2 = f3;
}
```

Die Klasse *VZLangzahl*

Eine Klasse ***VZLangzahl***, welche beliebig große, ganze (also auch negative) Zahlen enthalten soll, ist relativ leicht zu erstellen, wenn man auf die Klasse *Langzahl* zurückgreifen kann.

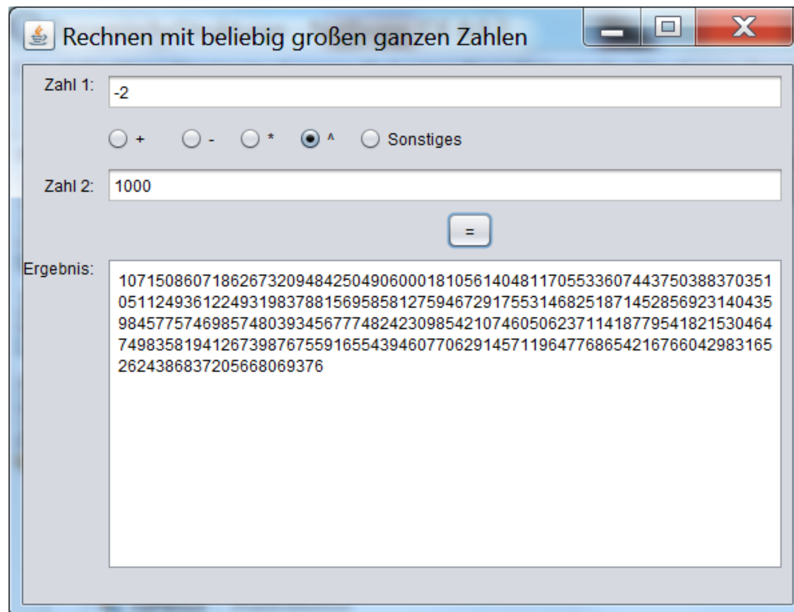
VZLangzahl
- Langzahl betrag - int vz
+ ! init() + ! init(String s) + ? String toStr() + ! eliminiereFuehrendeNullen() + ? VZLangzahl plus(VZLangzahl vzlz) + ? VZLangzahl minus(VZLangzahl vzlz) + ? int stellenanzahl() + ? boolean isGreater(VZLangzahl vzlz) + ? boolean isLess(VZLangzahl vzlz) + ? boolean isEqual(VZLangzahl vzlz) + ? VZLangzahl mal(int faktor) + ? VZLangzahl hoch(int exponent)

```
public class VZLangzahl {

 int vz; // enthält den Wert 1 oder -1
 Langzahl betrag;


 public VZLangzahl()   {
   vz = 1;
   betrag = new Langzahl();
 }


 public VZLangzahl(String s)  {
   if (s.charAt(0) == '-') {
      vz = -1;
      s = s.substring(1);
   }
```

```
   else if (s.charAt(0) == '+') {
          vz = 1;
          s = s.substring(1);
        }
        else vz = 1;
        betrag = new Langzahl(s);
 }



public String toStr()  {
  String s = "";
  if (vz == -1) s = "-";
  s = s + betrag.toStr();
  return s;
}



public void eliminiereFuehrendeNullen()  {
  betrag.eliminiereFuehrendeNullen();
}



public VZLangzahl plus(VZLangzahl vzlz)  {
  VZLangzahl ergebnis = new VZLangzahl();
  if (this.vz == vzlz.vz) {
     ergebnis.betrag = this.betrag.plus(vzlz.betrag);
     ergebnis.vz = this.vz;
  }
  else if (this.betrag.isGreater(vzlz.betrag))   {
          ergebnis.betrag = this.betrag.minus(vzlz.betrag);
          ergebnis.vz = this.vz;
       }
       else {
          ergebnis.betrag = vzlz.betrag.minus(this.betrag);
          ergebnis.vz = vzlz.vz;
       }
  return ergebnis;
}



public VZLangzahl minus(VZLangzahl vzlz)  {
   VZLangzahl hilf = vzlz.mal(-1);
   return this.plus(hilf);
}
```

```
public int stellenanzahl()  {
  return this.betrag.stellenanzahl();
}



public boolean isGreater(VZLangzahl vzlz)  {
   if (this.vz == 1 && vzlz.vz == 1)
      return this.betrag.isGreater(vzlz.betrag);
   else if (this.vz == 1 && vzlz.vz == -1)  return true;
        else if (this.vz == -1 && vzlz.vz == 1)  return false;
             else return this.betrag.isLess(vzlz.betrag);
}



public boolean isLess(VZLangzahl vzlz)  {
   return vzlz.betrag.isGreater(this.betrag);
}



public boolean isEqual(VZLangzahl vzlz)  {
   return (! this.isGreater(vzlz) && ! this.isLess(vzlz));
}



public VZLangzahl mal(int faktor)  {
   VZLangzahl ergebnis = new VZLangzahl();
   int vorzeichen = 1;
   if (faktor < 0) vorzeichen = -1;
   ergebnis.vz = this.vz * vorzeichen;
   ergebnis.betrag = this.betrag.mal(Math.abs(faktor));
   return ergebnis;
}



public VZLangzahl hoch(int exponent)  {
   VZLangzahl ergebnis = new VZLangzahl();
   ergebnis.betrag = this.betrag.hoch(exponent);
   if (exponent % 2 == 0) ergebnis.vz = 1;
   else ergebnis.vz = this.vz;
   return ergebnis;
}

}   // Ende der Klasse VZLangzahl
```

Aufgabe:
Erstelle eine Dokumentation für die Klasse *VZLangzahl*




```
public class VZLangzahlProgrammNeu extends javax.swing.JFrame {

  VZLangzahl zahl1, zahl2, ergebnis;

  public VZLangzahlProgrammNeu() {
    initComponents();

    ergebnis = new VZLangzahl();
    tf1.setText("");
    tf2.setText("");
  }


private void btBerechneMouseClicked(java.awt…) {
  int faktor, exponent;

  taErgebnis.setText("");
  zahl1 = new VZLangzahl(tf1.getText());

  if (rbPlus.isSelected())   {
    zahl2 = new VZLangzahl(tf2.getText());
    ergebnis = zahl1.plus(zahl2);
```

```
      taErgebnis.setText(ergebnis.toStr());
    }
    else if (rbMinus.isSelected())    {
           zahl2 = new VZLangzahl(tf2.getText());
           ergebnis = zahl1.minus(zahl2);
         taErgebnis.setText(ergebnis.toStr());
       }
       else if (rbMal.isSelected())    {
              faktor = Integer.parseInt(tf2.getText());
              ergebnis = zahl1.mal(faktor);
              taErgebnis.setText(ergebnis.toStr());
            }
            else if (rbHoch.isSelected())    {
                   exponent = Integer.parseInt(tf2.getText());
                   ergebnis = zahl1.hoch(exponent);
                   taErgebnis.setText(ergebnis.toStr());
                 }
                 else if (rbSonstiges.isSelected()) {
                        zahl2 = new VZLangzahl(tf2.getText());
                        if (zahl1.isGreater(zahl2))
                          taErgebnis.setText("Zahl1 ist größer");
                        else if (zahl1.isLess(zahl2))
                              taErgebnis.setText("Zahl1 ist kleiner");
                             else
                               taErgebnis.setText("Zahl1 =  Zahl2");
                      }
    }
```

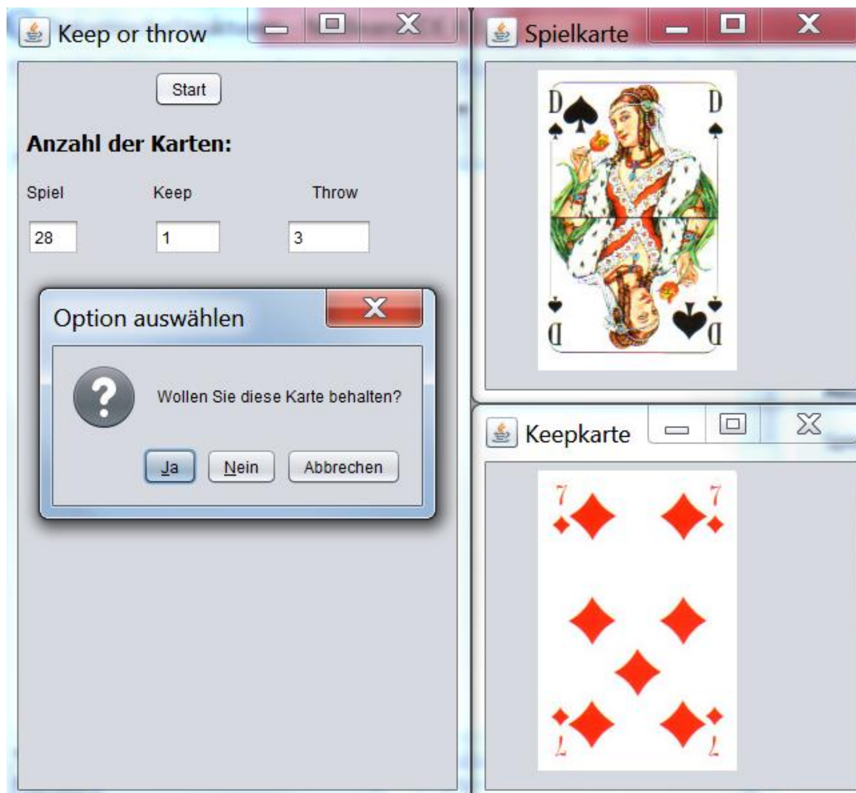
Für die Erstellung der folgenden Klasse ***LangDezimalBruch*** benötigt man wesentlich mehr Zeit und Arbeit. Deren Objekte sind Dezimalzahlen mit genau einer Stelle vor dem Komma und einer vorher festgelegten Anzahl von Stellen hinter dem Komma. Erstelle diese Klasse und löse damit die folgenden Aufgaben:

1. Berechne den Kehrwert einer natürlichen Zahl.

2. Berechne die Kreiszahl π. Es gilt: $\frac{\pi}{4} = 1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \frac{1}{9} - + \cdots$

3. Berechne die Eulersche Zahl e. Es gilt: $e = 1 + \frac{1}{1!} + \frac{1}{2!} + \frac{1}{3!} + \frac{1}{4!} + \cdots$

Keep or Throw



Keep or Throw ist ein (noch mäßig bekanntes) Kartenspiel für einsame Stunden. Es wird nach folgenden Regeln gespielt:
Gegeben ist ein durchmischter *Spielkartenstapel*. Der Spieler nimmt sich die oberste Karte und entscheidet, ob er die Karte behält (keep) oder wegwirft (throw). Behält er die Karte, wird sie auf einem *Keep-Stapel* abgelegt, ansonsten auf dem *Throw-Stapel*. Ziel ist es, möglichst viele Karten zu behalten. Dabei muss aber folgende Bedingung beachtet werden: Die Karten im *Keep-Stapel* müssen in aufsteigender Reihenfolge vorliegen. Die größte Karte liegt oben auf dem *Keep-Stapel*. Die Größe der Karten ist dabei folgendermaßen festgelegt:

Kreuz-As > Kreuz-König > … > Kreuz-7 > Pik-As > … > Pik-7 > … > Karo-7

Äußerst ungeschickt wäre es, direkt am Anfang eine "hohe" Karte wie Kreuz-König zu behalten. Die meisten der folgenden Karten müssten dann auf den *Throw-Stapel* gelegt werden.

Implementiere ein möglichst vollständiges Programm zur Realisierung dieses Spiels! Das Programm teilt dem Spieler (immer ?) mit, welche Karte gerade oben auf dem *Kartenstapel* liegt und fragt den Spieler gegebenenfalls, ob

er diese Karte behalten will. Natürlich sollen keine überflüssigen Fragen gestellt werden. Am Schluss des Spieles wird dem Spieler die Anzahl der behaltenen Karten mitgeteilt.

Programmierhinweise:
Stelle die Spielkarten einfach durch die Zahlen 1 bis 32 (aus der Klasse *IntegerElement*) dar! Den Zusammenhang mit den entsprechenden Bildern kann man einfach herstellen, weil alle Bilddateien die Namen *s1.jpg* bis *s32.jpg* haben.
Alle Bilder sind in einem Unterordner namens *Kartenspiel* gespeichert, welcher sich in dem Ordner des Projektes befinden soll.

Zeige die Kartenbilder in eigenen Formblättern an! Eine Darstellung im Haupt-Formblatt bringt Probleme, weil aufgrund der Dialog-Abfragefenster diese Bilder immer wieder gelöscht werden.

Lösung

```
import javax.swing.*;    //wegen Dialogfenster
import java.awt.*;       // für die Grafiken

public class KeepOrThrow extends javax.swing.JFrame {

  Stack<IntegerElement> Keep, Throw, Spiel;
  // das kleingeschriebene throw wäre eine Java-Anweisung
  int anzahlSpiel, anzahlKeep, anzahlThrow;
  javax.swing.JFrame Spielplan, Keepplan;
  Graphics zeichenagentSpiel, zeichenagentKeep;


  public KeepOrThrow() {
    initComponents();
    Spielplan = new javax.swing.JFrame();
    Spielplan.setBounds(this.getWidth(),0,300,300);
    Spielplan.setTitle("Spielkarte");
    Spielplan.setVisible(true);
    zeichenagentSpiel = Spielplan.getGraphics();

  Keepplan = new javax.swing.JFrame();
  Keepplan.setBounds(this.getWidth(),Spielplan.getHeight(),
                                                    300,300);
  Keepplan.setTitle("Keepkarte");
  Keepplan.setVisible(true);
  zeichenagentKeep = Keepplan.getGraphics();
}



  private void btStartMouseClicked(java.awt….)  {
    Spiel = new Stack<>();
    Keep = new Stack<>();
    Throw = new Stack<>();
    anzahlSpiel = 32;
    anzahlKeep = 0;
    anzahlThrow = 0;

    tfAnzahlSpiel.setText("32");
```

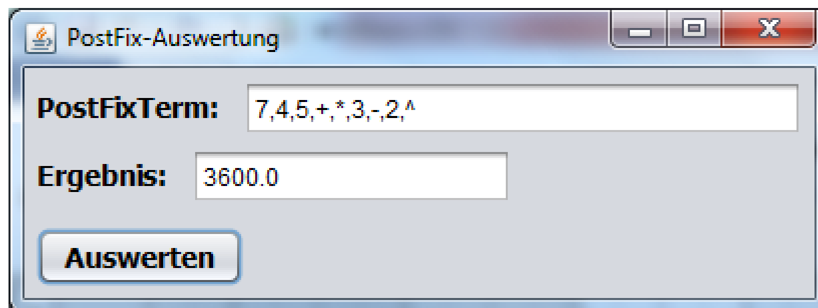
```
  tfAnzahlKeep.setText("0");
  tfAnzahlThrow.setText("0");

  IntegerElement e;
  for (int i=1; i<=32;i++) {
     e = new IntegerElement(i);
     Spiel.push(e);
  }
  mischen();
  spielen();
}



private void mischen()  {
   IntegerElement e;
   int zufall;
   Stack<IntegerElement> hilfA = new Stack<>();
   Stack<IntegerElement> hilfB = new Stack<>();
   Stack<IntegerElement> hilfC = new Stack<>();
   Stack<IntegerElement> hilfD = new Stack<>();

   for (int i=1; i<10; i++)  {
      while (! Spiel.isEmpty())  {
         e = Spiel.top();
         Spiel.pop();
         zufall = (int) (Math.random()*4);
         switch(zufall)  {
            case 0: hilfA.push(e); break;
            case 1: hilfB.push(e); break;
            case 2: hilfC.push(e); break;
            case 3: hilfA.push(e); break;
            default: break;
         }
      }  // End of while

       while (! hilfA.isEmpty())   {
          e = hilfA.top();
          hilfA.pop();
          Spiel.push(e);
       }
       while (! hilfB.isEmpty())   {
          e = hilfB.top();
          hilfB.pop();
```

```
            Spiel.push(e);
        }
        while (! hilfC.isEmpty())    {
            e = hilfC.top();
            hilfC.pop();
            Spiel.push(e);
        }
        while (! hilfD.isEmpty())    {
            e = hilfD.top();
            hilfD.pop();
            Spiel.push(e);
        }
    } // of for

}



void spielen()  {
    IntegerElement e;
    int blatt;

    while (! Spiel.isEmpty())  {
        e= Spiel.top();
        Spiel.pop();
        blatt = e.getInhalt();
        zeigeSpiel(blatt);

        if (! Keep.isEmpty() && e.isLess(Keep.top()))  {
            Throw.push(e);
            anzahlThrow++;
            tfAnzahlThrow.setText(anzahlThrow +"");
        }
        else {
            int wahl =0;
            wahl = JOptionPane.showConfirmDialog(this ,
                             "Wollen Sie diese Karte behalten?");
            if (wahl == 0)  {
                Keep.push(e);
                zeigeKeep(blatt);
                anzahlKeep++;
                tfAnzahlKeep.setText(anzahlKeep +"");
            }
            else  {
```

```
                Throw.push(e);
                anzahlThrow++;
                tfAnzahlThrow.setText(anzahlThrow +"");
            }
        }  // Ende von else

        anzahlSpiel--;
        tfAnzahlSpiel.setText(anzahlSpiel +"");
    }  // End von while

    zeichenagentSpiel.setColor(Color.WHITE);
    zeichenagentSpiel.fillRect(0,0,300,300);
    // Bild der letzten Karte gelöscht
}



private void zeigeSpiel(int nummer)  {
    String name ="Kartenspiel/s" + nummer +".jpg";
    // alle Bilder sind in einem Unterordner namens Kartenspiel gespeichert
    // sie haben die Dateinamen s1.jpg bis s32.jpg
    Image bild = new ImageIcon(name).getImage();
    zeichenagentSpiel.drawImage(bild,50,50,null);
}


  private void zeigeKeep(int nummer)  {
     String name ="Kartenspiel/s" + nummer +".jpg";
     Image bild = new ImageIcon(name).getImage();
     zeichenagentKeep.drawImage(bild,50,50,null);
  }


}  // Ende des Programms
```

Umgekehrte polnische Notation

Auswertung mit Hilfe eines Stacks



Die polnische Notation wurde nach dem polnischen Mathematiker *Jan Lukasiewicz (1878 – 1956)* benannt. Ihre wichtigste Eigenschaft besteht darin, dass die Reihenfolge, in der die Operationen durchzuführen sind, vollständig durch die Reihenfolge der Operatoren und Operanden bestimmt ist und keinerlei Klammern oder Prioritätenregeln notwendig sind. *Lukasiewicz* schrieb den Operator vor den Operanden (in sog. *Präfixnotation*), also statt A+B wie in der üblichen sog. *Infixnotation* schrieb er +AB. Bei der umgekehrten polnischen Notation wird der Operator hinter die beiden Operanden geschrieben (deswegen auch *Postfixnotation* genannt): AB+
Der obige Beispielterm entspricht der *Infixnotation* $[7 \cdot (4 + 5) - 3]^2$

Bemerkung: Falls der ganze Postfixterm nicht nur aus einer einzigen Zahl besteht, ist das letzte Zeichen ein Operator.

Zum Lösungsalgorithmus: Als Trennzeichen fungiert das Komma. Als Endezeichen wird an den Term eine schließende Klammer angehängt.
Der Term wird nun von links nach rechts folgendermaßen abgearbeitet:

Wiederhole

Wird eine Zahl gefunden, so wird diese auf den Stack gelegt.

Wird ein Operator ⊗ gefunden, dann

- entnehme dem Stack die oberste Zahl A und die zweitoberste Zahl B,
- berechne B ⊗ A,
- lege das Ergebnis wieder auf den Stack.

bis das Endezeichen erreicht wird.
Gib als Endergebnis die einzige, noch auf dem Stack vorhandene Zahl aus.

Einschränkungen: Es werden nur binäre Operatoren zugelassen (also keine Vorzeichen). Das nachfolgende Programm beschränkt sich zunächst auf das Rechnen mit einstelligen Zahlen. Dadurch wird das Einlesen der Zahlen einfacher.

Hinweis: Im Programm sollten die Zahlen vom Typ *double* sein, weil Divisionen vorkommen dürfen und somit das Ergebnis auch keine natürliche Zahl mehr sein könnte.

```
public class PostFix extends javax.swing.JFrame {

 String ziffernmenge = "0123456789";
 String operatorenmenge = "+-*/^";
 Stack<RealElement> stapel;
 String term;

 public PostFix() {
   initComponents();
   stapel = new Stack<>();
   tfEingabe.setText(""); //wichtig: keine Leerzeichen am Anfang!
 }

 private void rechne(char ch)   {
   double ergebnis, zahl1, zahl2;
   RealElement e;

   e = stapel.top();
   zahl1 = e.getInhalt();
   stapel.pop();
   e = stapel.top();
   zahl2 = e.getInhalt();
   stapel.pop();
   ergebnis = 0;
   switch (ch)   {
     case '+': ergebnis = zahl1 + zahl2; break;
     case '-': ergebnis = zahl2 - zahl1; break;
     case '*': ergebnis = zahl1 * zahl2; break;
     case '/': ergebnis = zahl2 / zahl1; break;
     case '^': ergebnis = Math.pow(zahl2, zahl1);
   }
   e = new RealElement(ergebnis);
   stapel.push(e);
 }
```

```
private void btAuswertenMouseClicked(java…….) {
  char ch;
  RealElement e;
  term = tfEingabe.getText() + ')';

  do  {
    ch = term.charAt(0);

    if (ziffernmenge.indexOf(ch) > -1)  {
      e = new RealElement(Double.parseDouble(""+ch));
      stapel.push(e);
    }
    else if (operatorenmenge.indexOf(ch) > -1) rechne(ch);

    if (term.length() > 1) term = term.substring(1);
    // damit wurde das nächste Trennzeichen gelöscht
  }
  while (ch != ')');

  e = stapel.top();
  stapel.pop();
  tfErgebnis.setText(""+e.getInhalt());
}
```

Aufgaben

1. Erweitere das Programm so, dass im Eingabeterm nicht nur einzelne Ziffern, sondern mehrstellige, natürliche Zahlen stehen können. Dabei soll allerdings der positive Integer-Bereich nicht überschritten werden.
Hinweis: prüfe zunächst das erste Zeichen in der Variable *Term* im obigen Programm. Ist dieses Zeichen eine Ziffer, so bilden die ersten Zeichen von *Term* eine Zahl. Diese wird erst durch ein Komma oder durch das Endezeichen „)“ beendet.

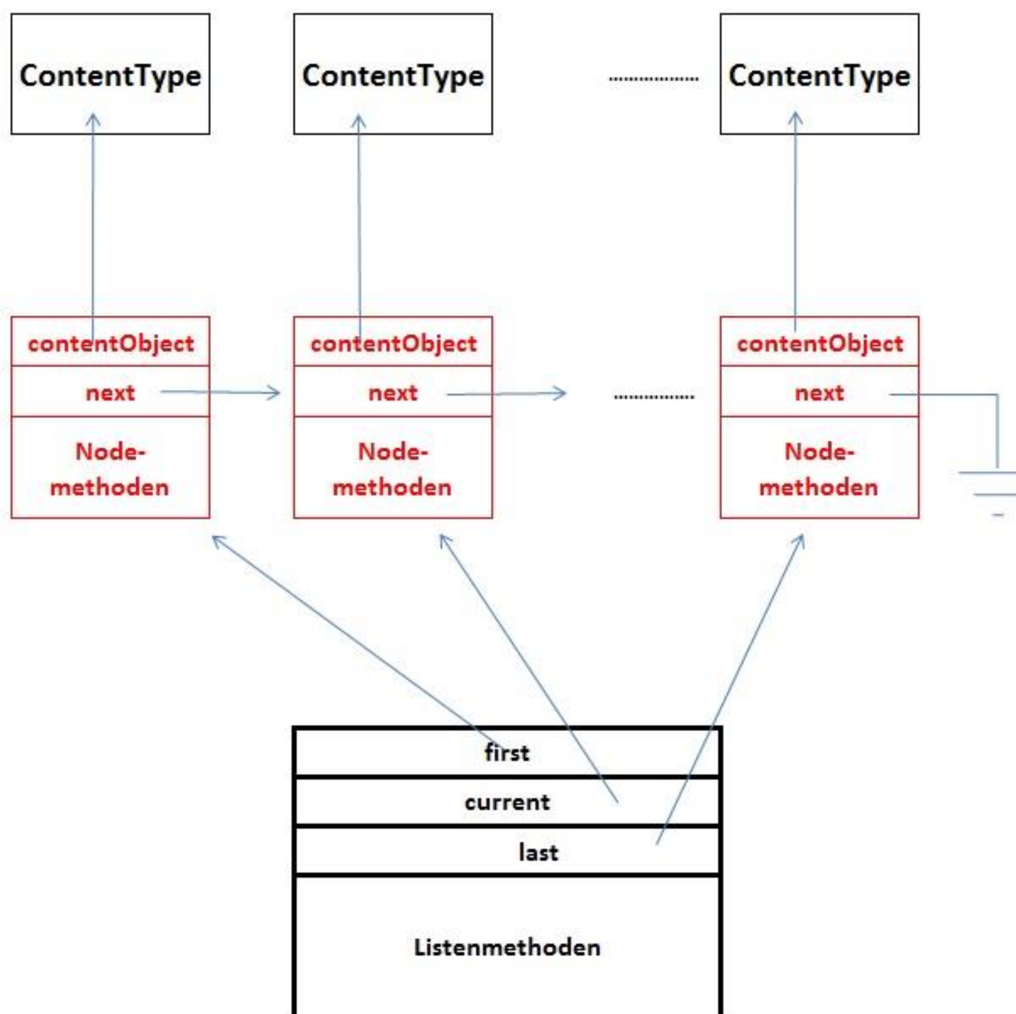
2. Der Eingabeterm soll nun auch ganze Zahlen (mit Vorzeichen) enthalten können.

Die Klasse *List*, generische Version

Eine lineare Liste ist ähnlich aufgebaut wie eine Schlange oder ein Stack. Der wesentliche Unterschied besteht darin, dass man bei einer Liste auch auf ein beliebiges Listenelement zugreifen kann. Es können auch Elemente mitten in der Liste gelöscht oder eingefügt werden.

Ein Objekt der Klasse *List* verwaltet beliebige Objekte nach einem Listenprinzip. Ein interner Positionszeiger wird durch die Listenstruktur bewegt, seine Position markiert ein aktuelles Objekt. Die Lage des Positionszeigers kann abgefragt, verändert und die Objektinhalte an den Positionen können gelesen oder verändert werden.

Es gibt mehrere gute Methoden, eine Liste zu implementieren. Für die zentralen Abiturklausuren in NRW ab dem Jahr 2017 wurde nachfolgend beschriebene Implementation eingeführt:



Generische Klasse List<ContentType>
Ein Objekt dieser generischen Klasse verwaltet beliebig viele linear angeordnete Objekte vom Typ ContentType. Auf höchstens eines davon, aktuellesObjekt genannt, kann jeweils zugegriffen werden.

Wenn eine Liste leer ist, vollständig durchlaufen wurde oder das aktuelle Objekt am Ende der Liste gelöscht wurde, gibt es kein aktuelles Objekt.

Das erste oder das letzte Objekt einer Liste können durch einen Auftrag zum aktuellen Objekt gemacht werden. Außerdem kann das dem aktuellen Objekt folgende Listenobjekt zum neuen aktuellen Objekt werden.

Das aktuelle Objekt kann gelesen, verändert oder gelöscht werden. Außerdem kann vor dem aktuellen Objekt ein Listenobjekt eingefügt werden.

Autor: Qualitäts- und UnterstützungsAgentur - Landesinstitut für Schule, Materialien zum schulinternen Lehrplan Informatik SII.
@version Generisch_06 2015-10-25

```
public class List<ContentType> {

/* --------- Anfang der privaten inneren Klasse -------------- */
  private class ListNode {

    private ContentType contentObject;
    private ListNode next;

    public ListNode(ContentType pContent) {
      contentObject = pContent;
      next = null;
    }

    public ContentType getContentObject() {
      return contentObject;
    }
```

```
public void setContentObject(ContentType pContent) {
   contentObject = pContent;
}
```

** Der Nachfolgeknoten wird zurückgeliefert.*

```
public ListNode getNextNode() {
   return this.next;
}
```

** Der Verweis wird auf das Knotenobjekt, das als Parameter übergeben wird, gesetzt.*

```
public void setNextNode(ListNode pNext) {
   this.next = pNext;
}
```

} */* ----------- Ende der privaten inneren Klasse -------------- */*

```
ListNode first;
ListNode last;
ListNode current;
```

```
public List() {
  first = null;
  last = null;
  current = null;
}
```

** Die Anfrage liefert den Wert true, wenn die Liste keine Objekte enthält, sonst liefert sie den Wert false.*

```
public boolean isEmpty() {
  // Die Liste ist leer, wenn es kein erstes Element gibt.
  return first == null;
}
```

** Die Anfrage liefert den Wert true, wenn es ein aktuelles Objekt gibt, sonst liefert sie den Wert false.*

```
public boolean hasAccess() {
  // Es gibt keinen Zugriff, wenn current auf kein Element verweist.
  return current != null;
}
```

** Falls die Liste nicht leer ist, es ein aktuelles Objekt gibt und dieses nicht das letzte Objekt der Liste ist, wird das dem aktuellen Objekt in der Liste folgende Objekt zum aktuellen Objekt, andernfalls gibt es nach Ausführung des Auftrags kein aktuelles Objekt, d.h. hasAccess() liefert den Wert false.*

```
public void next() {
  if (this.hasAccess()) {
    current = current.getNextNode();
  }
}
```

** Falls die Liste nicht leer ist, wird das erste Objekt der Liste aktuelles Objekt. Ist die Liste leer, geschieht nichts.*

```
  public void toFirst() {
    if (!isEmpty()) {
      current = first;
    }
  }
```

** Falls die Liste nicht leer ist, wird das letzte Objekt der Liste aktuelles Objekt. Ist die Liste leer, geschieht nichts.*

```
public void toLast() {
  if (!isEmpty()) {
    current = last;
  }
}
```

** Falls es ein aktuelles Objekt gibt (hasAccess() == true), wird das aktuelle Objekt zurückgegeben, andernfalls (hasAccess() == false) gibt die Anfrage den Wert null zurück.*

```
public ContentType getContent() {
  if (this.hasAccess()) return
                        current.getContentObject();
  else return null;
}
```

** Falls es ein aktuelles Objekt gibt (hasAccess() == true) und pContent ungleich null ist, wird das aktuelle Objekt durch pContent ersetzt. Sonst geschieht nichts.*

```
public void setContent(ContentType pContent) {
// Nichts tun, wenn es keinen Inhalt oder kein aktuelles Element gibt.
  if (pContent != null && this.hasAccess()) {
    current.setContentObject(pContent);
  }
}
```

** Falls es ein aktuelles Objekt gibt (hasAccess() == true), wird ein neues Objekt vor dem aktuellen Objekt in die Liste eingefügt. Das aktuelle Objekt bleibt unverändert.*
** Wenn die Liste leer ist, wird pContent in die Liste eingefügt und es gibt weiterhin kein aktuelles Objekt (hasAccess() == false).*
** Falls es kein aktuelles Objekt gibt (hasAccess() == false) und die Liste nicht leer ist oder pContent gleich null ist, geschieht nichts.*

```
public void insert(ContentType pContent) {
  if (pContent != null) { // Nichts tun, wenn es keinen Inhalt gibt.
    if (this.hasAccess()) { // Fall: Es gibt ein aktuelles Element.
      // Neuen Knoten erstellen.
      ListNode newNode = new ListNode(pContent);

      if (current != first) { // Fall: Nicht an 1. Stelle einfügen.
        ListNode previous =this.getPrevious(current);
        newNode.setNextNode(previous.getNextNode());
        previous.setNextNode(newNode);
      }
      else { // Fall: An 1. Stelle einfügen.
        newNode.setNextNode(first);
        first = newNode;
      }
    }   // end of   es gibt ein aktuelles Element
    else { //Fall: Es gibt kein aktuelles Element.
       if (this.isEmpty()) { // Fall: In leere Liste einfügen.
        // Neuen Knoten erstellen.
         ListNode newNode = new ListNode(pContent);
         first = newNode;
         last = newNode;
       }

    }
  }
}
```

** Falls pContent gleich null ist, geschieht nichts.*
** Ansonsten wird ein neues Objekt pContent am Ende der Liste eingefügt.*
** Das aktuelle Objekt bleibt unverändert.*
** Wenn die Liste leer ist, wird das Objekt pContent in die Liste eingefügt*
** und es gibt weiterhin kein aktuelles Objekt (hasAccess() == false).*

```
public void append(ContentType pContent) {
  if (pContent != null) { // Nichts tun, wenn es keine Inhalt gibt.
      if (this.isEmpty()) // Fall: An leere Liste anfügen.
        this.insert(pContent);
      else { // Fall: An nicht-leere Liste anfügen.
        // Neuen Knoten erstellen.
        ListNode newNode = new ListNode(pContent);
        last.setNextNode(newNode);
        last = newNode; // Letzten Knoten aktualisieren.
      }
  }
}
```

** Falls es sich bei der Liste und pList um dasselbe Objekt handelt, pList null oder eine leere Liste ist, geschieht nichts.*
** Ansonsten wird die Liste pList an die aktuelle Liste angehängt.*
** Anschliessend wird pList eine leere Liste. Das aktuelle Objekt bleibt unverändert. Insbesondere bleibt hasAccess identisch.*

```
  public void concat(List<ContentType> pList) {
    if (pList != this && pList != null &&
                                      !pList.isEmpty()) {
// Nichts tun, wenn pList und this identisch, pList leer oder nicht existent.

      if (this.isEmpty()) { // Fall: An leere Liste anfügen.
        this.first = pList.first;
        this.last = pList.last;
      } else { // Fall: An nicht-leere Liste anfügen.
        this.last.setNextNode(pList.first);
        this.last = pList.last;
      }
```

```
  // Liste pList löschen.
      pList.first = null;
      pList.last = null;
      pList.current = null;
    }
  }
```

* Wenn die Liste leer ist oder es kein aktuelles Objekt gibt (hasAccess() == false), geschieht nichts.
* Falls es ein aktuelles Objekt gibt (hasAccess() == true), wird das aktuelle Objekt gelöscht und das Objekt hinter dem gelöschten Objekt wird zum aktuellen Objekt.
* Wird das Objekt, das am Ende der Liste steht, gelöscht, gibt es kein aktuelles Objekt mehr.

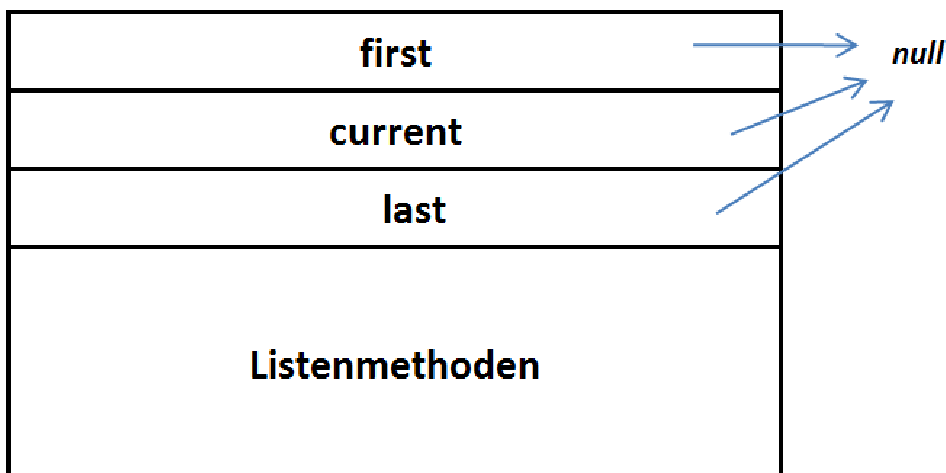
```
  public void remove() {
    // Nichts tun, wenn es kein aktuelles Element gibt oder die Liste leer ist.
    if (this.hasAccess() && !this.isEmpty()) {
      if (current == first)
        first = first.getNextNode();
      else {
        ListNode previous = this.getPrevious(current);
        if (current == last)  last = previous;
        previous.setNextNode(current.getNextNode());
      }

      ListNode temp = current.getNextNode();
      current.setContentObject(null);
      current.setNextNode(null);
      current = temp;

      //Beim Löschen des letzten Elements last auf null setzen.
      if (this.isEmpty())  last = null;
    }
  }
```

** Liefert den Vorgängerknoten des Knotens pNode. Ist die Liste leer, pNode == null, pNode nicht in der Liste oder pNode der erste Knoten der Liste, wird null zurückgegeben.*

```
  private ListNode getPrevious(ListNode pNode) {
    if (pNode != null && pNode != first &&
                                  !this.isEmpty()) {
      ListNode temp = first;
      while (temp != null &&
                        temp.getNextNode() != pNode) {
        temp = temp.getNextNode();
      }
      return temp;
    }
    else {
      return null;
    }
  }

}
```

Bei nichtleerer Liste zeigt der Zeiger *first* immer auf das erste Listenelement. Der interne Positionszeiger *current* hingegen kann auf ein beliebiges Element zeigen. Nach dem Durchlaufen der Liste zeigt *current* auf *null*.

Eine existierende, aber leere Liste sieht so aus:



Objekte der Klasse ***List<ContentType>*** verwalten beliebig viele, linear angeordnete Objekte. Auf höchstens ein Listenobjekt, aktuelles Objekt genannt, kann jeweils zugegriffen werden. Wenn eine Liste leer ist, vollständig durchlaufen wurde oder das aktuelle Objekt am Ende der Liste gelöscht wurde, gibt es kein aktuelles Objekt. Das erste oder das letzte Objekt einer Liste können durch einen Auftrag zum aktuellen Objekt gemacht werden. Außerdem kann das dem aktuellen Objekt folgende Listenobjekt zum neuen aktuellen Objekt werden.
Das aktuelle Objekt kann gelesen, verändert oder gelöscht werden. Außerdem kann vor dem aktuellen Objekt ein Listenobjekt eingefügt oder ein Listenobjekt an das Ende der Liste angefügt werden.

Dokumentation der Klasse List<ContentType>

Konstruktor **init**
Eine leere Liste wird erzeugt.

Anfrage **boolean isEmpty()**
Die Anfrage liefert den Wert *true*, wenn die Liste keine Objekte enthält, sonst liefert sie den Wert *false*.

Anfrage **boolean hasAccess()**
Die Anfrage liefert den Wert *true*, wenn es ein aktuelles Objekt gibt, sonst liefert sie den Wert *false*.

Auftrag **void next()**
Falls die Liste nicht leer ist, es ein aktuelles Objekt gibt und dieses nicht das letzte Objekt der Liste ist, wird das dem aktuellen Objekt in der Liste folgende Objekt zum aktuellen Objekt, andernfalls gibt es nach Ausführung des Auftrags kein aktuelles Objekt, d.h. *hasAccess()* liefert den Wert *false*.

Auftrag **void toFirst()**
Falls die Liste nicht leer ist, wird das erste Objekt der Liste aktuelles Objekt. Ist die Liste leer, geschieht nichts.

Auftrag **void toLast()**
Falls die Liste nicht leer ist, wird das letzte Objekt der Liste aktuelles Objekt. Ist die Liste leer, geschieht nichts.

Anfrage **Object getContent()**
Falls es ein aktuelles Objekt gibt (*hasAccess() == true*), wird das aktuelle Objekt zurückgegeben, andernfalls gibt die Anfrage den Wert *null* zurück.

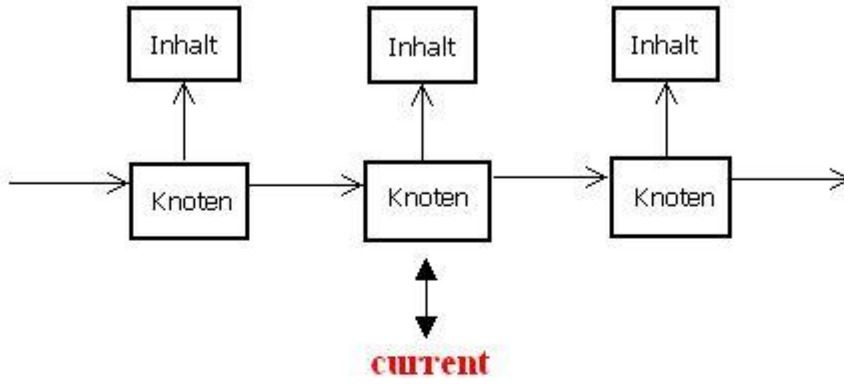
Auftrag **void setContent(ContentType p)**
Falls es ein aktuelles Objekt gibt (*hasAccess() == true*) und p ungleich *null* ist, wird das aktuelle Objekt durch p ersetzt. Sonst bleibt die Liste unverändert.

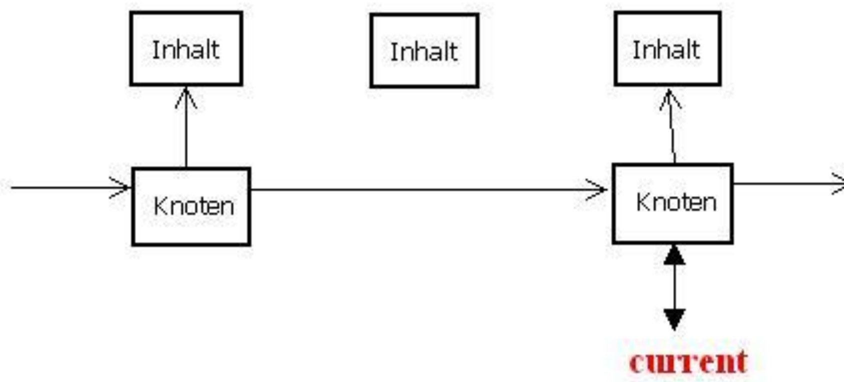
Auftrag **void append(ContentType p)**
Ein neues Objekt p wird am Ende der Liste eingefügt. Das aktuelle Objekt bleibt unverändert. Wenn die Liste leer ist, wird das Objekt p in die Liste eingefügt und es gibt weiterhin kein aktuelles Objekt (*hasAccess() == false*). Falls p gleich *null* ist, bleibt die Liste unverändert.

Auftrag **void insert(Object p)**
Falls es ein aktuelles Objekt gibt (hasAccess == true), wird ein neues Objekt vor dem aktuellen Objekt in die Liste eingefügt. Das aktuelle Objekt bleibt unverändert. Falls die Liste leer ist und es somit kein aktuelles Objekt gibt (*hasAccess == false*), wird p in die Liste eingefügt und es gibt weiterhin kein aktuelles Objekt. Falls es kein aktuelles Objekt gibt (*hasAccess == false*) und die Liste nicht leer ist oder p gleich *null* ist, bleibt die Liste unverändert.

Auftrag **concat(List<ContentType> pList)**
Falls es sich bei der Liste und pList um dasselbe Objekt handelt, pList *null* oder eine leere Liste ist, geschieht nichts. Ansonsten wird die Liste pList an die aktuelle Liste angehängt. Anschliessend wird pList eine leere Liste. Das aktuelle Objekt bleibt unverändert. Insbesondere bleibt hasAccess identisch.

Auftrag **void remove()**
Falls es ein aktuelles Objekt gibt (*hasAccess == true*), wird das aktuelle Objekt aus der Liste gelöscht und das Objekt hinter dem gelöschten Objekt wird zum aktuellen Objekt.
Wird das Objekt, das am Ende der Liste steht, gelöscht, gibt es kein aktuelles Objekt mehr (*hasAccess == false*).
Wenn die Liste leer ist oder es kein aktuelles Objekt gibt (*hasAccess == false*), bleibt die Liste unverändert.


Vor dem Löschen
Inhalt
Inhalt
Inhalt
Knoten
Knoten
Knoten
current


Nach dem Löschen
Inhalt
Inhalt
Inhalt
Knoten
Knoten
current

Aufgaben

Die nachfolgenden Aufgaben sollen nur mit den öffentlichen Listenmethoden gelöst werden! Es sollen keine weiteren Methoden implementiert werden!

1. Stelle von einer Liste A eine unabhängige Kopie B her! A und B sollen also nicht nur zwei unterschiedliche Namen für dieselbe Liste sein! Wenn man eine der beiden Listen löscht, soll die andere weiterhin existieren!

2. Schreibe das Programm für die im Folgenden abgebildete Anwendung!

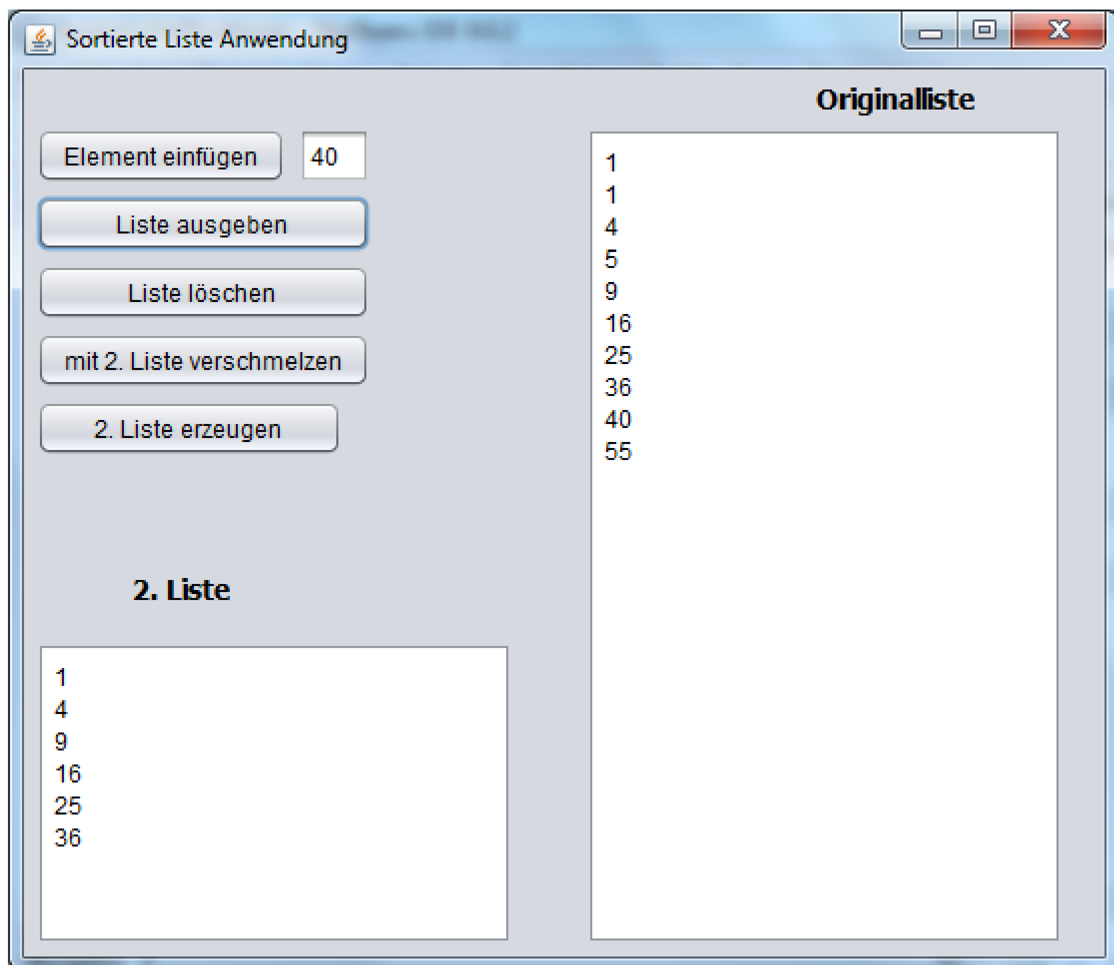
3. Geschwindigkeitsmessung
Erzeuge ein Array mit 10 000 Zufallszahlen. Kopiere dieses Array in eine lineare Liste! Sortiere nun jeweils und vergleiche die benötigten Zeiten!

4. Implementiere eine generische Klasse namens *QueueNeu<ContentType>* derart, dass sie als privates Attribut ein Objekt vom Typ *List<ContentType>* besitzt!

5. Implementiere eine generische Klasse namens *StackNeu<ContentType>* derart, dass sie als privates Attribut ein Objekt vom Typ *List<ContentType>* hat!

6. Implementiere eine Klasse *SortierteListe<ContentType>* derart, dass sie als privates Attribut ein Objekt vom Typ *List<ContentType>* hat!
Die alten Methoden *setObject* und *append* gibt es nicht mehr und *insert* wird neu angepasst. Die Methode *concat* wird ersetzt durch die neue Methode *meltWith(SortierteListe p)*. Diese Klasse *SortierteListe<ContentType>* soll Objekte der Klasse *Item* verwalten können.

Teste deine Implementation mit folgendem Hauptprogramm:



7. Implementiere die Klasse *QueueNeu2<ContentType>* als Unterklasse von *List<ContentType>* !

8. Implementiere die Klasse *StackNeu2<ContentType>* als Unterklasse von *List<ContentType>* !

9. Implementiere wie in Aufgabe 6 eine Klasse *SortierteListeNeu2<ContentType>,* diesmal aber als direkte Unterklasse von *List<ContentType>* !

Sehr problematisch an den letzten drei Aufgaben 7, 8 und 9 ist Folgendes:
In der Unterklasse sollten diejenigen Methoden der Oberklasse, welche nicht mehr dem Anwender zur Verfügung stehen sollen, deaktiviert werden. Dies kann dadurch geschehen, dass man sie einfach überschreibt mit einem Quellcode, der nichts bewirkt. Unschön dabei ist allerdings, dass der Anwender diese Methoden durchaus aufrufen kann, sie bewirken nur nichts und, was schlimmer ist, dem Anwender wird dies nicht mitgeteilt.
Außerdem muss man darauf achten, dass diese Methoden in der Oberklasse nicht innerhalb irgendwelcher anderen Methoden benötigt werden. Denn dann würden auch diese anderen Methoden nicht funktionieren, wenn sie von einem Objekt der Unterklasse aufgerufen werden.

10. Was bewirkt das folgende Programmstück? Gehe davon aus, dass die Variable *liste* ein paar natürliche Zahlen enthält!

```
private void ausgabe()  {
  if (liste.hasAccess())  {
    IntegerElement e = liste.getContent();
    int zahl = e.getInhalt();
    liste.next();
    ausgabe();
    taAusgabe.append(zahl + "\n");
  }
}
```

Lösungen

Aufgabe 2

```
import javax.swing.*;

public class ListenAnwendung extends javax.swing.JFrame {

  List<IntegerElement> liste;

  public ListenAnwendung() {
    initComponents();
  }



  private void btZufallslisteErzeugenMouseClicked(java......)   {
    IntegerElement e;
    if (liste == null) liste = new List<>();
    for (int i = 1; i <= 7; i++)   {
      e = new IntegerElement((int) (100*Math.random()));
      liste.append(e);
    }
  }



  private void btListeAusgebenMouseClicked(java......)   {
    IntegerElement e;
    taAusgabe.setText("");
    if (liste != null)  {
      liste.toFirst();
      while (liste.hasAccess())  {
         e = liste.getContent();
         taAusgabe.append(e.getInhalt()+"\n");
         liste.next();
      }
    }
  }


  private void btListeLoeschenMouseClicked(java......)   {
     liste = null;
  }
```

```
private void btListeSortierenMouseClicked(java......)   {
  List<IntegerElement> hilfliste;
  IntegerElement e;

  if (liste == null)
     JOptionPane.showMessageDialog(this, "Liste ist leer");
  else   {
    hilfliste = new List<>();
    liste.toFirst();
    while (liste.hasAccess())  {
       e = liste.getContent();
       if (hilfliste.isEmpty())  hilfliste.append(e);
       else  {
         hilfliste.toFirst();
         while (hilfliste.hasAccess() &&
                                   (e.isGreater(hilfliste.getContent())))
           hilfliste.next();
         if (hilfliste.hasAccess())  hilfliste.insert(e);
         else hilfliste.append(e);
       }
       liste.remove();
    } // of while
    liste = hilfliste;
  } // of else
}


private void btVorneEinfuegenMouseClicked(java......)   {
  IntegerElement e;
  int zahl = Integer.parseInt(tfVorneEinfuegen.getText());
  e = new IntegerElement(zahl);
  liste.toFirst();
  liste.insert(e);
}


private void btHintenAnhaengenMouseClicked(java......)   {
  IntegerElement e;
  int zahl = Integer.parseInt(tfHintenAnhaengen.getText());
  e = new IntegerElement(zahl);
  liste.append(e);
}
```

```
private void btElementeanzahlMouseClicked(java……)    {
  int count = 0;

  if (liste != null)   {
    liste.toFirst();
    while (liste.hasAccess())   {
       count++;
       liste.next();
    }
  }
  tfElementeanzahl.setText(""+ count);
}



private void btSuchenMouseClicked(java……)   {
  IntegerElement e;
  int zahl = Integer.parseInt(tfSuchen.getText());
  boolean gefunden;

  if (liste != null)   {
    liste.toFirst();
    gefunden = false;
    while (liste.hasAccess() && !gefunden)  {
       e = liste.getContent();
       if (e.getInhalt() == zahl)  gefunden = true;
       liste.next();
    }
    if (gefunden) JOptionPane.showMessageDialog(this, "Die
              Zahl " + tfSuchen.getText() + " ist enthalten");
    else JOptionPane.showMessageDialog(this, "Die Zahl " +
                 tfSuchen.getText() + " ist nicht enthalten");
  }
}
```

```
private void btLoeschenMouseClicked(java……)   {
  IntegerElement e;
  int zahl = Integer.parseInt(tfLoeschen.getText());

  if (liste != null)  {
     liste.toFirst();
     while (liste.hasAccess())  {
        e = liste.getContent();
        if (e.getInhalt() == zahl)  liste.remove();
        liste.next();
     }
  }
}



private void btNteZahlMouseClicked(java……)   {
  IntegerElement e;
  if (liste == null)
    JOptionPane.showMessageDialog(this, "Die Liste ist leer");
  else   {
    int zahl = Integer.parseInt(tfNteZahlEingabe.getText());
    liste.toFirst();
    //Vorsicht: zahl <= Elementeanzahl?
    for (int i = 1; i< zahl; i++)  liste.next();
    e = liste.getContent();
    tfNteZahlAusgabe.setText(""+e.getInhalt());
  }
}
```

Aufgabe 4

```
public class QueueNeu<ContentType> {

  private  List<ContentType> hatList;

  public QueueNeu()   {
     hatList = new List<>();
  }


  public void enqueue(ContentType p)  {
      hatList.append(p);
  }


  public void dequeue()   {
      hatList.toFirst();
      hatList.remove();
  }


  public Object front()  {
      hatList.toFirst();
      return hatList.getContent();
  }


  public boolean isEmpty()   {
     return hatList.isEmpty();
  }
}
```

Aufgabe 5

```
public class StackNeu<ContentType> {
  List<ContentType> hatList;

  public StackNeu()   {
    hatList = new List<>();
  }



  public void push(ContentType p)   {
     hatList.toFirst();
     hatList.insert(p);
  }



  public void pop()   {
     hatList.toFirst();
     hatList.remove();
  }



  public Object top()   {
     hatList.toFirst();
     return hatList.getContent();
  }



  public boolean isEmpty()   {
     return hatList.isEmpty();
  }

}
```

Aufgabe 6

```
public class SortierteListe<ContentType> {

  private List<ContentType>  hatList;

  public SortierteListe()   {
     hatList = new List<>();
  }


  public boolean isEmpty()   {
     return hatList.isEmpty();
  }


  public boolean hasAccess()  {
     return hatList.hasAccess();
  }


  public void next()  {
      hatList.next();
  }


  public void toFirst()  {
     hatList.toFirst();
  }


  public void toLast()  {
     hatList.toLast();
  }


  public ContentType getContent()  {
      return hatList.getContent();
  }
```

```
  public void insert(ContentType p)  {
     Item e, neuE;

     if (hatList.isEmpty()) hatList.insert(p);
     else  {
        hatList.toFirst();
        neuE = (Item) p;
        do  {
           e = (Item) hatList.getContent();
           if (! neuE.isGreater(e)) hatList.insert(p);
           else hatList.next();
        }
        while (neuE.isGreater(e) && hatList.hasAccess());

        if (! hatList.hasAccess()) hatList.append(p);
     } // of else
  }


  public void meltWith(SortierteListe<ContentType> p)  {
     p.toFirst();
     while (p.hasAccess())  {
        this.insert(p.getContent());
        p.next();
     }
  }


  public void remove()  {
     hatList.remove();
  }
}
```

Aufgabe 6 (Hauptprogramm

```
public class SortierteListeAnwendung extends
javax.swing.JFrame {

  private SortierteListe<IntegerElement> sortlist, sortlist2;

  public SortierteListeAnwendung() {
        initComponents();
  }


  private void btEinfuegenMouseClicked(java.awt…) {
    IntegerElement e;
    if (sortlist == null) sortlist = new SortierteListe<>();
    int zahl = Integer.parseInt(tfElement.getText());
    e = new IntegerElement(zahl);
    sortlist.insert(e);
  }


  private void btAusgabeMouseClicked(java.awt….) {
    IntegerElement e;
    taOriginal.setText("");
    if (sortlist != null)  {
      sortlist.toFirst();
      while (sortlist.hasAccess())  {
         e = sortlist.getContent();
         taOriginal.append(e.getInhalt()+"\n");
         sortlist.next();
      }
    }
  }


  private void btLoeschenMouseClicked(java.awt….) {
    sortlist = null;
    taOriginal.setText("");
  }


  private void btListe2ErzeugenMouseClicked(java.awt….) {
    IntegerElement e;
```

```
  sortlist2 = new SortierteListe<>();
  taListe2.setText("");
  for (int i = 1; i<7; i++)  {
     e = new IntegerElement(i*i);
     sortlist2.insert(e);
     taListe2.append(e.getInhalt() + "\n");
  }
}


private void btVerschmelzenMouseClicked(java.awt….) {
  sortlist.meltWith(sortlist2);
}
```

Aufgabe 8

```
public class StackNeu2 <ContentType> extends List<ContentType> {

  public void push(ContentType p)  {
    super.toFirst();
    super.insert(p);
  }


  public void pop()  {
    super.toFirst();
    super.remove();
}


  public ContentType top()  {
    super.toFirst();
    return super.getContent();
}


  @Override
  public boolean hasAccess()  {
    return true;
    // wird hiermit deaktiviert, sollte nicht benutzt werden.
  }


  @Override
  public void next()  {
    // wird hiermit deaktiviert, sollte nicht benutzt werden.
}


  @Override
  public void toFirst()  {
    // wird hiermit deaktiviert, sollte nicht benutzt werden.
  }
```

```
    @Override
    public void toLast()  {
        // wird hiermit deaktiviert, sollte nicht benutzt werden.
    }


    @Override
    public ContentType getContent()  {
        return null;
        // wird hiermit deaktiviert, sollte nicht benutzt werden.
    }


    @Override
    public void setContent(ContentType p)  {
        // wird hiermit deaktiviert, sollte nicht benutzt werden.
    }


    @Override
    public void append(ContentType p) {
        // wird hiermit deaktiviert, sollte nicht benutzt werden.
    }


    @Override
    public void insert(ContentType p)   {
        // wird hiermit deaktiviert, sollte nicht benutzt werden.
    }


    @Override
    public void concat(List<ContentType> p)   {
        // wird hiermit deaktiviert, sollte nicht benutzt werden.
    }


    @Override
    public void remove()  {
        // wird hiermit deaktiviert, sollte nicht benutzt werden.
    }
}
```

Aufgabe 9

```
public class SortierteListeNeu<ContentType>
                                  extends List<ContentType>  {

  @Override
  public void insert(ContentType p)  {
    Item e, neuE;

    if (this.isEmpty()) super.append(p);
    /* wichtig: hier darf nicht super.insert aufgerufen werden, weil diese Methode
       wiederum die Methode append aufruft, welche hier in dieser Unterklasse deaktiviert
       wurde. */
    else  {
       toFirst();
       neuE = (Item) p;
       do  {
         e = (Item) this.getContent();
         if (! neuE.isGreater(e)) super.insert(p);
         // in diesem Fall wird append nicht aufgerufen
         else next();
       }
       while (neuE.isGreater(e) && this.hasAccess());

       if (! this.hasAccess()) super.append(p);
    }  // end of else
 }



  public void meltWith(SortierteListe<ContentType> p)  {
    p.toFirst();
    while (p.hasAccess())  {
       this.insert(p.getContent());
       p.next();
    }
  }



  @Override
  public void setContent(ContentType p)   {
    // wird hiermit deaktiviert
  }
```

```
    @Override
    public void append(ContentType p)  {
       // wird hiermit deaktiviert
    }



    @Override
    public void concat(List<ContentType> p)  {
       // wird hiermit deaktiviert
    }

}
```

Projekt Vokabelliste

Erstelle folgendes Projekt, welches eine Liste von Vokabeln verwaltet!

Aufgaben

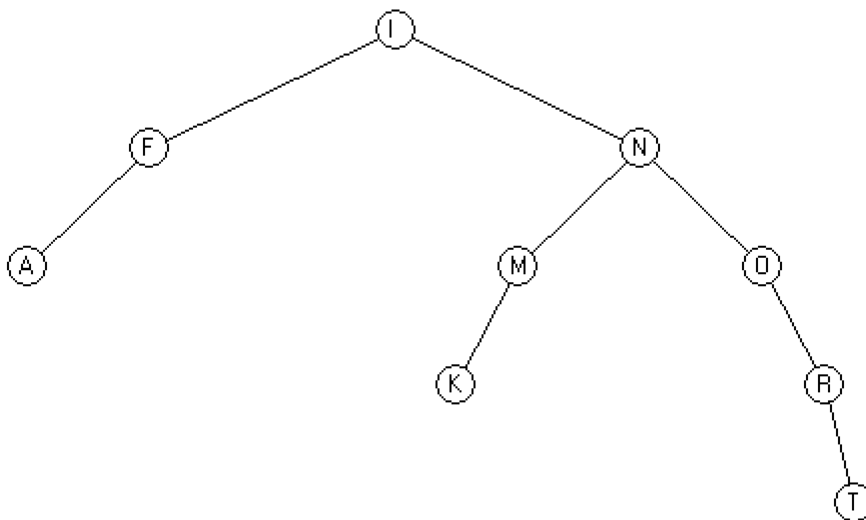
1. Erweitere das Programm so, dass man nicht nur nach englischen, sondern auch nach deutschen Begriffen suchen kann.

2. Die Vokabelliste soll auch nach deutschen Begriffen sortiert werden.

Nicht-Lineare Strukturen

Binärbäume

Die einzelnen Elemente eines Baumes heißen ***Knoten***, wobei die untersten Knoten keine Nachfolger haben und als ***Blätter*** bezeichnet werden.
Der oberste Knoten heißt auch ***Wurzel*** des Baumes.
Jeder ***innere Knoten*** eines Baumes hat somit mindestens einen Nachfolger.
Die Tiefe bzw. Höhe eines Baumes wird in der Literatur leider nicht immer identisch festgelegt. Wir definieren: Unter der ***Tiefe eines Knotens*** versteht man die Anzahl der ***Kanten*** von diesem Knoten bis hoch zur Wurzel. Die Wurzel selbst besitzt die Tiefe 0.
Als ***Höhe des Baumes*** bezeichnet man die Tiefe des Blattes mit der größten Tiefe. Die Höhe eines leeren Baumes soll **-1** sein.
Jeder Knoten besitzt einen (evtl. leeren) linken und rechten ***Teilbaum***.



Der obige Beispielsbaum besitzt 9 Knoten und 3 Blätter. Er besitzt die Höhe 4. Die Wurzel enthält den Buchstaben „I“.
Dieser Beispielsbaum ist ein sog. ***geordneter Baum*** oder ***Suchbaum***: Wenn man irgendeinen beliebigen Knoten betrachtet, so sind alle Elemente des linken Teilbaumes kleiner, und alle Elemente des rechten Teilbaumes größer als der Knoten selbst.
Der obige Beispielsbaum entsteht übrigens, wenn man das Wort *„INFORMATIK“* buchstabenweise in einen zunächst leeren Binärbaum einordnet. Ein Suchbaum enthält jede Information (z.B. den Buchstaben *„I“*) nur einmal.

Aufgaben

1. Zeichne alle möglichen Binärbaumstrukturen mit drei Knoten!

2. Gegeben sei ein Binärbaum der Höhe n. Erinnerung: ein Binärbaum, welcher nur aus der Wurzel besteht, hat die Höhe 0.
Wie viele Knoten besitzt dieser Baum mindestens und höchstens?

An obigem Binärbaum kann man auch schon unsere nächsten zu lösenden Aufgaben im Unterricht erkennen:

- Wie stellt man einen Binärbaum graphisch dar?
- Wie kann man feststellen, ob der Binärbaum ein bestimmtes Element enthält (Suchen)?
- Wie kann man ein Element löschen?
- Wie kann man ein neues Element einfügen?
- Wie kann man einen Binärbaum extern speichern?
- In welcher Reihenfolge soll man einen Binärbaum durchlaufen?
- Wie kann man einen Binärbaum optimieren? (Das Wort *Informatik* lässt sich sicherlich auch in einem kleineren Binärbaum darstellen!)

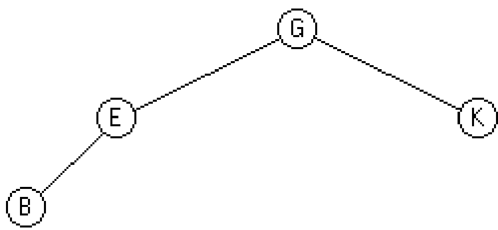
Definitionen:

(Adelson, Velskii und Landis):
Ein Baum ist genau dann *ausgeglichen* oder ***ausgewogen*, wenn sich für jeden Knoten die Tiefen der beiden Teilbäume um höchstens 1 unterscheiden.**

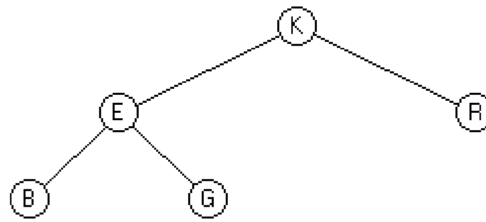
Derartige Binärbäume werden ***AVL-Bäume*** genannt.

Ein Binärbaum heißt *vollständig ausgeglichen* bzw. ***vollständig ausgewogen*, wenn sich für jeden Knoten die Zahl der Knoten in seinem linken und rechten Teilbaum um höchstens 1 unterscheiden.**

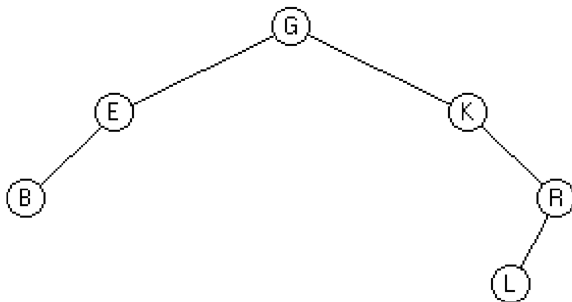
Beispiel a)



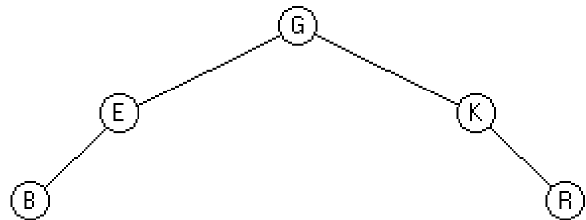
Beispiel b)



Beispiel c)



Beispiel d)



Aufgabe: Zeichne einen Binärbaum mit 11 Knoten, der
a) ausgewogen, aber nicht vollständig ausgewogen ist.
b) vollständig ausgewogen ist.

Im Folgenden wollen wir uns zunächst mit ungeordneten Binärbäumen beschäftigen.

Die Klasse *BinaryTree,* generische Version

Bemerkung: Die folgende Dokumentation und Implementation entspricht leider nicht den auf dem Softwaremarkt üblichen Darstellungen. Allerdings ist sie gültig für die Abiturjahrgänge in NRW ab dem Jahr 2017.

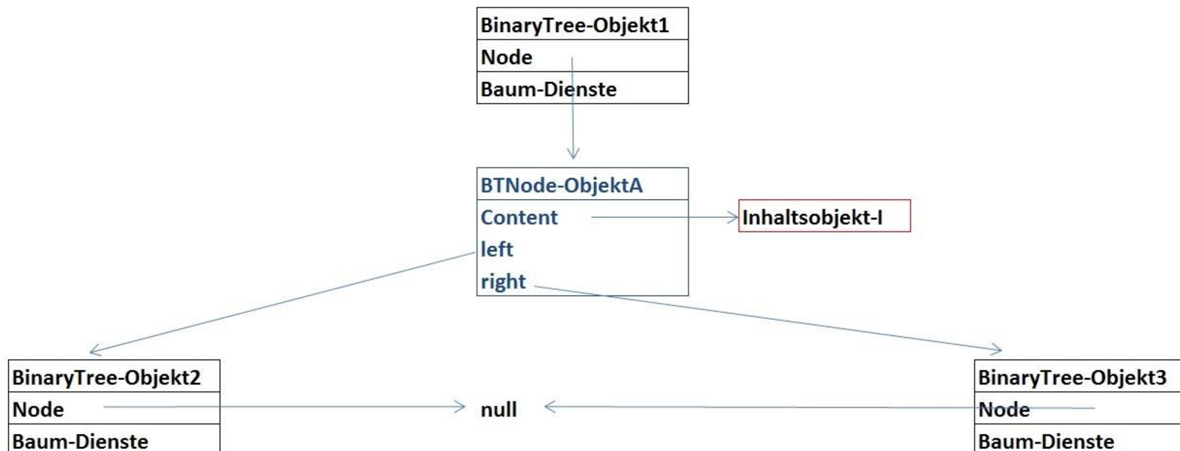
Die Klasse BinaryTree besitzt intern eine eigene Knotenklasse. Diese Knoten enthalten sowohl den eigentlichen Hinweis auf das zu speichernde Objektelement als auch zwei weitere Zeiger auf die untergeordneten Teilbäume.

Ein leerer Baum sieht so aus:

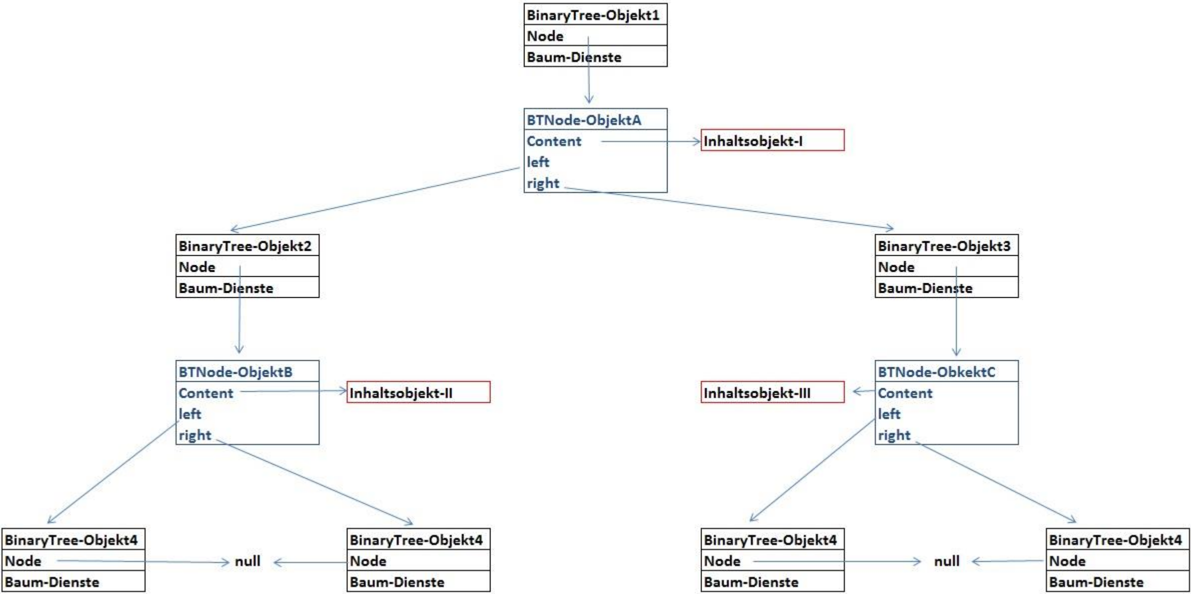
Der Knoten eines leeren Baumes hat den Wert *null*. Das zugehörige BinaryTree-Objekt für diesen leeren Baum ist allerdings nicht *null*. Ein leerer Baum ist auch ein Baum!
Ein Baum wird als leer erkannt, wenn sein Knoten gleich *null* ist.

Alle Blätter eines Baumes haben einen **leeren** linken und einen **leeren** rechten Unterbaum. Insbesondere sind diese beiden Teilbäume also nicht *null*.

Ein Binärbaum mit nur einem einzigen Element (also ein Blatt) sieht so aus:



Das folgende Bild zeigt einen Binärbaum mit drei Elementen:



Dokumentation der Klasse BinaryTree<ContentType>

Mithilfe der Klasse *BinaryTree<ContentType>* können beliebig viele Inhaltsobjekte verwaltet werden. Ein Objekt dieser Klasse stellt entweder einen leeren Baum dar oder verwaltet ein Inhaltsobjekt sowie einen linken und einen rechten Teilbaum, die ebenfalls Objekte der Klasse *BinaryTree<ContentType>* sind.

`Konstruktor` **`init()`**
Nach dem Aufruf existiert ein leerer Binärbaum.

`Konstruktor` **`init(ContentType p)`**
Wenn der Parameter p ungleich *null* ist, existiert nach dem Aufruf der Binärbaum und hat p als Inhaltsobjekt und zwei leere Teilbäume. Falls der Parameter *null* ist, wird nur ein leerer Binärbaum erzeugt.

`Konstruktor` **`init(ContentType p; BinaryTree pLeftTree; BinaryTree pRightTree)`**
Wenn der Parameter p ungleich *null* ist, wird ein Binärbaum mit p als Inhaltsobjekt und den beiden Teilbäume pLeftTree und pRightTree erzeugt. Sind pLeftTree oder pRightTree gleich *null*, wird der entsprechende Teilbaum als leerer Binärbaum eingefügt. Wenn der Parameter p gleich *null* ist, wird ein leerer Binärbaum erzeugt.

`Anfrage` **`boolean isEmpty()`**
Diese Anfrage liefert den Wahrheitswert ***true***, wenn der Binärbaum leer ist, sonst liefert sie den Wert ***false***.

`Auftrag` **`setContent(ContentType p)`**
Wenn p *null* ist, geschieht nichts. Ansonsten: Wenn der Binärbaum leer ist, wird der Parameter p als Inhaltsobjekt sowie ein leerer linker und rechter Teilbaum eingefügt. Ist der Binärbaum nicht leer, wird das Inhaltsobjekt durch p ersetzt. Die Teilbäume werden dann nicht geändert.

`Anfrage` **`ContentType getContent()`**
Diese Anfrage liefert das Inhaltsobjekt des Binärbaums. Wenn der Binärbaum leer ist, wird *null* zurückgegeben.

Auftrag **setLeftTree(BinaryTree<ContentType> pTree)**
Falls der Parameter *null* ist, geschieht nichts. Wenn der Binärbaum leer ist, wird pTree nicht angehängt. Andernfalls erhält der Binärbaum den übergebenen Baum als linken Teilbaum.

Auftrag **setRightTree(BinaryTree<ContentType> pTree)**
Falls der Parameter *null* ist, ändert sich nichts. Wenn der Binärbaum leer ist, wird pTree nicht angehängt. Andernfalls erhält der Binärbaum den übergebenen Baum als rechten Teilbaum.

Anfrage **BinaryTree<ContentType> getLeftTree()**
Diese Anfrage liefert den linken Teilbaum des Binärbaumes. Der Binärbaum ändert sich nicht. Wenn der Binärbaum leer ist, wird *null* zurückgegeben.

Anfrage **BinaryTree<ContentType> getRightTree()**
Diese Anfrage liefert den rechten Teilbaum des Binärbaumes. Der Binärbaum ändert sich nicht. Wenn der Binärbaum leer ist, wird *null* zurückgegeben.

Materialien zu den zentralen NRW-Abiturprüfungen ab 2017.
Generische Klasse BinaryTree<ContentType>

Mithilfe der generischen Klasse BinaryTree<ContentType> können beliebig viele Inhaltsobjekte vom Typ ContentType in einem Binärbaum verwaltet werden.
Ein Objekt der Klasse stellt entweder einen leeren Baum dar oder verwaltet ein Inhaltsobjekt sowie einen linken und einen rechten Teilbaum, die ebenfalls Objekte der generischen Klasse BinaryTree<ContentType> sind.

Author: Qualitäts- und UnterstützungsAgentur - Landesinstitut für Schule, Materialien zum schulinternen Lehrplan Informatik SII
@version Generisch_03 2014-03-01

```
public class BinaryTree<ContentType> {

/* --------- Anfang der privaten inneren Klasse -------------- */

/* Durch diese innere Klasse kann man dafür sorgen, dass ein leerer Baum null ist, ein nicht-
   leerer Baum jedoch immer eine nicht-null-Wurzel sowie nicht-null-Teilbäume, ggf. leere
   Teilbäume hat.  * /

  private class BTNode<CT> {

    private CT content;
    private BinaryTree<CT> left, right;

    public BTNode(CT pContent) {
    /* Der Knoten hat einen linken und einen rechten Teilbaum, die beide von null
       verschieden sind. Also hat ein Blatt immer zwei leere Teilbäume unter sich.*/
      this.content = pContent;
      left = new BinaryTree<CT>();
      right = new BinaryTree<CT>();
    }
  } /* ----------- Ende der privaten inneren Klasse -------------- */


private BTNode<ContentType> node;

// Nach dem Aufruf des Konstruktors existiert ein leerer Binärbaum.
public BinaryTree() {
   this.node = null;
}
```

/ Wenn der Parameter pContent ungleich null ist, existiert nach dem Aufruf des Konstruktors der Binärbaum und hat pContent als Inhaltsobjekt und zwei leere Teilbäume. Falls der Parameter null ist, wird ein leerer Binärbaum erzeugt. */*

```
public BinaryTree(ContentType pContent) {
  if (pContent != null) {
    this.node = new BTNode<ContentType>(pContent);
  }
  else  this.node = null;
}
```

/ Wenn der Parameter pContent ungleich null ist, wird ein Binärbaum mit pContent als Inhalt und den beiden Teilbäume pLeftTree und pRightTree erzeugt. Sind pLeftTree oder pRightTree gleich null, wird der entsprechende Teilbaum als leerer Binärbaum eingefuegt. So kann es also nie passieren, dass linke oder rechte Teilbaeume null sind. Wenn der Parameter pContent gleich null ist, wird ein leerer Binärbaum erzeugt. */*

```
public BinaryTree(ContentType pContent,
                  BinaryTree<ContentType> pLeftTree,
                  BinaryTree<ContentType> pRightTree) {

  if (pContent != null) {
    this.node = new BTNode<ContentType>(pContent);
    if (pLeftTree != null) this.node.left = pLeftTree;
    else this.node.left = new BinaryTree<ContentType>();

    if (pRightTree != null) this.node.right = pRightTree;
    else this.node.right = new BinaryTree<ContentType>();
  }
  else {
  // Da der Inhalt null ist, wird ein leerer BinarySearchTree erzeugt.
     this.node = null;
  }
}
```

/ Diese Anfrage liefert den Wahrheitswert true, wenn der Binärbaum leer ist, sonst liefert sie den Wert false.*/*

```
public boolean isEmpty() {
  return this.node == null;
}
```

/ Wenn pContent null ist, geschieht nichts.*
Ansonsten: Wenn der Binärbaum leer ist, wird der Parameter pContent als Inhaltsobjekt sowie ein leerer linker und rechter Teilbaum eingefügt.
*Ist der Binaerbaum nicht leer, wird das Inhaltsobjekt durch pContent ersetzt. Die Teilbäume werden nicht geändert. */*

```
public void setContent(ContentType pContent) {
  if (pContent != null) {
    if (this.isEmpty()) {
      node = new BTNode<ContentType>(pContent);
      this.node.left = new BinaryTree<ContentType>();
      this.node.right = new BinaryTree<ContentType>();
    }
    this.node.content = pContent;
  }
}
```

/ Diese Anfrage liefert das Inhaltsobjekt des Binärbaums. Wenn der Binärbaum leer ist, wird null zurückgegeben. */*

```
public ContentType getContent() {
  if (this.isEmpty()) return null;
  else return this.node.content;
}
```

/ Falls der Parameter null ist, geschieht nichts. Wenn der Binärbaum leer ist, wird pTree nicht angehängt. Andernfalls erhält der Binärbaum den übergebenen BinaryTree als linken Teilbaum. */*

```
public void setLeftTree(BinaryTree<ContentType> pTree) {
  if (!this.isEmpty() && pTree != null) {
    this.node.left = pTree;
  }
}
```

/ Falls der Parameter null ist, geschieht nichts. Wenn der Binärbaum leer ist, wird pTree nicht angehängt. Andernfalls erhält der Binärbaum den übergebenen BinaryTree als rechten Teilbaum. */*

```
public void setRightTree(BinaryTree<ContentType> pTree) {
  if (!this.isEmpty() && pTree != null) {
    this.node.right = pTree;
  }
}
```

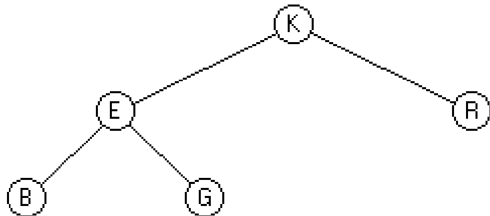
/ Diese Anfrage liefert den linken Teilbaum des Binärbaumes. Wenn der Binärbaum leer ist, wird null zurückgegeben. */*

```
public BinaryTree<ContentType> getLeftTree() {
  if (!this.isEmpty()) return this.node.left;
  else return null;
}
```

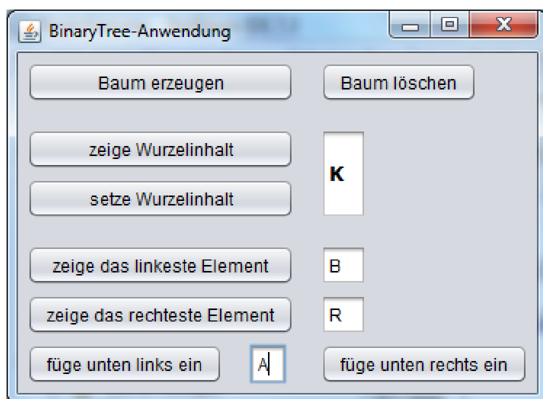
/ Diese Anfrage liefert den rechten Teilbaum des Binärbaumes. Wenn der Binärbaum (this) leer ist, wird null zurückgegeben. */*

```
public BinaryTree<ContentType> getRightTree() {
  if (!this.isEmpty()) return this.node.right;
  else return null;
}

}
```

Um die Methoden der Klasse *BinaryTree<ContentType>* kennenzulernen, arbeiten wir mit der breits bekannten Klasse *ZeichenElement*, welche einzelne Buchstaben enthalten kann.
Da die Klasse *BinaryTree<ContentType>* noch keine Ordnung kennt, gibt es auch noch keine sinnvolle Einfügemethode. Deshalb muss jeder Baum noch umständlich zusammengesetzt werden. Im folgenden Beispielprogramm soll mit dem nachstehend dargestellten Baum gearbeitet werden:



Die Konstruktion dieses Baumes geschieht so:
Erzeuge zuerst die beiden Einzelbäume B und G, danach den Baum E! Analog wird dann der Baum R erzeugt und zuletzt der Baum K.



Um zu überprüfen, ob die implementierte Klasse *BinaryTree* auch wie gewünscht funktioniert, programmiere das obenstehende Hauptprogramm!

Die beiden letzten Buttons sollen jeweils einen weiteren Knoten in den Baum einfügen. Ob das geklappt hat, kann man anschließend z.B. mit dem Button *„zeige das linkeste Element“* überprüfen.

```
public class BinaryTreeAnwendung extends javax.swing.JFrame {

  BinaryTree<ZeichenElement> baum;

  public BinaryTreeAnwendung() {
     initComponents();
     baum = null;
  }

  private void btBaumErzeugenMouseClicked(java……)  {
    ZeichenElement e;
    BinaryTree<ZeichenElement> baumB, baumG, baumE, baumR;

    e = new ZeichenElement('B');
    baumB = new BinaryTree<>(e);
    e = new ZeichenElement('G');
    baumG = new BinaryTree<> (e);
    e = new ZeichenElement('E');
    baumE = new BinaryTree<> (e, baumB, baumG);
    e = new ZeichenElement('R');
    baumR = new BinaryTree<> (e);
    e = new ZeichenElement('K');
    baum = new BinaryTree<> (e, baumE, baumR);
  }



  private void btBaumLoeschenMouseClicked(java……)  {
    if (baum != null)  baum = null;
    tfWurzel.setText("");
    tfLinkestesElement.setText("");
    tfRechtestesElement.setText("");
  }



  private void btZeigeWurzelMouseClicked(java……)  {
    ZeichenElement e = baum.getContent();
    tfWurzel.setText("" + e.getInhalt());
  }
```

```
private void btSetzeWurzelMouseClicked(java.awt……)  {
  ZeichenElement e=new ZeichenElement(tfWurzel.getText().charAt(0));
  baum.setContent(e);
}


private void btLinkestesElementMouseClicked(java……)  {
  BinaryTree<ZeichenElement> lauf = baum;
  ZeichenElement e;
  if (! lauf.isEmpty())  {
     while (! lauf.getLeftTree().isEmpty())
          lauf = lauf.getLeftTree();
     e = lauf.getContent();
     if (e!=null) tfLinkestesElement.setText("" + e.getInhalt());
  }
}


private void btRechtestesElementMouseClicked(java……)  {
   BinaryTree<ZeichenElement> lauf = baum;
   ZeichenElement e;
   if (! lauf.isEmpty())  {
      while (! lauf.getRightTree().isEmpty())
           lauf = lauf.getRightTree();
      e = lauf.getContent();
      if (e !=null) tfRechtestesElement.setText(""+e.getInhalt());
   }
}


private void btFuegeUntenLinksEinMouseClicked(java……)  {
   BinaryTree<ZeichenElement> lauf, neu;
   ZeichenElement e;

   lauf = baum;
   if (! lauf.isEmpty())   {
      while (!lauf.getLeftTree().isEmpty())
          lauf = lauf.getLeftTree();
      e = new ZeichenElement(tfEingabe.getText().charAt(0));
      //Vorsicht, wenn Editfeld leer ist!
      neu = new BinaryTree<>(e);
      lauf.setLeftTree(neu);
   }
}
```

```
private void btFuegeUntenRechtsEinMouseClicked(java......)  {
   BinaryTree<ZeichenElement> lauf, neu;
   ZeichenElement e;

   lauf = baum;
   if (! lauf.isEmpty())   {
      while (!lauf.getRightTree().isEmpty())
          lauf = lauf.getRightTree();
      e = new ZeichenElement(tfEingabe.getText().charAt(0));
      //Vorsicht, wenn Editfeld leer ist!
      neu = new BinaryTree(e);
      lauf.setRightTree(neu);
   }
}
```

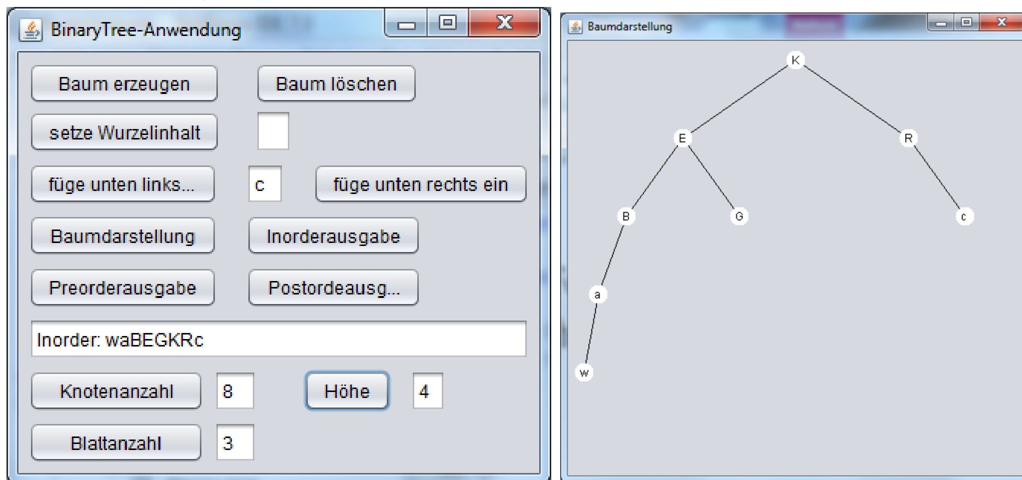
Hinweis: Die obige Implementierung setzt voraus, dass nur sinnvolle Aktionen durchgeführt werden. Zum Beispiel sollte man kein Element einfügen, nachdem vorher der Baum komplett gelöscht worden ist. Allerdings lässt sich das Programm auch relativ einfach so ändern, dass auch bei nicht sinnvollen Eingaben keine Fehlermeldungen auftreten.

Aufgabe: Programmiere die letzten vier Methoden rekursiv!

Lösung: *// rekursive Version nur für das linkeste Element*

```
private BinaryTree zeigerAufLinkestesElement(BinaryTree zeiger)  {
  if (zeiger.getLeftTree().isEmpty()) return zeiger;
  else return zeigerAufLinkestesElement(zeiger.getLeftTree());
}


private void btLinkestesElementMouseClicked(java……) {
  ZeichenElement e;
  BinaryTree<ZeichenElement> pointer = baum;
  if (baum != null)  {
     pointer = zeigerAufLinkestesElement(baum);
     e = pointer.getContent();
     tfLinkestesElement.setText("" + e.getInhalt());
  }
}


private void btFuegeUntenLinksEinMouseClicked(java……)  {
 ZeichenElement e=new ZeichenElement(tfEingabe.getText().charAt(0));
 BinaryTree<ZeichenElement>  neu = new BinaryTree<>(e);
 BinaryTree<ZeichenElement>  pointer = baum;
 if (baum != null)  {
     pointer = zeigerAufLinkestesElement(baum);
     pointer.setLeftTree(neu);
 }
}
```

Im Folgenden wird unser Hauptprogramm durch einige sehr wichtige, rekursive Methoden ergänzt.




```
import java.awt.*;    //wird für Baumdarstellung benötigt.

public class BinaryTreeAnwendung extends javax.swing.JFrame {

  BinaryTree<ContentType> baum;
  Graphics zeichenagent;
  javax.swing.JFrame Plan;


  public BinaryTreeAnwendung() {
    initComponents();
    baum = null;
    Plan = new javax.swing.JFrame();
    // Plan.setSize(500,500);
    Plan.setBounds(this.getWidth(),0,500,500);
    Plan.setTitle("Baumdarstellung");
    // Plan.getContentPane().setBackground(Color.WHITE);
    Plan.setVisible(true);
    zeichenagent = Plan.getGraphics();
  }



  private void btBaumErzeugenMouseClicked(java.awt......)   {
    ZeichenElement e;
    BinaryTree baumB, baumG, baumE, baumR;

    e = new ZeichenElement('B');
    baumB = new BinaryTree(e);
```

```
    e = new ZeichenElement('G');
    baumG = new BinaryTree(e);
    e = new ZeichenElement('E');
    baumE = new BinaryTree(e, baumB, baumG);
    e = new ZeichenElement('R');
    baumR = new BinaryTree(e);
    e = new ZeichenElement('K');
    baum = new BinaryTree(e, baumE, baumR);
  }



private void btBaumLoeschenMouseClicked(java.awt……)  {
  if (baum != null)  baum = null;
  tfWurzel.setText("");

  // Plan.getContentPane().setBackground(Color.WHITE);
  // Die Farbe funktioniert nicht beim 2. Mal löschen
  zeichenagent.fillRect(0,0,Plan.getWidth(),Plan.getHeight());
}



private void btSetzeWurzelMouseClicked(java.awt…..)  {
  ZeichenElement e=new ZeichenElement(tfWurzel.getText().charAt(0));
  baum.setContent(e);
}



private void zeichne(BinaryTree<ZeichenElement> pBaum,
                                       int xl,int xr,int y)  {
  int deltay = Plan.getHeight()/6; // willkürliche Tiefen-Begrenzung
  int xm;
  ZeichenElement e;

  if (! pBaum.isEmpty())   {
    xm = (xl + xr)/2;
    if (! pBaum.getLeftTree().isEmpty())
       zeichenagent.drawLine(xm,y,(xl + xm)/2,y + deltay);
    if (! pBaum.getRightTree().isEmpty())
       zeichenagent.drawLine(xm,y,(xr + xm)/2,y + deltay);
    // Diese Kanten werden anschließend teilweise von den gefüllten Kreisen überzeichnet.
    zeichenagent.setColor(Color.white);
```

```
      zeichenagent.fillOval( xm-10, y - 10, 20, 20);
      zeichenagent.setColor(Color.black);
      e = pBaum.getContent();
      if (e != null)
         zeichenagent.drawString(""+e.getInhalt(), xm-4, y + 5);
      zeichne(pBaum.getLeftTree(), xl, xm, y + deltay);
      zeichne(pBaum.getRightTree(), xm, xr, y + deltay);
   } // of if (! pBaum.isEmpty())
}

private void btBaumdarstellungMouseClicked(java.awt…....)  {
   int xl = 10;
   int xr = Plan.getWidth() - 10;
   // die Koordinaten werden angepasst, so dass es am Rand keine Probleme gibt.
   zeichne(baum, xl, xr, 50);
}



private void preorder(BinaryTree<ZeichenElement> pBaum)  {
   ZeichenElement e;

   if (! pBaum.isEmpty()) {
      e = pBaum.getContent();
      if (e != null)
         tfBaumausgabe.setText(tfBaumausgabe.getText()
                                                + e.getInhalt());
      preorder(pBaum.getLeftTree());
      preorder(pBaum.getRightTree());
   }
}

private void btPreorderMouseClicked(java.awt…...)  {
   tfBaumausgabe.setText("Preorder: ");
   if (baum != null) preorder(baum);
}



private void inorder(BinaryTree<ZeichenElement> pBaum)  {
   ZeichenElement e;

   if (! pBaum.isEmpty()) {
      inorder(pBaum.getLeftTree());
```

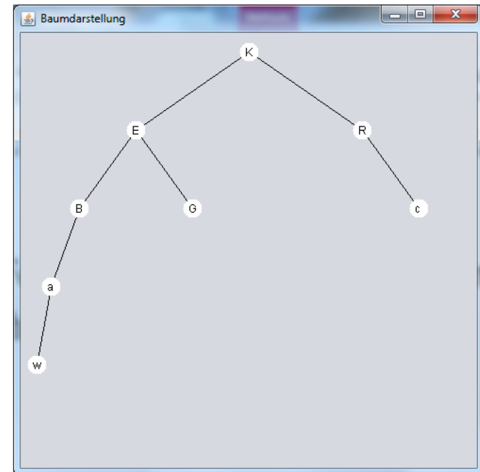
```
      e = pBaum.getContent();
      if (e != null)
         tfBaumausgabe.setText(tfBaumausgabe.getText()
                                                  + e.getInhalt());

      inorder(pBaum.getRightTree());
   }
}

private void btInorderMouseClicked(java.awt……)  {
   tfBaumausgabe.setText("Inorder: ");
   if (baum != null) inorder(baum);
}



private void postorder(BinaryTree<ZeichenElement> pBaum)  {
   ZeichenElement e;

   if (! pBaum.isEmpty()) {
      postorder(pBaum.getLeftTree());
      postorder(pBaum.getRightTree());

      e = pBaum.getContent();
      if (e != null)
         tfBaumausgabe.setText(tfBaumausgabe.getText()
                                                  + e.getInhalt());
   }
}

private void btPostorderMouseClicked(java.awt…....)  {
   tfBaumausgabe.setText("Postorder: ");
   if (baum != null) postorder(baum);
}



private int anzahl(BinaryTree<ZeichenElement> pBaum)   {
   if (pBaum.isEmpty()) return 0;
   else return (anzahl(pBaum.getLeftTree())
                          + anzahl(pBaum.getRightTree()) + 1);
}
```

```
private void btKnotenanzahlMouseClicked(java.awt……)  {
  if (baum != null) tfKnoten.setText("" + anzahl(baum));
  else tfKnoten.setText("-1");
}



private int blattAnzahl(BinaryTree<ZeichenElement> pBaum)   {
  if (pBaum.getLeftTree().isEmpty()&&pBaum.getRightTree().isEmpty())
     return 1;
  else if (pBaum.getLeftTree().isEmpty())
         return blattAnzahl(pBaum.getRightTree());
       else if (pBaum.getRightTree().isEmpty())
              return blattAnzahl(pBaum.getLeftTree());
            else return (blattAnzahl(pBaum.getLeftTree())
                         + blattAnzahl(pBaum.getRightTree()));
}



private void btBlattanzahlMouseClicked(java.awt……)  {
  if (baum != null)
     if (! baum.isEmpty()) tfBlatt.setText(""+ blattAnzahl(baum));
}



private int hoehe(BinaryTree pBaum)  {
  if (pBaum.isEmpty()) return -1;
  else return Math.max(hoehe(pBaum.getLeftTree()),
                           hoehe(pBaum.getRightTree())) + 1;
}

private void btHoeheMouseClicked(java.awt……)  {
  tfHoehe.setText("" + hoehe(baum));
}
```

Aufgaben

1. Gegeben sei ein nicht geordneter Binärbaum mit 9 Knoten.
Inorderausgabe: E A C K F H D B G
Preorderausgabe: F A E K C D H G B
Zeichne den Baum!

2. Gib (in Inorder-Reihenfolge) jedes Element eines Binärbaumes zusammen mit seiner Ebenennummer aus! Für den nebenstehenden Binärbaum sähe die Ausgabe so aus:
w5, a4, B3, E2, G3, K1, R2, c3

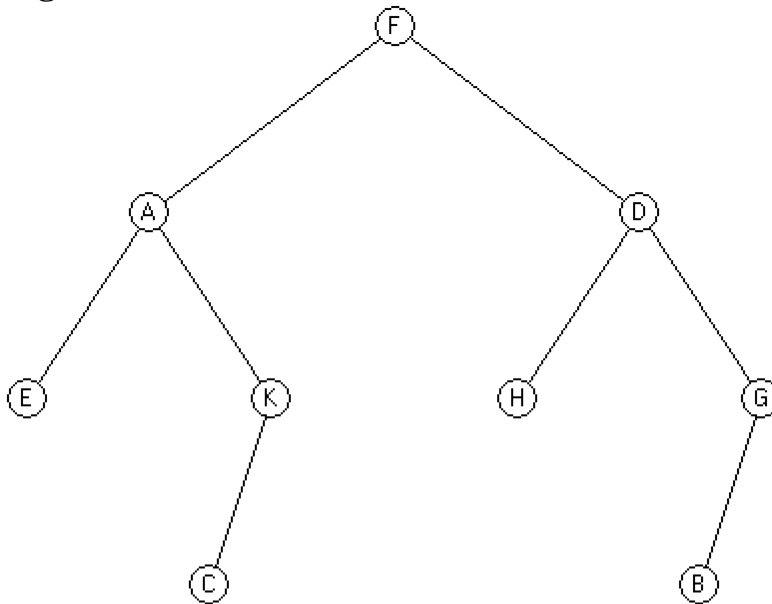


3. Gib (in Inorder-Reihenfolge) alle Elemente eines Binärbaumes zusammen mit ihren Pfadnamen aus. Dabei steht **L** für links und **R** für rechts. Für obigen Binärbaum sähe die Ausgabe so aus:

w	LLLL
a	LLL
B	LL
E	L
G	LR
K	
R	R
c	RR

Lösungen

Aufgabe 1



Aufgabe 2

```
private void ausgabe(BinaryTree<ZeichenElement> pBaum, int n){
  ZeichenElement e;
  if (! pBaum.isEmpty())  {
     ausgabe(pBaum.getLeftTree(), n+1);
     e = pBaum.getContent();
     if (e != null)
        tfBaumausgabe.setText(tfBaumausgabe.getText()
                                   + e.getInhalt()+n + ", ");
     ausgabe(pBaum.getRightTree(), n+1);
  }
}

private void jButton1MouseClicked(java……)  {
  tfBaumausgabe.setText("");
  ausgabe(baum, 1);
}
```

Aufgabe 3

```
private void pfadausgabe(BinaryTree<ZeichenElement> pBaum,
                                                  String s)   {
  ZeichenElement e;
  if (! pBaum.isEmpty())  {
     pfadausgabe(pBaum.getLeftTree(), s+"L");
     e = pBaum.getContent();
     if (e != null) taAusgabe.append(e.getInhalt()+"  "+s+"\n");
     pfadausgabe(pBaum.getRightTree(), s+"R");
  }
}


private void btAufgabe3MouseClicked(java.awt…..)  {
  taAusgabe.setText("");
  pfadausgabe(baum, "");
}
```

Morsecode-Aufgabe

Lateinische Buchstaben	
Buchstabe	**Code**
A	· —
B	— · · ·
C	— · — ·
D	— · ·
E	·
F	· · — ·
G	— — ·
H	· · · ·
I	· ·
J	· — — —
K	— · —
L	· — · ·
M	— —
N	— ·
O	— — —
P	· — — ·
Q	— — · —
R	· — ·
S	· · ·
T	—
U	· · —
V	· · · —
W	· — —
X	— · · —
Y	— · — —
Z	— — · ·

Der Morsecode kann in einem Binärbaum dargestellt werden. Dabei stehen in den Knoten die Buchstaben. Die Wurzel enthält keinen Buchstaben. Ein Punkt im Morsecode entspricht der Verzweigung nach links, ein Strich der Verzweigung nach rechts.

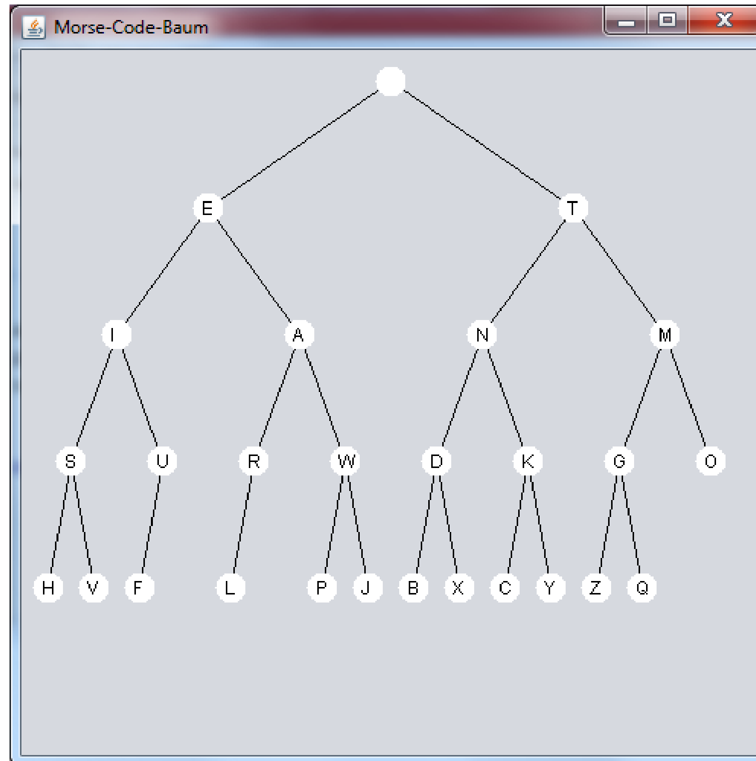
a) Zeichne zunächst den Morsecode-Baum auf Papier!

b) Erzeuge anschließend diesen Morsecode-Baum so, wie es im letzten Haupt-programm vorgestellt wurde! Beachte die Methode `btBaumErzeugenMouseClicked;`

c) Schreibe ein Programm, welches mit Hilfe des Morsecode-Baumes den in einem Memofeld stehenden Morsecode, z.B.
.−−./..../−.−−/.../../−.−/
dekodiert. Als Trennzeichen dient dabei der Querstrich (oder ein Leerzeichen).

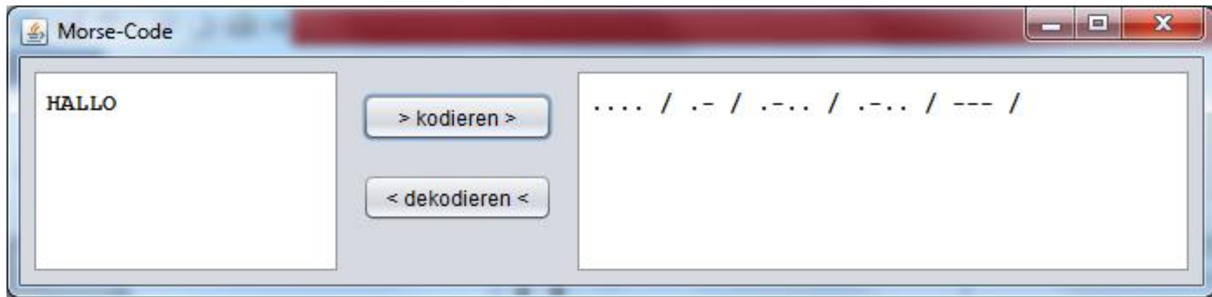
d) Der in einem Memofeld stehende Original-text soll in den Morsecode codiert werden.

Lösung der Morse-Code-Aufgabe:




```
private void MorsebaumErzeugen()  {
  ZeichenElement e;
  BinaryTree<ZeichenElement> baumL, baumR, baum3, baum4,
                                      baum5, baumE, baumT;

  e = new ZeichenElement('H');
  baumL = new BinaryTree<>(e);
  e = new ZeichenElement('V');
  baumR = new BinaryTree<> (e);
  e = new ZeichenElement('S');
  baum3 = new BinaryTree<> (e, baumL, baumR);
  e = new ZeichenElement('F');
  baumL = new BinaryTree<> (e);
  e = new ZeichenElement('U');
  baum4 = new BinaryTree<> (e, baumL, null);
  e = new ZeichenElement('I');
  baum5 = new BinaryTree<> (e, baum3, baum4);

  e = new ZeichenElement('L');
  baumL = new BinaryTree<> (e);
  e = new ZeichenElement('R');
```

```
  baum3 = new BinaryTree<> (e, baumL, null);
  e = new ZeichenElement('P');
  baumL = new BinaryTree<> (e);
  e = new ZeichenElement('J');
  baumR = new BinaryTree<> (e);
  e = new ZeichenElement('W');

  baum4 = new BinaryTree<> (e, baumL, baumR);
  e = new ZeichenElement('A');
  baum4 = new BinaryTree<> (e, baum3, baum4);
  e = new ZeichenElement('E');
  baumE = new BinaryTree<> (e, baum5, baum4);

  e = new ZeichenElement('B');
  baumL = new BinaryTree<> (e);
  e = new ZeichenElement('X');
  baumR = new BinaryTree<> (e);
  e = new ZeichenElement('D');
  baum3 = new BinaryTree<> (e, baumL, baumR);
  e = new ZeichenElement('C');
  baumL = new BinaryTree<> (e);
  e = new ZeichenElement('Y');
  baumR = new BinaryTree<> (e);
  e = new ZeichenElement('K');
  baum4 = new BinaryTree<> (e, baumL, baumR);
  e = new ZeichenElement('N');
  baum5 = new BinaryTree<> (e, baum3, baum4);

  e = new ZeichenElement('Q');
  baumR = new BinaryTree<> (e);
  e = new ZeichenElement('Z');
  baumL = new BinaryTree<> (e);
  e = new ZeichenElement('G');
  baum3 = new BinaryTree<> (e, baumL, baumR);
  e = new ZeichenElement('O');
  baum4 = new BinaryTree<> (e);
  e = new ZeichenElement('M');
  baum4 = new BinaryTree<> (e, baum3, baum4);
  e = new ZeichenElement('T');
  baumT = new BinaryTree<> (e, baum5, baum4);

  e = new ZeichenElement(' '); // Leerzeichen
  baum = new BinaryTree<> (e, baumE, baumT);
}
```



Hinweis: Im Morsecode-Fenster sollte die Eigenschaft *LineWrap* (Zeilenumsprung) gesetzt werden.

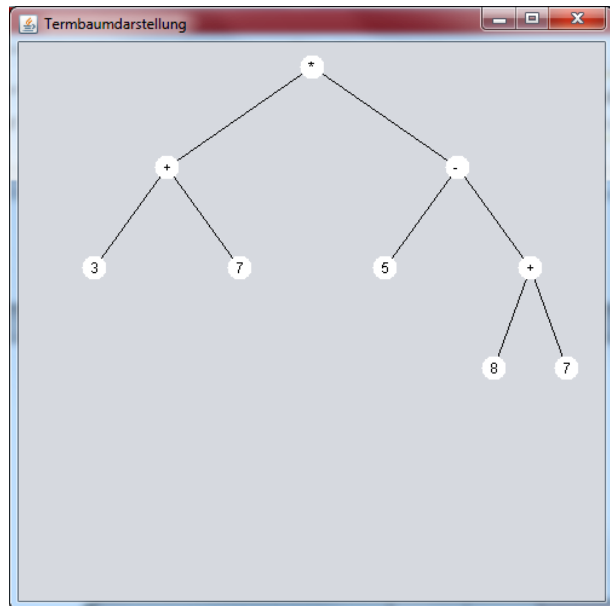
```
public class MorseCode extends javax.swing.JFrame {
  BinaryTree<ZeichenElement> baum;
  String Alphabet = "ABCDEFGHIJKLMNOPQRSTUVWXYZ";

  public MorseCode() {
    initComponents();
    MorsebaumErzeugen();
  }

  private String dekodierterText(String pCode)  {
    int zaehler = 0;
    BinaryTree<ZeichenElement> hilfsbaum = baum;
    ZeichenElement e;
    String ergebnis = "";
    while (zaehler < pCode.length())  {
       if (pCode.charAt(zaehler) == '.')
                 hilfsbaum = hilfsbaum.getLeftTree();
       else if (pCode.charAt(zaehler) == '-')
                hilfsbaum = hilfsbaum.getRightTree();
            /*Wenn der Benutzer Eingaben ins Memofeld schreibt, werden leider auch
              zusätzlich die ASCII-Codes von Return (13) und Linefeed (10) gespeichert.
            */
            else  {
               e = hilfsbaum.getContent();
               if (e != null)
                  if (Alphabet.indexOf(e.getInhalt()) != -1)
                      ergebnis = ergebnis + e.getInhalt();
               hilfsbaum = baum;
            }
       zaehler++;
    }  // of while
    return ergebnis;
  }
```

```
private void MorsebaumErzeugen() // siehe oben


private void btDekodierenMouseClicked(java.awt.....)  {
  String CodeText = taMorsecode.getText() + "$";
  // zusätzliches Endezeichen erleichtert die Auswertung.
  taOriginal.setText(dekodierterText(CodeText));
}



private String Zeichencode(BinaryTree<ZeichenElement> hilfsbaum,
                                         char ch, String weg)  {
  ZeichenElement e;
  String ergebnis = "";

  if (hilfsbaum != null)  {
    e = hilfsbaum.getContent();
    if (e != null)  if (ch == e.getInhalt()) ergebnis = weg;
    if (ergebnis == "")
      ergebnis = Zeichencode(hilfsbaum.getLeftTree(),ch, weg + ".");
    if (ergebnis == "")
      ergebnis = Zeichencode(hilfsbaum.getRightTree(),ch,weg + "-");
  }
  return ergebnis;
}



private void btKodierenMouseClicked(java.awt.....)  {
  String OriginalText = taOriginal.getText();
  taMorsecode.setText("");
  for (int i = 0; i < OriginalText.length(); i++)
     if (Alphabet.indexOf(OriginalText.charAt(i)) != -1)
        taMorsecode.setText(taMorsecode.getText()+
        Zeichencode(baum,OriginalText.charAt(i), "") + " / ");
}
```

Termbäume

Ein Termbaum besteht entweder nur aus einem Blatt, welches eine Zahl enthält oder aber die Wurzel enthält einen Operator und die Teilbäume links und rechts sind ebenfalls Termbäume.

Die nachfolgende Implementation stammt ursprünglich von Horst Hildebrecht und wurde hier etwas bearbeitet.




```
public class Termbaum  {

  BinaryTree<String> kenntBinaryTree;


  public Termbaum(BinaryTree<String> pBinaerbaum) {
     kenntBinaryTree = pBinaerbaum;
  }


  // Der Inhalt der Wurzel des Binärbaums wird als Zeichenkette geliefert.
  private String wurzelString(BinaryTree<String> pBinaryTree)
   {
      return pBinaryTree.getContent();
   }


  // vorher: Die Wurzel des Binärbaums enthält einen Operator.
  // nachher: Der Inhalt der Wurzel des Binärbaums wurde als Zeichen geliefert.
  public char wurzelOperator(BinaryTree<String> pBinaryTree) {
    return this.wurzelString(pBinaryTree).charAt(0);
  }
```

```
// vorher: Die Wurzel des Binärbaums enthält eine Zahl.
// nachher: Der Inhalt der Wurzel des Binärbaums wurde als Zahl vom Typ double
   geliefert.
public double wurzelZahl(BinaryTree<String> pBinaryTree)  {
  return Double.parseDouble(this.wurzelString(pBinaryTree));
}


// Der Wert des Termbaums wird geliefert.
public double wert()   {
   return this.wert(kenntBinaryTree);
}



private double wert(BinaryTree<String> pBinaryTree)   {
   if (pBinaryTree.getLeftTree().isEmpty() &&
          pBinaryTree.getRightTree().isEmpty())  // Blatt, also Zahl
       return this.wurzelZahl(pBinaryTree);
   else  { // Kein Blatt, also Operator mit Teilbäumen
      double lWertLinks =
                       this.wert(pBinaryTree.getLeftTree());
      double lWertRechts =
                      this.wert(pBinaryTree.getRightTree());
      switch (this.wurzelOperator(pBinaryTree))  {
          case '+': return lWertLinks + lWertRechts;
          case '-': return lWertLinks - lWertRechts;
          case '*': return lWertLinks * lWertRechts;
          case '/': return lWertLinks / lWertRechts;
      }
   }
   return 0;   // muss da sein, falls switch-Wert nicht definiert
  }



// Die Darstellung des Termbaums in umgekehrter polnischer Notation wird geliefert.
public String inUPN()   {
   return this.inUPN(kenntBinaryTree);
}
```

```
  private String inUPN(BinaryTree<String> pBinaryTree)  {
    if (pBinaryTree.isEmpty())  return "";
      else
        return this.inUPN(pBinaryTree.getLeftTree()) +
              this.inUPN(pBinaryTree.getRightTree()) +
              this.wurzelString(pBinaryTree) + " ";
  }
```

} *// Ende der Klasse Termbaum*

Aufgabe: Schreibe eine Dokumentation über die Klasse Termbaum!

Ein TermbaumPflanzer wandelt eine Zeichenkette (in fehlerfreier Infixnotation) in einen BinaryTree um, welcher den mathematischen Term als Baum darstellt.
** @author Horst Hildebrecht*
** bearbeitet von Dieter Lindenberg*

```
public class TermbaumPflanzer  {
  char STOP = 1;    //als zusätzliches Endezeichen des Eingabeterms: Keine Ziffer, kein
                    //Vorzeichen, kein Operator

  BinaryTree<String> hatBinaryTree;

  int zPosition;
  String zTerm;


  // Konstruktor
  // Wenn der angegebene Term korrekt war, hat der TermbaumPflanzer einen
  // entsprechenden BinaryTree erstellt.
  public TermbaumPflanzer(String pTerm)   {
    zTerm = pTerm + STOP;
    zPosition = 1;
    hatBinaryTree = this.baumAusTerm();
  }
```

```
 public BinaryTree<String> baum()   {
    return hatBinaryTree;
 }
```

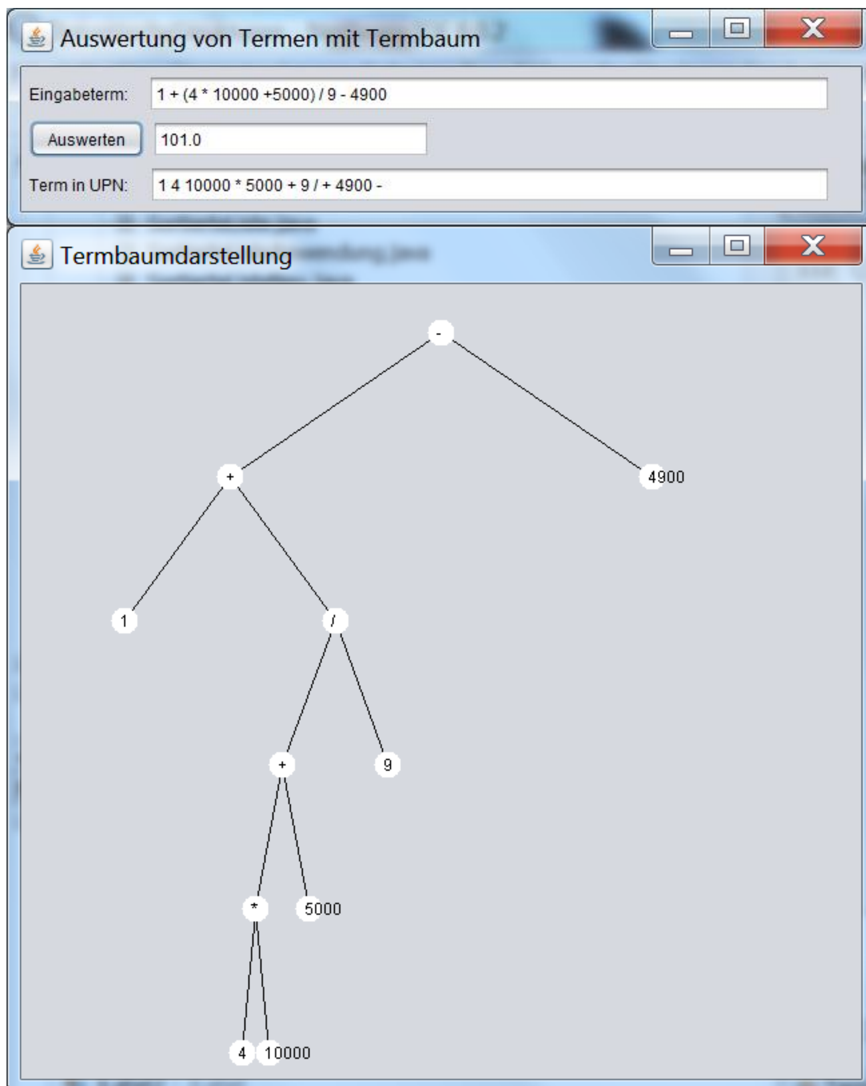
// Der Rest erstellt den BinaryTree, ist daher privat.

```
 private char naechstesZeichen()   {
   while (zTerm.charAt(zPosition) == ' ')  zPosition++;
   return zTerm.charAt(zPosition);
 }


 private boolean istZiffer()  {
   return this.naechstesZeichen() >= '0' &&
                            this.naechstesZeichen() <= '9';
 }


 private BinaryTree<String> baumAusTerm()  {
   char lVorzeichen = this.naechstesZeichen();
   if (lVorzeichen == '+' || lVorzeichen == '-') zPosition++;
   BinaryTree<String> lLinkerBinaryTree =
                                     this.baumAusSummand();
   if (lVorzeichen == '-')
      lLinkerBinaryTree = new BinaryTree<String>("-", new
                 BinaryTree<String>("0"), lLinkerBinaryTree);
   while (this.naechstesZeichen() == '+' ||
                            this.naechstesZeichen() == '-')
     {
      char lOperator = this.naechstesZeichen();
      zPosition++;
      BinaryTree<String> lrechterBinaryTree =
                                     this.baumAusSummand();

      if (lOperator == '+')
         lLinkerBinaryTree = new BinaryTree<String>("+",
                      lLinkerBinaryTree, lrechterBinaryTree);
      else
          lLinkerBinaryTree = new BinaryTree<String>("-",
                      lLinkerBinaryTree, lrechterBinaryTree);
     }
   return lLinkerBinaryTree;
 }
```

```
private BinaryTree<String> baumAusSummand()  {
  BinaryTree<String> lLinkerBinaryTree = this.baumAusFaktor();

  while (this.naechstesZeichen() == '*' ||
                              this.naechstesZeichen() == '/')
  {
     char lOperator = this.naechstesZeichen();
     zPosition++;
     BinaryTree<String> lrechterBinaryTree =
                                          this.baumAusFaktor();

     if (lOperator == '*')
        lLinkerBinaryTree = new BinaryTree<String>("*",
                       lLinkerBinaryTree, lrechterBinaryTree);
     else
        lLinkerBinaryTree = new BinaryTree<String>("/",
                      lLinkerBinaryTree, lrechterBinaryTree);
  }
  return lLinkerBinaryTree;
}



private BinaryTree<String> baumAusFaktor()   {
  if (this.istZiffer())
           return new BinaryTree<String>(this.zahl());
  if (this.naechstesZeichen() == '(')   {
     zPosition++;
     BinaryTree<String> lBinaryTree = this.baumAusTerm();
     if (this.naechstesZeichen() == ')')   {
        zPosition++;
        return lBinaryTree;
     }
     else  return null;
  }
  else  return null;
}
```

```
  private String zahl()   {
    String lZahl = "";
    while (this.istZiffer())  {
       lZahl = lZahl + this.naechstesZeichen();
       zPosition++;
    }
    if (this.naechstesZeichen() == '.')    {
       lZahl = lZahl + this.naechstesZeichen();
       zPosition++;
       if (this.istZiffer())
          while (this.istZiffer())   {
              lZahl = lZahl + this.naechstesZeichen();
              zPosition++;
          }
    }
    return lZahl;
  }

}   // Ende der Klasse TermbaumPflanzer
```

Nun folgt die Implementation des eigentlichen Auswertungsprogramms:



Man erkennt, dass man die Darstellung von großen Zahlen im Programm durchaus noch verbessern könnte.

```
import java.awt.*;    // wird für zweites Formblatt benötigt
public class Termauswertung extends javax.swing.JFrame {

  String term = "";
  Graphics zeichenagent;
  javax.swing.JFrame Plan;

  public Termauswertung() {
     initComponents();

     Plan = new javax.swing.JFrame();
     Plan.setBounds(0,this.getHeight(),
                            this.getWidth(),this.getWidth());
     Plan.setTitle("Termbaumdarstellung");
     Plan.setVisible(true);
     zeichenagent = Plan.getGraphics();
  }



  private void btAuswertenMouseClicked(java.awt…..)    {
    term = tfEingabe.getText();
    TermbaumPflanzer lTermbaumPflanzer = new
                                   TermbaumPflanzer(term);
    BinaryTree<String> lBinaerbaum = lTermbaumPflanzer.baum();
    Termbaum lTermbaum = new Termbaum(lBinaerbaum);

    tfErgebnis.setText(""+lTermbaum.wert());
    tfUPNTerm.setText(lTermbaum.inUPN());

    int xl = 10;
    int xr = Plan.getWidth() - 10;
    // die Koordinaten werden angepasst, so dass es am Rand keine Schwierigkeiten gibt.
    zeichne(lTermbaum.kenntBinaryTree, xl, xr, 80);
  }




  private void zeichne(BinaryTree<String> pBaum,
                                  int xl, int xr, int y)  {
     int deltay = Plan.getHeight()/6; // willkürliche Tiefen-Begrenzung
     int xm;
     String e = "";

     if (! pBaum.isEmpty())    {
        xm = (xl + xr)/2;
        if (! pBaum.getLeftTree().isEmpty())
           zeichenagent.drawLine(xm,y,(xl + xm)/2,y + deltay);
        if (! pBaum.getRightTree().isEmpty())
```

```
            zeichenagent.drawLine(xm,y,(xr + xm)/2,y + deltay);

        // Diese Kanten werden anschließend teilweise von den gefüllten Kreisen
        // überzeichnet.
        zeichenagent.setColor(Color.white);
        zeichenagent.fillOval( xm-10, y - 10, 20, 20);
        zeichenagent.setColor(Color.black);

        e =  pBaum.getContent();
        zeichenagent.drawString(""+ e,xm-3,y+5);

        zeichne(pBaum.getLeftTree(), xl, xm, y + deltay);
        zeichne(pBaum.getRightTree(), xm, xr, y + deltay);
      }
   }   //Ende der Methode


}    // Ende der Klasse Termauswertung
```

Aufgaben zu Termbäumen

1. Zeichne folgende Termbäume:

$3 \cdot 4 + 5$, $\quad 3 + 4 \cdot 5$, $\quad 3 - 4 \cdot \frac{5}{6}$, $\quad 3 + 4 \cdot 5^6$, $\quad (3 \cdot 4 + 5) \cdot (6 \cdot 7 - 8)^9$

Suchbäume

Ein *Suchbaum* oder *geordneter Baum* ist ein spezieller Binärbaum, in dem die Daten geordnet abgelegt sind. Dabei gilt, dass die Daten im linken Teilbaum kleiner als die Daten in der Wurzel und die Daten im rechten Teilbaum größer als in der Wurzel sein müssen. Dies gilt dann auch wieder für alle Teilbäume. Daraus folgt, dass ein Suchbaum jedes Element nur einmal enthalten kann. Alle Teilbäume sind ebenfalls Suchbäume.

Voraussetzung ist natürlich, dass alle Elemente im Baum überhaupt der Größe nach geordnet werden können.

Die Klasse *BinarySearchTree*

Bemerkung: diese Klasse ist noch nicht generisch

In einem Objekt der Klasse *BinarySearchTree* werden beliebig viele Objekte in einem Binärbaum (binärer Suchbaum) entsprechend einer Ordnungsrelation verwaltet. Ein Objekt der Klasse stellt entweder einen leeren Baum dar oder verwaltet ein Inhaltsobjekt sowie einen linken und einen rechten Teilbaum, die ebenfalls Objekte der Klasse *BinarySearchTree* sind. Dabei gilt:

Die Inhaltsobjekte sind Objekte einer Unterklasse von *Item*, in der durch Überschreiben der drei Vergleichsmethoden *isLess, isEqual, isGreater* eine eindeutige Ordnungsrelation festgelegt sein muss.
Alle Objekte im linken Teilbaum sind kleiner als das Inhaltsobjekt des Binärbaumes. Alle Objekte im rechten Teilbaum sind größer als das Inhaltsobjekt des Binärbaumes.
Diese Bedingung gilt auch in beiden Teilbäumen.
Die Klasse *BinarySearchTree* ist keine Unterklasse der Klasse *BinaryTree*, sodass deren Methoden nicht zur Verfügung stehen.

Dokumentation der Klasse *BinarySearchTree*

Konstruktor **`init()`**
nachher Der geordnete Binärbaum existiert und ist leer.

Anfrage **`boolean isEmpty()`**
nachher Diese Anfrage liefert den Wahrheitswert *true*, wenn der Suchbaum leer ist, sonst liefert sie den Wert *false*.

Auftrag **`insertItem(Item pItem)`**
nachher Falls ein mit *pItem* übereinstimmendes Objekt im geordneten Baum enthalten ist, passiert nichts. Andernfalls wird das Objekt *pItem* entsprechend der vorgegebenen Ordnungsrelation in den Baum eingeordnet. Falls der Parameter *null* ist, ändert sich nichts.

Anfrage **`Item search(Item pItem)`**
nachher Falls ein bezüglich der verwendeten Ordnungsrelation mit *pItem* übereinstimmendes Objekt im geordneten Baum enthalten ist, liefert die Anfrage dieses, ansonsten wird *null* zurückgegeben. Falls der Parameter *null* ist, wird *null* zurückgegeben.

Auftrag **`remove(Item pItem)`**
nachher Falls ein bezüglich der verwendeten Ordnungsrelation mit *pItem* übereinstimmendes Objekt im geordneten Baum enthalten ist, wird dieses (nur) aus dem Baum entfernt. Falls der Parameter *null* ist, ändert sich nichts.

Anfrage **`Item getItem()`**
nachher Diese Anfrage liefert das Inhaltsobjekt des Suchbaumes. Wenn der Suchbaum leer ist, wird *null* zurückgegeben.

Anfrage **`BinarySearchTree getLeftTree()`**
nachher Diese Anfrage liefert den linken Teilbaum des Suchbaumes. Der Suchbaum ändert sich nicht. Wenn er leer ist, wird *null* zurückgegeben.

Anfrage	**`BinarySearchTree getRightTree()`**
nachher	Diese Anfrage liefert den rechten Teilbaum des Suchbaumes. Der Suchbaum ändert sich nicht. Wenn er leer ist, wird *null* zurückgegeben.

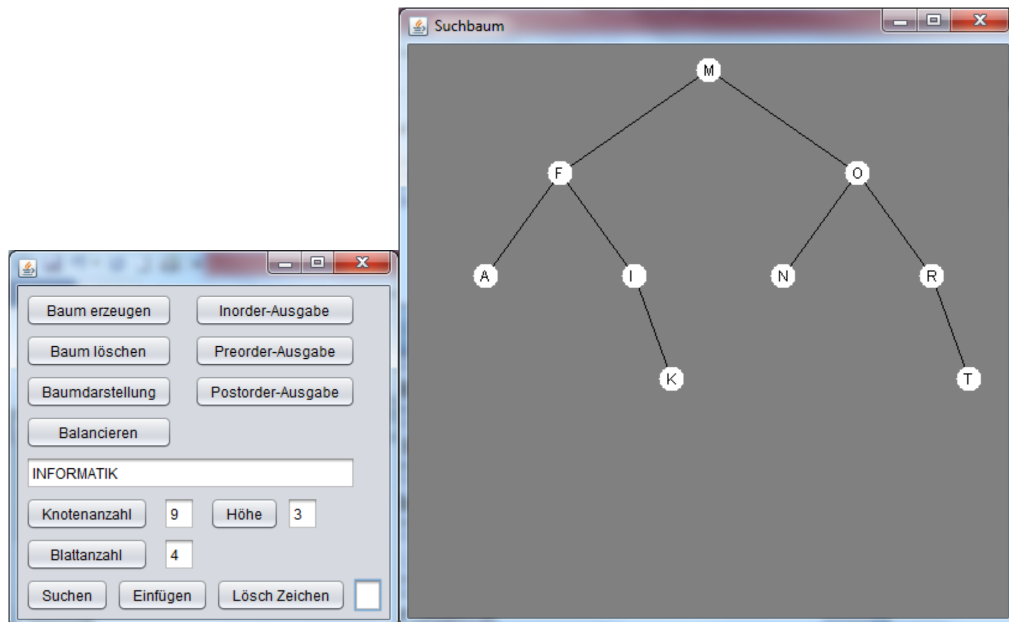
Implementation der Klasse *BinarySearchTree*

```
public class BinarySearchTree {

  BinaryTree binTree;

  public BinarySearchTree()  {
    binTree = new BinaryTree();
  }



  public boolean isEmpty()   {
    return binTree.isEmpty();
  }



  public void insert(Item pItem)  {
    Item lItem;
    BinarySearchTree lTree, rTree;

    if (pItem != null)  {
       if (binTree.isEmpty()) binTree.setObject(pItem);
       else  {
          lItem = (Item) binTree.getObject();
          if (pItem.isLess(lItem))  {
             lTree = this.getLeftTree();
             lTree.insert(pItem);
             this.binTree.setLeftTree(lTree.binTree);
          }
          else  {
             if (pItem.isGreater(lItem))  {
                rTree = this.getRightTree();
                rTree.insert(pItem);
```

```
                this.binTree.setRightTree(rTree.binTree);
              }
           } // of else
        } // wenn Baum nicht leer ist
     }  // wenn pItem != null
  }



public Item search(Item pItem)  {
  Item lItem;
  if (binTree.isEmpty() || (pItem == null)) return null;
  else  {
     lItem = (Item) binTree.getObject();
     if (pItem.isLess(lItem))
                      return this.getLeftTree().search(pItem);
     else if (pItem.isGreater(lItem))
                     return this.getRightTree().search(pItem);
          else return lItem;
  }
}



public Item getItem()  {
  if (this.isEmpty()) return null;
  else return (Item) binTree.getObject();
}



public BinarySearchTree getLeftTree()  {
  BinarySearchTree lTree;
  if (this.isEmpty()) return null;
  else  {
     lTree = new BinarySearchTree();
     lTree.binTree = this.binTree.getLeftTree();
     return lTree;
  }
}
```

```
public BinarySearchTree getRightTree()  {
  BinarySearchTree lTree;
     if (this.isEmpty()) return null;
     else  {
        lTree = new BinarySearchTree();
        lTree.binTree = this.binTree.getRightTree();
        return lTree;
     }
  }



public void remove(Item pItem)  {
  BinaryTree lKnoten, glKnoten;
  BinarySearchTree lTree, rTree;
  Item lInhalt;

  if (! this.isEmpty() && (pItem != null))  {
    lInhalt = this.getItem();
    if (lInhalt.isEqual(pItem))  {
      if (binTree.getRightTree().isEmpty() &&
          binTree.getLeftTree().isEmpty()) binTree.setEmpty();
      else  {
         if (binTree.getRightTree().isEmpty())  {
           lKnoten = binTree.getLeftTree();
           binTree.setObject(lKnoten.getObject());
           binTree.setLeftTree(lKnoten.getLeftTree());
           binTree.setRightTree(lKnoten.getRightTree());
         }
         else  {
            if (binTree.getLeftTree().isEmpty())  {
              lKnoten = binTree.getRightTree();
              binTree.setObject(lKnoten.getObject());
              binTree.setLeftTree(lKnoten.getLeftTree());
              binTree.setRightTree(lKnoten.getRightTree());
            }
            else  {
               glKnoten = binTree.getLeftTree();
               while (! glKnoten.getRightTree().isEmpty())
                  glKnoten = glKnoten.getRightTree();
               binTree.setObject(glKnoten.getObject());
               this.getLeftTree().remove((Item)
                                      glKnoten.getObject());
            } // of else
```

```
            }   // of else
         }  // of else
      }  // of second if
      else  {
        if (lInhalt.isLess(pItem))  {
          rTree = this.getRightTree();
          rTree.remove(pItem);
        }
        else  {
           lTree = this.getLeftTree();
           lTree.remove(pItem);
        }
     }
  }  // of if
}

}  // der Klasse
```



{ In dieser Aufgabe wird nur der spezielle Suchbaum behandelt, der als Inhalt einzelne Zeichen enthält. }

```
import java.awt.*;
import javax.swing.*;

public class BinarySearchTreeAnwendung extends javax.swing.JFrame {

  BinarySearchTree baum;
  Graphics zeichenagent;
  javax.swing.JFrame Plan;
  int elementeAnzahl;    // wird nur für das Ausbalancieren benötigt
  char [] A;  // wird nur für das Ausbalancieren benötigt


  public BinarySearchTreeAnwendung() {
    initComponents();
    baum = null;
    Plan = new javax.swing.JFrame();
    Plan.setBounds(this.getWidth(),0,500,500);
    Plan.setTitle("Suchbaum");
    // Plan.getContentPane().setBackground(Color.WHITE);
    Plan.setVisible(true);
    zeichenagent = Plan.getGraphics();
  }
```

```
private void btErzeugenMouseClicked(java…..)  {
  ZeichenElement e;
  if (baum == null)  baum = new BinarySearchTree();
  e = new ZeichenElement('I');
  baum.insert(e);
  e = new ZeichenElement('N');
  baum.insert(e);
…………

}



private void btDarstellungMouseClicked(java…..)  {
  zeichenagent.setColor(Color.gray);
  zeichenagent.fillRect(0,0,Plan.getWidth(),Plan.getHeight());
  zeichenagent.setColor(Color.black);
  int xl = 10;
  int xr = Plan.getWidth() - 10;
  if (baum != null)  zeichne(baum, xl, xr, 50);
}

public void zeichne(BinarySearchTree pBaum, int xl, int xr, int y) {
  int deltay = Plan.getHeight()/6; // willkürliche Tiefen-Begrenzung
  int xm;
  ZeichenElement e;

  if (! pBaum.isEmpty())  {
     xm = (xl + xr)/2;
     if (! pBaum.getLeftTree().isEmpty())
        zeichenagent.drawLine(xm, y, (xl+xm)/2, y + deltay);
        // Die Kanten werden anschließend teilweise von den Kreisen überzeichnet.
     if (! pBaum.getRightTree().isEmpty())
        zeichenagent.drawLine(xm, y, (xr+xm)/2, y + deltay);
     zeichenagent.setColor(Color.white);
     zeichenagent.fillOval( xm-10, y - 10, 20, 20);
     zeichenagent.setColor(Color.black);
     e = (ZeichenElement) pBaum.getItem();
     if (e != null) zeichenagent.drawString
                              (""+e.getInhalt(), xm-4, y + 5);
     zeichne(pBaum.getLeftTree(), xl, xm, y + deltay);
     zeichne(pBaum.getRightTree(), xm, xr, y + deltay);
  }
}
```

```
private void btSuchenMouseClicked(java…..)  {
  ZeichenElement e = new
               ZeichenElement(tfZeichen.getText().charAt(0));
  ZeichenElement f = (ZeichenElement) baum.search(e);
  if (f == null)  JOptionPane.showMessageDialog(this,
                                          "nicht gefunden");
  else JOptionPane.showMessageDialog(this, "gefunden");
}
```

```
private void btEinfuegenMouseClicked(java…..)  {
  ZeichenElement e = new
               ZeichenElement(tfZeichen.getText().charAt(0));
  if (baum == null) baum = new BinarySearchTree();
  baum.insert(e);
}
```

```
private void btLoeschZeichenMouseClicked(java…..)  {
  ZeichenElement e = new
               ZeichenElement(tfZeichen.getText().charAt(0));
  baum.remove(e);
}
```

```
private void btLoeschenMouseClicked(java…..)  {
   baum = null;
}
```

```
private void btBalancierenMouseClicked(java.....)  {
  A = new char[100];
  elementeAnzahl = 0;
  if (! baum.isEmpty())  inOrderInsFeld(baum);
  baum = null;   // eigentlich überflüssig
  baum = new BinarySearchTree();
  balancieren(1, elementeAnzahl);
}

private void inOrderInsFeld(BinarySearchTree pBaum)  {
  char zeichen;
  ZeichenElement e;
  if (! pBaum.isEmpty())  {
    inOrderInsFeld(pBaum.getLeftTree());
    elementeAnzahl++;
    e = (ZeichenElement) pBaum.getItem();
    zeichen = e.getInhalt();
    A[elementeAnzahl] = zeichen;
    inOrderInsFeld(pBaum.getRightTree());
  }
}

private void balancieren(int li, int re)  {
  int mitte;
  ZeichenElement e;
  if ( li == re)  {
     e = new ZeichenElement(A[li]);
     baum.insert(e);
  }
  else if (li + 1 == re)  {
          e = new ZeichenElement(A[li]);
          baum.insert(e);
          e = new ZeichenElement(A[re]);
          baum.insert(e);
       }
       else  {
          mitte = (li + re)/2;
          e = new ZeichenElement(A[mitte]);
          baum.insert(e);
          if (mitte - 1 >= li) balancieren(li, mitte - 1);
          balancieren(mitte + 1, re);
       }
}
} // Programmende
```

Suchbaumaufgaben

1. Gegeben sei folgende Liste von Buchstaben
J, R, D, G, T, E, M, H, P, A, F, Q, die nacheinander in einen zunächst leeren, binären Suchbaum eingeordnet werden sollen.
Konstruiere den Suchbaum!
Man kann einen inneren Knoten, der zwei Nachfolger besitzt, löschen, indem man ihn z.B. durch das kleinste Element aus seinem rechten Teilbaum ersetzt.
Zeichne obigen Suchbaum, nachdem man derart die Knoten M und D gelöscht hat!