

# Hilfestellung zur Programmierung mit dem Graphentester



Hier findest du die vollständige Dokumentation für die Methoden, die der Graphentester mitbringt.

## Grundsätzliche Struktur

Damit deine Algorithmen überhaupt kompilieren können und vom Graphentester erkannt werden, muss jede Klasse ein paar bestimmte Befehle enthalten. Grundsätzlich gilt folgende Struktur.

```
package eigeneAlgorithmen;

import graph.*;
import algorithmen.*;

public class GraphAlgo_MeineKlasse extends GraphAlgo
{
    Graph g;

    public String getBezeichnung() {
        return "MeineKlasse";
    }

    public void fuehreAlgorithmusAus() {
        g = getGraph();
        // ...
    }
}
```

Folgende Punkte sind dabei wichtig:

- Das package ergibt sich automatisch aus dem Verzeichnis
- Die beiden import-Statements benötigt man, um die bereits vorgefertigten Graphen-Algorithmen nutzen zu können.
- **Der Klassenname muss mit GraphAlgo\_ beginnen**
- Die Klasse muss die Elternklasse GraphAlgo erweitern
- Wegen der müssen die folgenden zwei Klassen geerbt werden:
  - `public String getBezeichnung()`: Darin wird der leserliche Name definiert/zurückgegeben, der in der GUI auf deinen Algorithmus verweist.
  - `public void fuehreAlgorithmusAus()`: Hierin schreibst du deinen Algorithmus (das ersetzt sozusagen die main-Methode).

## Wichtige Tasks

### Aktion auf allen Knoten/Kanten

Soll der Algorithmus mit allen Knoten des Graphen etwas Bestimmtes machen, dann verwendet man die Methode `getAlleKnoten()` bzw. `getAlleKanten()` des Objekts `g` (Klasse `Graph`):

```
import java.util.List;

// Eine Liste aller Knoten anfordern
List<Knoten> alleKnoten = g.getAlleKnoten();

// Schleife über alle Knoten der Liste
for (Knoten aktuellerKnoten: alleKnoten) {
    // mache etwas mit dem aktuellen Knoten, z.B. markieren
    aktuellerKnoten.setMarkiert(true);
    // Ausführung unterbrechen
    step();
}
```

### Aktion auf bestimmten Knoten/Kanten

Soll der Algorithmus mit bestimmten Knoten/Kanten des Graphen etwas bestimmtes machen, dann holt man sich eine Liste aller Knoten/Kanten, die einer Bedingung genügen. Diese Bedingung kann als Prädikat, d.h. einem Lambda-Ausdruck, der `true` oder `false` zurück liefert, angegeben werden.

```
import java.util.List;

// Eine Liste aller markierten Knoten anfordern
List<Knoten> markierteKnoten = g.getAlleKnoten(k->k.isMarkiert());

// Schleife über alle Knoten der Liste
for (Knoten aktuellerKnoten: markierteKnoten ) {
    // mache etwas mit dem aktuellen Knoten, z.B. Markierung löschen
    aktuellerKnoten.setMarkiert(false);
    // Ausführung unterbrechen
    step();
}
```

Diese Beschränkung auf eine Teilmenge mittels eines Lambda-Ausdrucks funktioniert auch bei den Methoden:

- `getAlleKanten(...)`
- `getAusgehendeKanten(...)`
- `getEingehendeKanten(...)`
- `getNachbarknoten(...)`

## Knoten/Kanten sortieren oder Aktion mit Knoten/Kante mit kleinstem/größtem Wert

Möchte man eine Knoten- oder Kantenliste sortieren oder den größten/kleinsten wert ermitteln, kann man folgendermaßen vorgehen:

Zunächst holt man sich eine Liste der Knoten/Kanten. Mit `Collections.sort(...)` kann man die Kanten nach ihrem Gewicht und die Knoten nach ihrem Wert aufsteigend sortieren lassen. Das erste Element der Liste (Index=0) ist dann das mit dem kleinsten Wert, das letzte das mit dem größten Wert (Index=(Länge der Liste-1)).

```
import java.util.List;
import java.util.Collections;

// Liste aller Kanten holen
List<Kante> kanten = g.getAlleKanten();
// Kanten nach Gewicht aufsteigend sortieren
Collections.sort(kanten);
// Kante mit minimalem bzw. maximalem Gewicht ermitteln
Kante min = kanten.get(0);
Kante max = kanten.get(kanten.size()-1);
```

Benötigt man eine andere Sortierung, kann man die Klasse `Comparator` benutzen und angeben nach welchem Wert sortiert werden soll:

```
import java.util.Comparator;

//aufsteigend sortieren
knoten.sort(Comparator.comparing(Knoten::getIntWert));
//absteigend sortieren
knoten.sort(Comparator.comparing(Knoten::getIntWert).reversed());
```

## Arbeiten mit einer ToDo-Liste

In vielen Fällen arbeitet man die Knoten in einer vorher nicht feststehenden Reihenfolge ab. Die Knoten werden während des Algorithmus einer ToDo-Liste hinzugefügt und dann der Reihe nach abgearbeitet, bis die ToDo-Liste leer ist.

```
import java.util.ArrayList;

// ToDo-Liste erzeugen und mit Startknoten füllen
ArrayList<Knoten> todo = new ArrayList<Knoten>();
todo.add(getStartKnoten());

// Solange die ToDo-Liste nicht leer ist...
while(todo.size()>0) {
    // ersten Knoten aus der Liste herausnehmen
    Knoten k = todo.remove(0);

    // markiere ihn als besucht
```

```
k.setBesucht(true);

// füge alle nicht besuchten Nachbarknoten der Liste hinzu
for(Knoten Nachbar : g.getNachbarknoten(k)) {
    if(!Nachbar.isMarkiert() && !todo.contains(Nachbar)) {
        todo.add(0, n); //füge am Anfang der Liste hinzu
        // oder
        // todo.add(n) //füge am Ende der Liste hinzu
    }
}
step(); // Ausführung unterbrechen
}
```

Alternativ kann man statt einer ArrayList auch die ADTs Stapel (Stack) oder Schlange (Queue) benutzen, die automatisch das Einfügen an der richtigen Stelle übernehmen.

## Rekursiv arbeiten

Standardmäßig wird beim Start des Algorithmus einmal die Methode `fuehreAlgorithmusAus()` aufgerufen. Um rekursiv zu arbeiten, benötigt man neben `fuehreAlgorithmusAus()` noch eine weitere Methode - `fuehreAlgorithmusAus()` ruft diese mit dem Startknoten als Parameter auf, im folgenden kann diese Methode sich dann wiederum rekursiv selbst aufrufen:

```
import java.util.ArrayList;
import java.util.Collections;

public void fuehreAlgorithmusAus() {
    rekursiveMethode(getStartKnoten());
}

public void rekursiveMethode(Knoten k){
    // Abbruchbedingung
    if (k.isMarkiert()) {
        // Ausführung unterbrechen
        step();
        return true;
    } else {
        // Aktion mit Knoten durchführen, z.B. markieren
        k.setMarkiert(true);
        // Ausführung unterbrechen
        step();

        // Rekursiver Aufruf mit allen Nachbarknoten
        for(Knoten Nachbar : g.getNachbarknoten(k)) {
            rekursiveMethode(Nachbar);
        }
    }
}
```

## Backtracking

Backtracking lässt sich am leichtesten rekursiv implementieren.

```
import java.util.ArrayList;

public void fuehreAlgorithmusAus() {
    // Starte Backtracking mit Startknoten
    ArrayList<String> loesung = backtracking(getStartKnoten());
    // Wenn Lösung gefunden, dann anzeigen
    if(loesung != null) g.restoreStatus(loesung);
    step();
}

public ArrayList<String> backtracking(Knoten k){
    // Bisher keine Lösung gefunden
    ArrayList<String> loesung = null;

    // Abbruchbedingung: Lösung gefunden
    if (k.isMarkiert()) {
        step(); // Ausführung unterbrechen
        loesung = g.saveStatus(); // Lösung merken
    }
    // Rekursionsschritt
    else {
        // aktuellen Zustand sichern
        ArrayList<String> aktuellerZustand = g.saveStatus();

        // Probiere alle Möglichkeiten
        // hier alle nicht markierten, ausgehenden Kanten
        ArrayList<Kante> ausgehend = g.getAusgehendeKanten(k);
        ArrayList<Kante> nichtMarkiert = g.beschaenkeKantenAuf(ausgehend,
Graph.NICHTMARKIERT, Graph.BELIEBIG);

        for(Kante ausgehendeKante : nichtMarkiert) {
            // Führe Aktionen aus und anzeigen
            k.setMarkiert(true);
            ausgehendeKante.setMarkiert(true);
            step();

            // Rekursion
            Knoten nachbar = ausgehendeKante.getAnderesEnde(k);
            loesung = backtracking(nachbar);

            // Rückschritt
            g.restoreStatus(aktuellerZustand);
            step();
            if(loesung != null) break;
        }
    }
}
```

```
return loesung;  
}
```

## Ausgabe von Informationen/Ergebnissen

Um Informationen oder Ergebnisse eigener Algorithmen auszugeben, bietet der Graphentester drei Möglichkeiten an:

### Java Konsole

Man kann mit `System.out.println()` Dinge auf die Konsole schreiben. Zu beachten ist dabei, dass die JavaFX-Applikation die Konsole nicht automatisch öffnet, wenn man etwas ausgibt - man muss diese im BlueJ-Menü Ansicht oder mit der Tastenkombination CTRL - T öffnen, um die Ausgabe sehen zu können.

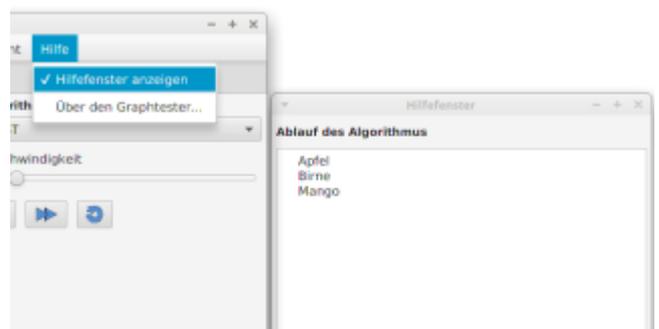
### melde()

Mit der Methode `melde(String Ausgabe)` erzeugt man ein Fenster mit O.K. Button, in dem der Ausgabestring angezeigt wird. Dabei kann man innerhalb des Ausgabestrings auch `\n` verwenden, um eine neue Zeile zu beginnen.

```
melde("Apfel\nBirne\nMango");
```



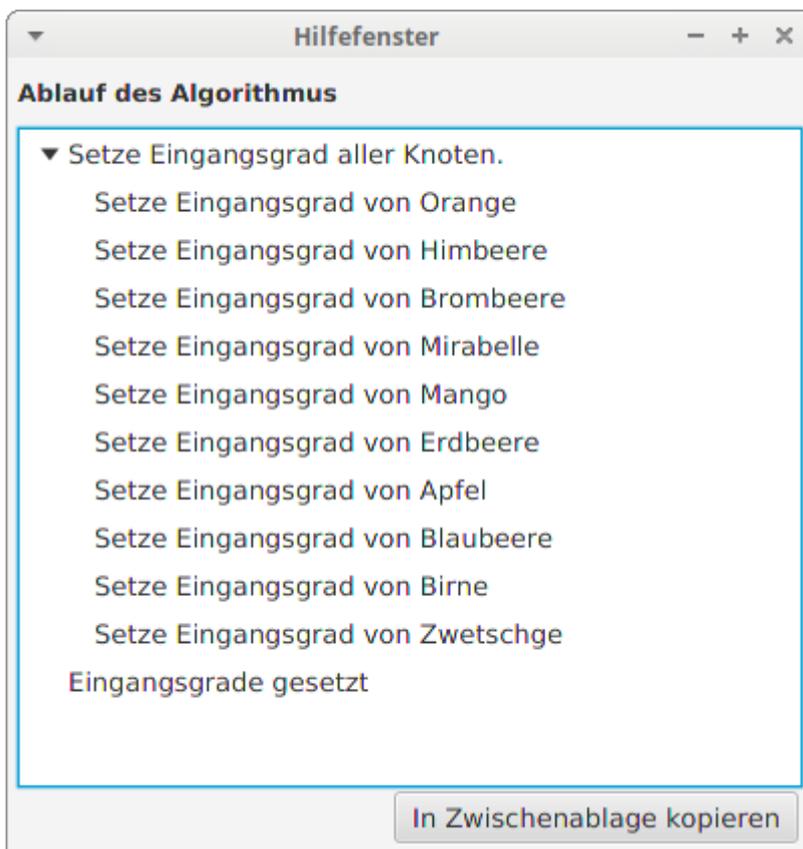
### info()



Mit Hilfe der Methoden `info(String s)`, `resetInfo()`, `infoIndentMore()` und `infoIndentLess()` kann man Informationen - auch während des Programmablaufs - innerhalb des Hilfefensters ausgeben. Das Hilfe-Fenster kann man im Graphentester im Menü Hilfe→Hilfefenster anzeigen aktivieren

Beispiel:

```
info("Setze Eingangsgrad aller Knoten.");
infoIndentMore();
for(Knoten k: alleKnoten) {
    int Eingangsgrad = g.getEingehendeKanten(k).size();
    info("Setze Eingangsgrad von " + k.getInfotext());
    k.setWert(Eingangsgrad);
}
infoIndentLess();
info("Eingangsgrade gesetzt");
```



## Code-Beispiele

### Setze alle Knotenwerte

Alle Knoten kann man direkt in einer Schleife verarbeiten:

```
// Setze alle Knotenwerte auf 0
for(Knoten k : g.getAlleKnoten()) {
    k.setWert(0);
}
```

```
// Setze alle Knotenwerte auf "unendlich"
for(Knoten k : g.getAlleKnoten()) {
    k.setWert(Double.POSITIVE_INFINITY);
}
```

From:  
<https://info-bw.de/> -

Permanent link:  
<https://info-bw.de/faecher:informatik:oberstufe:graphen:zpg:hilfekarten:start?rev=1724938238>

Last update: **29.08.2024 13:30**

