Grundlage der Programmierung II (Java)

Dr. Klaus Höppner

1. Zusammenfassung

Inhaltsverzeichnis

1 Collections

Für Datencontainer unterschiedlichen Typs bietet Java Collections. Diese sind seit Java 5 als Generics definiert, können also außer in der Rohversion (*raw type*) auch für eine Parameterklasse verwendet werden.

Als Collections werden bezeichnet:

sequenzielle Container die in der Regel einen Zugriff über einen Index erlauben. Es existieren aber auch Stack und Queue, die zwar sequenziell sind, aber bei denen nur das erste bzw. letzte Element direkt zugänglich sind, und für die kein Indexzugriff möglich ist.

Mengentypen bei deinen nur relevant ist, ob ein Objekt im Container enthalten ist oder nicht.

assoziative Container die einem Schlüssel einen Wert zuordnen. Diese sind keine Collections im engeren Sinne, da sie nicht das Interface `Collection` implementieren.

1.1 Das Interface Collection

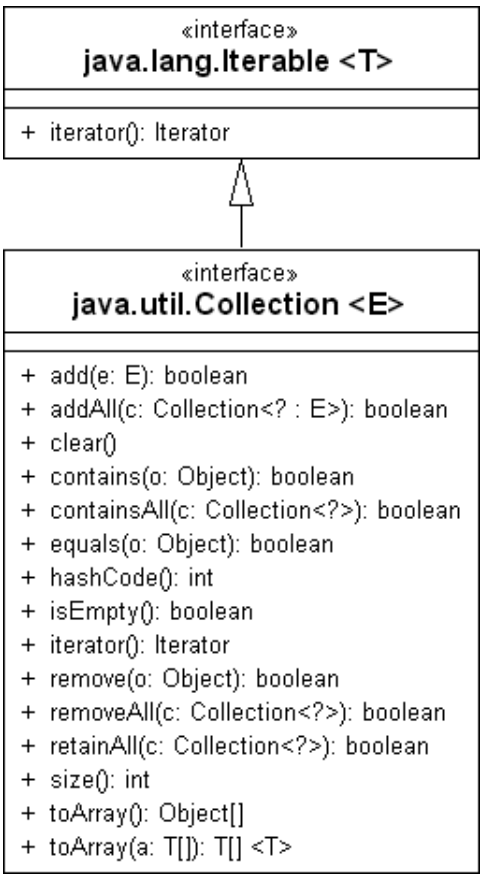

Abbildung 1: Das Interface `Collection`

Das Interface `Collection` bietet im Wesentlichen folgende Möglichkeiten:

- Hinzufügen eines Elementes (mit einem Wahrheitswert als Rückgabewert, da dies nicht bei allen Implementierungen von `Collection` erfolgreich sein muss, z. B. bei Mengen, in denen keine zwei äquivalente Elemente enthalten sein fürfen),
- Entfernen eines Elementes,
- Prüfen, ob ein Element vorhanden ist.

Abb. 1 stellt das Interface als UML-Diagramm dar.

Da `Collection` vom Interface `Iterable` erbt, kann für eine Collection die erweiterte **for**-Schleife verwendet werden. Hierbei sind die beiden Code-Fragmente im folgenden Beispiel äquivalent:

```
Collection<String> c = ...;
for (String s : c) {
    System.out.println(s);
}

Iterator<String> iter = c.iterator();
while (iter.hasNext()) {
    String s = iter.next();
    System.out.println(s);
}
```

Interfaces, die von Collection erben

Von `Collection` erben insbesondere `List` und `Set`. Ersteres ist das Analogon zu konventionellen Arrays, allerdings mit variabler Länge. `Set` stellt hingegen den Mengentyp dar.

Nebenbei existieren noch Interfaces für Stapel und Warteschlangen, die in Abb. 2 mit dargestellt sind.

1.2 Listen

Es existieren zwei Standardimplementierungen von `List`.

- `LinkedList` als doppelt verkettete Liste,
- `ArrayList` die intern auf einem Array basiert, für den bei Bedarf dynamisch mehr Platz geschaffen wird.

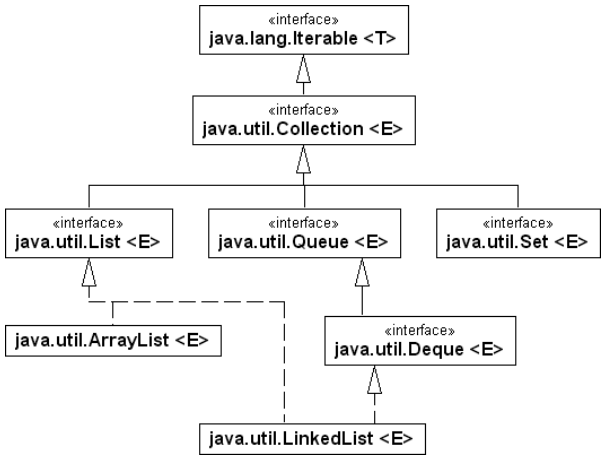

Abbildung 2: Interfaces, die von `Collection` erben.

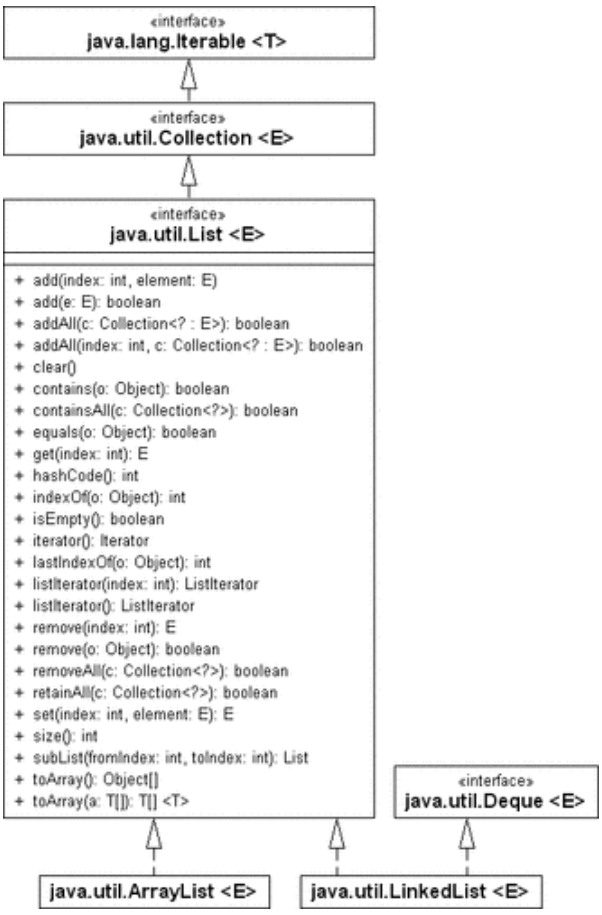

Abbildung 3: Das Interface `List`.

Beide Implementierungen sind insofern komplementär, als sie für verschiedene Situation optimiert sind. Die `ArrayList` hat ihre Stärken beim wahlfreien Zugriff auf Elemente über den Index, leidet aber beim Einfügen und Entfernen von Elementen außer am Ende unter Performanceverlusten. Bei letzterem hat die `LinkedList` Vorteile, da hier im Gegensatz zur `ArrayList` nicht viele Elemente hin und her kopiert werden müssen. Dafür ist der Zugriff auf Elemente über den Index langsam, da hier jeweils vom ersten oder letzten Element aus abgezählt werden muss.

Im Interface `List` kommen indexbezogene Methoden hinzu, die in Abb. 3 dargestellt sind.

1.3 Mengen

Im Interface `Set` kommen keine neuen Methoden hinzu. Eine Menge ist eine spezielle Collection, in der keine zwei äquivalente Elemente enthalten sein dürfen. Ist also das Hinzufügen eines neuen Elementes mit `add` z. B. in eine `List` per Definition immer erfolgreich, so kann es sein, dass `add` bei einer Menge fehlschlägt, da schon ein äquivalentes Element im `Set` enthalten ist. In diesem Fall ist der Rückgabewert von `add` unwahr.

Es existieren zwei Implementierungen von `Set`, die technisch unterschiedlich realisiert sind:

- `HashSet` als eine Menge, bei der für die schnelle Entscheidung, ob ein Element vorhanden ist oder nicht, ein Hashwert verwendet wird. Hierfür werden die Elemente in kleine Päckchen (so genannte *Buckets*) aufgeteilt. Wird jetzt nach einem Element gesucht, so wird zunächst anhand des Hashwerts das Päckchen lokalisiert, in dem sich das Element potenziell befinden könnte. Die Elemente in diesem Päckchen werden dann linear durchsucht.

  Für den Hashwert wird die Methode `hashCode` verwendet, die einen **`int`** zurück gibt. Hierbei gilt die Regel, dass äquivalente Objekt denselben Hashcode haben müssen. Wird also für eine eigene Klasse die Methode `equals` überladen, so muss dies auch für `hashCode` so geschehen, dass diese Bedingung erfüllt ist.

- `TreeSet` verwendet für die Menge einen (ausbalancierten) binären Baum. Damit dies möglich ist, reicht es nicht aus, dass für zwei Objekte vom Elementtyp der Menge definiert ist, wann sie äquivalent sind, sondern es muss eine Ordnungsfunktion existieren. Dies kann die natürliche Ordnung sein, wenn der Elementtyp das Interface `Comparable` implementiert. Alternativ kann die Menge einen `Comparator` verwenden, der für den Elementtyp die Methode `compare` zur Verfügung stellt:

$$\texttt{compare(a, b)} = \begin{cases} -1 & \text{falls } a < b \\ 0 & \text{falls } a = b \\ 1 & \text{falls } a > b \end{cases}$$

1.4 Assoziative Container

Für assoziative Container existiert das Interface `Map`. Da Maps sich auf Schlüssel-Wert-Paare beziehen, ist `Map` eine generisches Interface für zwei Parameterklassen.

Technisch ist eine Map ein Container von Schlüssel-Werte-Paaren. Hierfür ist innerhalb von `Map` das innere Interface `Entry` definiert, vgl. Abb. 4. Auf die Elemente innerhalb der Map sind drei Sichten möglich:

- `entrySet()` gibt die Menge der Schlüssel-Werte-Paare zurück (`Set<Map.Entry<K,V>>`),
- `keySet()` gibt die Menge aller Schlüssel zurück (`Set<K>`),
- `values()` gibt eine Collection aller Werte zurück (`Collection<V>`).

Bezüglich der Zuweisungs-Logik besteht ein erheblicher Unterscheid zwischen `Set` und `Map`. Bei `Set` geht der Versuch schief, ein Element hinzuzufügen, für das schon ein äquivalentes Element enthalten ist. Wird hingegen für `Map` die Methode `put(K, V)` aufgerufen, und es existiert bereits ein äquivalenter Schlüssel, so wird die bestehende Assoziation mit einem Wert durch den neuen Wert ersetzt und der alte als Rückgabewert zurück gegeben (andernfalls **`null`**).

Analog zu `Set` existieren zwei Implementierungen von `Map`;

- `HashMap`, wobei die Optimierung bezüglich des Schlüssels durch Aufteilung in Teilpakete anhand des Hashcodes des Schlüssels geschieht,
- `TreeMap` der als binärer Baum bezüglich des Schlüssels organisiert ist.

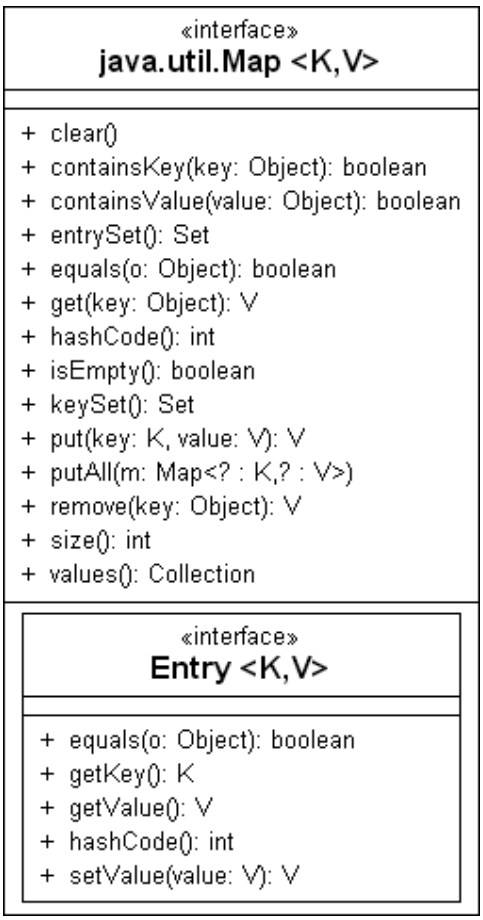

Abbildung 4: Das Interface `Map` mit dem inneren Interface `Entry`.

Die bei `Set` gemachten Aussagen bezüglich Überladen von `hashCode` und der Existenz einer Ordnungsfunktion gelten daher analog für den Schlüsseltyp.

2 Grafische Oberflächen mit Swing

2.1 Einführung

Java besitzt zwei integrierte Toolkits für grafische Oberflächen:

- AWT
- Swing

Hierin liegt ein Vorteil gegenüber anderen Sprachen, wo GUI-Bibliotheken in der Regel Zusatzkomponenten darstellen, die häufig speziell für einzelne Systeme sind. Im Gegensatz hierzu sind mit AWT und Swing erstellte Oberflächen portabel auf verschiedenen Plattformen nutzbar.

Unterschiede zwischen AWT und Swing:

AWT ist das ursprüngliche GUI-Toolkit von Java, das sich bei den unterstützten Komponenten auf den kleinsten gemeinsamen Nenner der unterstüzten Betriebssysteme verlässt, da das Zeichnen der Komponenten dem darunter liegenden System überlassen wird. Als Folge entspricht das Look & Feel der Anwendung derjenigen des Systems.

Swing wurde in Kooperation von Sun und Netscape entwickelt und ist seit Version 1.2 in Java integriert. Hier wird das Zeichnen der Komponenten von den Swing-Bibliotheken selbst realisiert, so dass nur geringe Anforderungen an das darunter liegende System gestellt werden. Swing-Anwendungen sehen auf allen Plattformen identisch aus, weichen dafür aber vom Look & Feel der nativen Anwendungen für das Betriebssystem ab.

Mittlerweile werden Oberflächen in der Regel in Swing implementiert. AWT ist allerdings nicht überflüssig geworden, da Swing an einigen Stellen auf AWT zurück greift (z. B. Eventbehandlung). Es existieren noch weitere GUI-Toolkits, insbesondere das *Standard Wideget Toolkit* (SWT), das innerhalb von Eclipse verwendet wird.

2.2 Toplevelfenster

Swing kennt insgesamt drei *Toplevel*-Elemente:

JFrame ein Fenster mit Rahmen und Titel

Selbst geschriebene Anwendungen mit grafischer Oberfläche befinden sich meist innerhalb eines `JFrame`. Daher erbt die Hauptklasse einer Anwendung oft von `JFrame`.

JDialog für Meldungsfenster

JApplet für Java-Applets, die innerhalb eines (Java-fähigen) Webbrowsers laufen.

Innerhalb eines Toplevel-Fensters können beliebige Komponenten (außer anderen Toplevel-Fenstern) angeordnet werden.

Eine weitere wichtige Komponente ist `JPanel`, die einen Container zur Verfügung stellt, der kein Toplevel-Element ist, und innerhalb einer Anwendung benutzt werden kann, um in einem Unterbereich z. B. einen anderen Layout-Manager zu verwenden.

Zur Platzierung von Komponenten innerhalb eines Fensters wird ein Layout-Manager verwendet. Übliche sind:

FlowLayout ein einfacher Layout-Manager, der die Komponenten einfach von links nach rechts anordnet und daher nur selten verwendet wird.

BorderLayout zur Anordnung der Komponenten nach Himmelsrichtungen.

GridLayout bei dem die Elemente anhand eines Rasters zeilen- und spaltenweise angeordnet werden. Hierbei sind alle Zeilen gleich groß.

GridBagLayout eine flexiblere Form des GridLayouts, da hier Zellen sich auch über mehrere Spalten und Zeilen erstrecken können. Die Größe der Zeilen ist durch Gewichtsfaktoren festgelegt werden kann variabel festlegbar.

Der prinzipielle Quelltext für eine Anwendung sieht so aus, dass in einer eigenen Klasse von einer der Toplevel-Klassen geerbt wird und dort die der LayoutManager eingestellt und das Fenster sichtbar gemacht wird:

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

public class MyApp extends JFrame {
    public MyApp(String title) {
        super(title);
        setLayoutManager(new BorderLayout());

        // Komponenten hinzufügen

        pack();
        setVisible(true);
    }

    public static void main(String[] args) {
        MyApp app = new MyApp("Titel");
    }
}
```

2.3 Swing-Komponenten

Die wichtigsten Komponenten von Swing sind:

JLabel ein normaler Text.

JTextField zur Eingabe einzeiliger Texte.

JEditorPane, JTextPane für mehrzeilige, evtl. editierbare Texte.

JButton für Schaltflächen.

JCheckBox für Ankreuzfelder.

JRadioButton für (sich gegenseitig ausschließende) Optionsfelder.

JComboBox für Pull-Down-Listen zur Auswahl, evtl. mit Möglichkeit zur zusätzlichen Eingabe von freiem Text.

JList für Auswahllisten.

JTable für Tabellen.

JScrollpane für (horizontal und/oder vertikal) scrollbare Bereiche.

2.4 Menüs

Das Toplevel-Fenster `JFrame` kann am oberen Rand eine Menü-Leiste beinhalten. Diese kann über die Methode `addMenuBar(JMenuBar menuBar)` mit einer Instanz von `JMenuBar` als Parameter mit Inhalt gefüllt werden.

Ein Menü kann dabei im Wesentlichen bestehen aus Instanzen von:

JMenu als ein Menüelement, das ein Untermenü öffnet. Dies ist in der Menüleiste die Regel, allerdings kann auch ein `JMenu` wiederum andere Instanzen von `JMenu` enthalten, um Untermenüs zu realisieren.

JMenuItem als ein Menüeintrag, das dann tatsächlich eine Aktion auslöst. Da `JMenuItem` wie `JButton` von `AbstractButton` erbt, stehen hier die selben Methoden wie für eine Schaltfläche zur Verfügung.

Zusätzlich können Toolbars und Kontextmenüs realisiert werden.

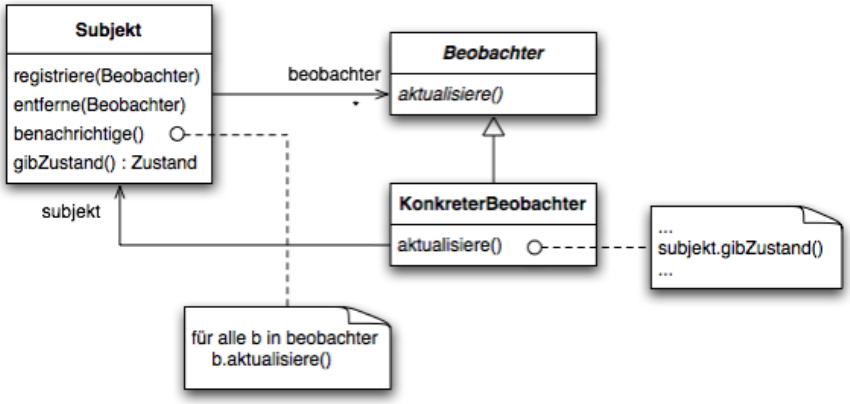

Abbildung 5: Illustration des Observer-Patterns

2.5 Events und deren Behandlung

Bei grafischen Anwendungen findet eine Trennung zwischen dem Layout und der Aktions-Logik statt. Das Layout wird durch die oben beschriebenen Komponenten bestimmt. Für die eigentliche Logik der Anwendung sind durch Aktionen (Tastatur, Mausklicks, Window-Aktionen) ausgelöste Events und davon benachrichtigte Listener verantwortlich.

Dies wird in Java durch das in Abb. 5 dargestellte Observer-Pattern realisiert. Bei den Komponenten werden die Listener als Beobachter angemeldet, beim Auftreten von Events wird für alle vorhandenen Beobachter eine bestimmte Methode aufgerufen. Die folgende Tabelle listet für die verschiedenen Arten von Listenern Methoden auf, die ein entsprechender Listener dann überladen kann.

Listener	Methoden	Parameter	Implementiert durch
ActionListener	`actionPerformed`	`ActionEvent`	

Listener	Methoden	Parameter	Implementiert durch
KeyListener	keyTyped keyPressed keyReleased	KeyEvent	KeyAdapter
MouseListener	mouseClicked mouseEntered mouseExited mousePressed mouseReleased	MouseEvent	MouseAdapter
MouseMotionListener	mouseDragged mouseMoved	MouseEvent	MouseMotionAdapter
MouseInputListener	...	MouseEvent	MouseInputAdapter
	(Kombination aus MouseListener und MouseMotionListener)		
WindowListener	windowActivated windowClosed windowClosing windowDeactivated windowDeiconified windowIconified windowOpened	WindowEvent	WindowAdapter
WindowFocusListener	windowGainedFocus windowLostFocus	WindowEvent	WindowAdapter
ListSelectionListener	valueChanged	ListSelectionEvent	

In den meisten Fällen, z. B. normaler Linksklick auf eine Schaltfläche oder Eingabe in einem Textfeld (mit anschließendem Pressen von „Enter“), reicht ein ActionListener zum Verarbeiten der Events aus. Sollen jedoch z. B. auf mittlere oder rechte Maustaste reagiert werden oder Modifizierer wie gleichzeitig gedrückte „Strg“- oder „Alt“-Taste ausgewertet werden, sind die spezielleren Listener notwendig.

2.6 Exkurs: Innere und anonyme Klassen

In einer Anwendung tritt häufig eine Vielzahl von Events auf. Soll auf diese unterschiedlich reagiert werden, so stehen zwei grundsätzliche Strategien zur Verfügung:

- Wenige Listener lauschen auf viele Events, müssen aber dann in der Behandlung unterscheiden, wo die Quelle des Events liegt und dann darauf reagieren.
- Für jedes Event wird ein eigener Listener verwendet, so dass jweils die Quelle des Events eindeutig ist.

Im zweiten Fall würde die Anwendung aus vielen Klassen (jeweils in eigenen Quelltextdateien) bestehen, wenn diese als reguläre Klassen realisiert werden. Um dies zu vermeiden, besteht die Möglichkeit, Klassen auch als *innere Klassen* oder *anonyme Klassen* zu implementieren.

Eine innere Klasse wird mit ihren Methoden, Attributen und evtl. Konstruktor innerhalb der Klasse definiert. Besonderheit ist hierbei, dass innerhalb der inneren Klasse sogar auf private Attribute der äußeren Klasse zugegriffen werden darf.

Beispiel

```
public class MyApp extends JFrame {
    private JLabel label1;
    private JButton button1;

    public MyApp() {
        ...
        button1.addActionListener(new MyListener());
    }

    class MyListener extends ActionListener {
        @Override
        public void actionPerformed(ActionEvent event) {
            label1.setText("Hallo");
        }
    }
}
```

In einfachen Fällen ist die Definition einer Klasse mit Namen überflüssig, wenn diese nur einmal benötigt wird. Dann kann man eine anonyme Klasse verwenden, die dann von einem Interface oder einer Elternklasse erbt. Für anonyme Klassen gilt ebenfalls, dass sie auf private Attribute der äußeren Klasse zugreifen kann. Besonderheit hier ist, dass die Definition eines Konstruktors nicht möglich ist. Alle dem Konstruktor übergebenen Parameter werden automatisch an den Superkonstruktor übergeben werden. Dies bedeutet, dass bei anonymen Klassen, die direkt ein Interface implementieren, nur der Standardkonstruktor verwendet werden kann.

Beispiel

```
public class MyApp extends JFrame {
    private JLabel label1;
    private JButton button1;

    public MyApp() {
        ...
        button1.addActionListener(new ActionListener() {
            @Override
            public void actionPerformed(ActionEvent event) {
                label1.setText("Hallo");
            }
        });
    }
}
```

2.7 Die Klasse Action

Häufig kann ein und dieselbe Aktion auf verschiedene Wege ausgelöst werden:

- durch einen Menüpunkt,
- durch Klicken auf ein Icon im Toolbar,
- durch eine Tastenkombination (z. B. `Strg-O`)

Um dies zu vereinfachen, existiert das Interface `Action`, das vom Interface `ActionListener` erbt und von der Klasse `AbstractAction` implementiert wird. Hierbei muss die abstrakte Methode `actionPerformed` überladen werden. Weiterhin können Icon, Tooltiptext, Kurzwahltaste etc. gesetzt werden, die dann für verschiedene Komponenten verwendet werden. Hierfür dient die Methode `putValue(String, Object)` :

`NAME`	Name (benutzt für Menüeinträge, Buttons, Checkboxen, ...)
`SMALL_ICON`	Icon (für Menüeinträge, Buttons, Checkboxen, ...)
`LONG_ICON_KEY`	Icon (für Buttons, Checkboxen, ...)
`ACCELERATOR_KEY`	Shortcut für Menüeinträge
`MNEMONIC_KEY`	Mnemonic (für Buttons, Checkboxen, ...)
`SHORT_DESCRIPTION`	Tooltip-Text
`ACTION_COMMAND_KEY`	String für ActionCommand

2.8 Listen und Tabellen

Durch die Swing-Komponenten `JList` und `JTable` können Listen bzw. Tabellen grafisch dargestellt werden. Weiterhin sind beide ein Beispiel dafür, dass Widgets mit Präsentations- und Kontrollschicht von dem Datenmodell getrennt werden können.

Listen

Bei Auswahllisten, die mit `JList` realisiert sind, gibt es zwei Auswahlmodi:

- Einzelselektion von Elementen,
- Mehrfachselektion von Elementen möglich.

Da in der Praxis die Einzelselektion die Regel ist, wird hier nur diese behandelt.

Um eine `JList` mit einem Datenmodel zu verknüpfen, wird dem Konstruktor von `JList` eine Instanz einer Klasse übergeben, die das Interface `ListModel` implementiert. In vielen Fällen wird einfach die Klasse `DefaultListModel` verwendet, die sich bezüglich des Hinzufügens, Ersetzen und Entfernen von Datenelementen lose am `List`-Interface aus den Java-Collections orientiert.

Das Datenmodell bestimmt dabei, aus wievielen und welchen Elementen die Auswahlliste besteht, wobei man im Fall von `DefaultListModel` die Listenelemente einfach nach und nach

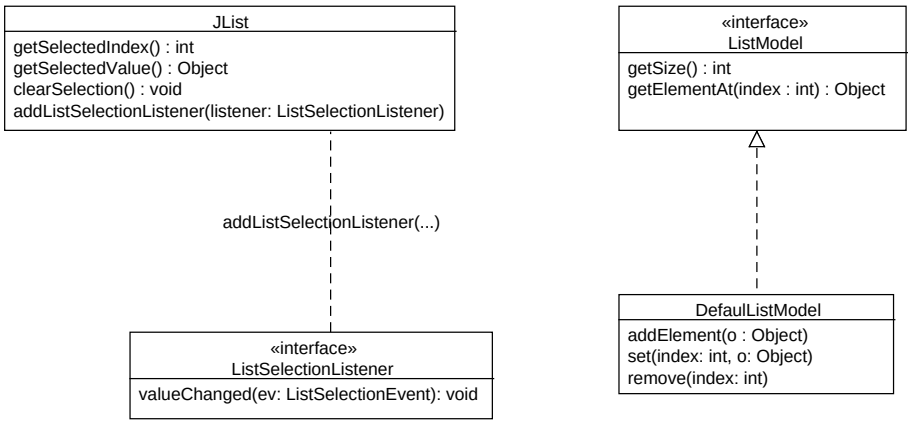

Abbildung 6: Trennung von Layout in `JList` und Daten der Auswahlelement im `ListModel`.

mit der Methode `addElement` hinzu fügen kann. Die grundlegende Struktur von `JList` und Datenmodell ist in Abb. 6 dargestellt.

Mit der Methode `addListSelectionListener` kann der `JList` ein Listener hinzugefügt werden, für den bei Auswahl eines Elementes die Methode `valueChanged` aufgerufen wird. Dieser wird ein `ListSelectionEvent` als Parameter übergeben. Im folgenden Beispiel wird in der Auswahlliste beim Selektieren eines Listenelements jeweils zwischen Groß- und Kleinschreibung des Wortes gewechselt.

Beispiel für eine Auswahliste

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

public class MyList extends JFrame implements ListSelectionListener {
    private DefaultListModel data = new DefaultListModel();
    private JList list = new JList(data);

    public MyList() {
        setLayout(new BorderLayout());
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        data.addElement("hallo");
        data.addElement("welt");
        data.addElement("echo");
        data.addElement("otto");
        add(list,BorderLayout.CENTER);
        list.addListSelectionListener(this);
        pack();
        setVisible(true);
    }
```

```
    public void valueChanged(ListSelectionEvent event) {
        if (event.getValueIsAdjusting()) return;

        int index = list.getSelectedIndex();
        if (index != -1) {
            String s = (String) list.getSelectedValue();
            if (s.equals(s.toLowerCase())) {
                data.set(index, s.toUpperCase());
            } else {
                data.set(index, s.toLowerCase());
            }
            list.clearSelection();
        }
    }
}
```

JTable

Eine `JTable` wird als Tabelle dargestellt. Auch hier findet eine Verknüpfung mit einem Datenmodell statt. Dieses wird durch das Interface `TableModel` beschrieben, das folgende Methoden abstrakt definiert:

int getColumnCount() Zahl der darzustellenden Spalten,

String getColumnName(int col) Überschrift der Spalte `col`,

int getRowCount() Zahl der darzustellenden Zeilen],

Object getValueAt(int row, int col) der in der entsprechenden Zeile und Spalte darzustellende Wert,

boolean isCellEditable(int row, int col) gibt an, ob die entsprechnde Zelle editiert werden kann,

void setValueAt(int row, int col, Object o) wird nach dem Editieren einer Zelle mit dem neuen Objekt o aufgerufen.

Dieses Interface wird von der Klasse `AbstractTableModel` implementiert, analog zur Klasse `DefaultListModel` gibt es die Klasse `DefaultTableModel`, die ein einfaches Datenmodell auf Basis einer zweidimensionalen `List` implementiert (s. Abb. 7).

Im Folgenden findet sich der Quelltext für eine Klasse `QuadratTabelle`, die das Datenmodell für eine x-y-Tabelle mit $y = x^2$ implementiert und anschließend die Anwendung in einem grafischen Fenster.

Quelltext: `QuadratTabelle.java`

```
import javax.swing.table.AbstractTableModel;

public class QuadratTabelle extends AbstractTableModel {
    private int count;
```

```
public QuadratTabelle(int count) {
    this.count = count;
}

public int getColumnCount() {
    return 2;
}

@Override
public String getColumnName(int col) {
    String colname = null;
    switch (col) {
    case 0:
        colname = "x";
        break;
    case 1:
        colname = "x^2";
        break;
    }
    return colname;
}

public int getRowCount() {
    return count;
```

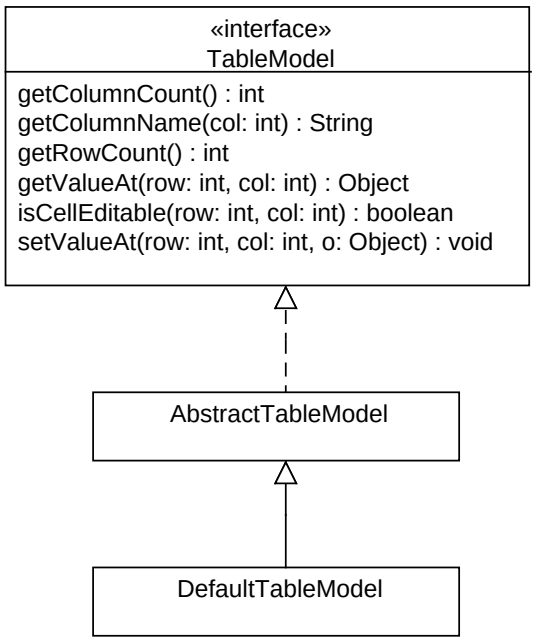

Abbildung 7: Hierarchie der Datenmodell-Klassen für Tabellen

```
    }

    public Object getValueAt(int row, int col) {
        Integer result = new Integer(0);
        switch (col) {
        case 0:
            result = row;
            break;
        case 1:
            result = row*row;
            break;
        }
        return result;
    }

    @Override
    public boolean isCellEditable(int row, int col) {
        return false;
    }

}
```

Quelltext: MyTable.java

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

public class MyTable extends JFrame {
    private TableModel data = new QuadratTabelle(10);
    private JTable table = new JTable(data);

    public MyTable() {
        setLayout(new BorderLayout());
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        add(table,BorderLayout.CENTER);
        pack();
        setVisible(true);
    }
}
```

3 Threads

3.1 Einführung

Neben Multitasking (quasi-parallele Ausführung unabhängiger Prozesse mit eigenem Adressraum) unterstützen moderne Betriebssysteme auch *Multithreading*.

Hier laufen innerhalb eines Programms mehrere Ausführungsstränge (*Threads*, dt. Fäden) nebenläufig ab.

Diese Thread besitzen zwar einen eigenen Stack, teilen sich aber die sonstigen Ressourcen des Programms. Da die Threads den gleichen Adressraum verwenden, ist somit eine direkte Kommunikation zwischen Threads möglich. Threads können sich so aber auch negativ beeinflussen. Daher müssen manchmal kritische Bereiche geschützt werden, was wieder zu Verklemmungen (*deadlock*) führen kann.

Java unterstützt seit jeher Threads, sogar unter Betriebssystemen, die nicht multithreaded sind (dann übernimmt die JVM die Threadverwaltung, d. h. Threadumschaltung einschließlich Stackverwaltung).

3.2 Benutzen von Threads in Java

In Java existiert die Klasse `Thread`, die ein Objekt erwartet, das das Interface `Runnable` implementiert.

Das Interface `Runnable` definiert nur eine einzige Methode, nämlich **`void`** `run()`. Deren Implementation enthält dann die Funktionalität, die dann innerhalb eines Threads ablaufen soll.

Beispiel: Runnable

```
Runnable r = new Runnable() {
    public void run() {
        while (!Thread.currentThread().isInterrupted()) {
            System.out.println(new java.util.Date());
        }
    }
};

Thread t = new Thread(r);
t.start();
```

Wird für einen Thread die Methode `start()` aufgerufen, so führt dieser für das verwaltete Objekt vom Typ `Runnable` die Methode `run()` nebenläufig aus.

Tatsächlich implementiert `Thread` selbst auch `Runnable`. Der vorherige Quellcode würde daher kürzer lauten:

Beispiel: Thread

```
Thread t = new Thread() {
    public void run() {
        while (!isInterrupted()) {
            System.out.println(new java.util.Date());
        }
    }
};
t.start();
```

Ein Thread kann per `interrupt()` zum Beenden aufgefordert werden. Dass ein Thread dieser Aufforderung folgt, muss explizit implementiert sein, wie im vorigen Beispiel die **while**-Schleife endet, sobald `isInterrupted()` wahr ist.

Die JVM läuft für eine Anwendung so lange, bis alle Thread beendet sind. Es kann daher vorkommen, dass auch nach dem Ende von `main(...)` die Anwendung weiter läuft. Ein Sonderfall stellen hierbei Dämonen dar, d. h. Threads, die *vor* dem Start mit `setDaemon(true)` als Daemon gekennzeichnet werden. Diese werden von der JVM bei der Prüfung auf aktive Threads nicht mitgezählt. Sobald alle normalen Threads (*user threads*) beendet sind, endet also die JVM und alle *daemon threads* werden gestoppt.

3.3 Kritische Bereiche

Da bei Threads nur der Stack von `run()` lokal ist, während sie sich die anderen Ressourcen teilen, kann es zu Konflikten kommen, wenn mehrere Threads gleichzeitig auf bestimmte Daten zugreifen.

Bereiche, in denen sicher gestellt werden muss, dass sie jeweils nur von einem Thread abgearbeitet werden, nennt man *kritische Bereiche*.

Java kennt dabei zwei wesentliche Methoden zum Schutz kritischer Bereiche:

- Direkt über ein `Lock`-Objekt,
- über ein **synchronized** für einen Bereich mit einem beliebigen Objekt als Monitor.

Das Interface `Lock` stellt (unter anderem) die Methoden `lock()` und `unlock()` zur Verfügung. Über ein Objekt, das das Interface `Lock` implementiert (meist vom Typ `ReentrantLock`) kann dann ein kritischer Bereich beim Betreten durch einen Thread verriegelt werden. Ein anderer Thread, der diesen Bereich (oder einen sonstigen über denselben `Lock` geschützten kritischen Bereich) betreten will, wartet dann, bis der erste Thread den kritischen Bereich abgearbeitet hat und den `Lock` wieder entriegelt.

Bei **synchronized** werden kritische Bereiche über ein Objekt (häufig sogar von der Klasse `Object`) geschützt. Hierbei lautet die Syntax:

```
synchronized (o) {
    ...
}
```

wobei o das Objekt ist, das für die Synchronization des kritischen Bereichs benutzt wird.

Es kann auch eine ganze Methode als **synchronized** definiert werden. Hierbei entspricht z. B.

```
public synchronized void foo(o) {
    ...
}
```

letztlich

```
public void foo(o) {
    synchronized (this) {
        ...
    }
}
```

Beim Schützen von kritischen Bereichen muss beachtet werden, dass es nicht zu Verklemmungen (*dead locks*) kommen darf, weil mehrere Threads sich gegenseitig blockieren.

3.4 Timer

Periodisch auszuführende Aktionen können zwar direkt über einen Thread implementiert werden, bequemer geht dieses aber über Timer, die intern auf Threads aufsetzen.

Die universellste Form von `Timer` befindet sich im Paket `java.util`:

```
java.util.Timer timer = new java.util.Timer(false);
timer.schedule(new TimerTask() {
    @Override
    public void run() {
        System.out.println(new java.util.Date());
    }
},100,500);
```

Obiges Besipiel gibt nach einer Verzögerungszeit von 100 ms alle 0,5 s auf der Konsole das aktuelle Datum und Uhrzeit aus. Der Parameter **false** gibt an, dass es sich nicht um einen User-, sondern einen Daemon-Thread handelt.

Nebenbei existiert noch die Klasse `javax.swing.Timer`, die auf einem ActionListener aufsetzt. Während der Timer aus `java.util` allgemeiner ist, hat der Swing-Timer den Vorteil, dass sich mehrere einen Thread teilen können.

3.5 Signale

Als Erweiterung können `Lock`-Objekte auch zum Austausch von Signalen zwischen Threads benutzt werden. Hierfür stellt Java seit Version 5 das Interface `Condition` zur Verfügung, wobei `Lock` die Methode `newCondition` zum Anlegen eines solchen Objektes bereit stellt.

Das grundlegende Vorgehen sieht wie folgt aus:

- Erzeugen eines Lock-Objekts `lock`
- Erzeugen eines Condition-Objekts:

  ```
  Condition cond = lock.newCondition()
  ```

- Threads dürfen nur dann in Warteposition geschoben werden, wenn sie den Lock besitzen. Es muss also zunächst `lock.lock()` aufgerufen werden!
- Mit `cond.await()` wird gewartet (und der Lock wieder freigegeben!), bis der Thread von einem anderen mit `cond.signal()` bzw. `cond.signalAll()` benachrichtigt wird.
- Der Thread wartet nun, bis er den Lock wieder hat und setzt die Arbeit fort.
- Am Ende das `lock.unlock()` nicht vergessen!

Hierbei wird mit `cond.signal()` *ein* Thread aufgeweckt wird, der auf das entsprechende Signal wartet, bei `signalAll()` alle wartenden Threads hintereinander.

Signale zwischen Threads werden oft bei Szenarien eingesetzt, in denen einige Threads Daten produzieren (Erzeuger) und andere diese Daten entgegen nehmen und verarbeiten (Verbraucher).