Helge Peters, Matthias Taulien, Georg Wilke

Science-Academy Baden-Württemberg 2004
Sommerakademie für die Mittelstufe

ROBOTIK MIT JAVA

Anleitungen

Mai 2004

Impressum

Autoren: Dipl.-Ing. Helge Peters
Universität Karlsruhe
Institute of Process Control and Robotics
E-Mail: peters@ira.uka.de

OStR Matthias Taulien
Carl-Friedrich-Gauß-Gymnasium, Hockenheim
Hector-Seminar, Mannheim
E-Mail: Taulien@hector-seminar.de

OStR Georg Wilke
Bunsen-Gymnasium, Heidelberg
Hector-Seminar, Heidelberg
E-Mail: Wilke@hector-seminar.de

Vorwort

Dieses Manuskript wurde für die Teilnehmerinnen und Teilnehmer des Robotik-Kurses an der Sommerakademie für die Mittelstufe der Science-Academy Baden-Württemberg 2003 geschrieben. Es sollen dort kleine *LEGO-Mindstorms*- oder *Fischer-Technik*-Roboter gebaut werden, die mit Hilfe der Programmiersprache Java gesteuert werden. Das Manuskript dient in erster Linie als Nachschlagewerk und als Gedächtnisstütze für die Nachbereitung zu Hause, was vor allem die Bezugsquellen der verschiedenen verwendeten Werkzeuge betrifft, als auch deren Installation und Konfiguration beschreiben soll. Im Anhang ist eine ausführliche Liste der verschiedenen Internet-Adressen der verwendeten Dateien aufgelistet. Zu diesem Manuskript wird eine CD angeboten, die das Herunterladen aus dem Internet erspart. Manche dieser Dateien sind einer häufigen Änderung unterzogen, so dass man hin und wieder nach entsprechenden Updates im Internet nachschauen sollte. Manche der Dateien auf der CD sind von den Autoren selbst erstellt. Sie enthalten entweder Ergänzungen oder Korrekturen der zu den Programmen aus dem Internet.

Da alle Programme, die im Manuskript erwähnt sind, sich auf der CD befinden bzw. deren Internet-Adresse im Anhang aufgeführt ist, wurde darauf verzichtet, die Bezugsadresse im Text extra zu erwähnen. Das macht die Beschreibungen etwas schlanker.

Dieses Manuskript kann keine Programmieranleitung oder Java-Einführung sein. Das würde den Umfang zu sehr vergrößern. Vielleicht ergibt es sich aber, dass die Teilnehmerinnen und Teilnehmer selbst ihre Erfahrungen, Anleitungen, Tipps und Programme aufschreiben möchten. Dann würden wir uns freuen, wenn sie Ihre Materialien als Ergänzung zu diesem Manuskript zur Verfügung stellen würden.

Helge Peters, Matthias Taulien und Georg Wilke

Heidelberg im August 2003

Inhaltsverzeichnis

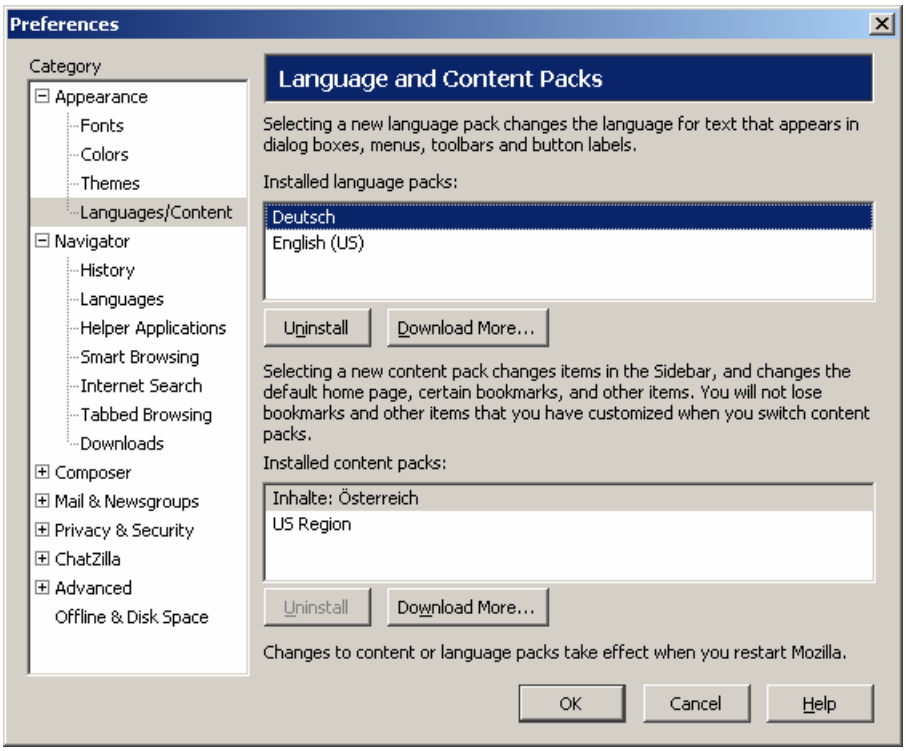
1 Installationsanleitung

Java und alle für die Programmierung der *LEGO-Mindstorm*-Roboter oder der *Fischer-Technik*-Roboter benötigten Programme gibt es kostenlos im Internet. Die Internetadressen sind im Anhang aufgelistet.

1.1 Installation von *Mozilla*

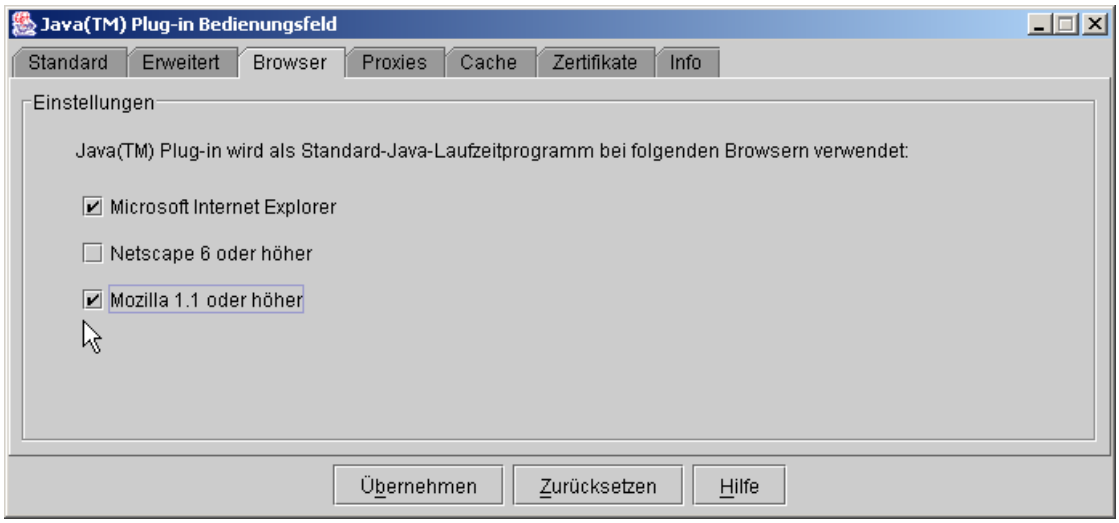
Will man neben Java-Anwendungen auch Applets erstellen, die in einem Browser betrachtet werden sollen, ist es sinnvoll, wenn man *zuvor* einen Applet-freundlichen Browser wie *Mozilla* installiert hat. Der *Internet-Explorer* (IE) von *Microsoft* ist für das Ausführen von Applets nur schwer zu konfigurieren. Aus diesem Grund startet die Installationsanleitung mit der Installation von *Mozilla.*
Durch Doppelklick auf die Datei `mozilla-win32-1.6-installer.exe` startet man die Installation des Web-Browsers. Es wird zunächst die englischsprachige Version installiert. Man sollte bei den Installationsoptionen CUSTOM(= BENUTZERDEFINIERT) auswählen, da man dann die Installation am besten kontrollieren kann.

Nach Abschluss der Installation startet man *Mozilla*. Möglicherweise wird man gefragt, ob man *Mozilla* zum Standard-Browser machen möchte, die Antwort sei jedem Benutzer selbst überlassen.
Damit *Mozilla* auch deutsch sprechen kann, öffnet man im Browser einfach über FILE – OPEN FILE die Datei `mozilla-1.6-lang-de-AT.xpi` auf der CD.
Daraufhin fragt *Mozilla* nach, ob das deutschsprachige Paket installiert werden soll, was man entsprechend bestätigt. Unter dem Menüpunkt EDIT → PREFERENCES und dort unter CATEGORY → APPEARANCE unter dem Punkt LANGUAGE/CONTENT wählt man Deutsch als Sprache und Österreich als Inhaltspaket. Beim nächsten Start erscheint *Mozilla* dann in Deutsch.



Bemerkung: Wurde *Mozilla* **nach** der Installation des *JDK* installiert, ist die Unterstützung von Java-Applets noch nicht eingerichtet. Dazu startet man in der Systemsteuerung das Java-Plugin Bedienungsfeld,

wählt die Registerkarte BROWSER



und setzt in das Kontrollkästchen MOZILLA 1.1 ODER HÖHER ein Häkchen. Anschließend klickt man auf ÜBERNEHMEN. Sollte der Eintrag *Mozilla* mit Kontrollkästchen nicht erscheinen, ist die Installation von *Mozilla* nicht erfolgreich verlaufen und sollte nochmals wiederholt werden.

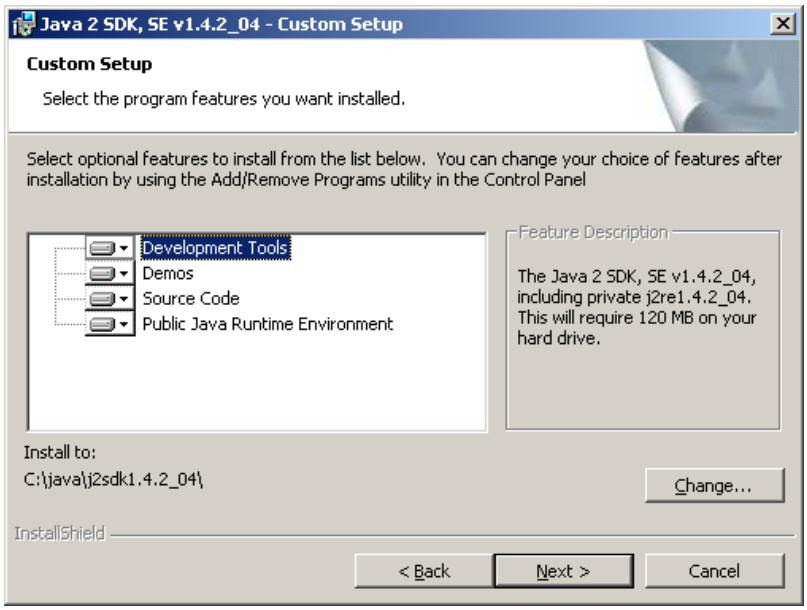
1.2 Java Grundinstallation

1.2.1 Das *Java Development Kit* (JDK) installieren

Um mit Java programmieren zu können, benötigt man das *Java Development Kit* (JDK) von der Firma *Sun Microsystems*, aktuell ist gerade die Version 1.4.2_04. Es enthält den Java-Compiler, der die erstellten Programme für den Computer übersetzt (compiliert), den Java-Interpreter, der den vom Compiler erstellten Byte-Code ausführt und viele andere für den Programmierer benötigten Hilfsprogramme.
Das *JDK* sollte nicht mit dem *JRE* (Java Runtime Environment) verwechselt werden. Das *JRE* ist nur für die Ausführung von Java-Programmen zuständig, man kann mit ihm keine Java-Programme erstellen.
Das *JDK* gibt es für viele Plattformen. Um es auf *Windows*-Rechnern zu installieren, benötigt man die Datei `j2sdk-1_4_2_04.windows-i586-p.exe` und die zugehörige Dokumentation `j2sdk1_4_2-doc.zip`. Mit einem Doppelklick auf `j2sdk-1_4_2_04.windows-i586-p.exe` startet man die Installation.
Im Folgenden gehen wir davon aus, dass alle zum Java-Paket gehörigen Anwendungen in das Verzeichnis `C:\java\` installiert werden sollen. Für alle selbst geschriebenen Java-Programme sollte man noch ein separates Verzeichnis erzeugen. Es kann durchaus auch auf einem anderen Laufwerk liegen wie z. B. `D:\Java`. Letzteres wird in dieser Anleitung verwendet.

Wichtig: Alle Ordner, die Java-Dateien enthalten, und Dateinamen von Java-Programmen sollten keine Leer- oder Sonderzeichen wie z. B. Umlaute oder ß enthalten

Als Installationspfad für JDK wählt man also `C:\java\j2sdk1.4.2_04` und beantwortet alle Fragen entsprechend mit „Ja“.



1.2.2 Installation der Online-Dokumentation des JDK

Da man die Fülle der Klassen und Methoden, die Java mitbringt, unmöglich alle wissen kann, benötigt man die zugehörige Online-Dokumentation. Dazu packt man die Datei `j2sdk1_4_2-doc.zip` in den Ordner `C:\java\j2sdk1.4.2_04` aus, wobei automatisch der Ordner `C:\java\j2sdk1.4.2_04\docs` erzeugt wird.

1.2.3 Die Java-Unterstützung für die serielle und parallele Schnittstelle installieren

Wenn man mit Java auf die Hardware-Schnittstellen zugreifen will, benötigt man das Paket *Communications API*. Alle nötigen Dateien sind in `javacomm20-win32.zip` enthalten. Diese packt man am besten nach `C:\java` aus. Dabei wird der Ordner `C:\java\commapi` erstellt.
Danach wird die Datei `C:\java\commapi\win32com.dll` nach `C:\java\j2sdk1.4.2\bin` kopiert. Ebenso kopiert man die beiden Dateien `C:\java\commapi\comm.jar` und `C:\java\commapi\javax.comm.properties` nach `C:\java\j2sdk1.4.2\lib`.

1.2.4 Den *Jikes*-Compiler installieren

Jikes ist ein alternativer Compiler von *IBM*. Man benötigt ihn nicht unbedingt. Allerdings wird auf der Webseite zu *Jikes* behauptet, dass er viel strenger die Java-Spezifikationen einhalte als der mit dem *JDK* mitgelieferten Compiler `javac.exe`. Auf jeden Fall compiliert er viel schneller als der Standard-Compiler `javac.exe`.
Die Installation ist einfach, man packt die Datei `jikes-1.20-1.windows.zip` nach `C:\java` aus. Dabei wird der Ordner `C:\java\jikes-1.20` erstellt.

1.2.5 Eigene Java-Pakete installieren

Auf der CD werden zwei Pakete `eingabe` und `turtle` mitgeliefert, die zum einen einige hilfreiche Methoden zur Eingabe von Werten von der Konsole aus, zum anderen Übungen zum Einstieg in die Java-Programmierung enthalten. Man packt daher die Datei user.zip in den Ordner `C:\java` aus. Dabei werden die Ordner `C:\java\user\lib` und `C:\java\user\docs` erstellt. Der erste Ordner enthält die Paket-Dateien, der zweite Ordner die zugehörigen Dokumentationen.
In `C:\java\user\lib\<paketname>` kann man je nach Bedarf weitere selbst erstellte Pakete ablegen. Die zugehörige Dokumentation speichert man unter `C:\java\user\docs\<paketname>`.

1.2.6 Installation wichtiger Tutorials

Auf der Webseite von *Sun Microsystems* findet man für alle Bereiche in Java entsprechende Tutorials. Sie enthalten sehr weitreichende Informationen und Hilfestellungen, wenn auch auf Englisch. Sie sind alle im html-Format verfasst und können somit direkt in einem Browser online gelesen werden. Die meisten Tutorials werden auch zum Download angeboten. Das Standard-Tutorial von *Sun* ist in der Datei `tutorial.zip` enthalten. Beim Auspacken muss man aufpassen, dass man es in einen eigenen Ordner auspackt um nicht die Übersicht über die vielen einzelnen Unterordner zu verlieren. Man kann die Datei `tutorial.zip` z. B nach `C:\java\Tutorials\JavaTutorial` auspacken. Zum Lesen öffnet man die Datei `C:\java\Tutorials\JavaTutorial\index.html`.

Ein anderes Tutorial ist auf Deutsch. Es ist das *Java-Buch* von Guido Krüger. Es kommt in zwei Dateien daher:
`hjp3html.zip` packt man nach `C:\java\Tutorials\JavaBuch\html` aus und
`hjp3exam.zip` nach `C:\java\Tutorials\JavaBuch\examples`.
Zum Lesen öffnet man die Datei `C:\java\Tutorials\JavaBuch\html\cover.html`.

1.3 Java und *LEGO-Mindstorms*

Achtung: Die folgenden Installationen sind nur notwendig, wenn man Lego-Mindstorms-Roboter programmieren möchte!

1.3.1 Installation von *leJOS* für *LEGO*

leJOS kann mit dem *JDK* von *Sun Microsystems* verglichen werden. *leJOS* enthält einen Compiler `lejosc.exe` analog zu `javac.exe`, der einen Byte-Code erzeugt, und einen Interpreter `lejos.exe` der ähnlich wie `java.exe` diesen Byte-Code liest und ihn zum *LEGO*-RCX-Baustein überträgt.
Bei der Installation des *JDK* wurde schon eine JVM (Java Virtual Machine) installiert, die der einzige Plattform abhängige Teil von Java ist und dafür sorgt, dass der Byte-Code auf den jeweiligen Plattformen tatsächlich ausgeführt werden kann. Ebenso benötigt der RCX-Baustein ein Grundprogramm (*leJOS* JVM), das den Java-Byte-Code ausführen kann. Dieses Grundprogramm muss einmalig auf den RCX-Baustein übertragen werden, was von dem Programm `lejosfirmdl.exe` erledigt wird. Da der RCX-Baustein erheblich wenige Speicher als ein Computer besitzt, ist der Umfang von *leJOS* erheblich eingeschränkt und es ist klar, dass man für den RCX-Baustein eine eigene JVM benötigt. So hat der RCX-Baustein nur 32 kB RAM-Speicher, wovon noch 4 kB abgezogen werden müssen, die von einigen Routinen aus dem ROM-Speicher benötigt werden. Die *leJOS* JVM belegt im RAM selbst 16 kB, somit bleiben für die selbst geschrieben Programme nur 12 kB übrig. Das ist nicht gerade viel, für die meisten Roboter reicht der Speicherplatz aber aus.
leJOS bekommt man gepackt in der Datei `lejos_win32_2_1_0.zip`, die man z. B. nach `C:\Lego` auspackt, wobei der Unterordner `C:\Lego\lejos` erstellt wird, der alle weitern Dateien enthält.
Auch hier ist die Online-Dokumentation sehr wichtig. Sie erhält man mit dem Archiv `lejos_win32_2_1_0.doc.zip`, das man ebenfalls nach `C:\Lego` auspackt. Dabei wird man einige Male gefragt, ob schon vorhanden Dateien überschrieben werden sollen, was man getrost bejahen kann. Unter `C:\Lego\lejos` gibt es jetzt einige Dokumentations-Ordner, der wichtigste ist `C:\Lego\lejos\apidocs` und dort die Datei `index.html`.

1.3.2 Umgebungsvariable für *leJOS* setzen

Um mit *leJOS* arbeiten zu können, müssen noch einige Umgebungsvariablen gesetzt werden (siehe dazu auch die Hilfe-Datei `C:\Lego\leJOS\CLICKME.html`).
Als erstes muss die `PATH`-Variable von *Windows* um den Pfad zu *leJOS* `C:\Lego\leJOS\bin` ergänzt werden.
Als nächstes muss eine neue Umgebungsvariable `RCXTTY` erstellt und definiert werden, falls der Infrarot-Sender nicht an der seriellen Schnittstelle[1] `COM1`, angeschlossen ist, was *leJOS* standardmäßig voraussetzt. Diese Information ist wichtig, denn beim Übertragen der Programme auf den RCX-Baustein muss *leJos* wissen, an welche Schnittstelle es das Programm senden soll. In unserer Beispiel-Installation ist der Sender an `COM4` angeschlossen. Deshalb muss die Umgebungsvariable `RCXTTY` auf den Wert `COM4` gesetzt werden.

[1] Man nennt die serielle Schnittstelle manchmal auch COM-Port.

1.3.2.1 *Windows* 95/ 98/ ME

Unter *Windows* 95/ 98/ ME öffnet man mit einem Texteditor dazu in die Datei `C:\autoexec.bat` und fügt eine neue Zeile hinter einer eventuell schon vorhandenen Zeile PATH ein:

```
PATH=%PATH%;C:\java\j2sdk1.4.2\bin;C:\Lego\leJOS\bin
```

Falls der Eintrag für das *JDK* schon in der PATH-Variablen eingetragen sein sollt, kann man diesen Eintrag in der nächsten Zeile natürlich weglassen. Wichtig ist, dass die verschiedenen Pfade durch ein Semikolon voneinander getrennt werden.

Dann fügt mein eine weitere Zeile ein:

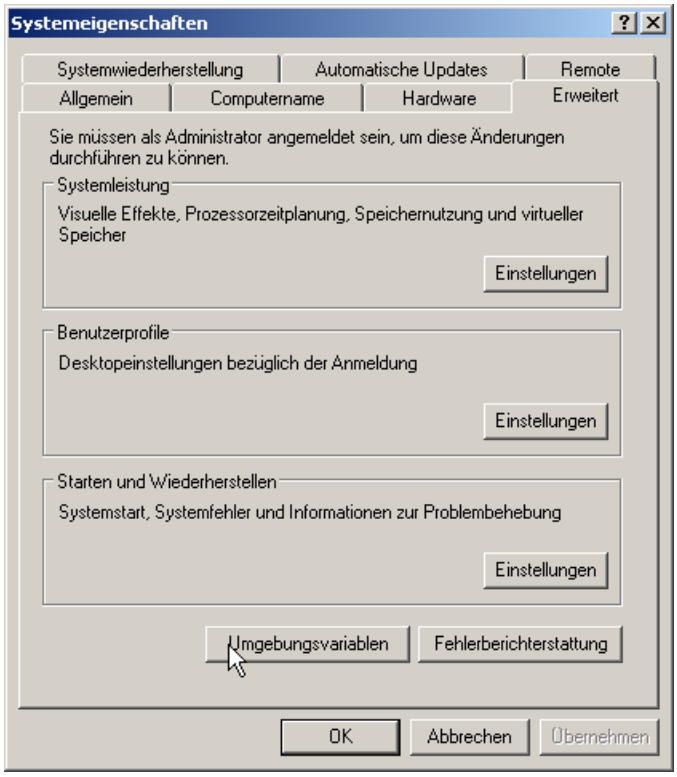
```
set RCXTTY=COM4
```

Hat man einen Infrarot-Sender, der an eine USB-Schnittstelle angeschlossen wird, lautet der Eintrag:

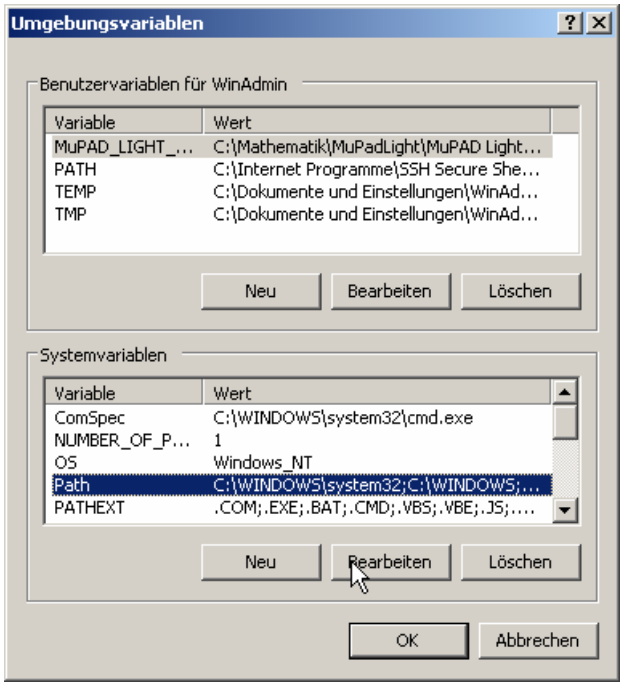
```
set RCXTTY=usb.
```

1.3.2.2 *Windows* NT/ 2000/ XP

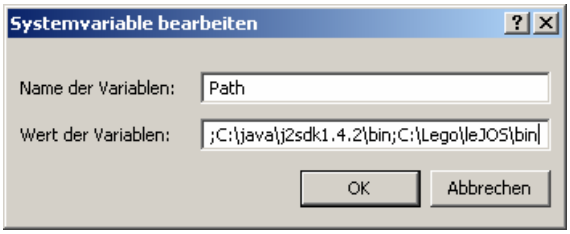
Um die benötigten Einträge machen zu können, muss man Administrator-Rechte besitzen. Man öffnet in der Systemsteuerung SYSTEM → ERWEITERT.



Anklicken der Schaltfläche UMGEBUNGSVARIABLEN öffnet folgendes Fenster:

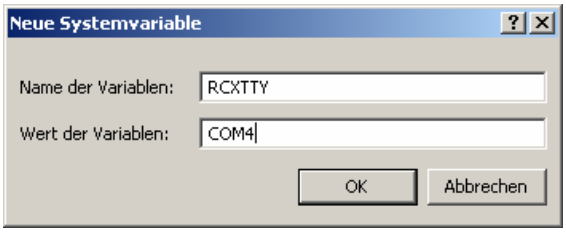


Man wählt im Fenster SYSTEMVARIABLEN den Eintrag PATH und klickt auf BEARBEITEN. Dann öffnet sich folgendes Fenster:



Man ergänzt die Einträge im Eingabefeld WERT DER VARIABLEN um `C:\Lego\leJOS\bin`. Der Eintrag für das *JDK* sollte schon vorhanden sein. Falls nicht, muss man auch diesen Eintrag hinzufügen.
Wichtig:
Alle Pfade müssen durch ein Semikolon voneinander getrennt werden.
Anschließend bestätigt man die Änderung mit OK.

Um eine neue Umgebungsvariable zu erstellen, klickt man auf die Schaltfläche NEU, worauf sich das gleiche Fenster wie eben, jedoch mit zwei leeren Eingabefeldern öffnet:



Man trägt als NAME DER VARIABLEN RCXTTY und als WERT DER VARIABLEN `COM4` ein. Dann auf OK klicken, und alle weiteren Fenster ebenfalls mit OK schließen, das war's.

1.3.3 Ergänzungen zum Paket *leJOS*

Es hat sich gezeigt, dass die Funktionsweise nicht von allen *leJOS*-Klassen korrekt ist. Insbesondere funktioniert der `TimingNavigator` nicht richtig. Somit sollte man das Java-Archiv `C:\Lego\leJOS\lib\classes.jar` umbenennen in z. B. `classes.jar.sic` und durch `classes.jar` von der CD ersetzen.

1.4 Java und *Fischer-Technik*

1.4.1 Installation des Pakets `FtSerialPort` für *Fischer-Technik*

fischertechnik-Roboter werden meist über eine Direktverbindung zu einem PC gesteuert. Hierzu wird die *fischertechnik*-Steuerung an den Seriellen Port des Computers angeschlossen. Weil Java eine Betriebssystem- und daher auch Hardware-unabhängige Sprache ist, bietet sie keine Möglichkeit, direkt auf die serielle Schnittstelle zuzugreifen. Um diese Einschränkung zu umgehen, gibt das kleine, in der Sprache C++ geschriebene Programm `FtSerialPort.dll`, das man in den Ordner `C:\Windows\System32` kopieren muss. In die Java-Projekte muss man später immer die Datei `FtSerialPort.java einbinden`. Die darin enthaltene Klasse benutzt dann `FtSerialPort.dll` um dem Java-Programm Zugriff auf den Seriellen Anschluss zu geben.

1.4.2 Installation des Pakets `ft.comm` für *Fischer-Technik*

Fischer-Technik-Roboter werden meist mit Direktverbindung zu einem PC gesteuert. Weil man hier keine Speicherplatzprobleme zu befürchten hat, käme man mit der gewöhnlichen Java-Installation aus, ergänzt um das Paket `commapi`. Dann müsste man aber die Ansteuerung der Motoren und der analogen und digitalen Eingänge am *Fischer-Technik*-Interface mühsam selbst programmieren. Das Paket `ft.comm` von Axel T. Schreiner stellt hierfür geeignete Klassen und Methoden zur Verfügung. Man erhält es als Archiv `code.zip`. Leider ist in diesem Archiv die Dateistruktur nicht sehr übersichtlich. Deshalb sollte man diese Datei zunächst ein einen temporären Ordner z. B. `C:\Temp` auspacken. Für das weitere Vorgehen legt man einen Ordner `C:\FischerTechnik` an und verschiebt den gesamten Ordner `C:\Temp\code\ft` samt Inhalt nach `C:\FischerTechnik`.
`ft.comm` stellt darüber hinaus noch eine Datei `ft.comm.dll` zur Verfügung, die in der Programmiersprache C++ geschrieben wurde und einige native Routinen[2] enthält. So kann die Zugriffsrate auf die serielle Schnittstelle gegenüber der Zugriffrate aus dem Java `commapi`-Paket erhöht werden.

1.4.3 Installation des Pakets `JavaFish` für *Fischer-Technik*

`JavaFish` von Ulrich Müller ist eine andere Java-Erweiterung für *Fischer-Technik*-Roboter. Auch dieses Paket stellt eine Reihe von nativen Routinen zur Verfügung, die im Paket `ftcomputing` in der Klasse `JavaFish` durch Java-Methoden ergänzt wurden.
Es gibt eine Installationsdatei `umfish20setup.exe` und ein Archiv `JavaFish.zip`. Die Installationsdatei enthält ein fertiges Programm, mit dem man die Verbindung zum *Fischer-Technik*-Interface testen kann, sowie weitere Hilfsmittel für andere Programmiersprachen wie C++, Delphi, VisualBasic und auch Java. Da hier jedoch nur die Erweiterungen für Java benötigt werden, verzichtet man besser auf die Installation von `umfish20setup.exe`. Außerdem enthält die Archivdatei `javafish.zip` eine korrigierte Version von `JavaFish.java`, sowie die zugehörige Dokumentation. Leider muss man die Java-Quelldateien alle noch selbst compilieren. Deshalb packt man stattdessen die auf CD mitgelieferte Datei `JavaFish_erw.zip` nach z. B. `C:\FischerTechnik` aus, die alles Notwendige enthält. Die eigentlichen nativen Routinen sind in den beiden Dateien `javaFish.dll` und `umFish20.dll` enthalten. Damit diese Dateien von den Anwendungen auch gefunden werden, muss man sie in den Ordner `C:\Windows\System32` kopieren. Dazu benötigt man Administrator-Rechte.

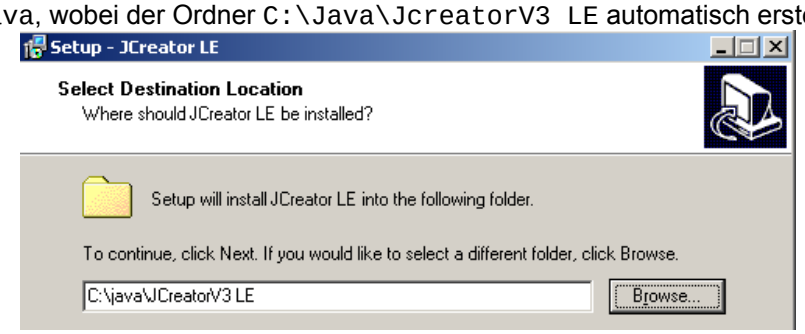
[2] Native Routinen werden einer anderen Programmiersprache geschrieben und bieten eine Schnittstelle zu Java an, so dass Java diese Methoden wie eigene ansprechen kann.

1.5 Der Editor *JCreator*

JCreator ist ein Editor-Programm zum Erstellen von Java-Programmen. Es gibt ihn in einer kostenlosen Version *JCreator LE* und in einer etwas mächtigeren Version, die man kaufen muss. Man kann ihn gut den eigenen Bedürfnissen anpassen, insbesondere erlaubt er die Einbindung von *leJOS*. Dann bietet er die Möglichkeit, ganze Projekte zu verwalten und sehr umfangreiche Projektvorlagen zu erstellen.

1.5.1 *JCreator* installieren

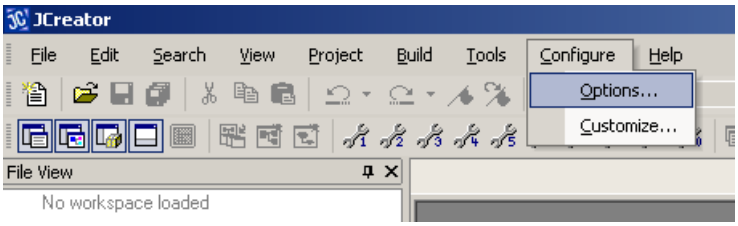
Zum Installieren von *JCreator LE* packt man das Archiv `jcrea310.zip` in einen temporären Ordner aus und starten mit einem Doppelklick auf `setup.exe` die Installation. Als Installationsordner wählt man z. B. `C:\Java`, wobei der Ordner `C:\Java\JcreatorV3 LE` automatisch erstellt wird.



Beim ersten Start nach der Installation muss man einige Fragen beantworten, z. B. ob Dateien mit der Endung `*.java` mit *JCreator* verknüpft werden sollen oder nicht. Diese Frage wird bejaht. Im folgenden wird der Pfad zum installierten *JDK* wird automatisch erkannt.

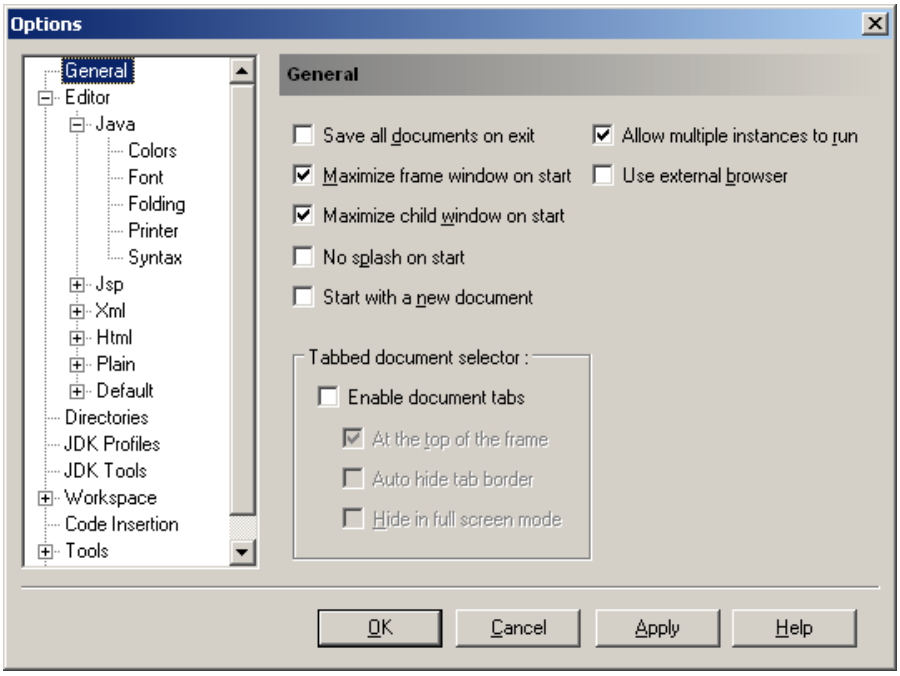
1.5.2 *JCreator* konfigurieren

Etwas umfangreicher gestalten sich die Konfigurationseinstellungen von *JCreator* und seine Vorbereitung für den Umgang mit *LEGO*- bzw. *Fischer-Technik*-Programmen. Man startet *JCreator LE* und wechselt mit CONFIGURE → OPTIONS ins Konfigurations-Menü:

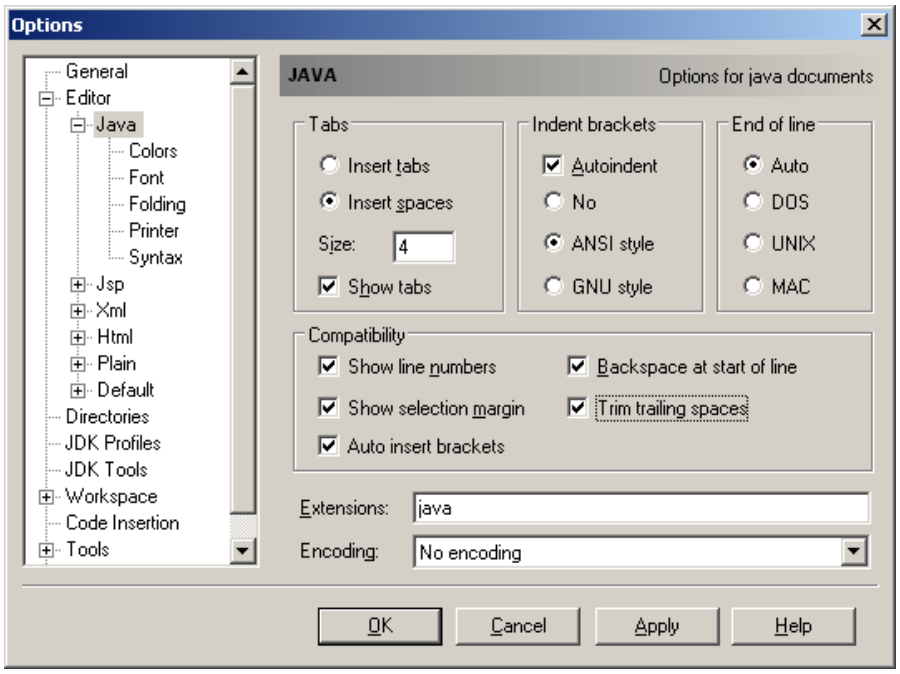


Dann nimmt man sich jeden Eintrag in der linken Spalte nacheinander vor.

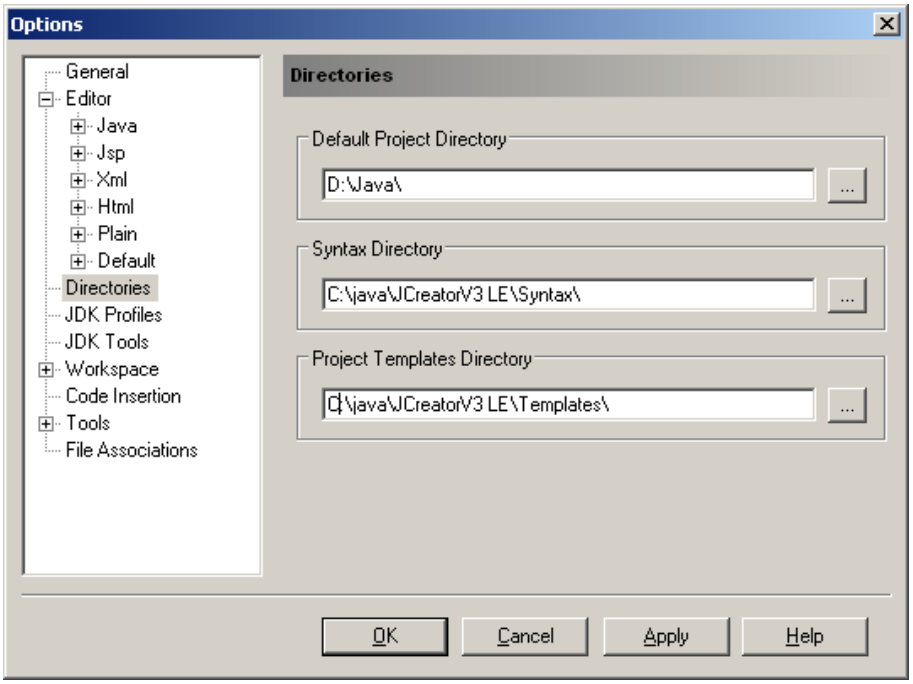
Das Menü GENERAL:



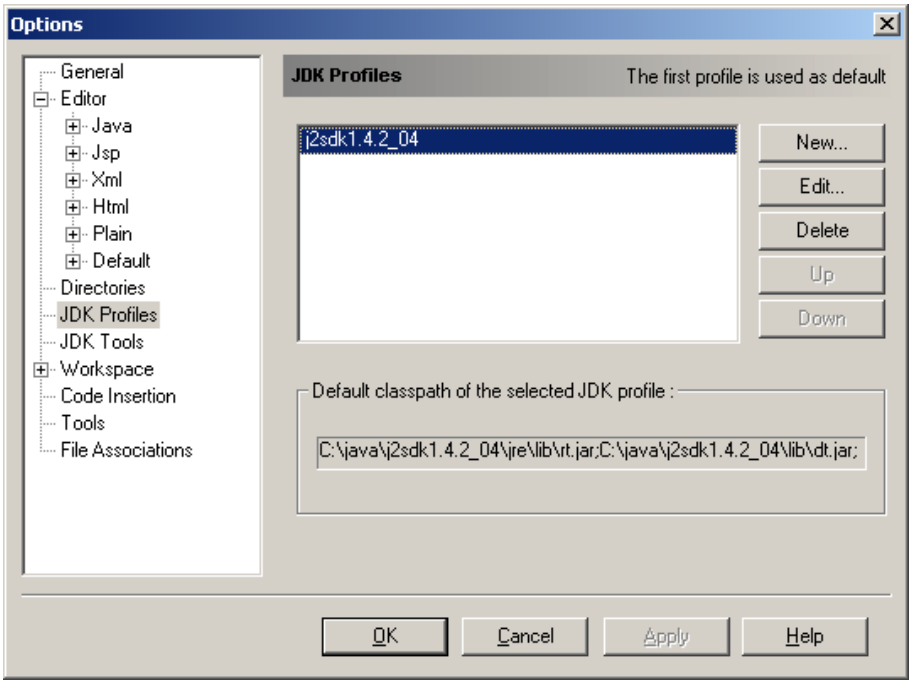
Das Menü EDITOR → JAVA:



Das Menü DIRECTORIES: Als DEFAULT PROJEKT DIRECTORY muss natürlich der zuvor angelegte Odner für eigene Programme eangegeben werden, hier: `D:\Java\` .



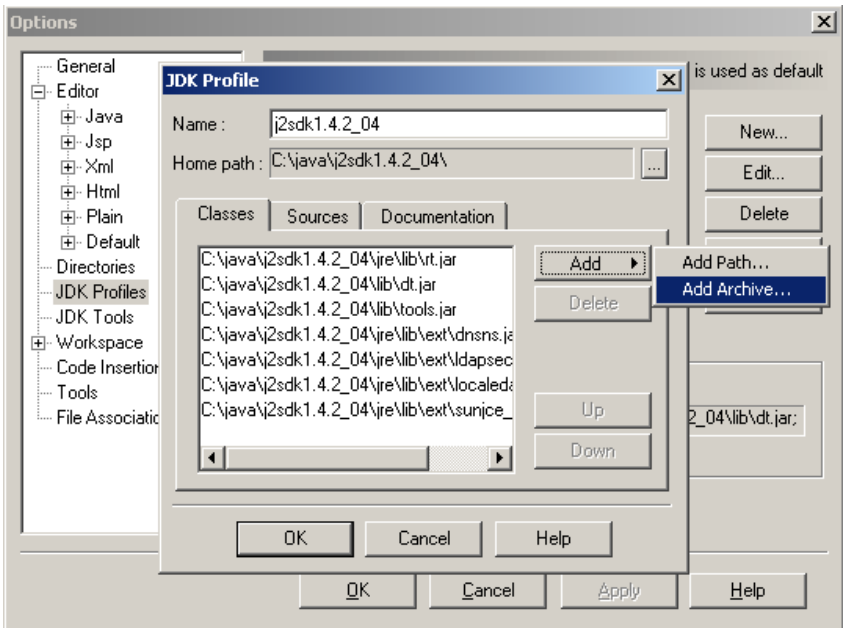
Das Menü JDK PROFILES:
Hier ist das installierte *JDK* schon eingetragen. Wenn nicht kann man es mit NEW hinzufügen.



Man wählt im Textfeld rechts den Eintrag `j2sdk1.4.2` aus und klickt auf EDIT. Es öffnet sich ein Fenster. In der Registerkarte CLASSES sind schon alle Archiv-Dateien mit der Endung `*.jar` des *JDK* eingetragen.

```
C:\java\j2sdk1.4.2_04\jre\lib\rt.jar;
C:\java\j2sdk1.4.2_04\lib\dt.jar;
C:\java\j2sdk1.4.2_04\lib\tools.jar;
C:\java\j2sdk1.4.2_04\jre\lib\ext\dnsns.jar;
C:\java\j2sdk1.4.2_04\jre\lib\ext\ldapsec.jar;
C:\java\j2sdk1.4.2_04\jre\lib\ext\localedata.jar;
C:\java\j2sdk1.4.2_04\jre\lib\ext\sunjce_provider.jar;
```

Diese Eintragungen werden der CLASSPATH-Variablen übergeben. Wir benötigen noch weitere Einträge. Dazu klickt man auf ADD und für Archiv-Dateien auf ADD ARCHIVE. Dazu wählt man nacheinander die benötigten Archiv-Dateien aus. Liegen in einem Ordner mehrere `jar`-Dateien, kann man diese auch auf einmal auswählen und mit OK hinzufügen.



Der folgende Eintrag fügt das Java Communications API hinzu:
C:\java\j2sdk1.4.2_04\lib\comm.jar;

Dann müssen die *LEGO*- und *Fischer-Technik*-Pakete hinzugefügt (ADD ARCHIVE) werden:

```
C:\Lego\leJOS\lib\vision.jar;
C:\Lego\leJOS\lib\classes.jar;
C:\Lego\leJOS\lib\jtools.jar;
C:\Lego\leJOS\lib\pcrcxcomm.jar;
C:\Lego\leJOS\lib\rcxrcxcomm.jar;
C:\FischerTechnik\ft\comm\ft.comm.jar
```

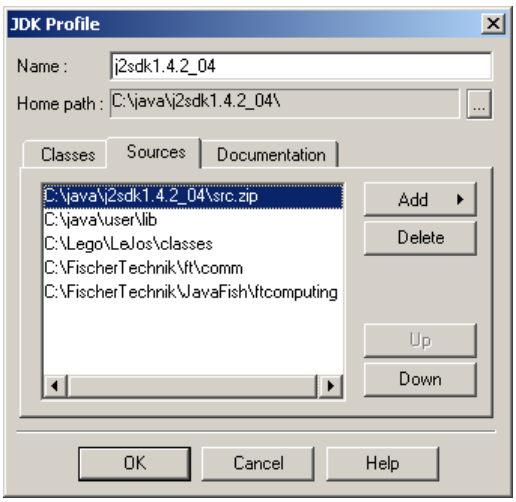
Da nicht alle *Fischer-Technik*-Pakete in Archiven abgelegt sind, muss man den Pfad zu diesen Paketen mit ADD PATH hinzufügen:
C:\FischerTechnik

```
C:\FischerTechnik\JavaFish
```

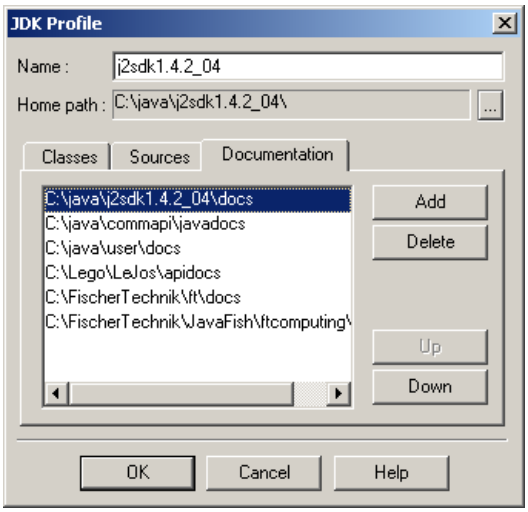
Auch der Pfad zu den eigenen Paketen sollten nicht vergessen werden:

```
C:\java\user\lib
```

Gelegentlich möchte man sich die Quellcodes der verwendeten Archive anschauen. Dazu wählt man im Menü JDK PROFILE die Registerkarte SOURCES aus und fügt die entsprechenden Pfade mit jeweils ADD PATH hinzu.

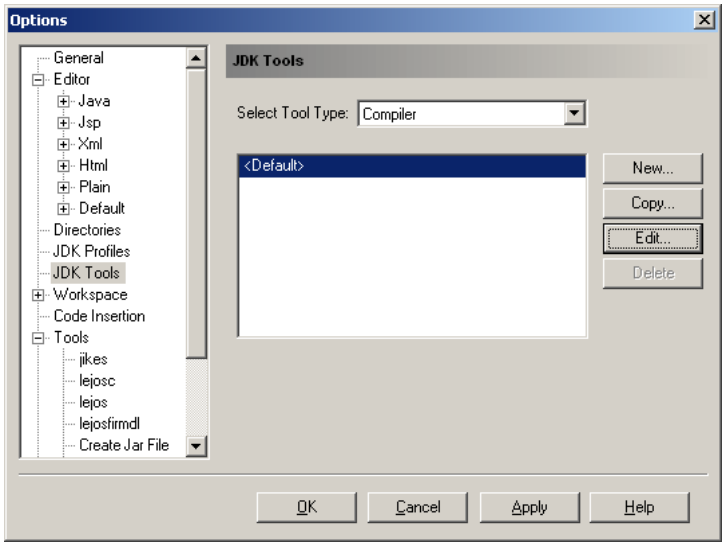


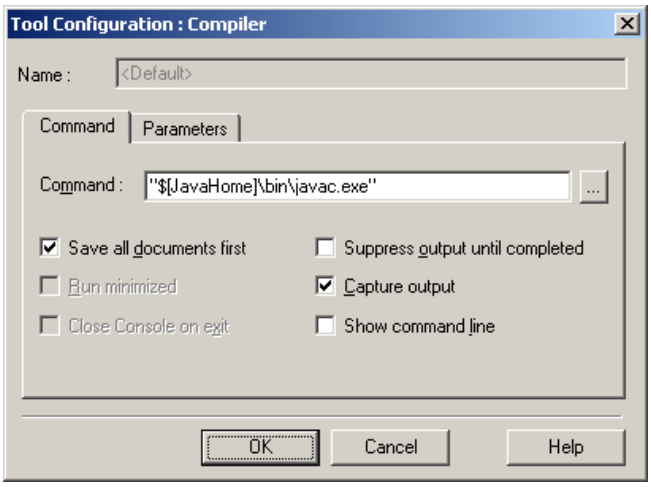
Ganz wichtig sind natürlich die jeweiligen Dokumentationen. Wenn man hier die Pfade richtig eingetragen hat, kann man später im Editor bei einem Schlüsselwort auf <Strg>+F1 klicken und man bekommt in der zugehörige Dokumentation den Eintrag zu diesem Schlüsselwort angezeigt.

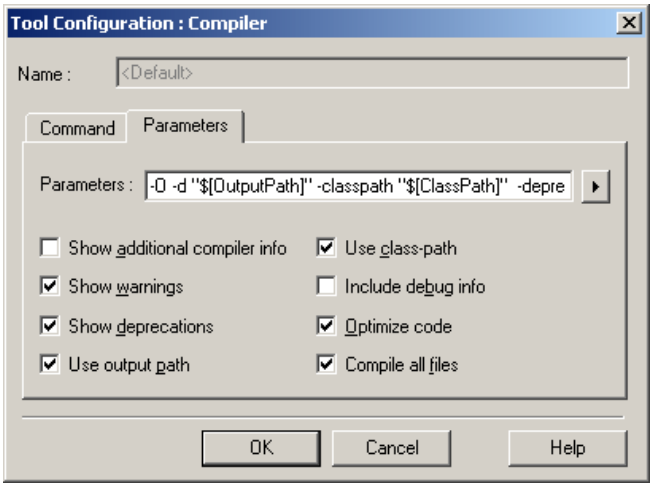


Anschließend bestätigt man mit OK, im Menü JDK PROFILES jedoch nur mit APPLY damit sich das Fenster nicht schließt.

Als nächstes werden die Eintragungen für den Java-Standard-Compiler vorgenommen. Dazu wählt man im Menü JDK TOOLS den Eintrag <DEFAULT> und klickt auf EDIT:





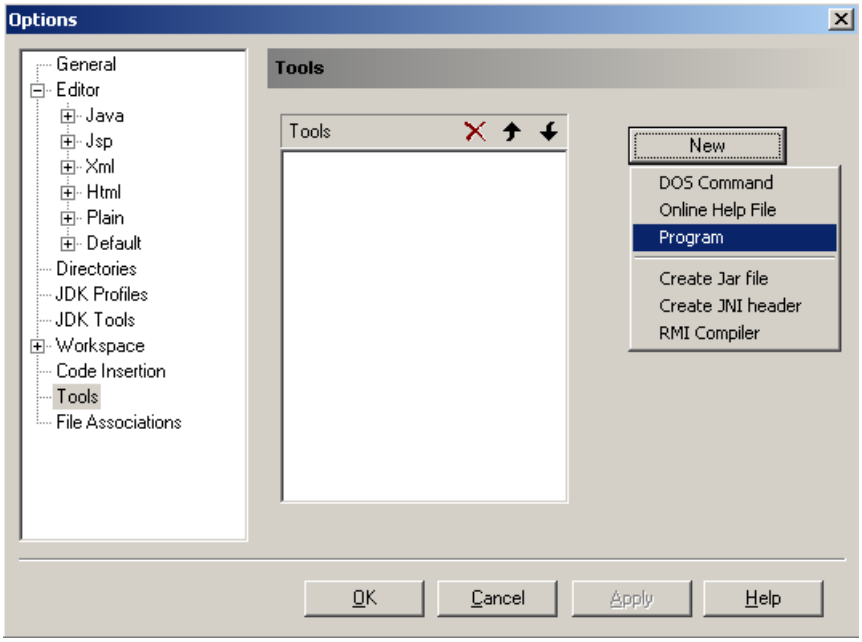


Dabei werden im Eingabefeld PARAMETERS folgende Eintragungen gemacht:

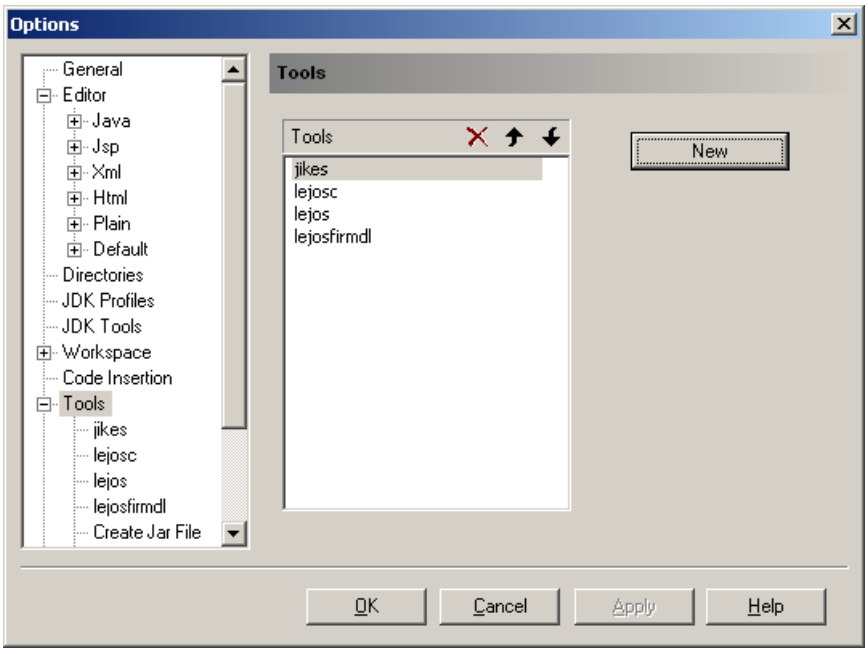
```
-O -d "$[OutputPath]" -classpath "$[ClassPath]" -deprecation $[JavaFiles]
```

Die Eintragungen werden durch einmal klicken auf OK und einmal klicken auf APPLY übernommen.

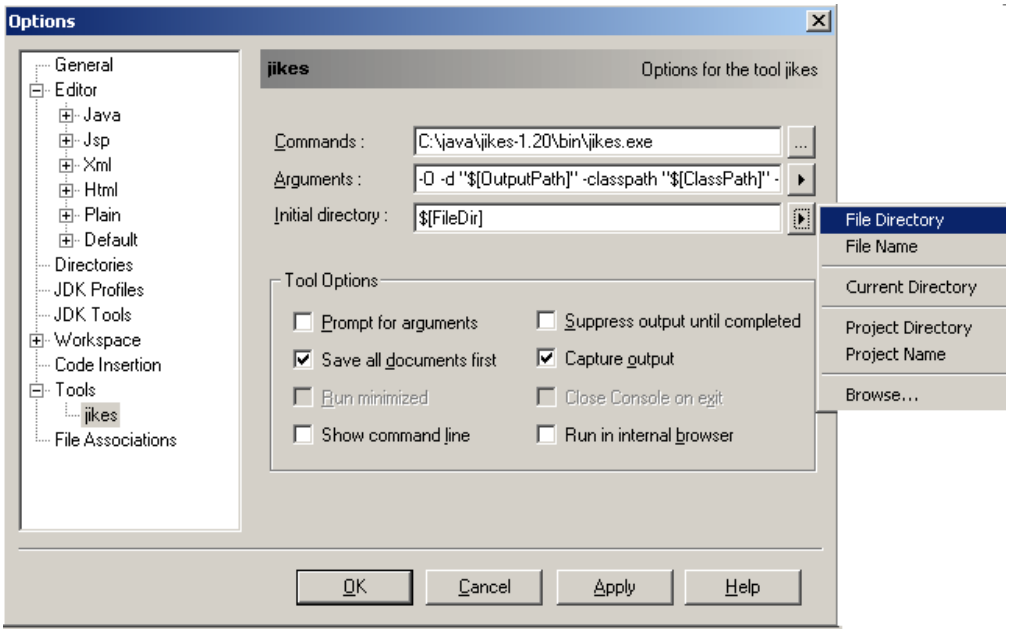
Zum Schluss müssen noch externe Programme wie der *Jikes*-Compiler, der *leJOS*-Compiler, der *leJOS*-Interpreter und das *leJOS*-Firmeware-Übertragungsprogramm eingetragen werden.
dazu gibt es das Menü TOOLS. Durch Klicken auf NEW und Auswahl von PROGRAM fügt man die entsprechenden Einträge der benutzerdefinierten Symbolleiste hinzu.



Im sich öffnenden Verzeichnisbaum wählt man nacheinander die Dateien `jikes.exe`, `lejosc.exe`, `lejos.exe` und `lejosfirmdl.exe` aus.



Anschließend müssen diese Einträge noch konfiguriert werden.
Für *Jikes* wählt man diesen Eintrag aus und kopiert am besten die Eintragungen für den Standard-Java-Compiler. Insbesondere das Eingabefeld ARGUMENTS sollte man durch copy & paste aus dem entsprechenden Feld des Java-Compilers übernehmen.



Der Eintrag unter ARGUMENTS lautet dann:
`-O -d "$[OutputPath]" -classpath "$[ClassPath]" -deprecation $[JavaFiles]`
Den Eintrag `$[FileDir]` im Eingabefeld INITIAL DIRECTORY kann man aus einer Liste rechts neben dem Eingabefeld auswählen.
Die Auswahlfelder SAVE ALL DOCUMENTS FIRST und CAPTURE OUTPUT werden ausgewählt.

Die Einträge für `lejosc.exe` lauten:

COMMAND:	`C:\Lego\leJOS\bin\lejosc.exe`
ARGUMENTS:	`-classpath $[FileDir];$[ClassPath] $[FileName]` `-d $[OutputPath]`
INITIAL DIRECTORY:	`$[FileDir]`

Die Auswahlfelder SAVE ALL DOCUMENTS FIRST und CAPTURE OUTPUT werden ausgewählt.

Die Einträge für `lejos.exe` lauten:

COMMAND:	`C:\Lego\leJOS\bin\lejos.exe`
ARGUMENTS:	`$[CurClass]`
INITIAL DIRECTORY:	`$[FileDir]`

Keines der Auswahlfelder wird ausgewählt.

Die Einträge für `lejosfirmdl.exe` lauten:

COMMAND:	`C:\Lego\leJOS\bin\lejosfirmdl.exe`
ARGUMENTS:	
INITIAL DIRECTORY:	

Keines der Auswahlfelder wird ausgewählt.

1.5.2.1 Weitere Benutzeranwendungen zu *JCreator* hinzufügen

- **mehrere Dateien auf einmal compilieren:** Wenn ein *LEGO*-Projekt aus mehreren Dateien besteht, möchte man manchmal **alle** Dateien des Projekts **auf einmal** compilieren. Dazu muss man nochmals die Anwendung `lejosc.exe` dem Menü TOOLS mit NEW → PROGRAM hinzufügen. Allerdings startet man dann lejosc.exe mit anderen Parametern.

  Die Einträge für `lejos.exe` lauten:

  COMMAND: `C:\Lego\leJOS\bin\lejos.exe`
  ARGUMENTS: `-classpath $[FileDir];$[ClassPath] $[JavaFiles] -d $[OutputPath]`
  INITIAL DIRECTORY: `$[FileDir]`

  Die Auswahlfelder SAVE ALL DOCUMENTS FIRST und CAPTURE OUTPUT werden ausgewählt.

- **Dokumentation:** Wenn man zu den Quelltexten der eigenen Programme so genannte Dokumentations-Kommentare hinzufügt, kann man sehr bequem auf „Tastendruck" eigene Dokumentations-Dateien erstellen. Dieses erledigt das Programm `javadoc.exe` in `C:\java\j2sdk1.4.2_04\bin`. Wie eben beschrieben, fügt man im Menü TOOLS mit NEW → PROGRAM das Programm `javadoc.exe` hinzu. Wenn man ein neues JDK installiert, möchte man nicht alle Eintragungen nochmals machen. Es besser wenn man mit relativen Pfaden statt mit absoluten Pfaden arbeitet. Deshalb verwendet man die JCreator-Variable `[JavaHome]`.

  Die Einträge für `javadoc.exe` lauten:

  COMMAND: `"$[JavaHome]\bin\javadoc.exe"`
  ARGUMENTS: `-author -version -d docs -classpath $[ClassPath] $[JavaFiles]`
  INITIAL DIRECTORY: `$[PrjDir]`

  Die Auswahlfelder SAVE ALL DOCUMENTS FIRST und CAPTURE OUTPUT werden ausgewählt.

- **Jar-Archive:** Manchmal möchte man eigene `jar`-Archive, d.h. gepackte Java-Projekte, erstellen. Das erledigt das Programm `jar.exe` in `C:\java\j2sdk1.4.2_04\bin`. Man fügt es mit NEW → CREATE JAR FILE den Benutzer-Tools hinzu. Dabei werden alle nötigen Einstellungen automatisch vorgenommen.

  Die Einträge für `jar.exe` lauten:

  COMMAND: `"$[JavaHome]\bin\javadoc.exe"`
  ARGUMENTS: `cvf $[PrjName].jar .`
  INITIAL DIRECTORY: `"$[OutputPath]"`

  Die Auswahlfelder CAPTURE OUTPUT und SHOW COMMAND LINE werden automatisch ausgewählt, das Auswahlfeld SAVE ALL DOCUMENTS FIRST, sollte man noch nachträglich auswählen.

- **Dokumentation ansehen:** Weiterhin ist es nützlich, wenn man die Dokumentation, die man mit JavaDoc erstellt hat schnell öffnen kann. Da diese als html-Datei vorliegt, muss man sie mit einem Browser anschauen. Für den nächsten Menüpunkt gehen wir davon aus, dass *Mozilla* schon installiert ist.
  Mit NEW → PROGRAM FILE fügt man das Programm `mozilla.exe` hinzu und übergibt ihm die Index-Datei der Dokumentation als Parameter.

  Die Einträge für `mozilla.exe` lauten:

  COMMAND: `C:\Programme\Mozilla\mozilla.exe`
  ARGUMENTS: `file:///$[PrjDir]/docs/index.html`
  INITIAL DIRECTORY:

  Die Auswahlfelder RUN MINIMIZED und CLOSE CONSOLE ON EXIT werden ausgewählt.

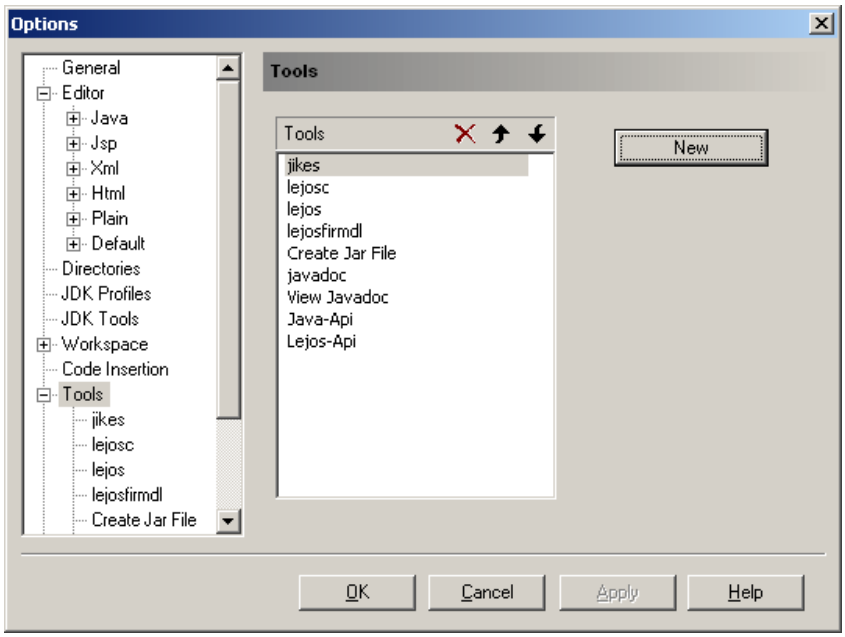
- **Java-API:** Ebenso will man die Dokumentation zur Java-API schnell öffnen können. Auch diese schaut man sich mit *Mozilla* an.
  Wie oben fügt man mit NEW → PROGRAM fügt man das Programm `mozilla.exe` hinzu und übergibt ihm die Index-Datei der API-Dokumentation als Parameter.

  Die Einträge für `mozilla.exe` lauten:
  COMMAND: `C:\Programme\Mozilla\mozilla.exe`
  ARGUMENTS: `file:///$[JavaHome]/docs/api/index.html`
  INITIAL DIRECTORY:
  Die Auswahlfelder RUN MINIMIZED und CLOSE CONSOLE ON EXIT werden ausgewählt.

- **LeJos-API:** Die letzte noch freie Tools-Schaltfläche kann man mit der Dokumentation zur *leJOS*-API belegen. Auch hier wird die entsprechende Datei dem Programm `mozilla.exe` übergeben.

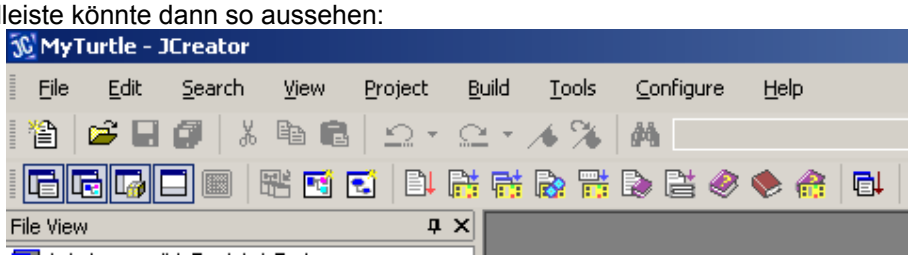
  Die Einträge für `mozilla.exe` lauten:
  COMMAND: `C:\Programme\Mozilla\mozilla.exe`
  ARGUMENTS: `file:///C|/Lego/leJOS/apidocs/index.html`
  INITIAL DIRECTORY:
  Die Auswahlfelder RUN MINIMIZED und CLOSE CONSOLE ON EXIT werden ausgewählt.

Um die Reihenfolge der Schaltflächen zu ändern, kann man im Menü CONFIGURE → OPTIONS → TOOLS mit den Schaltflächen MOVE ITEM UP bzw. MOVE ITEM DOWN hoch oder runter schieben.



Wenn man mag, kann man die Symbole der User-Tools noch ändern. Das macht man mit CONFIGURE → CUSTOMIZE... Dann klickt man mit der rechten Maustaste auf das zu ändernde Symbol und wählt BUTTON APPEARENCE ... In dem sich öffnenden Fenster wählt man die Option SELECT USER DEFINED IMAGE und sucht sich ein passendes Symbol aus. Man kann mit NEW auch selbst Symbole erstellen.

Da die Anzahl der mitgelieferten vordefinierten Symbole etwas klein ist, gibt es auf der CD eine Erweiterung. Zunächst schließt man *JCreator*. Dann kopiert von CD die Datei `UserImages.bmp` in das Installationsverzeichnis von *JCreator*. Zur Sicherheit sollte man die dort vorhandene gleichnamige Datei unter einem anderen Namen sichern.
Wenn man *JCreator* jetzt wieder startet, stehen viele neue Symbole zur Verfügung. Eine veränderte Tool-Symbolleiste könnte dann so aussehen:



1.5.3 Vorlagendateien für *JCreator* erstellen

Im Ordner `C:\java\JcreatorV3 LE\Templates` befinden sich nach der Installation vier weitere Unterordner `Template_1`, `Template_2`, `Template_3` und `Template_4`.

1.5.3.1 Vorlagendateien für *Lejos* erstellen

Man kopiert den gesamten Ordner `Templates_1` mit Inhalt und fügt ihn als `Template_5` oder besser, weil aussagekräftiger, als `Template_lejos` ein. `Template_lejos` enthält jetzt eine Datei `setup.tst` und im Unterordner `classes` eine Datei `Project_Name.java`. Die Datei `Project_Name.java` kann man mit einem gewöhnlichen Texteditor bearbeiten.

Nach den Änderungen sollte `Project_Name.java` folgenden Inhalt haben:

```
import josx.platform.rcx.*;
import josx.robotics.*;

/**
  * LEGO-Projekt mit leJOS <PROJECT_NAME>.java
  * @version 1.0 vom <%d>.<%m>.<%y>
  * @author N.N.
  */
public class <PROJECT_NAME> {

    public static void main (String[] args) {

    }
}
```

Die Datei `setup.tst` kann man auch mit einem einfachen Texteditor bearbeitet werden:

```
; This file contains the setup information for a template project.

[LABEL]    leJOS Application
```

1.5.3.2 Vorlagendateien für *Fischer-Technik* mit *ft* erstellen

Um eine Vorlage für *Fischer-Technik*-Programme mit dem Paket `ft.comm` zu erstellen, kopiert man den Ordner `Templates_lejos` und fügt ihn als `Templates_ft` ein.

`Project_Name.java` ändert man wie folgt:

```
import ft.*;

/**
  * Fischer-Technik mit ft.comm <PROJECT_NAME>.java
  * @version 1.0 vom <%d>.<%m>.<%y>
  * @author N.N.
  */
public class <PROJECT_NAME> {

    public static void main (String[] args) {

    }
}
```

`setup.tst` ändert man ebenfalls:

```
; This file contains the setup information for a template project.

[LABEL]    FischerTechnik ft.comm Application
[DESTPATH].."classes"
```

1.5.3.3 Vorlagendateien für *Fischer-Technik* mit *JavaFish* erstellen

Um eine Vorlage für *Fischer-Technik*-Programme zu erstellen, kopiert man den Ordner `Templates_ft` und fügt ihn als `Templates_JavaFish` ein.

`Project_Name.java` ändert man wie folgt:

```
import ftcomputing.*;

/**
  * Fischer-Technik mit JavaFish <PROJECT_NAME>.java
  * @version 1.0 vom <%d>.<%m>.<%y>
  * @author N.N.
  */
public class <PROJECT_NAME> {

    public static void main (String[] args) {

    }
}
```

`setup.tst` ändert man ebenfalls:

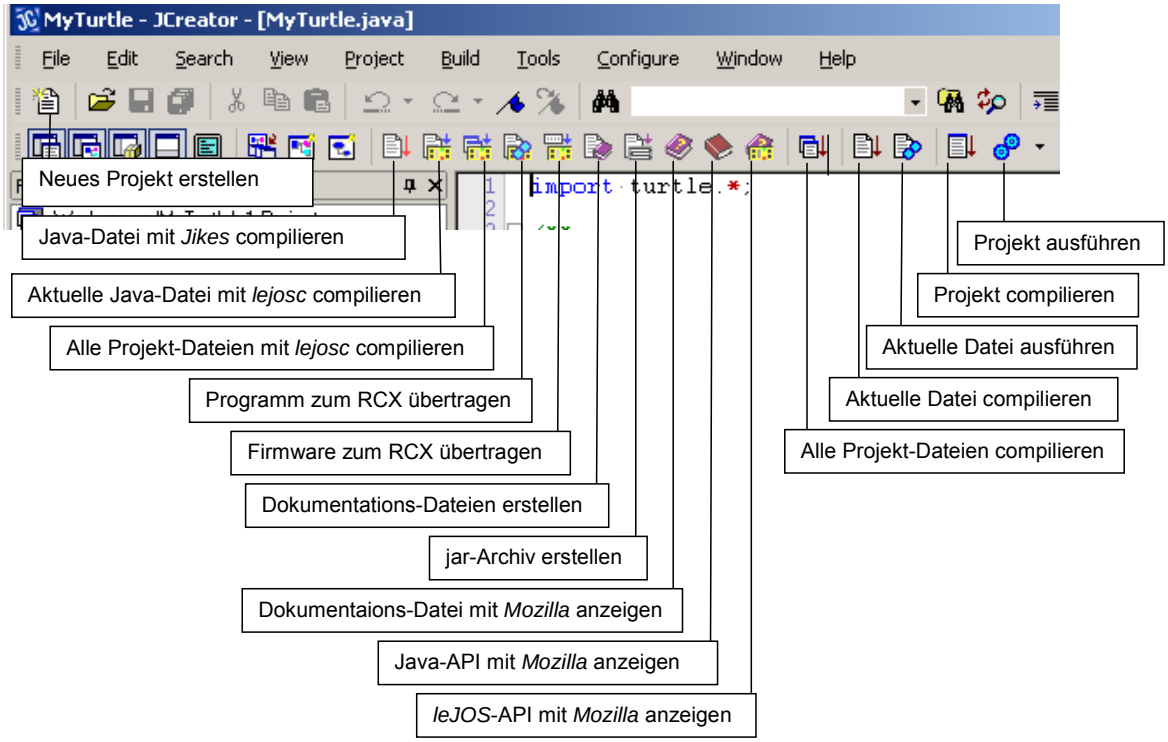
```
; This file contains the setup information for a template project.

[LABEL]    FischerTechnik JavaFish Application
[DESTPATH].."classes"
```

Für N.N. trägt man jeweils den eigenen Namen ein.

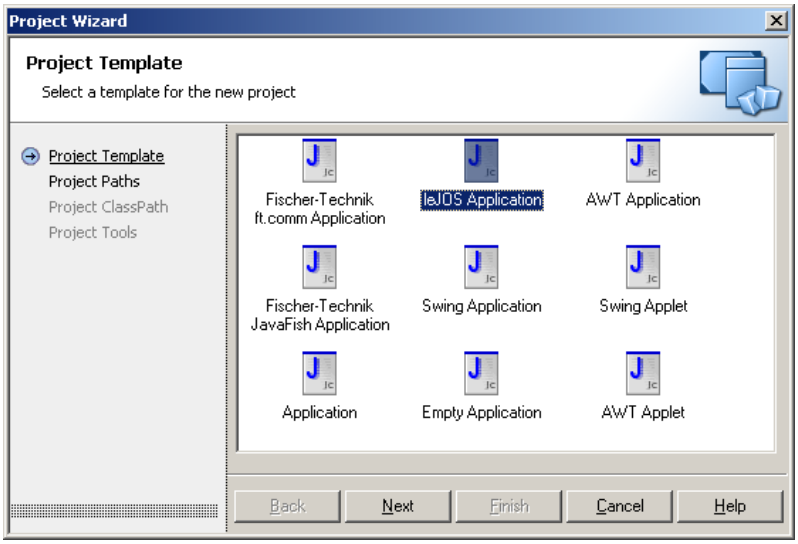
Anstatt diese Vorlagen selbst zu erstellen kann man auch das Archiv `Templates.zip`, das diese und weitere Vorlagen enthält, von der CD nach `C:\java\JcreatorV3 LE\Templates` auspacken.

1.5.4 Projekte mit *JCreator* erstellen

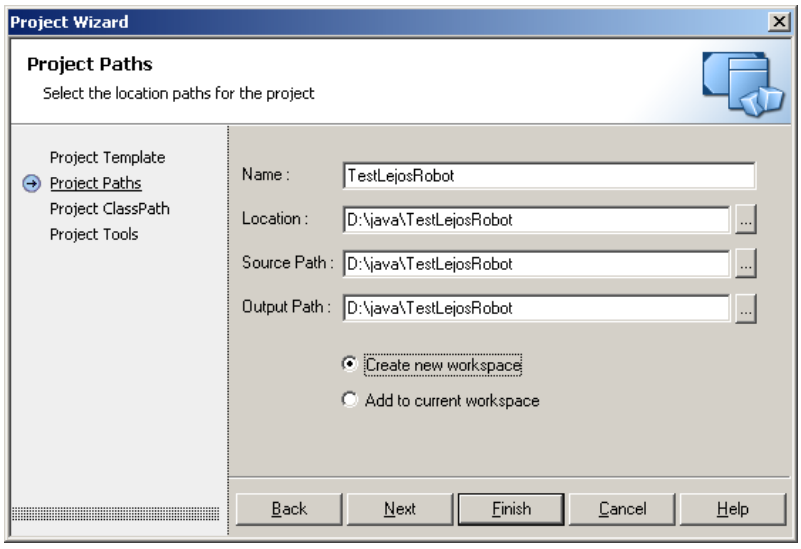
Die Abbildung zeigt die Bedeutung der wichtigsten Schaltflächen. Die benutzerdefinierten Schaltflächen `lejosc`, `lejos` und `lejosfirmdl` benötigt man für *LEGO*-Projekte. Für *Fischer-Technik*-Projekte verwendet man die Standardschaltflächen.



Ein neues Projekt beginnt man durch Klicken auf die Schaltfläche NEW (Neues Projekt erstellen). Es öffnet sich ein Fenster. In der Registerkarte PROJECT TEMPLATE wählen wir `leJOS Application` .



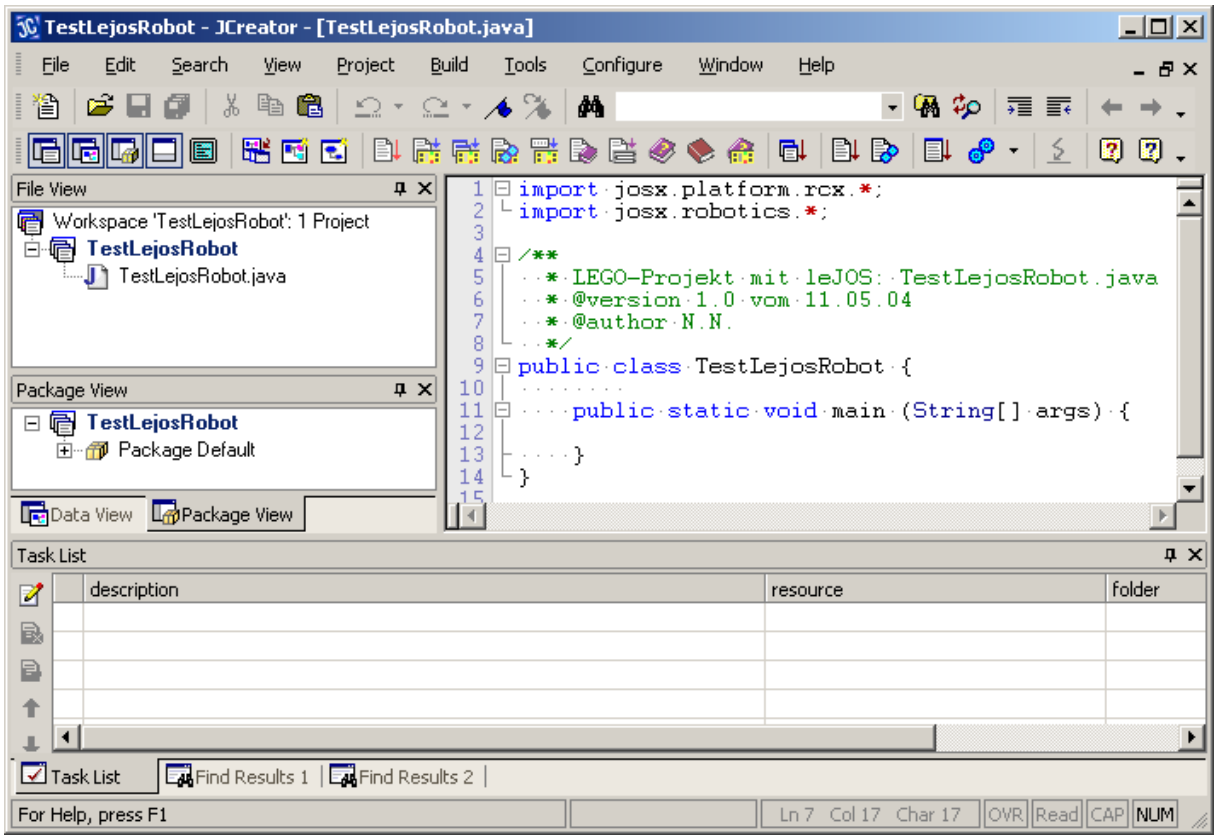
Im Register PROJECT PATHS tragen wir im Eingabefeld PROJECT NAME `TestLejosRobot` ein. Dadurch wird der Pfad `D:\Java` automatisch zu `D:\Java\TestLejosRobot` ergänzt. Anschließend bestätigt man mit `Next`.



Die beiden folgenden Fenster bleiben unverändert.

Durch Doppelklick auf `TestLejosRobot.java` im linken Fenster öffnet sich der Editor mit dem Quelltext, der die Eintragungen aus der gewählten Vorlagendatei enthält. Man kann den Quelltext nun weiter bearbeiten.

Durch Klicken auf die zweite benutzerdefinierte Schaltfläche wird das Programm mit `lejosc.exe` compiliert. Die dritte benutzerdefinierte Schaltfläche überträgt das Programm zum RCX. Falls die Firmware noch nicht übertragen war, wird man dazu aufgefordert, was man mit der vierten benutzerdefinierten Schaltfläche macht.



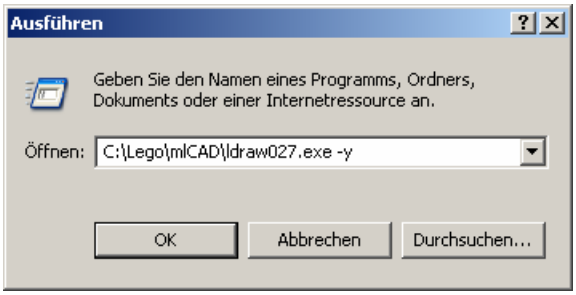
Ein *Fischer-Technik*-Projekt erstellt man genauso. Man wählt dann natürlich eine der beiden Vorlagen für die `FischerTechnik Applications` und compiliert und startet mit den entsprechenden Java-Werkzeugen.

1.6 *LDraw* und *mlCAD*

LDraw ist ein Programm, mit dessen Hilfe man *LEGO*-Modelle und -Bauanleitungen in 3D-Ansicht erstellen kann. Es kennt sämtliche *LEGO*-Bauteile, die man zu einem Modell zusammenfügen kann. *mlCad* ist ein *Windows*-Programm, das auf *LDraw* aufbaut und mit dessen Hilfe man die Modelle sehr komfortabel direkt auf dem Bildschirm zusammenbauen und anschauen kann. Dabei lässt sich zudem das Modell in alle Richtungen drehen.

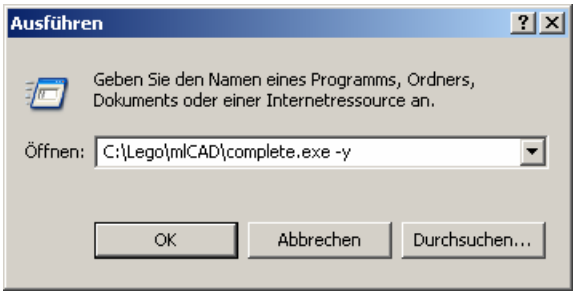
1.6.1 *LDraw* installieren

Zunächst sollte man einen neuen Basis-Ordner für die Programme erstellen, in unserem Fall z. B. `C:\Lego\mlCAD`. Dorthin hinein werden die beiden Installationsdateien `ldraw027.exe` und `complete.exe` kopiert. Die Installation startet man über START → AUSFÜHREN. In das Feld Öffnen gibt man die Befehlszeile `C:\Lego\mlCAD\ldraw027.exe -y` ein und bestätigt mit OK.



Dadurch werden die *LDraw*-Programm-Dateien in den Ordner `C:\Lego\mlCAD\LDRAW` entpackt.

Die Beschreibungs-Dateien der Lego-Bauteile befinden sich in der Datei `complete.exe`, die über die Befehlszeile `C:\Lego\mlCAD\ldraw027.exe -y` entpackt wird:



Anschließend können die beiden Dateien `ldraw027.exe` und `complete.exe` gelöscht werden, da sie nicht mehr benötigt werden.

1.6.2 *mlCAD* installieren

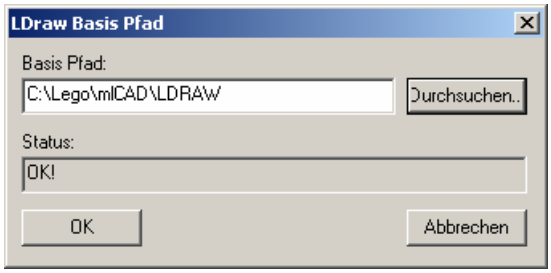
Nachdem das Basisprogramm *LDraw* installiert wurde, kann man die Installation des *Windows*-Programms *mlCAD* in Angriff nehmen. Dazu wird die Datei `mlcad300.zip` in den Ordner `C:\Lego\mlCAD entpackt.`
Für den vereinfachten Start des Programms ist es sinnvoll eine Verknüpfung auf den Desktop oder in das Startmenü anzulegen. Dazu klickt man mit der rechten Maustaste auf `MLCAD.exe` und wählt VERKNÜPFUNG ERSTELLEN. Diese Verknüpfung kann man anschließend z. B auf den Desktop ziehen.

1.6.3 Das Tutorial zu *mlCAD* installieren

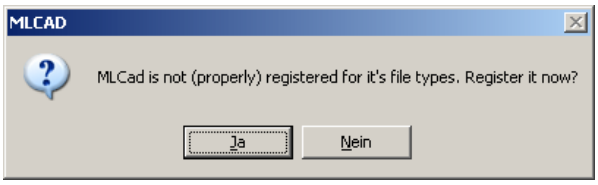
Eine kurze Anleitung zu *mlCAD* ist in dem Archiv `tutorial.zip` enthalten. Dieses wird z. B. in das Verzeichnis `C:\Lego\mlCAD\tutorial` entpackt. Die Start-Seite heißt `tut_ger.html`. Auch für diese Datei kann man eine Verknüpfung anlegen.

1.6.4 *mlCAD* konfigurieren

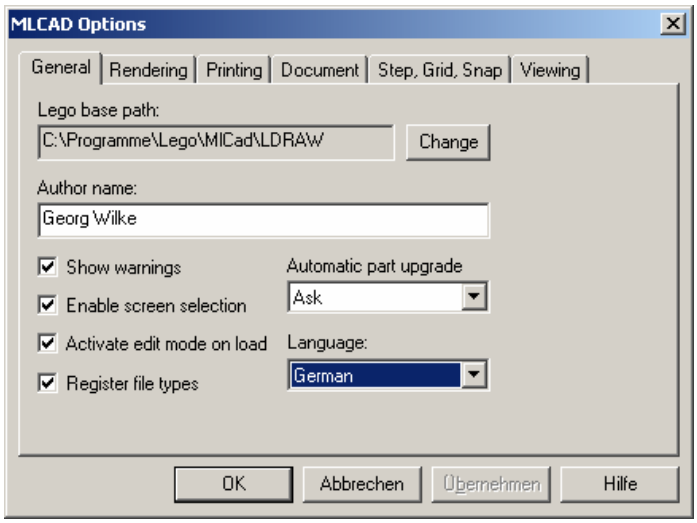
Zunächst wird *mlCAD* gestartet. Beim ersten Start wird man nach dem Pfad zu den *LDraw*-Dateien gefragt. Hier gibt man `C:\Lego\mlCAD\LDRAW` an.



Die Frage nach der Registrierung der entsprechenden Dateien kann man getrost bejahen



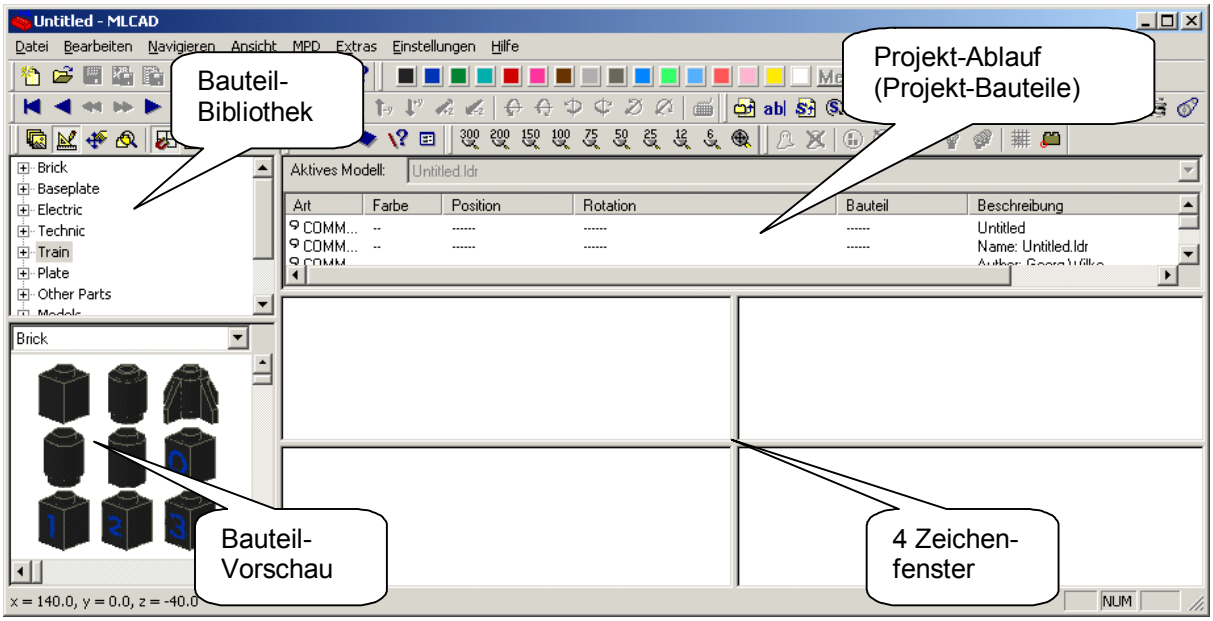
Gleich nach dem Start geht man in das Menü SETTINGS → GENERAL → CHANGE... Dort wählt man als SPRACHE (LANGUAGE) `Deutsch (German)` aus und unter AUTHOR gibt man seinen Namen ein.



Anschließend startet man *mlCAD* neu, damit die Änderungen Wirkung zeigen können.

Als nächstes müssen die *Lego*-Bauteile von *LDraw* nach *mlCAD* importiert werden. Dazu klickt man auf den Menüeintarg DATEI → SUCHE BAUTEILE. Die folgende Abfrage quittiert man mit JA.

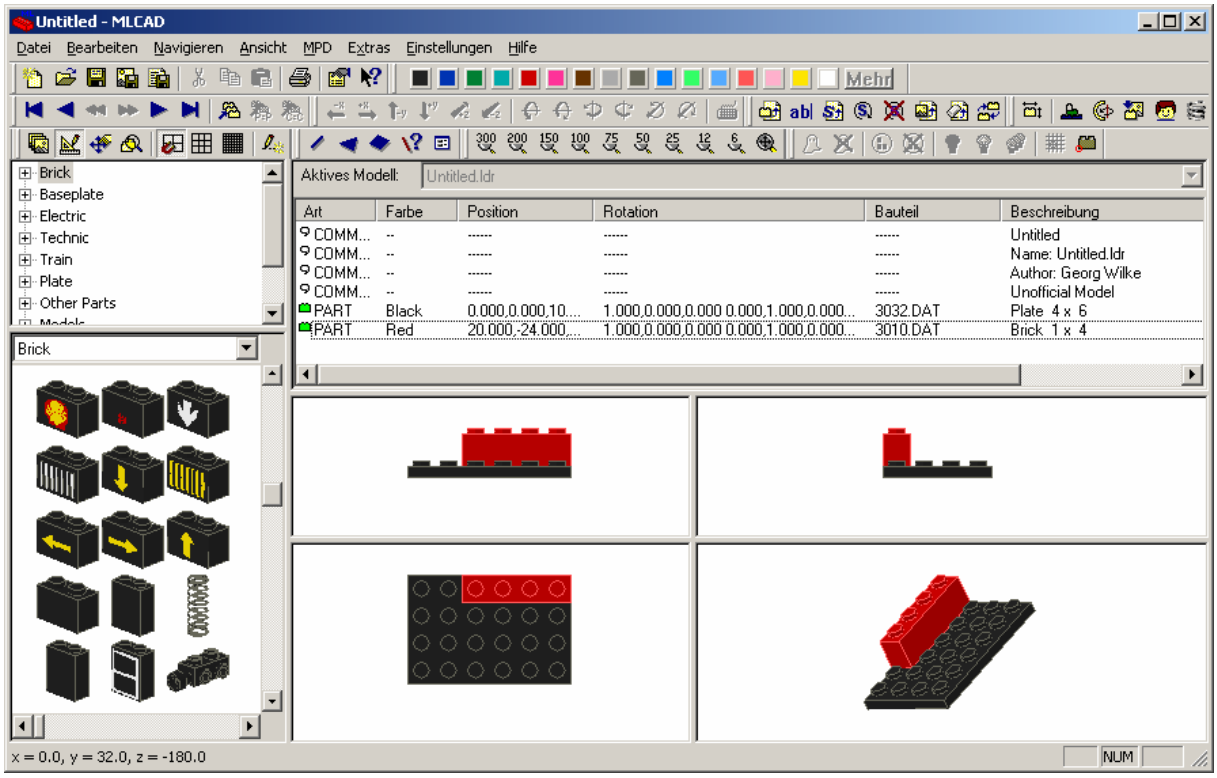
Nun sollte man noch die Symbolleisten noch etwas durch Drag-and-Drop mit der Maus ausrichten, um Platz für die unteren Fenster zu gewinnen. Nach Abschluss der Konfiguration sollte sich der Arbeitsbereich von *mlCAD* wie folgt präsentieren.



1.6.5 *mlCAD* benutzen

Der grundsätzliche Ablauf beim Erstellen eines neuen *LEGO*-Modells ist der folgende: Man wählt zunächst in der linken Spalte ein Bauteil aus (in der Vorschau oder der Bibliothek) und zieht es auf eines der vier Zeichenfenster. Es erscheint nun in drei Fenstern in jeweils einer Blickrichtung, im vierten (unten rechts) dagegen in 3D-Ansicht.
Ein zweites Teil wird nun hinzugefügt (evtl. wird die Farbe mit Hilfe der Farbsymbolleiste angepasst) und durch Verschieben so an das erste Teil gefügt, dass es wie zusammengebaut aussieht.
Achtung: es „rastet“ nicht ein!

Im Folgenden sieht man beispielsweise, wie eine kleine Platte (`Plate 4 x 6`) mit einem schmalen, roten „Vierer“ (`Brick 1 x 4`) zusammengefügt wurde.



Der Projektablauf gibt dabei die Bauschritte, die Position, die Stellung usw. des verwendeten Bauteils an.

Da eine weitergehende Erläuterung den Rahmen dieses Skriptes sprengen würde, sei hier dringend auf die Online-Hilfe von *mlCAD* und das Tutorial verwiesen. Dort wird auch beschrieben, wie man Teilmodelle (Submodels), Bauteillisten und Bildfolgen generiert.

2 Programmieren mit Java

Dieses Kapitel enthält nur eine sehr kurze Übersicht über einige Merkmale von Java. Eine „Java-Schule“ hätte den Umfang eines Buches und würde den Rahmen dieses Manuskriptes sprengen. Es spielt für das weitere Verständnis überhaupt keine Rolle, ob man nun *JavaEdit* oder *JCreator* verwendet, wir haben uns für *JCreator* entschieden.

2.1 Einfache Applikationen

2.1.1 Ein erstes, sehr einfaches Beispiel-Programm

Wir starten *JCreator,* klicken auf NEW, wählen in der Registerkarte PROJECTS des sich öffnenden Fensters als PROJEKTTYP `Empty Application`, geben im Eingabefeld PROJECT NAME `ApplicationDemos` ein und bestätigen mit OK. Dann klicken wir ein zweites Mal auf NEW, wählen aber jetzt die Registerkarte FILES und geben im Eingabefeld FILENAME `HalloWelt` ein. Dann öffnen wir die Datei `HalloWelt.java` und geben im Editor-Fenster den unten stehenden Text ein.

```
1         /**
2           * Ein sehr einfaches Beispiel
3           */
4         public class HalloWelt {
5             /**
6               * Gibt einen Text auf die Konsole aus
7               */
8             public static void main (String[] args) {
9                 System.out.println("Hallo Roboter-Welt!");
10            }
11        }
```

Anschließend klicken wir auf Compilieren (COMPILE FILE) und dann auf Ausführen (EXCECUTE FILE). Es öffnet sich ein Konsolen-Fenster:

Jedes Java-Programm besteht aus mindestens einer Klasse, deren Definition man mit dem Schlüsselwort `class` einleitet (Zeile 4). Der zugehörige Programmtext wird zwischen zwei geschweiften Klammern eingeschlossen. Falls das Java-Programm kein Applet ist, muss die ausführbare Klasse die statische Methode `main` enthalten (Zeile 8). Sie muss `public` („öffentlich“) sein, genau wie die Klasse[3], zu der sie gehört. Methoden haben nach ihrem Namen immer ein rundes Klammerpaar, das einen Parameter, eine durch Kommata getrennte Parameterliste enthalten oder auch leer sein kann. `main` hat genau einen Parameter `args`. `args` ist ein Feld von Zeichenketten (`String`), was man am eckigen Klammerpaar hinter `String` erkennen kann. Anschließend wird der Programmtext, der zur Methode gehört, in ein geschweiftes Klammerpaar eingeschlossen. Unsere Methode `main` enthält nur eine Anweisung (Zeile 9). Sie bedeutet, dass die Zeichenkette (erkennbar an den Anführungszeichen) `"Hallo Roboter-Welt!"` auf die Konsole ausgegeben wird.
Die Zeilen 1 bis 3 und 5 bis 7 enthalten so genannte Dokumentations-Kommentare. Kommentare erleichtern dem Programmierer die Arbeit weil sie Gedanken festhalten, die man später schnell vergessen hat, sie werden aber vom Compiler nicht beachtet.

[3] Man könnte `public` vor der Klassendefinition hier auch weglassen. Dennoch sollte man auf die korrekte Verwendung der Attribute `public` (= sichtbar aus allen Klassen), `protected` (= sichtbar nur innerhalb des eigenen Pakets) und `private` (= nur sichtbar innerhalb der eigenen Klasse) gleich von Anfang an achten. Bei der Objektorientierten Programmierung (OOP) spielen diese Attribute eine wichtige Rolle.

2.1.2 Anwendung der Programm-Parameter

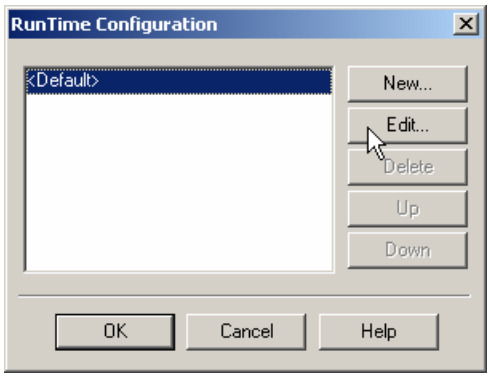
Um die Verwendung des Parameters `args` zu illustrieren, erstellen wir ein neues Java-Programm `ArgsDemo.java` und compilieren es.

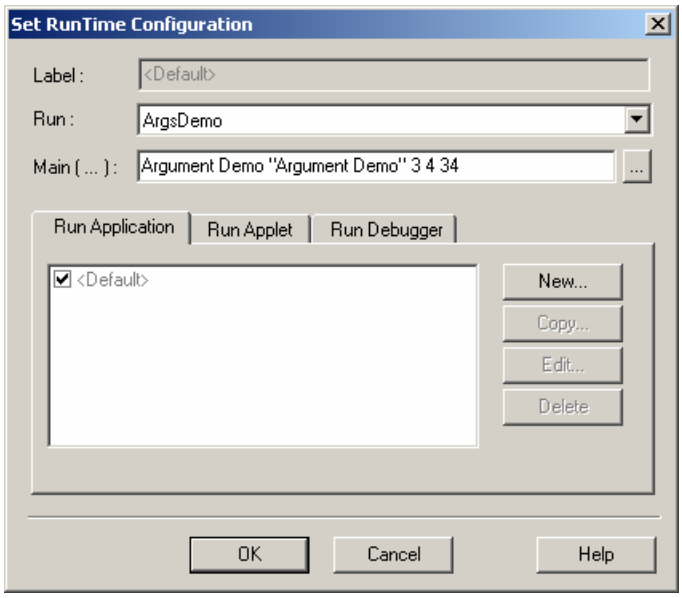
```
1         public class ArgsDemo {
2             public static void main(String[] args) {
3                 for(int i = 0; i < args.length; i++) {
4                     System.out.println(
5                         "Argument Nr. " + i + " = " + args[i]);
6                 }
7             }
8         }
```

Da wir dem Programm einige Parameter übergeben wollen, müssen wir hierzu das Menü BUILD RUNTIME CONFIGURATION ... aufrufen. In dem sich öffnenden Fenster wählen wir den Eintrag <DEFAULT> aus und klicken auf EDIT... Dann tragen wir im Feld MAIN(...); beliebige Buchstaben, Zahlen oder Wörter ein, jeweils durch Leerzeichen getrennt, wie z. B.

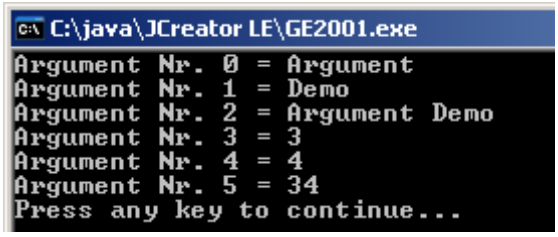
```
Argument Demo "Argument Demo" 3 4 34
```

(man beachte die Funktionsweise der Anführungszeichen!), schließen alle Fenster durch jeweils OK und starten das Programm.





Die Ausgabe ist folgende:



Jetzt sollte man die Eintragungen für die Parameter wieder entfernen. Es schadet nicht, wenn man jetzt das Programm nochmals laufen lässt und dabei auf die Ausgabe achtet.

2.1.3 Demo-Programm zur Verwendung des Pakets `eingabe`.

Das nächste Programm soll die Einbindung eines Paketes (`package`), erklärt werden. Zusammengehörige Klassen fasst man in Paketen zusammen. Dazu werden die Klassen-Dateien in einem gemeinsamen Ordner gespeichert. Man muss dabei beachten, dass Paket-Name und Ordner-Name übereinstimmen. Paket-Namen schreibt man meist nur mit Kleinbuchstaben.

Damit eine Klasse zu einem Paket gehört, muss sie dies in der ersten Porammzeile zum Ausdruck bringen durch die `package`-Anweisung. Der Quelltext der Klasse Eingabe im Paket `Eingabe` beginnt z. B. wie folgt:

```
package eingabe;
public class Eingabe { …
```

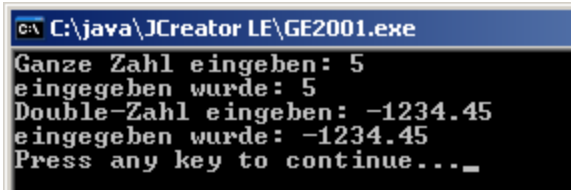
Listing des Beispiel-Programms `EingabeDemo`:

```
1   import eingabe.*;
2
3   public class EingabeDemo {
4       public static void main(String[] args) {
5           int i = Eingabe.readLineInt("Ganze Zahl eingeben: ");
6           System.out.println("eingegeben wurde: " + i);
7           double d = Eingabe.readLineDouble("Double-Zahl eingeben: ");
8           System.out.println("eingegeben wurde: " + d);
9       }
10  }
```

In Zeile 5 und 7 werden zwei Methoden aufgerufen, die in der Klasse Eingabe im Paket `eingabe` definiert sind. Um jetzt nicht jedes Mal

```
eingabe.Eingabe.readLineInt(…);
```

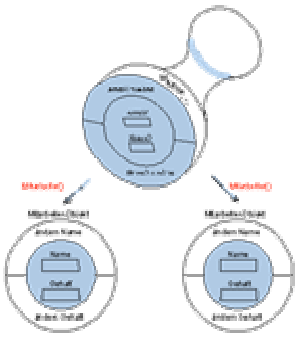
eingeben zu müssen, fügt man dem Quelltext am Anfang eine `import`-Anweisung hinzu (Zeile 1). Damit wird dem Compiler gesagt, wo er die betreffende Klasse zu suchen hat. Der Stern „`*`“ ist ein Platzhalter und bedeutet „alle Klassen im Paket `eingabe`“. Hat ein Paket weitere Unterpakete, muss jedes Verwendete Unterpaket getrennt eingegeben werden, wobei man UnterPakete durch einen Punkt „`.`“ voneinander trennt.
Die Ausgabe des Programms `EingabeDemo` lautet:



Aufgabe 1: Erstelle andere einfache Programme deiner Wahl.
Compiliere und starte sie.
Ergänze deine Programme mit Dokumentationskommentaren und erstelle dann mit `JavaDoc` die zugehörige Dokumentation.

2.1.4 Objektorientierte Programmierung, Klassen und Vererbung

Bären sind gefährliche Raubtiere, Eisbären sind ganz besondere Bären. Trotzdem braucht man vor den Begriffen „Bär“ und „Eisbär“ keine Angst zu haben. Erst wenn ich bei einem Zoobesuch erfahre, dass eben ein Eisbär, der auf den Namen „Fritz“ hört, aus seinem Gehege ausgebrochen ist, kommt bei mir leichte Panik auf. Und wenn er sich direkt vor mir aufrichtet, möchte ich rufen „Scotty, beam me up!“. Somit besteht ein gewaltiger Unterschied zwischen dem abstrakten Begriff „Eisbär“ und dem realen Exemplar „Fritz“.

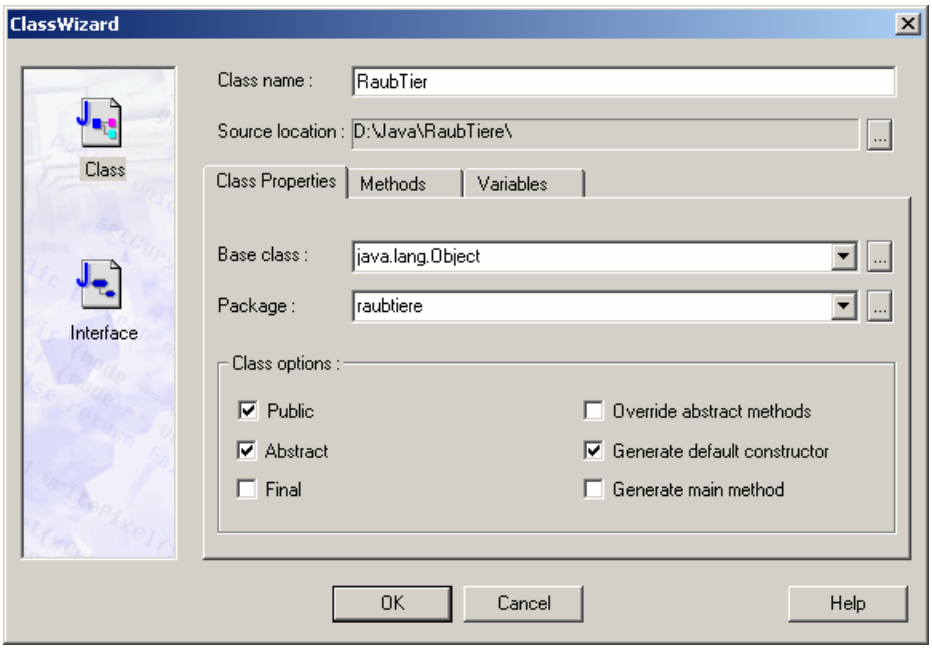


Genauso unterscheidet eine objektorientierte Programmiersprache zwischen Klassen und Objekten.[4] In der Objektorientierten Programmierung (OOP) sagen wir auch „Fritz“ ist eine Instanz oder Objekt der Klasse `EisBaer`.
Etwas allgemeiner können wir sagen, Klassen werden dazu benutzt, Dinge der realen und der gedachten Welt zu modellieren. Solche „Dinge“ können Mitarbeiter in einer Firma sein, oder eine komplizierte Gleichung in

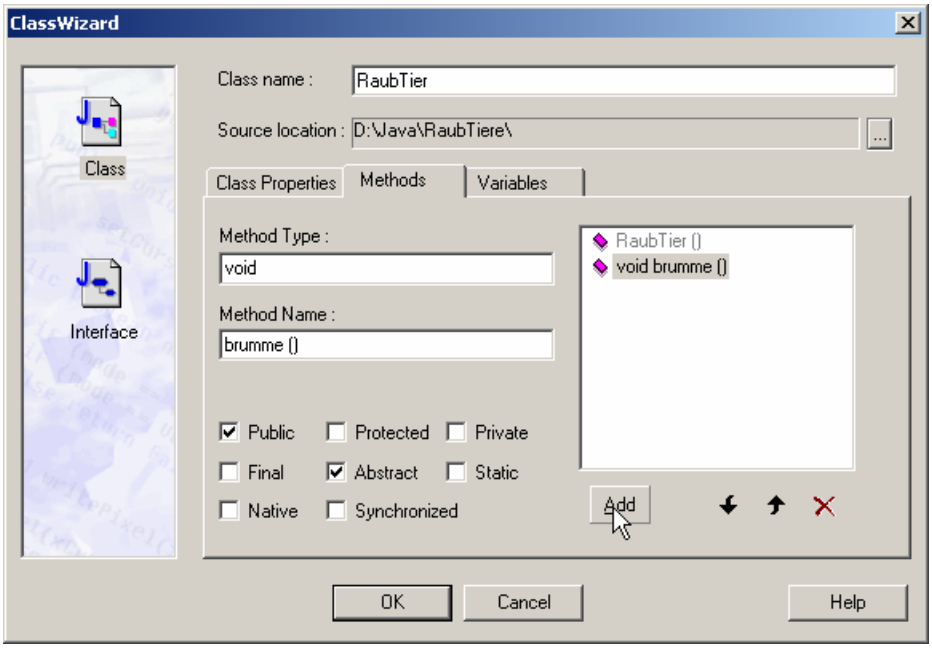
[4] Die Begriffe *Klasse* und *Objekt* werden hier im Sinne von Java verwendet. In der Theorie gibt es insbesondere für den Begriff *Objekt* andere Interpretationen.

der Mathematik. Eine Klasse können wir auch mit einem Stempel vergleichen. Jedes Mal, wenn wir den Stempel auf ein Papier drücken, erzeugen wir ein Objekt, für das der Stempel den Bauplan liefert. In die Felder „Name“ und „Gehalt“ (vgl. dazu die Abbildung) lassen sich dann konkrete Werte eintragen. Natürlich sind Objekte, die nicht auf dem Papier, sondern im Speicher eines Rechner existieren, wesentlich flexibler, so kann man mit Methoden den konkreten Namens- bzw. Gehaltseintrag wieder verändern, wenn dies z.B. durch Heirat bzw. Gehaltserhöhung angezeigt ist.

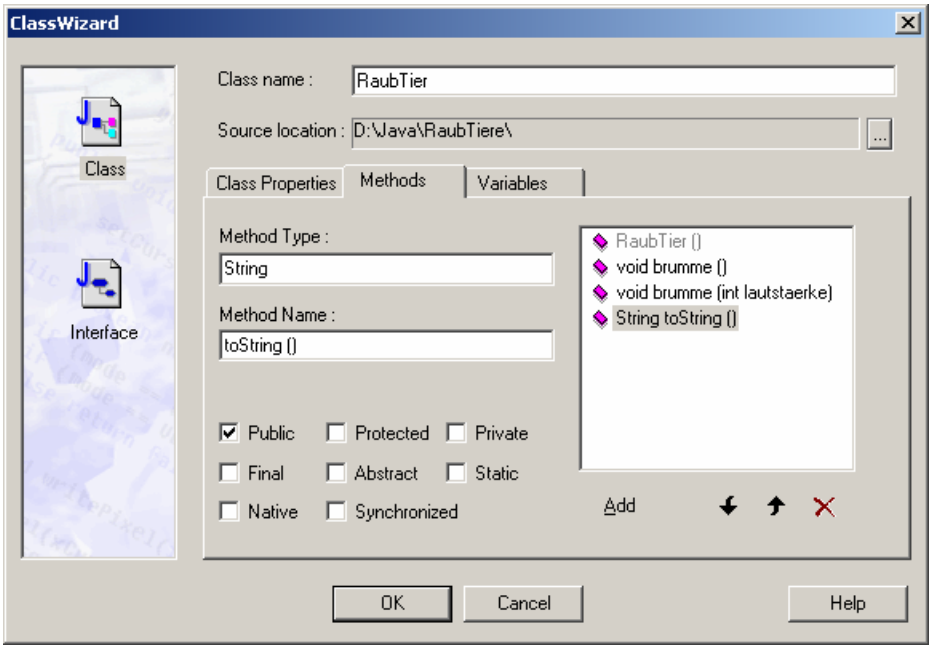
In dem folgenden Beispiel-Projekt werden insgesamt vier Klassen erzeugt, die alle irgendwie zusammenhängen. Dazu öffnet man *JCreator* und erzeugt mit NEW → PROJECTS → EMPTY APPLICATION ein neues Projekt `RaubTiere`.
Als nächstes erzeugt man mit PROJECTS → NEW CLASS die Klasse `RaubTier`, wobei man in der Registerkarte CLASS PROPERTIES die entsprechenden Eintragungen vornimmt. Die Auswahlfelder PUBLIC, ABSTRACT und GENERATE DEFAULT CONSTRUKTOR werden angekreuzt. `RaubTier` ist eine abstrakte Klasse, da eine ihrer Methoden als `abstract` deklariert ist. Wichtig ist der Eintrag `raubtiere` im Eingabefeld PACKAGE. Dadurch wird ein gleichnamiges Paket erzeugt und im Unterordner `raubtiere` von unserem Projektordner `D:\Java\RaubTiere` erstellt.



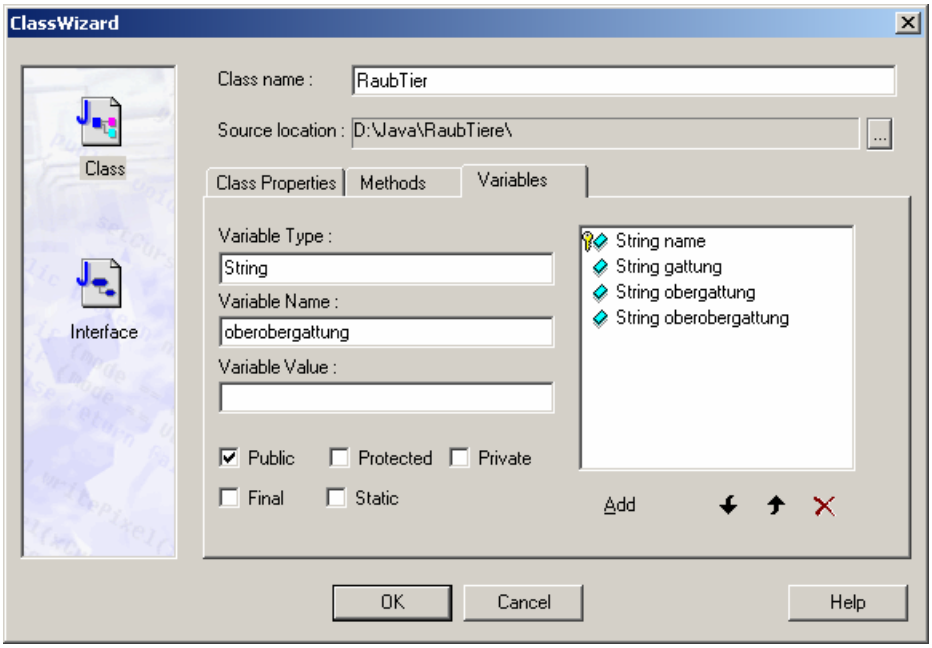
In der Registerkarte METHODS fügt man eine abstrakte Methode `brumme()` mit leerer Parameterliste hinzu.



Dann noch eine Methode `brumme(int lautstaerke)` und eine Methode `toString()`, die beide nicht abstrakt sind.



In der Registerkarte VARIABLES fügt man die benötigten Objektvariablen[5] hinzu, wobei `name` das Attribut `protected` erhält, da andere Klassen aus dem Paket `raubtiere` auf diese Variable zugreifen können sollen.



Nach entsprechenden Ergänzungen ist die Klasse Raubtier fertig.

[5] Von *Objektvariablen* spricht man, wenn eine Variable dem gesamten Objekt „gehört“. Für jede neue Instanz ein es Objekts, wir auch eine neue Objektvariable erzeugt.
Klassenvariablen dagegen „gehören“ allen Instanzen einer Klasse gleichzeitig. Eine Klassenvariable wird nur einmal im Speicher angelegt. Ändert eine Instanz den Wert einer Klassenvariablen, wirkt sich diese Änderung auch auf alle anderen Instanzen aus. Eine Klassenvariable erkennt man an dem Bezeichner `static`.

Das Listing zur RaubTier:

```
1   package raubtiere;
2   public abstract class RaubTier {
3       protected String name;
4       private String gattung, obergattung, oberobergattung;
5       public RaubTier(String name) {
6           this.name = name;
7           gattung = getClass().getName();
8           obergattung = getClass().getSuperclass().getName();
9           oberobergattung =
10              getClass().getSuperclass().getSuperclass().getName();
11      }
12      public abstract String brumme();
13      public String brumme(int lautstaerke) {
14          String roehr = "";
15          for(int i = 0; i< lautstaerke; i++)
16              roehr += "R";
17          return brumme() + roehr;
18      }
19      public String toString() {
20          return "Das ist ein " + gattung + "\n" +
21                 "Sein Vorfahr ist ein " + obergattung + "\n" +
22                 "Sein Ur-Vorfahr ist ein " + oberobergattung + "\n" +
23                 "Sein Name ist " + name;
24      }
25  }
```

Die Methode `brumme()` in Zeile 12 hat noch keinen Methoden-Rumpf, da man nicht wissen kann, wie ein Raubtier brummt, wenn man noch gar nicht weiß, um welche Art Raubtier es sich handelt. Das einzige, was wir wissen, ist wie ein Raubtier laut brummt, wenn wir wüssten, wie es denn überhaupt brummt. Somit kann die Methode `brumme(int lautstaerke)` in Zeile 13 schon erstellt und die Methode `brumme()`verwendet werden, obwohl sie erst später (d. h in von `RaubTier` abgeleiteten Unterklassen) erstellt wird.
In der Klasse `RaubTier` gibt es zwei Methoden, die `brumme` heißen. Man sagt deshalb, diese Methode ist überladen (Overloading). Java kann sie trotzdem unterscheiden, da sie sich in ihrer Signatur, d. h. in ihrer Parameter-Liste, unterscheiden.
In Zeile 3 steht vor der Objekt-Variablen `name` das Attribut `protected`. Somit kann aus allen Klassen im Paket `raubtiere` auf diese Variable zugegriffen werden, nicht jedoch von Klassen aus, die nicht zu `raubtiere` gehören. Die Klasse `RaubTierDemo` weiter unten sollte eigentlich nicht zum Paket `raubtiere` gehören, da sie quasi nur eine Starter-Datei für die Raubtier-Klassen darstellt. Man sollte dort die 1. Zeile `import raubtiere.*;` löschen. Dann würden aber die beiden Anweisungen in den Zeilen 7 und 12 von `RaubTierDemo` nicht mehr ausgeführt werden können. Um dieses Problem zu vermeiden, bietet man für Objektvariable immer entsprechende öffentliche Methoden `setXYZ` und `getXYZ` an. Die Vorsilben `set` und `get` sind inzwischen Standard selbst bei deutschen Bezeichnern geworden.

Unsere Beispielklasse `RaubTier` sollte man durch die beiden folgenden Methoden erweiten:

```
1   public void setName(String name) {
2       this.name = name;
3   }
4   public String getName() {
5       return name;
6   }
```

Anmerkung:
Da in der Methode `setName` der Parameter `name` denselben Bezeichner hat wie die Objektvariable `name`, muss man sie innerhalb von `setName` unterscheiden; `this.name` ist die Objektvariable und `name` der Parameter.

Das Gleiche gilt für die Variablen in Zeile 4 von `RaubTier`, sie haben das Attribut `private`. Auf diese Variablen kann nur innerhalb von RaubTier zugegriffen werden. In andern Klassen sind diese Variablen nicht bekannt. Das nennt man Kapselung (Information Hiding), ein ganz wichtiges Prinzip in der OOP.
Nur das Objekt selbst weiß, wie es mit seinen Daten umzugehen hat. Da könnte ein Zugriff von außen, z. B. durch eine unerlaubte Wertzuweisung fatale Folgen haben. Man stelle sich z. B. vor, dass in dem Beispiel des Angestellten von oben, man auf seinen Namen oder sein Gehalt beliebig Zugriff hätte. Man wird somit die Variable `gehalt` als `private` deklarieren und öffentliche Methoden (`public`) wie `getGehalt()` und `setGehalt(...)` erstellen. In `setGehalt(...)` kann man jetzt Absicherungen einbauen. Ein Gehalt sollte z. B. einen bestimmten Mindest- oder Höchstwert nicht unter- bzw. überschreiten.

Jetzt wollen wir eine Klasse `Baer` erstellen. Das machen wir ebenfalls mit PROJECT → NEW CLASS. Wir wissen, dass ein Bär ein Raubtier ist, wir glauben auch zu wissen, wie ein Bär brummt.
Damit gestaltet sich die Definition der neuen Klasse `Baer` recht einfach:

```
1   package raubtiere;
2   public class Baer extends RaubTier {
3       public Baer(String name) {
4           super(name);
5       }
6       public String brumme() {
7           return "wwwr";
8       }
9   }
```

In Zeile 2 steht das Schlüsselwort `extends`. Das Bedeutet, dass `Baer` eine Unterklasse von `RaubTier` ist und damit alle Eigenschaften von `RaubTier` hat. In Zeile 4 wird der Konstruktor der Oberklasse von `RaubTier` aufgerufen. Das geschieht durch das Schlüsselwort `super`. Die einzige wirkliche Arbeit in dieser Klasse ist den Bär zum Brummen zu bringen (Zeile 6).

Ein Eisbär ist ein Bär und ist ebenfalls ein Raubtier. Wie Eisbären brummen ist bekannt.

Listing von EisBaer:

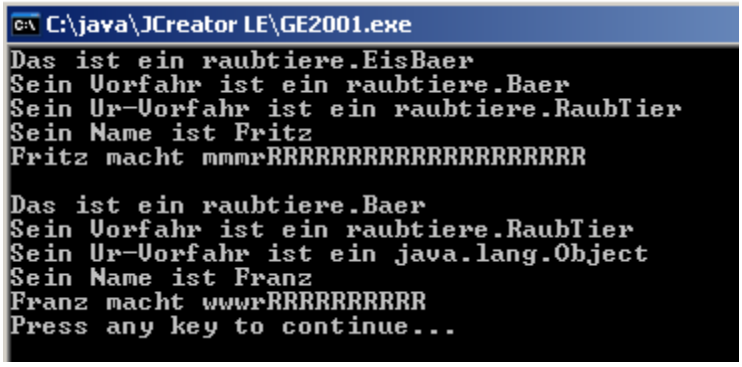
```
1   package raubtiere;
2   public class EisBaer extends Baer {
3       public EisBaer(String name) {
4           super(name);
5       }
6       public String brumme() {
7           return "mmmr";
8       }
9   }
```

Das sieht genauso aus wie bei `Baer`, nur dass `EisBaer` von `Baer` und nicht von `RaubTier` abgeleitet ist.
Die Klasse `Baer` leitet sich direkt von `RaubTier` ab und die Klasse `EisBaer` wieder direkt von `Baer`. Diese beiden Klassen sind nicht mehr abstrakt, da dort die Methode `brumme` jeweils implementiert ist. Von diesen Beiden Klassen lassen sich Instanzen wie der Eisbär „Fritz“ oder der Bär „Franz“ erstellen.
Anmerkung:
Eigentlich müsste `Baer` ein Grizzly- oder Braunbär sein um eine konkrete Instanz erzeugen zu können, dann kann man aber Eisbär nicht mehr von Braunbär anleiten und dann hinkt unser Beispiel etwas.

Jetzt benötigen wir nur noch eine Klasse `RaubTierDemo`, wo wir uns einen ganz bestimmten Eisbären „Fritz“ und einen Bären „Franz“ erzeugen:

```
1         package raubtiere;
2         public class RaubTierDemo {
3             public static void main (String[] args) {
4                 EisBaer fritz = new EisBaer("Fritz");
5                 System.out.println(fritz);
6                 System.out.println(
7                     fritz.name + " macht " + fritz.brumme(20));
8                 System.out.println();
9                 Baer franz = new Baer("Franz");
10                System.out.println(franz);
11                System.out.println(
12                    franz.name + " macht " + franz.brumme(10));
13            }
14        }
```

Die Ausgabe dieses Programms ist:



In den Zeilen 7 und 6 von `RaubTier` werden Ober- und Ober-Ober-Klasse von `EisBaer` bzw. `Baer` ermittelt. Bei `EisBaer` ist das klar: `Baer` und `RaubTier`.
Die Oberklasse von `Baer` ist auch klar: `RaubTier`. Welches ist aber die Ober-Klasse von `RaubTier`? Die Ausgabe schafft Klarheit: jede Klasse ist von der Klasse `Object` abgeleitet. Würde man nach der Oberklasse `Object` von fragen, erhielte man eine Fehlermeldung.

Aufgabe 2: Vielleicht kennst du noch andere Raubtiere.
Erzeuge die zugehörigen Klassen, berücksichtige dabei die Vererbungsstrukturen.
Ergänze das Hauptprogramm entsprechend und teste es.

2.1.5 Objektorientierte Programmierung bei GUI-Klassen, Ereignissteuerung

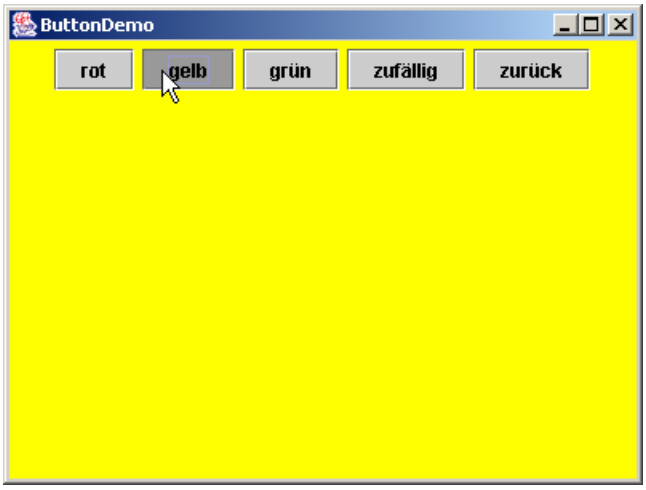
Klassen zur Erstellung von Windows-Applikationen stellen Fenster, Schaltflächen, Eingabefelder usw. dar. Das Werkzeug zu ihrer Erstellung nennt man Graphical User Interface (GUI). Das Paket `swing`, das im *JDK* enthalten ist, ist ein GUI. Viele seiner Klassen sind von dem älteren GUI-Paket AWT (Abstract Windowing Toolkit) abgeleitet. Die meisten Swing-Objekte haben ein äquivalentes Objekt in AWT. Zur Unterscheidung haben Swing-Klassen meist ein „J“ in ihrem Klassennamen vorangestellt. Die einzelnen Swing-Klassen sind über viele Ebenen voneinander abgeleitet. Für den Anwender ist es deshalb leicht, optisch ansprechende Programme zu erstellen, da die Funktionsweisen der einzelnen GUI-Elemente schon „vorgefertigt“ sind. Man muss von ihnen nur noch geeignete Unterklassen bilden und mit wenigen Programmzeilen ihnen Leben einhauchen.

Wir erstellen ein neues Projekt `ButtonDemo`. Das Grundgerüst wir an wenigen Stellen abgeändert und um einige Elemente ergänzt:

```
1         import java.awt.*;
2         import java.awt.event.*;
3         import javax.swing.*;
4         class ButtonDemo extends JFrame implements ActionListener {
5             JButton bnRot, bnGelb, bnGruen, bnZufall, bnReset;
6             JPanel panel;
7             Color bgColor;
```

```
 8              public ButtonDemo(String title) {
 9                 super(title);
10                 addWindowListener(new WindowAdapter() {
11                     public void windowClosing(WindowEvent evt) {
12                         dispose();
13                         System.exit(0);
14                     }
15                 });
16                 panel = new JPanel();
17                 panel.setLayout(new FlowLayout());
18                 setContentPane(panel);
19                 bgColor = panel.getBackground();
20                 bnRot = new JButton("rot");
21                 bnRot.addActionListener(this);
22                 panel.add(bnRot);
23                 bnGelb = new JButton("gelb");
24                 bnGelb.addActionListener(this);
25                 panel.add(bnGelb);
26                 bnGruen = new JButton("grün");
27                 bnGruen.addActionListener(this);
28                 panel.add(bnGruen);
29                 bnZufall = new JButton("zufällig");
30                 bnZufall.addActionListener(this);
31                 panel.add(bnZufall);
32                 bnReset = new JButton("zurück");
33                 bnReset.addActionListener(this);
34                 panel.add(bnReset);
35                 setSize(400, 300);
36                 setVisible(true);
37             }
38             public void actionPerformed(ActionEvent event) {
39                 Object obj = event.getSource();
40                 if (obj == bnRot) panel.setBackground(Color.RED);
41                 if (obj == bnGelb) panel.setBackground(Color.YELLOW);
42                 if (obj == bnGruen) panel.setBackground(Color.GREEN);
43                 if (obj == bnZufall) {
44                     float r = (float)Math.random();
45                     float g = (float)Math.random();
46                     float b = (float)Math.random();
47                     panel.setBackground(new Color(r, b, g));
48                 }
49                 if (obj == bnReset) panel.setBackground(bgColor);
50             }
51             public static void main (String[] args) {
52                 new ButtonDemo("ButtonDemo");
53             }
54         }
```

Das Ergebnis dieser Zeilen kann sich durchaus schon sehen lassen:



In den Zeilen 1 bis 3 werden die benötigen GUI-Pakete importiert. Zeile 4 leitet ButtonDemo von einem `JFrame` ab. Damit kann unser Programm schon alles, was ein richtiges *Windows*-Fenster auch kann, man kann es vergrößern, verkleinern, verschieben usw. Neu ist das Schlüsselwort `implements`. Mit `implements` wird ein `Interface` implementiert. `Interfaces` sind den Klassen recht ähnlich, mit einem wesentlichen Unterschied: keine Methode hat einen Methoden-Rumpf. Das ist nicht dasselbe, als wenn in einer Klasse alle Methoden `abstract` wären. Weitere Details hierzu würden jedoch den Rahmen dieses Manuskriptes sprengen. Nur soviel, mit der Verwendung von `Interfaces` kann man das Problem der Mehrfachvererbung[6] umgehen, die in Java nicht erlaubt ist. Mit `implements ActionListenener` wird unser `ButtonDemo` zusätzlich noch zu einem Aktionsabhorcher (`ActionListener`). `ButtonDemo` meldet sich bei den einzelnen Schaltflächen (`JButton`) in den Zeilen 21, 24, 27, 30 und 33 mit der Methode `addActionListener` bei den jeweiligen Schaltflächen als Empfänger einer Botschaft der jeweiligen Schaltfläche an. Übergeben wird dieser Methode der jeweilige Aktionsabhorcher. In diesem Fall ist es `ButtonDemo` selbst, erkennbar an dem Bezeichner `this`. Sobald eine Schaltfläche gedrückt wird, sendet diese eine Botschaft an alle angemeldeten Aktionsabhorcher. Diese können dann auf diese Botschaft individuell reagieren. Die Reaktionsweise auf eine Aktion wird in der Methode `actionPerformed` (Zeile 38) festgelegt. Da hier auf die Botschaft aller Schaltflächen mit derselben `actionPerformed`-Methode reagiert wird, muss man den Sender der Botschaft vor dem Reagieren erst ausfindig machen. Das geschieht mit der Methode `getSource` (Zeile 39). Wenn man dann weiß wer der Sender war, kann man auf dessen Botschaft entsprechend reagieren (Zeilen 40 bis 49).
Diese Verfahren zum Behandeln von Ereignissen nennt man in Java `Delegation Event Model`.

Aufgabe 3: Füge dem Programm `ButtonDemo` noch eine weitere Schaltfläche für „blau“ hinzu.

Aufgabe 4: Mache dich in der Java API-Dokumentation kundig, welche Swing-Objekte es noch gibt. Ergänze das Programm `ButtonDemo` durch weitere Swing-Objekte, wie z. B. Textfelder (`JTextField`), Textflächen (`JLabel`) oder Kontrollkästchen (`JCheckBox`).

[6] Bei der Mehrfachvererbung würde eine Klasse von mehreren Klassen erben. Dann ergibt sich aber ein Problem, wenn in den beiden übergeordneten Klassen gleiche Objektvariablen oder Methoden verwendet werden. Für welche Methode soll sich dann sie untergeordnete Klasse entscheiden, für die aus der einen oder anderen oder einer weiteren übergeordneten Klasse?

2.1.6 Kommentare in Java

Es gibt in Java drei Arten von Kommentaren:

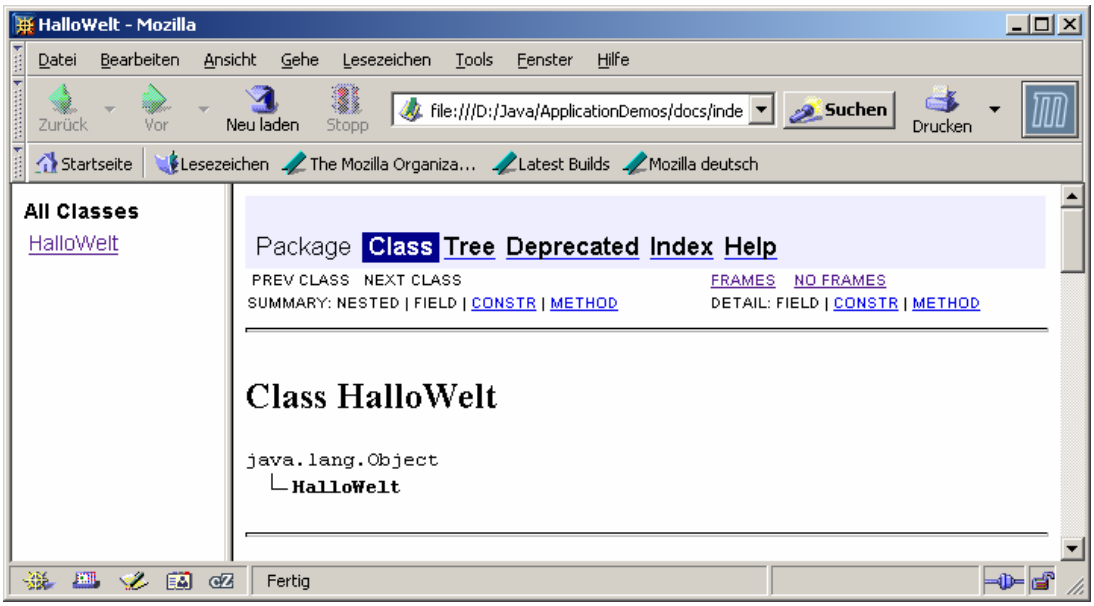
`// ...` Dieser Kommentar beginnt nach den beiden Schrägstrichen und gilt bis zum Ende der Zeile. Er enthält knappe Erläuterungen meist zu einer Programmzeile.

`/* ... */` Der Text zwischen Schrägstrich-Stern und Stern-Schrägstrich wird als Kommentar aufgefasst. Er kann sich auch über mehrere Zeilen erstrecken. Mit ihm lassen sich auch zu Testzwecken Programmteile ausblenden.

`/**... */` Dokumentations-Kommentar. Er kann sich auch über mehrere Zeilen erstrecken. Jede Kommentarzeile zwischen den Symbolen beginnt mit einem Stern, gefolgt con einem Leerzeichen.

Aus Dokumentations-Kommentaren kann man mit Hilfe des Werkzeugs `javadoc.exe` sehr einfach Dokumente im html-Format erstellen. Durch Klicken auf das entsprechende selbst User-Tool-Symbol geschieht das automatisch. Wenn man anschließend auf das User-Tool-Symbol zum Anzeigen der selbst erstellten Dokumentation klickt, wird diese in *Mozilla* angezeigt.

Aus der Beispiel-Datei `HalloWelt.java` wird dann folgende Dokumentation:

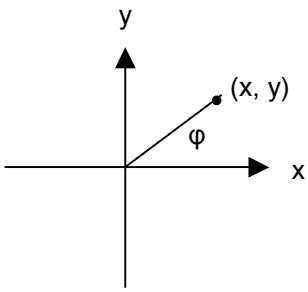


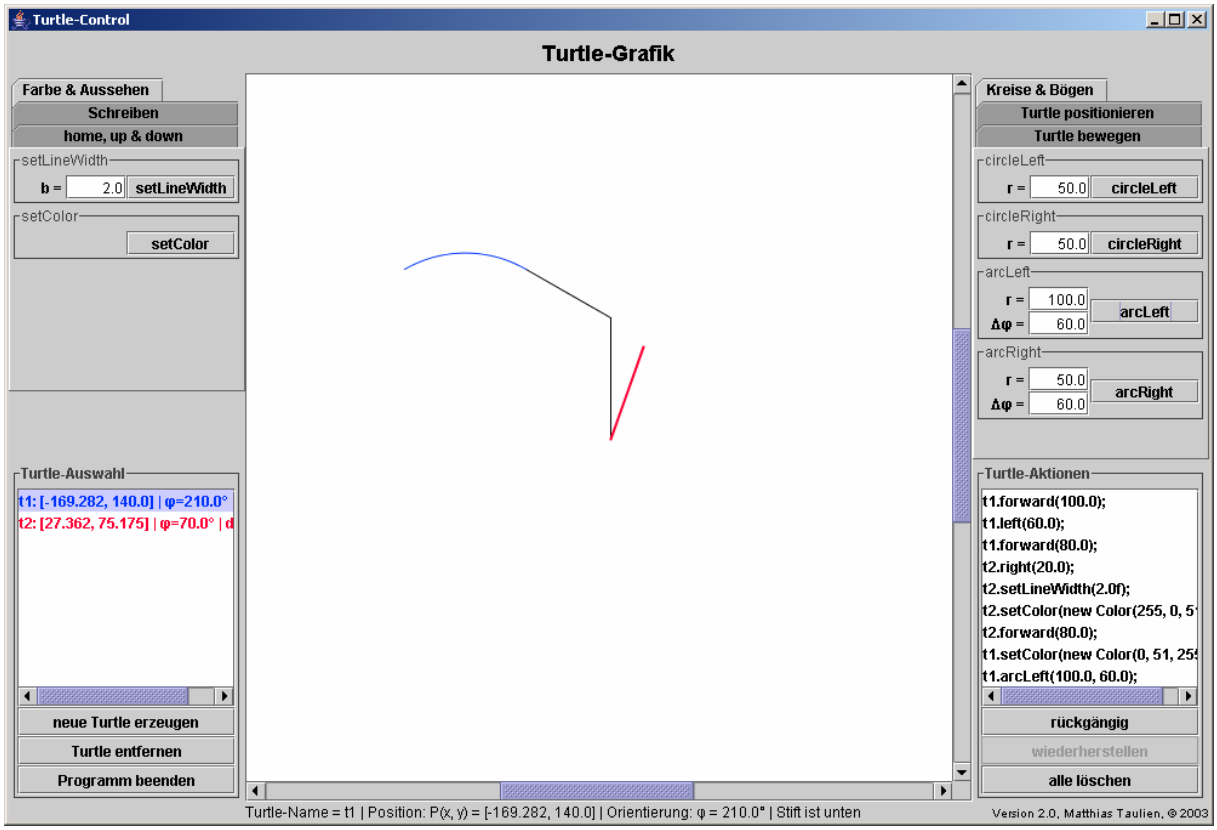
2.2 Programmieren lernen mit einer Turtle-Grafik

Um den Umgang mit Klassen, Objekten und Methoden zu erlernen haben wir ein Turtle-Projekt entwickelt.
Turtle-Grafiken wurden in den unterschiedlichsten Programmiersprachen entwickelt. Sehr interessant ist dabei die Programmiersprache *Logo*. Im Internet findet man ein interessantes Projekt *MSWLogo*, das Entwicklungsumgebung, Beispiele und Anleitungen kostenfrei auf Englisch oder Deutsch anbietet. Unsere Java-Turtle versteht die englischen Turtle-Anweisungen.
Eine Turtle (Schildkröte) macht genau das, was Schildkröten im allgemeinen tun, sie bewegt sich ein gewisses Stück vorwärts, dann dreht sie sich ein wenig nach links oder rechts, bewegt sich wieder ein bestimmtes Stück usw. Im Sand würde diese Schildkröte eine Spur hinterlassen. Unsere Java-Turtle hat dazu einen Stift, den sie auf eine Zeichenfläche absetzen kann, dann sieht man die Spur, sie kann den Stift aber auch hochnehmen, dann sieht man die Spur nicht.

2.2.1 Erlernen der Turtle-Befehle

Unsere Turtle kann eine ganze Menge mehr. Sie kennt die Himmelsrichtungen Norden, Süden, Westen, Osten, sie hat einen eingebauten Längen- und Winkelmesser, sie kennt Koordinaten und sie kann springen. Weiterhin kann sie die Farbe und die Dicke ihres Zeichenstifts wählen und sie kann sogar an eine bestimmte Stelle der Zeichenfläche einen kleinen Text schreiben. Auch kann sie auf Kreisen oder Kreisbögen entlanglaufen.
Dann können sich zu ihr weitere Turtles dazu gesellen, da alle auf derselben Zeichenfläche ihre Spuren hinterlassen.
Um sich mit den Turtle-Befehlen vertraut zu machen, gibt es auf der CD ein kleines Java-Programm `JTurtleControl.jar`, das man einfach mit einem Doppelklick starten kann, sofern das SDK bzw. JRE installiert ist.
Es öffnet sich folgendes Fenster:





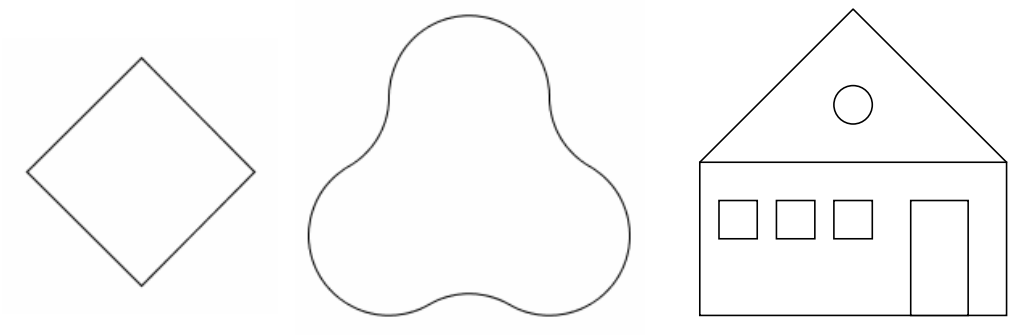
Die Schaltflächen tragen die Namen der Turtle-Befehle, falls die Parameter benötigen, kannst du sie in die entsprechenden Textfelder eingeben. Sobald du auf die Schaltfläche klickst, wird der Turtle-Befehl ausgeführt. Das Ergebnis siehst du auf der Zeichenfläche im mittleren Fenster, gleichzeitig

werden die Turtle-Anweisungen rechts mitprotokolliert. Falls du mehrere Turtles erzeugt hast, kannst du die jeweilige Turtle, die die nächste Aktion ausführen soll, im linken Fenster auswählen.

Aufgabe 5: Startet das Programm `JTurtleControl.jar` und macht euch mit der Funktionsweise der verschiedenen Turtle-Anweisungen vertraut.

Aufgabe 6: Die folgenden Grafiken sollen mit Hilfe von `JTurtleControl` erstellt werden. Die benötigten Anweisungen müssen dabei genau mitprotokolliert werden.

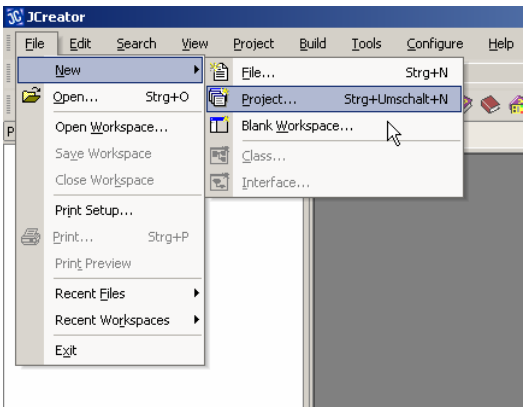
a) b) c)



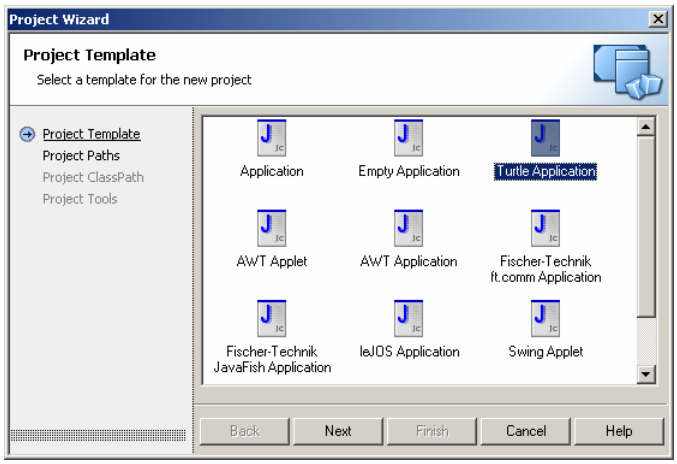
Entwirf auch eigene Grafiken und zeichnet sie mit der Turtle.

2.2.2 Eigene Java-Turtle Programme erstellen

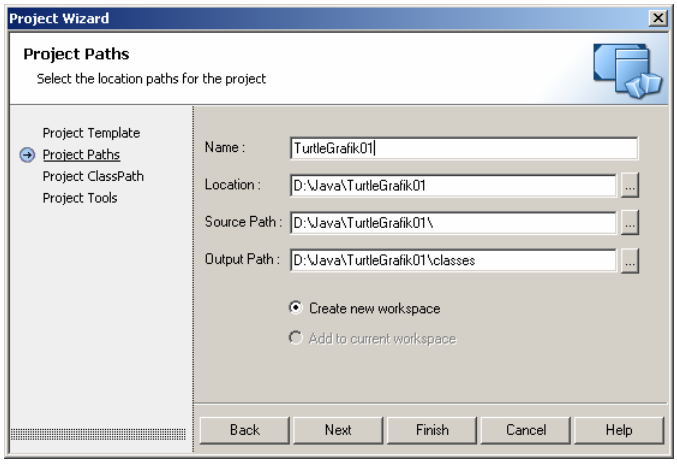
Die soeben erstellten Grafiken sollen jetzt programmiert werden.
Starte dazu das Programm `JCreator LE`. Mit FILE → NEW → PROJECT öffnet sich ein Fenster.



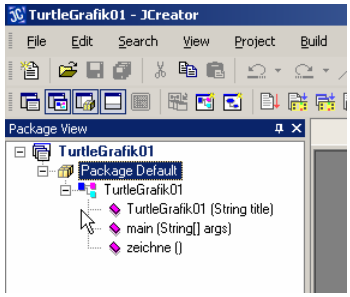
Wähle dabei die Projekt-Vorlage (Template) TURTLE APPLICATION und klicke auf NEXT.



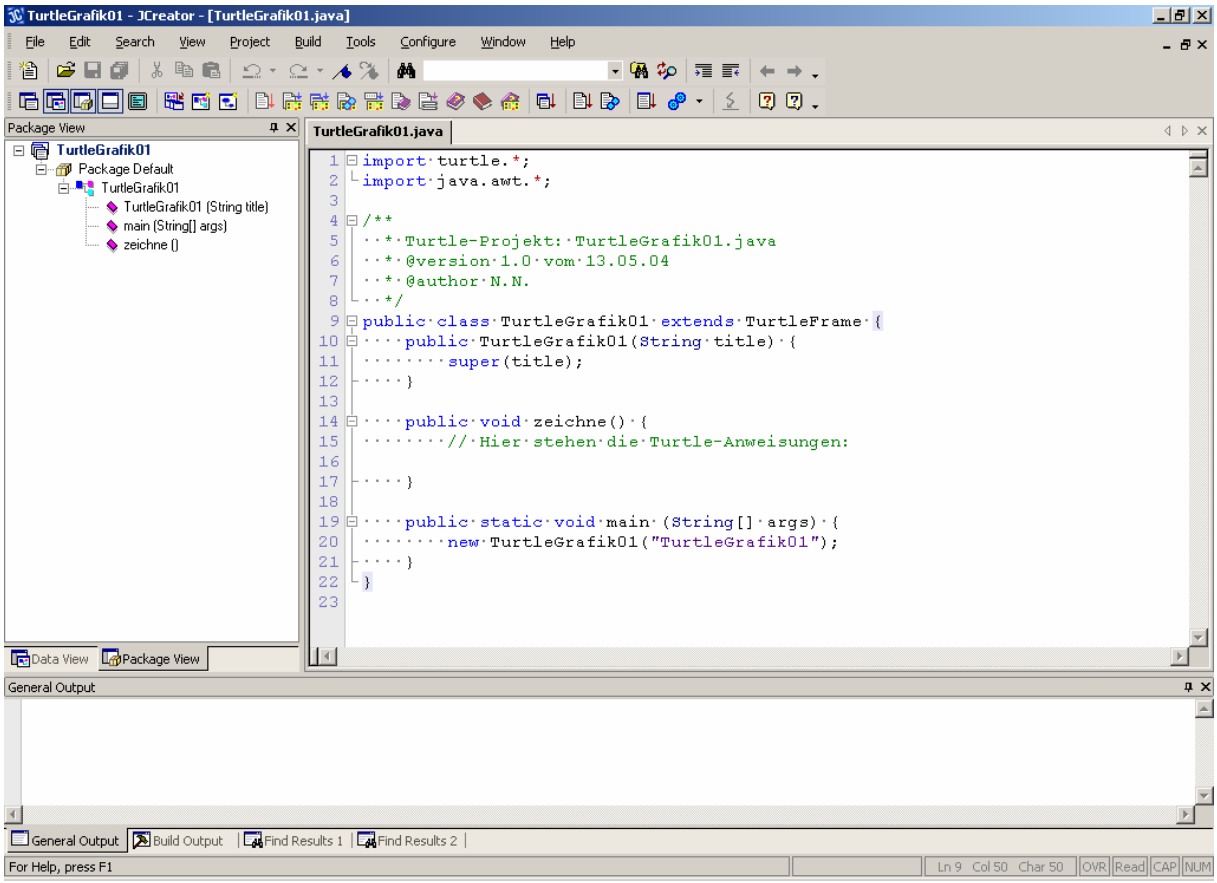
Gib unter NAME: `TurtleGrafik01` ein, die restlichen Einträge werden automatisch angepasst. Weiter geht es mit NEXT.



Es folgen Abfragen zu den Standardeinstellungen, die mit den aktuellen Voreinstellungen akzeptiert werden,
Im linken Fenster (PACKAGE VIEW) wird das erzeugte Projekt angezeigt. Klickt man auf die Plus-Zeichen, werden weitere Details des Projekts angezeigt.



Mit einem Doppelklick auf den Dateinamen `TurtleGrafik01` öffnet sich der Editor.



Klicke in die Zeile 16 und gib folgenden Text ein:

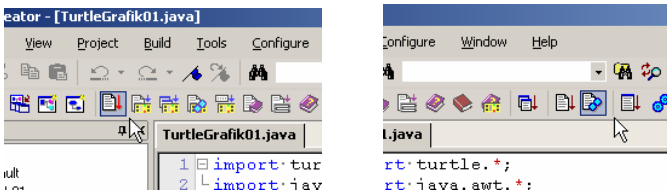
```
t.forward(100);
t.left(60);
t.forward(80);
```

Der gesamte Programmtext lautet dann:

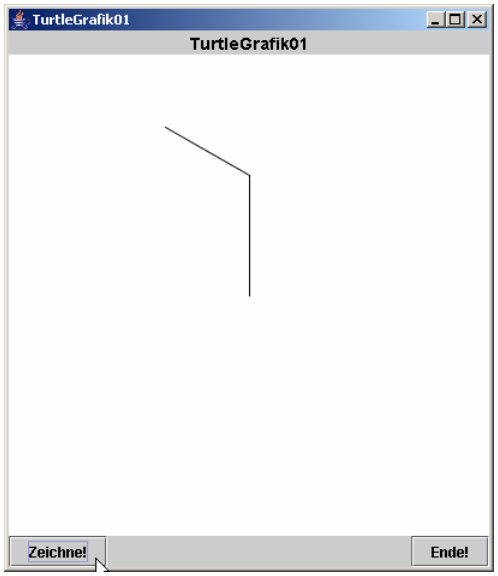
```
1   import turtle.*;
2   import java.awt.*;
3
4   /**
5     * Turtle-Projekt: TurtleGrafik01.java
6     * @version 1.0 vom 13.05.04
7     * @author N.N.
8     */
9   public class TurtleGrafik01 extends TurtleFrame {
10      public TurtleGrafik01(String title) {
11          super(title);
12      }
13
14      public void zeichne() {
15          // Hier stehen die Turtle-Anweisungen:
16          t.forward(100);
17          t.left(60);
18          t.forward(80);
19      }
20
21      public static void main (String[] args) {
22          new TurtleGrafik01("TurtleGrafik01");
23      }
24  }
```

In der Zeile 2 steht die Anweisung, das Paket `turtle` einzubinden, damit dem Programm alle Turtle-Klassen bekannt sind. In Zeile 9 steht die Definitionen einer eigenen Klasse `TurtleGrafik01`, die von einer Klasse `TurtleFrame` aus dem Paket `turtle` abgeleitet wird. TurtleFrame sorgt dafür, dass ein `JFrame` erstellt wird, der eine Zeichenfläche und die beiden Schaltflächen ZEICHNE! und ENDE enthält. Damit diese Turtle etwas zeichnen kann, müssen die Turtle-Befehle in der Methode `zeichne` (ab Zeile 14) aufgelistet werden.

Zunächst muss das Programm kompiliert werden (am besten mit `Jikes`), dann kann man es starten.



Klickt man auf ZEICHNE! wird die Turtle-Spur gezeichnet.



Schönere Bilder kann man diese zunächst mit `JTurtleControl` erstellen. Die mitprotokollierten Turtle-Anweisungen kann man im rechten Fenster mit der Maus markieren und mit <CTRL>-C in die Zwischenablage kopieren. Im Editorfenster von *JCreator* kann man sie anschließend, beginnend mit der Zeile 16, mit der Tastenkombination <CTRL>-V einfügen.
Da die Turtle-Grafik-Vorlage nur eine Turtle mit Namen `t` kennt, `JTurtleControl` die Turtle dagegen mit `t1` bezeichnet, muss man nach der Zeile 9 dem Programm mit

```
Turtle t1 = new Turtle(tWin);
```

den abweichenden Turtle-Namen noch bekannt machen.

Ein vollständiges Programm könnte z. B. so aussehen:

```
1  import turtle.*;
2  import java.awt.*;
3
4  /**
5    * Turtle-Projekt: TurtleGrafik01.java
6    * @version 1.0 vom 13.05.04
7    * @author N.N.
8    */
9  public class TurtleGrafik01 extends TurtleFrame {
10     Turtle t1 = new Turtle(tWin);
11
12     public TurtleGrafik01(String title) {
13         super(title);
14     }
15
16     public void zeichne() {
17         // Hier stehen die Turtle-Anweisungen:
18         t1.forward(100);
19         t1.left(90);
20         t1.forward(100);
21         t1.left(90);
22         t1.forward(100);
23         t1.left(90);
24         t1.forward(100);
25         t1.left(90);
26     }
27
28     public static void main (String[] args) {
29         new TurtleGrafik01("TurtleGrafik01");
30     }
31 }
```

Falls du weitere Turtles erzeugen möchtest, musst du die Zeile 10 entsprechend ergänzen:

```
Turtle t1 = new Turtle(tWin);
Turtle t2 = new Turtle(tWin);
Turtle t3 = new Turtle(tWin);
...
```

2.2.3 Eigene Turtle-Methoden erstellen

Das weiter oben gezeichnete Haus besteht aus vielen ähnlichen Elementen. Da wäre es doch praktisch, wenn man nicht immer das Gleiche eingeben müsste. Um z.B. ein Rechteck zu programmieren und es dann zwei Mal zu zeichnen kann man das Programm folgendermaßen abändern:

```
public void zeichne() {
    // Hier stehen die Turtle-Anweisungen:
    t.turnTo(Turtle.EAST);
    rechteck();
    t.up();
    t.forward(100);
    t.down();
    rechteck();
}
public void rechteck() {
    t.forward(50);
    t.left(90);
    t.forward(100);
    t.left(90);
    t.forward(50);
    t.left(90);
    t.forward(100);
    t.left(90);
}
```

In diesem Beispiel wurde eine eigene Methode `rechteck()` erstellt. `rechteck()` enthält die nötigen Turtle-Anweisungen, um ein Rechteck der Breite 50 und der Höhe 80 zu zeichnen. Damit die Turtle diese Rechtecke auch wirklich zeichnet, muss die Methode `rechteck()` innerhalb der Methode `zeichne()` aufgerufen werden, in unserem Beispiel sogar zwei Mal.

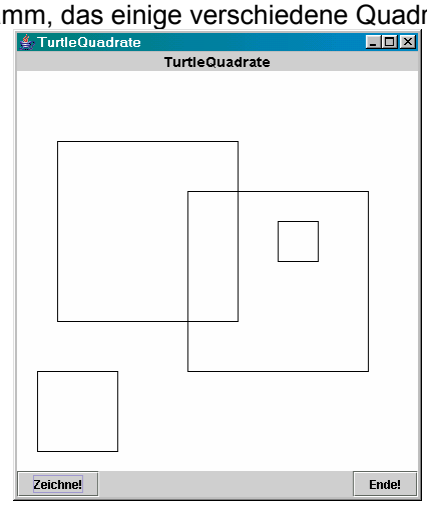
`rechteck()` kann leider immer nur 50×80-Rechtecke zeichnen. Will man Rechtecke mit anderen Kantenlängen zeichnen, muss man dieses der Methode `rechteck` mitteilen können. Dazu erstellen wir eine andere Methode `rechteck(double breite, double hoehe)`, die zwei Eingabeparameter `breite` und `hoehe` hat. Unser Programm zeichnet jetzt ein 50×100- und ein 80×120-Rechteck.

```
public void zeichne() {
    // Hier stehen die Turtle-Anweisungen:
    t.turnTo(Turtle.EAST);
    rechteck(50, 100);
    t.up(); t.forward(100); t.down();
    rechteck(80, 120 );
}

public void rechteck(double breite, double hoehe) {
    t.forward(breite);
    t.left(90);
    t.forward(hoehe);
    t.left(90);
    t.forward(breite);
    t.left(90);
    t.forward(hoehe);
    t.left(90);
}
```

Quadrate könnte man dann so programmieren:

```
public void quadrat(double breite) {
    t.forward(breite);
    t.left(90);
    t.forward(breite);
    t.left(90);
    t.forward(breite);
    t.left(90);
    t.forward(breite);
    t.left(90);
}
```

Aufgabe 7: Erstelle ein Programm, das einige verschiedene Quadrate zeichnet.



2.2.4 Wiederholungen programmieren

In unserer Methode `quadrat` werden dieselben Anweisungen mehrfach wiederholt. Das ist mühsam einzugeben, außerdem ist dieser Programmierstil fehleranfällig. Deshalb geht es kürzer:

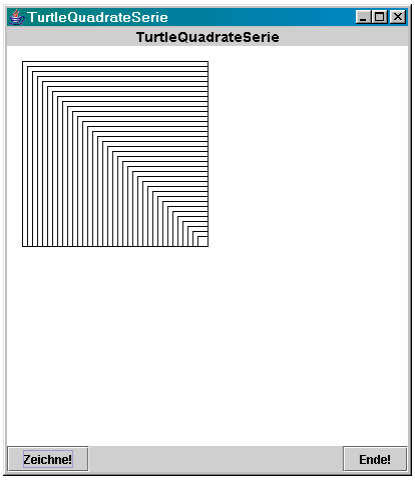
```
public void quadrat(double breite) {
    for (int i = 0; i < 4; i++) {
        t.forward(breite);
        t.left(90);
    }
}
```

Dabei bedeutet die zweite Zeile in der Methode `quadrat` folgendes:
Eine (ganzzahlige) Zählervariable `i` wird auf den Startwert 0 (`i = 0`) gesetzt. Wenn die Variable `i` noch nicht den Wert 4 (`i < 4`) erreicht hat, wird ihr Wert um eins erhöht (`i++`). Anschließend werden die Anweisungen zwischen den geschweiften Klammern ausgeführt. Das Ganze geschieht insgesamt vier Mal, wobei `i` nacheinander die Werte 0, 1, 2, und 3 annimmt.

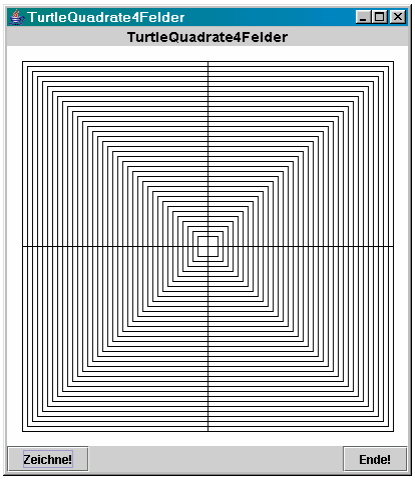
Man kann die Wiederholungsanweisungen sogar schachteln. Dabei kommen recht lustige Bilder heraus.

```
public void zeichne() {
    // hier stehen die Turtle-Anweisungen
    for(int j = 0; j < 36; j++) {
       quadrat(10 + j*5);
    }
}
```

Hier werden 36 ineinander geschachtelte Quadrate gezeichnet, das kleinste hat die Kantenlänge 10, jedes weitere vergrößert seine Kantenlänge um 5.



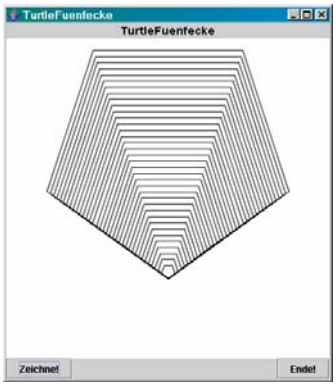
Aufgabe 8: Die eben gezeichnete Figur sieht nicht so schön aus, sie ist linkslastig.
Erstelle eine neue Datei `TurtleQuadrate4Felder.java` und binde sie in dein Projekt ein.
Ändere die Methode `zeichne` so ab, dass in allen 4 Feldern geschachtelte Quadrate gezeichnet werden.



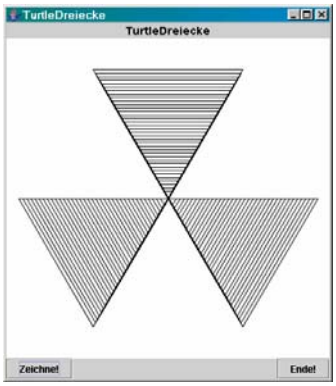
Hinweis:
Offensichtlich wird viermal das Gleiche gezeichnet. Es bietet sich an, zwei for-Schleifen zu schachteln:

```
public void zeichne() {
    // hier stehen die Turtle-Anweisungen
    for(int i = 0; i < 4; i++) {
        for(int j = 0; j < 36; j++) {
           quadrat(10 + j*5);
        }
        t.left(90);
    }
}
```

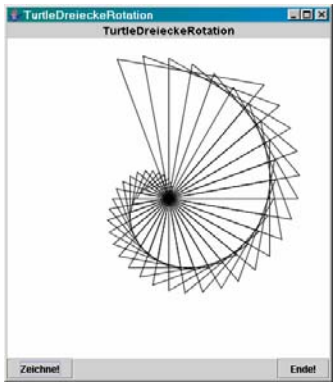
Aufgabe 9: Zeichne auf die selbe Weise andere Figuren, z. B.:



Ineinander geschachtelte Fünfecke



Geschachtelte Dreiecke dreimal wiederholt



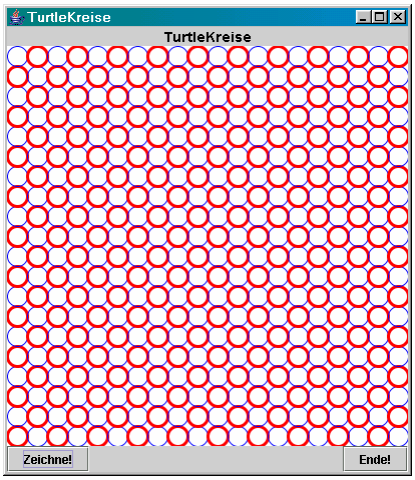
Verdrehte Dreiecke, die immer größer werden

Erfinde weitere Figuren, speichere neue Versionen jeweils unter einem anderen Namen ab.

2.2.5 Verzweigungen programmieren

Unsere bisherigen Programme waren sequenziell, d. h. ein Programmschritt folgte auf den nächsten. Manchmal ist es notwendig, den geradlinigen Programmverlauf zu verlassen. Abhängig von bestimmten Bedingungen, wie z. B. Rechenergebnissen, möchte man ein Programm so oder anders fortsetzen. Derartige Programmstrukturen nennt man Verzweigungen.

Unsere Turtle soll die gesamte Zeichenfläche mit kleinen Kreisen füllen. Dabei sollen die Kreise immer abwechselnd rot (im Schwarz/Weiß-Druck fett) und blau gezeichnet werden, wobei sich keine gleichfarbigen berühren sollen.



In Java schreibt man eine „Wenn-Dann“-Anweisung so:

```
if(Bedingung) {
    Anweisungen, falls Bedingung erfüllt ist.
}
else {
    Anweisungen, falls die Bedingung nicht erfüllt ist.
}
```

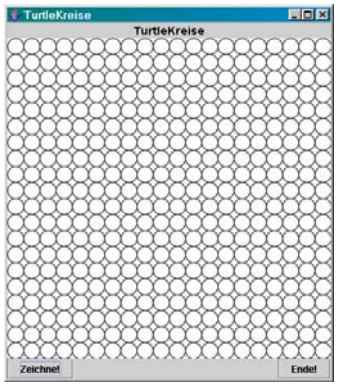
Tipp: Wenn die Summe aus Spaltennummer (s) und Zeilennummer (z) eine gerade Zahl ist, zeichne einen roten Kreis, andernfalls einen blauen.
In Java muss man dazu die Summe durch 2 teilen und schauen, ob als Rest 0 oder 1 herauskommt.:
Den Rest bei einer Division von Ganzzahlen erhält man mit dem Operator „%“.
Man vergleicht zwei Zahlen mit dem doppelten Gleichheitszeichen „==“.

Alles zusammen:

```
if((z + s) % 2 == 0) {
    t.setColor(java.awt.Color.RED);
}
else {
        t.setColor(java.awt.Color.BLUE);
}
```

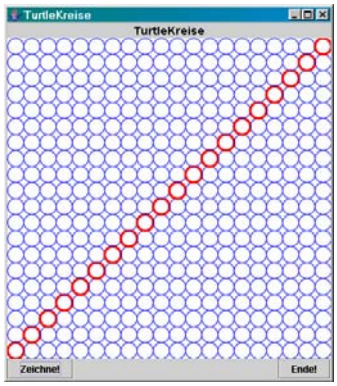
(Vorgefertigte Farben stellt Java im Paket `java.awt.Color` bereit.)

Aufgabe 10: Schließe alle Projekte. Erstelle dann ein neues Turtle-Projekt `TurtleKreise`.
Vorübung: Zeichne zunächst einen Kreis mit Radius 10, der genau in die Ecke links unten passt.
Zeichne dann viele Kreise, die die gesamte Zeichenfläche überdecken.
Hinweis:
Sie Zeichenfläche erstreckt sich normalerweise von –200 bis 200 in x-Richtung und von –200 bis 200 in y-Richtung.

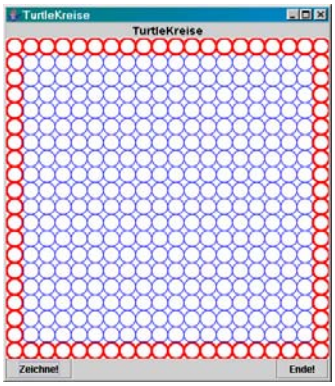


Probiere andere Kreisgrößen aus.

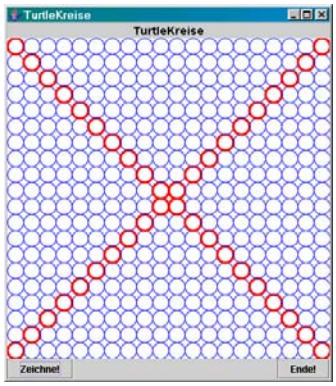
Aufgabe 11: Färbe die Kreise unterschiedlich ein:



Färbe die erste Diagonale rot; in Java:
```
if(z == s)
```



Färbe nur die Ränder rot; in Java:
```
if((z == 0) ||
   (z == 19) ||
   (s == 0) ||
   (s == 19))
```



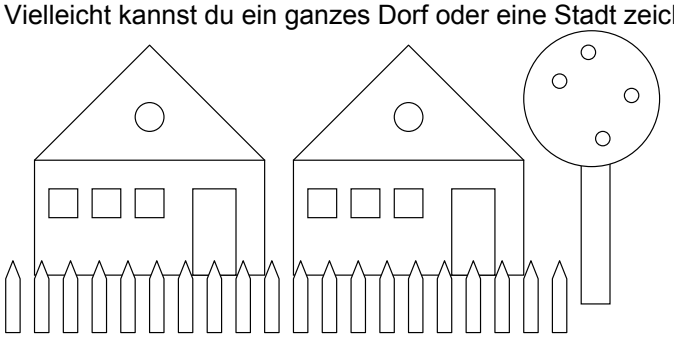
Färbe beide Diagonalen rot; in Java:
```
if((z == s) ||
  ((z+s+1)%20 == 0))
```

Erfinde eigene Aufgaben.

Was ist, wenn sich der Kreisradius ändert?

Hinweis: Man kann mehrere Bedingungen durch „und“ (&&) oder „oder“ (||)miteinander verbinden:
`Bed. 1` „und“ `Bed 2` laute in Java : `(Bed1) && (Bed2)`
`Bed. 1` „oder“ `Bed 2` laute in Java : `(Bed1) || (Bed2)`

Aufgabe 12: Das nachfolgende Bild kann sehr elegant programmiert werden. Man benötigt Methoden zum Zeichnen von Rechtecken, Quadraten, Dreiecken, ein Haus. Der Zaun besteht aus mehreren Zaunlatten …
Vielleicht kannst du ein ganzes Dorf oder eine Stadt zeichnen, oder …



Hinweis:
Es gibt noch weitere Turtle-Anweisungen, mit denen man die aktuelle Position und Orientierung der Turtle herausfinden kann:

- `double x = t.getX();` bzw.
  `double y = t.getY();`

speichern die aktuelle x- bzw. y-Koordinate der Turtle in einer Variablen `x` bzw. `y`,

- `double phi = t.getOrientation();`

speichert die aktuelle Orientierung der Turtle in einer Variablen `phi`.
Dabei bedeuten insbesondere:

0° (= `Turtle.EAST`):	Turtle zeigt nach rechts,
90° (= `Turtle.NORTH`):	Turtle zeigt nach oben,
180° (= `Turtle.WEST`):	Turtle zeigt nach links,
270° (= `Turtle.SOUTH`):	Turtle zeigt nach unten.

Natürlich sind auch alle anderen Werte zwischen 0° und 360° möglich.

3 Programmieren des RCX-Bausteins von *LEGO*

3.1 Einfache Testprogramme

3.1.1 Einfache Fahr-Tests

Am Roboter sind zwei Motoren an den Ausgängen A und B angeschlossen, die unabhängig voneinander jeweils ein Rad antreiben. Beim Starten mit Drücken auf RUN fährt der Roboter solange geradeaus (sollte er zumindest) bis RUN wieder gedrückt wird.
Sollte der Roboter nicht vorwärts fahren, sonder rückwärts oder sich drehen, sind die Anschlüsse an den Ausgängen A und C zu verdrehen.

Listing von `TravelTest`:

```
1          import josx.platform.rcx.*;
2
3          public class TravelTest {
4              public static void main (String[] args) {
5                  Motor.A.setPower(7);
6                  Motor.C.setPower(7);
7                  Motor.A.forward();
8                  Motor.C.forward();
9                  try {
10                     Button.RUN.waitForPressAndRelease();
11                 }catch(InterruptedException e) {}
12             }
13         }
```

Dieses Programm ist sehr einfach und vor allem noch kaum objektorientiert programmiert.
In den Zeilen 5 und 6 werden beide Motoren auf maximale Leistung geschaltet. Die Leistungsstufen sind wählbar in einem Bereich von 0 bis 7.
In den Zeilen 7 und 8 werden die Motoren eingeschaltet.
In der Zeile 10 wird der RUN-Schalter veranlasst zu warten bis er gedrückt und wieder losgelassen wurde. Solange verharrt der Roboter in seinem aktuellen Zustand (beide Motoren drehen vorwärts). Nach dem Loslassen von RUN wird das Programm beendet, der Roboter bleibt stehen. Die Methode `waitForPressAndRelease` könnte unter Umständen einen Fehler, man sagt Ausnahme; auslösen. Falls eine Ausnahme auftritt, muss man dem Programm sagen, wie es dann weiter zu machen hat. Das erledigt der Kontroll-Block `try`. Alles was nach `try` in dem geschweiften Klammerpaar steht, wird ausgeführt. Falls eine Ausnahme aufgetreten ist, wird alles, was im geschweiften Klammerpaar hinter `catch` steht, ausgeführt. Das Argument von `catch` enthält die Art der Ausnahme. Die einzig mögliche Ausnahme, die bei `waitForPressAndRelease` auftreten könnte ist, dass der `Thread`, der diese Methode kontrolliert, schon irgendwie unterbrochen ist; die Ausnahme heißt deshalb `InterruptedException`.

Aufgabe 13: Verändere die Werte in `setPower`.
Experimentiere auch mit dem Methoden `backward`, `stop`, `flt`, `reverseDirection` aus der Klasse `Motor`.
Wende sie einmal nur auf einen, dann auf beide Motoren an.
Der Roboter soll im Kreis fahren, links herum, rechts herum; und soll auf der Stelle drehen können
Hinweis durch die Anweisung `Thread.sleep(1000)` kann der weiter Programmablauf für 1000 Millisekunden = 1 Sekunde angehalten werden. Diese Anweisung muss eine `InterruptedException` behandeln.

3.1.2 Einfache Sensor-Steuerung mit Kontaktsensoren

Unser Roboter wird jetzt zusätzlich mit einem Kontaktsensor vorne (`Sensor.S1` am Eingang 1) und mit einem Kontaktsensor hinten (`Sensor.S3` am Eingang 3) ausgestattet. Er soll vorwärts fahren bis er an ein Hindernis stößt. Dann soll er rückwärts fahren bis er wieder an ein Hindernis stößt und dann anhalten.

Listing von BumpTest:

```
1         import josx.platform.rcx.*;
2
3         public class BumpTest implements SensorListener, SensorConstants {
4             private static Sensor sVorne = Sensor.S1;
5             private static Sensor sHinten = Sensor.S3;
6             public BumpTest() {
7                 sHinten.setTypeAndMode(SENSOR_TYPE_TOUCH, SENSOR_MODE_BOOL);
8                 sVorne.setTypeAndMode(SENSOR_TYPE_TOUCH, SENSOR_MODE_BOOL);
9                 sVorne.addSensorListener(this);
10                sHinten.addSensorListener(this);
11                sVorne.activate();
12                sHinten.activate();
13                Motor.A.setPower(7);
14                Motor.C.setPower(7);
15                Motor.A.forward();
16                Motor.C.forward();
17            }
18            public void stateChanged(Sensor bumper, int alt, int neu) {
19                if (bumper == sVorne) {
20                    Motor.A.backward();
21                    Motor.C.backward();
22                }
23                if (bumper == sHinten) {
24                    Motor.A.stop();
25                    Motor.C.stop();
26                }
27            }
28            public static void main (String[] args) {
29                new BumpTest();
30                try {
31                    Button.RUN.waitForPressAndRelease();
32                }catch(InterruptedException e) {}
33            }
34        }
```

Dieses Programm enthält die wesentlichen Elemente eines objektorientierten Programms. Seine Klasse hat einen eigenen Kostruktor und es implentiert einen Aktionsabhorcher (`SensorListener`). Mit Hilfe seiner beiden Klassenvariablen[7] `sVorne` (`Sensor.S1`) und `sHinten` (`Sensor.S2`) kann auf die Eingänge 1 und 3 zugegriffen werden. Dazu meldet sich `BumpTest` an den beiden Sensoren als Empfänger an (Zeilen 9 und 10) und wartet auf die Botschaft „Berührungssensor gedrückt“. Dass an den Eingängen tatsächlich Berührungssensoren angeschlossen sind, wird in den Zeilen 7 und 8 fest-

[7] Da die beiden Sensoren statisch sind, sind `Sensor.S1` und `Sensor.S3` Klassenvariablen, d. h. alle Instanzen von `BumpTest` greifen auf dieselben Variablen zu, während bei Objektvariablen mit jeder neu erzeugten Instanz auch eine neue Objektvariable erzeugt wird. Klassenvariablen sind sehr vorsichtig zu verwenden. Ändert eine Instanz den Wert einer Klassenvariablen, macht sich diese Änderung auch in allen anderen Instanzen bemerkbar. Da der RCX-Baustein genau drei Eingänge und drei Ausgänge hat, greifen alle Instanzen einer Klasse immer auf dieselben Ein- und Ausgänge zu. Somit müssen die zugehörigen Objekte statisch sein.

gelegt.
In Zeile 29 wird eine Instanz von `BumpTest` erzeugt, damit wird sein Konstruktor aufgerufen und in ihm werden die Motoren gestartet.

3.1.3 Einfache Sensor-Steuerung mit Lichtsensoren

Wir modifizieren unseren Roboter ein wenig. Statt der Kontaktsensoren erhält er Lichtsensoren. Sie zeigen auf den Boden und haben vom Boden wenige Millimeter Abstand. Der Roboter soll solange vorwärts fahren, bis er eine weiße Linie sieht, die sich vom dunklen Untergrund deutlich abhebt. Dann soll er solange rückwärts fahren, bis sein hinterer Sensor ebenfalls eine weiße Linie sieht und dann anhalten.
Der Quelltext von `WeisseLinienTest` lautet:

```
1         import josx.platform.rcx.*;
2
3         public class WeisseLinienTest implements SensorListener,
4                                                  SensorConstants {
5             private static Sensor sVorne = Sensor.S1;
6             private static Sensor sHinten = Sensor.S3;
7             private int schwellenWert;
8             public WeisseLinienTest(int schwellenWert) {
9                 this.schwellenWert = schwellenWert;
10                sVorne.setTypeAndMode(SENSOR_TYPE_LIGHT, SENSOR_MODE_RAW);
11                sHinten.setTypeAndMode(SENSOR_TYPE_LIGHT, SENSOR_MODE_RAW);
12                sVorne.addSensorListener(this);
13                sHinten.addSensorListener(this);
14                sVorne.activate();
15                sVorne.setPreviousValue(sVorne.readValue());
16                Motor.A.setPower(7);
17                Motor.C.setPower(7);
18                Motor.A.forward();
19                Motor.C.forward();
20            }
21            public void stateChanged(Sensor sLight, int alt, int neu) {
22                if (sLight == sVorne) {
23                    if (alt - neu > schwellenWert) {
24                        Motor.A.backward();
25                        Motor.C.backward();
26                        sVorne.passivate();
27                        sHinten.activate();
28                        sHinten.setPreviousValue(sHinten.readValue());
29                    }
30                }
31                if (sLight == sHinten) {
32                    if (alt - neu > schwellenWert) {
33                        Motor.A.stop();
34                        Motor.C.stop();
35                        sHinten.passivate();
36                    }
37                }
38            }
39            public static void main (String[] args) {
40                new WeisseLinieTest(10);
41                try {
42                    Button.RUN.waitForPressAndRelease();
43                }catch(InterruptedException e) {}
44            }
45        }
```

In den Zeilen 10 und 11 wird jeweils der Sensortyp festgelegt. Wir wählen Lichtsensoren, die wir im `raw`-Modus auslesen, wir erhalten somit Werte zwischen 0 und 1023. Die Methode `stateChanged` ist etwas aufwendiger als bei `BumpTest`. Da der Lichtsensor praktisch andauernd Lichtänderungen wahrnimmt, soll er nur reagieren, wenn die Änderung zwischen altem und neuem Wert eine bestimmte Schwelle überschreitet. Um ein wenig flexibler zu sein, übergeben wir diesen Schwellwert dem Konstruktor (Zeile 40), den wir als Objektvariable speichern. Wenn der vordere Sensor (Zeile 22) auf diesen Schwellenwert angesprochen hat (Zeile 23), lassen wir die Motoren rückwärts laufen, schalten den vorderen Sensor ab und den hinteren ein. Beim Einschalten der Sensoren ist darauf zu achten, dass als alter Wert der gerade momentan gemessene Wert abgespeichert wird (Zeilen 15 und 28). Andernfalls könnte schon beim Einschalten ein Änderungseffekt auftreten. Das wollen wir vermeiden. Jetzt fährt der Roboter solange rückwärts, bis sein hinterer Sensor (Zeile 31) eine weiße Linie sieht (Zeile 32). Dann hält er an.

Aufgabe 14: Ändere das Programm so ab, dass er jeweils bei einer schwarzen Linie (oder Tischkante) stoppt.

Aufgabe 15: Ändere das Programm so ab, dass er vorne bei einer weißen Linie, hinten bei einer schwarzen Linie stoppt.

3.1.4 Das *leJOS*-Interface *Behavior* und die *leJOS*-Klasse *Arbitrator*

Das *Behavior* API im Paket `josx.robotics`, besteht aus dem Interface `Behavior` (Verhalten) und der Klasse `Arbitrator` (Schiedsrichter).

`Behavior` enthält die folgenden drei Methoden:

- `public boolean takeControl()`
  falls ein bestimmtes Verhalten aktiv werden sollte, muss diese Methode den Wert `true` zurückliefern.
- `public void action()`
  In dieser Methode wird ein bestimmtes Verhalten programmiert, falls die `Behavior`-Klasse aktiv wird, d. h. wenn `takeControl()` den Wert `true` ergibt.
- `public void supress()`
  Beim Aufruf dieser Methode werden die Anweisungen in action() sofort abgebrochen.

`Arbitrator` enthält folgenden Konstruktor:

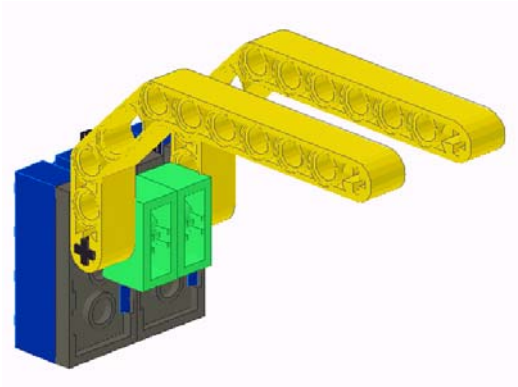
- `public Arbitrator(Behavior[] behaviors)`
  Der Konstruktor erzeugt ein „Schiedsrichter“-Objekt, dem als Parameter ein Feld von „Verhalten“-Objekten übergeben wird, wobei die Reihenfolge von Bedeutung ist. Das Objekt mit der höchsten Priorität steht an erster Stelle.

… und enthält folgende Methode:

- `public void start()`
  sie startet das “schiedsrichter”-Objekt.

3.1.5 `Pathfinder`, ein Beispiel zur Anwendung des *Behavior*-API

Am einfachsten wird diese Klasse anhand eines Beispiels erklärt. Das gibt uns die Gelegenheit auch einige erweiterte Funktionen in *JCreator* kennen zu lernen.
Unser Roboter hat zwei Motoren, angeschlossen an den Ausgängen A und C. Weiterhin hat er zwei Lichtsensoren, die an den Eingängen 1 und 3 angeschlossen sind. Er soll dabei einer schwarzen Linie folgen, deren Breite größer ist als der Abstand der Lichtsensoren. Die Lichtsensoren sind, wie die Abbildung zeigt, vorne angebracht.



Solange beide Sensoren „schwarz“ sehen ist alles in Ordnung. Der Roboter soll dann mit voller Leistung einfach geradeaus weiterfahren. Sieht der linke Sensor am Eingang 1 „weiß“, soll der Roboter ein nach rechts gegensteuern, entsprechend nach links, wenn der rechte Sensor am Eingang 3 „weiß“ sieht. Sehen beide Sensoren „weiß“ ist der Roboter von seinem Weg abgekommen. Dann soll er ihn suchen, indem er solange nach links dreht, bis er die Linie wieder gefunden hat. Der Roboter wird durch Drücken der Stopp-Taste angehalten.
Um diese verschiedenen Verhaltensweisen zu realisieren, erzeugen wir jeweils eine Klasse, die das Interface `Behavior` implementiert. Für das Geradeausfahren eine Klasse `GoAhead`. Falls der Roboter nach rechts vom weg abgekommen ist, muss er nach rechts drehen. Dazu erzeugen wir eine Klasse `TurnLeft`. Für den Fall, dass er nach links vom Weg abgekommen ist, erzeugen wir eine entsprechende Klasse `TurnRight`. Sollte er die Linie ganz verloren haben, versuchen wir diese wieder zu finden, indem wir den Roboter einfach um seine eigene Achse drehen lassen, bis er (hoffentlich) die Linie wieder gefunden hat. Dieses Verhalten realisieren wir mit Hilfe der Klasse `TurnAround`.
Jede so genannte „Verhaltens“-Klasse muss drei Methoden implementieren.

- `public boolean takeControl()` liefert den Wert `true` zurück, falls ein Zustand eingetreten ist, für den die jeweilige „Verhaltens“-Klasse die Kontrolle übernehmen will, andernfalls den Wert `false`.
- `public void action()` ist eine Methode, in der das Verhalten des Roboters festgelegt ist, für den Fall, dass der „Verhaltens“-Klasse die Ausführung übertragen wird.
- `public void suppress()` wird aufgerufen, wenn der „Verhaltens“-Klasse die weitere Ausführung der Roboter-Steuerung entzogen wird. Hier wird für die „Verhaltens“-Klasse, die als nächste an der Reihe ist, gewissermaßen aufgeräumt.

Damit die verschieden „Verhaltens“-Klassen richtig zusammenspielen, gibt es eine „Schiedsrichter“-Klasse `Arbitrator`. Dem Konstruktor von `Arbitrator` wird ein Feld (`Array`) dieser „Verhaltens“-Klassen übergeben, wobei die Priorität jeder „Verhaltens“-Klasse mit wachsendem Index steigt. Der letzte Eintrag hat demnach die höchste Priorität. Nachdem der `Arbitrator` mit der Methode `start` gestartet wurde, schaut er immer reihum, welche der `takeControl`-Methoden der „Verhaltens“-Klassen gerade `true` zurückliefet. Dann ruft er in dieser Klasse die Methode `action` auf. Liefert nun die `takeControl`-Methode einer anderen „Verhaltens“-Klasse `true` zurück, ruft `Arbitrator` die `suppress`-Methode der „Verhaltens“-Klasse auf, die bisher die Kontrolle hatte und anschließend die `action`-Methode der neuen „Verhaltens“-Klasse. Falls zwei Methoden um die Zuteilung der Kontrolle konkurrieren, bekommt diejenige „Verhaltens“-Klasse den Zuschlag, die die höhere Priorität hat.

Wir starten mit *JCreator* ein neues *leJOS*-Projekt `Pathfinder`. Dazu erzeugen wir ein *Workspace* `Pathfinder`. Dabei wird die Datei `Pathfinder.java` erzeugt, die wir aber erst später bearbeiten wollen. Wir erzeugen uns einige Hilfsklassen. Dazu öffnen wir in *JCreator* das Menü PROJECT → NEW CLASS und nehmen im Fenster CLASSWIZZARD die entsprechenden Eintragungen vor. Zunächst erzeugen wir die Datei `GoAhead.java`, die auch gleich im richtigen Unterordner `D:\Java\Pathfinder\behaviorctrl` abgelegt ist.

Nach einigen Anpassungen sieht der Inhalt von `GoAhead.java` so aus:

```
1   package behaviorctrl;
2   public class GoAhead implements Behavior {
3       public boolean takeControl() {
4           return true;
5       }
6       public void action() {
7           BehaviorInit.mLeft.setPower(7);
8           BehaviorInit.mRight.setPower(7);
9           BehaviorInit.mLeft.forward();
10          BehaviorInit.mRight.forward();
11      }
12      public void suppress() {
13          BehaviorInit.mLeft.setPower(1);
14          BehaviorInit.mRight.setPower(1);
15      }
16  }
```

Auf ähnliche Art und Weise erzeugen wir die Klasse `TurnLeft` …

```
1   package behaviorctrl;
2   import josx.platform.rcx.*;
3   import josx.robotics.*;
4   public class TurnLeft implements Behavior {
5       public boolean takeControl() {
6           return (BehaviorInit.sRight.readValue() >=
7                           BehaviorInit.getWhite()) &&
8                  (BehaviorInit.sLeft.readValue() <
9                           BehaviorInit.getBlack());
10      }
11      public void action() {
12          BehaviorInit.mLeft.backward();
13          BehaviorInit.mRight.forward();
14      }
15      public void suppress() {
16          BehaviorInit.mLeft.forward();
17          BehaviorInit.mRight.forward();
18      }
19  }
```

… und die Klasse `TurnRight`.

```
1   package behaviorctrl;
2   import josx.platform.rcx.*;
3   import josx.robotics.*;
```

```
4         public class TurnRight implements Behavior {
5             public boolean takeControl() {
6                 return (BehaviorInit.sLeft.readValue() >=
7                               BehaviorInit.getWhite()) &&
8                        (BehaviorInit.sRight.readValue() <
9                               BehaviorInit.getBlack());
10            }
11            public void action() {
12                BehaviorInit.mLeft.forward();
13                BehaviorInit.mRight.backward();
14            }
15            public void suppress() {
16                BehaviorInit.mLeft.forward();
17                BehaviorInit.mRight.forward();
18            }
19        }
```

Weiterhin soll der Roboter zum Suchen der schwarzen Linie sich um die eigene Achse drehen können. Das leistet die Klasse `TurnAround`.

```
1         package behaviorctrl;
2         import josx.platform.rcx.*;
3         import josx.robotics.*;
4         public class TurnAround implements Behavior {
5             public boolean takeControl() {
6                 return (BehaviorInit.sLeft.readValue() >=
7                               BehaviorInit.getWhite()) &&
8                        (BehaviorInit.sRight.readValue() >=
9                               BehaviorInit.getWhite());
10            }
11            public void action() {
12                BehaviorInit.mLeft.backward();
13                BehaviorInit.mRight.forward();
14            }
15            public void suppress() {
16                BehaviorInit.mLeft.forward();
17                BehaviorInit.mRight.forward();
18            }
19        }
```

Jetzt machen wir noch etwas Kosmetik. Alle Klassen in dem Paket `behaviorctrl` müssen auf dieselben Einstellungen für Sensoren und Motoren zurückgreifen können, insbesondere auf die Schwellenwerte für schwarz und weiß. Dazu erzeugen wir eine eigene Klasse `BehaviorInit`. Hier gibt es im Menü PROJECT → NEW CLASS wieder einiges zu tun. In der Registerkarte CLASS PROPERTIES wählen wir dieses Mal das Kontrollfeld GENERATE DEFAULT CONTRUCTOR aus. Besonders die Registerkarte VARIABLES kann man gut gebrauchen.

Nach dem Bestätigen mit OK und weiteren Anpassungen am Quelltext ist auch diese Klasse fertig.

```
1         package behaviorctrl;
2         import josx.platform.rcx.*;
3         import josx.robotics.*;
4         public class BehaviorInit implements SensorConstants {
5             protected static final Sensor sLeft = Sensor.S1;
6             protected static final Sensor sRight = Sensor.S3;
7             protected static final Motor mLeft = Motor.A;
8             protected static final Motor mRight = Motor.C;
```

```
1            private static int whiteVal = 40; // Prozent
2            private static int blackVal = 35; // Prozent
3            public BehaviorInit() {
4                sLeft.setTypeAndMode(SENSOR_TYPE_LIGHT, SENSOR_MODE_PCT);
5                sRight.setTypeAndMode(SENSOR_TYPE_LIGHT, SENSOR_MODE_PCT);
6                sLeft.activate();
7                sRight.activate();
8            }
9            public static void setWhite(int val) {
10               if (val >= 0 && val <= 100)
11                   whiteVal = val;
12           }
13           public static void setBlack(int val) {
14               if (val >= 0 && val <= 100)
15                   blackVal = val;
16           }
17           public static int getWhite() {
18               return whiteVal;
19           }
20           public static int getBlack() {
21               return blackVal;
22           }
23       }
```

Die Variablen `whiteVal` und `blackVal` speichern den Lichtwert in Prozent, den die Rotationssensoren bei weißem bzw. schwarzem Untergrund wahrnehmen. Nach gutem OOP-Stil wurden diese Variablen gekapselt. Nur über die öffentlichen Methoden `setWhite` und `getWhite` bzw. `setBlack` und `getBlack` kann auf sie zugegriffen werden. Dabei wird zugesichert, dass diese Variablen keine unsinnigen Werte annehmen können.

Jetzt muss nur noch die Klasse `Pathfinder` implementiert werden.

```
1        import josx.platform.rcx.*;
2        import josx.robotics.*;
3        import behaviorctrl.*;
4        public class Pathfinder {
5            public Pathfinder() {
6                new BehaviorInit();
7                BehaviorInit.setWhite(38);
8                BehaviorInit.setBlack(33);
9                Behavior goahead = new GoAhead();
10               Behavior turnleft = new TurnLeft();
11               Behavior turnright = new TurnRight();
12               Behavior turnaround = new TurnAround();
13               Behavior[] bArray =
14                  { goahead, turnleft, turnright, turnaround };
15               Arbitrator arby = new Arbitrator(bArray);
16               arby.start();
17           }
18           public static void main (String[] args) {
19               new Pathfinder();
20           }
21       }
```

In den Zeilen 1 und 2 werden die *leJOS*-Pakete, in der Zeile 3 das eigene Paket `behaviorctrl` eingebunden. Der Konstruktor (Zeile 5) erzeugt eine Instanz von `BehaviorInit` und legt in den Zeilen 7 und 8 die Werte für schwarz und weiß fest. Dann werden von jeder „Verhaltens“-Klasse je eine Instanz erzeugt. Diese Instanzen speichert man in einem Feld (engl.: `array`) `bArray` (Zeilen 13 und

14). In Zeile 15 wird eine Instanz von `Arbitrator` erzeugt, wobei seinem Konstruktor `bArray` als Parameter übergeben wird. Dieser „Schiedsrichter“ entscheidet, in welcher der „Verhaltens“-Klassen die jeweilige Methode `action` oder `supress` ausgeführt wird. Damit die Aktionen auch wirklich ausgeführt werden, muss man den „Schiedsrichter“ noch starten (Zeile 23).
Die `main`-Methode in `Pathfinder` erzeugt schließlich eine Instanz von `Pathfinder` selbst.

Aufgabe 16: Ändere das Programm `Pathfinder` so ab, dass der Roboter einer weißen Linie folgt.

Aufgabe 17: Ändere das Programm `Pathfinder` so ab, dass der Roboter einer sehr dünnen schwarzen Linie folgt.
Hinweis: Hier benötigst du drei Lichtsensoren, wobei der zusätzliche Sensor vor den anderen beiden befestigt wird.

3.1.6 Sensor-Steuerung mit Rotationssensoren

Tippy ist ein Roboter, der über zwei Motoren angetrieben wird. Jede Achse wird durch einen Rotationssensor überwacht, außerdem hat er einen Kontaktsensor, der bei Berührung den Roboter stoppt. Er fährt recht genau entlang der Kante eines Quadrats mit ein Meter Kantenlänge. Je nach Untergrund muss man mit dem Wert der Konstanten `driveLength` etwas anpassen. Diese Konstante enthält die Achslänge, d. h. den Abstand zwischen den Rädern, jeweils von Mitte zu Mitte der Laufflächen gemessen.

Das Listing von `Tippy`:

```
1   import josx.platform.rcx.*;
2   import josx.robotics.*;
3   public class Tippy extends RotationNavigator
4                       implements SensorListener, SensorConstants {
5       private static final float wheeldiameter = 4.96f;
6       private static final float driveLength = 8.0f;
7       private static final float ratio = 24/8;
8       private static Sensor stopSensor;
9       private static Button buttonRun;
10      public Tippy() {
11          super(wheeldiameter, driveLength, ratio);
12          Motor.A.setPower(7);
13          Motor.C.setPower(7);
14          stopSensor = Sensor.S2;
15          stopSensor.setTypeAndMode(
16              SENSOR_TYPE_TOUCH, SENSOR_MODE_BOOL
17          );
18          stopSensor.addSensorListener(this);
19          stopSensor.activate();
20      }
21      public void doTheWork() {
22          while(true) {
23              gotoPoint(100, 0);
24              LCD.showNumber((int)getAngle()); // 0°
25              gotoPoint(100, 100);
26              LCD.showNumber((int)getAngle()); // 90°
27              gotoPoint(0, 100);
28              LCD.showNumber((int)getAngle()); // 180°
29              gotoPoint(0, 0);
30              LCD.showNumber((int)getAngle()); // 270°
31              gotoAngle(0);
32              LCD.showNumber((int)getAngle()); // 0°
33              try {
34                  Button.RUN.waitForPressAndRelease();
35              }
36              catch (InterruptedException e) {}
37          }
38      }
```

```
39              public void stateChanged(Sensor s, int oldValue, int newValue) {
40                  stop();
41                  System. exit(0);
42              };
43              public static void main (String[] args) {
44                  Tippy tippy = new Tippy();
45                  tippy.doTheWork();
46              }
47          }
```

Aufgabe 18: `Tippy` soll jetzt ein gleichseitiges Dreieck durchfahren.

Aufgabe 19: `Tippy` soll jetzt einen Kreis mit vorgegebenem Radius durchfahren.

3.2 Übersicht über weitere *leJOS*-Programme

3.2.1 Ein- und Ausgabe-Testprogramm `IOTester`

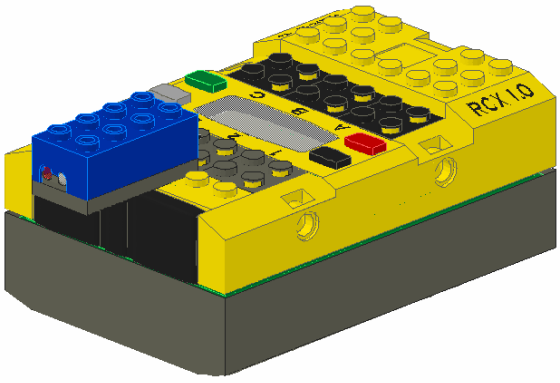
Dieses Programm dient zum testen der Ein- und Ausgänge. Man kann beliebige Sensoren an die Eingänge anschließen und durch Drücken der Tasten Run, View oder Prgm die Sensor-Typen und ihre Auslese-Modi verändern. Die von den Sensoren gemessenen Werte werden direkt auf dem Display des RCX dargestellt. Ebenso kann man Motoren an die Ausgänge anschließen, sie vorwärts oder rückwärts drehen lassen und ihre Energiezufuhr steuern.

Die Einstellungen im Einzelnen:
Beim Start haben sind alle Sensoren vom Typ: `RAW` im Modus: `RAW`
Die Motoren sind gestoppt, ihre Energie ist 0.

- `RUN`: schaltet den Sensor bzw. Motor ein/aus.
- `VIEW`: schaltet den Kanal um eins weiter (1 - 3, A - C).
- `PRGM`: schaltet den Sensor-Typ, bzw. die Motor-Energie um eins weiter
  Sensor-Typen: `Raw = 0`, `Touch = 1`, `Temp = 2`, `Light = 3`, `Rot = 4`
  Motor-Energie: (`0 - 7`)
- `Run + View`: schaltet den Auslesemodus um eins weiter
  (Auslese-Modi: `Raw = 0`, `Canonical = 1`, `Boolean = 2`),
  bzw. schaltet den Motor zwischen `stop` (= 0) und `float` (= 1) um.
- `View + Prgm`: schaltet den Sensor-Modus um eins höher
  (Sensor-Modi: `Raw = 0`, `Boolean = 32`, `Edge = 64`, `Pulse = 96`, `Prozent = 128`,
  Senor-Modi: `Grad-Celius = 160`, `Grad-Farenheit = 192` und `Winkel = 224` werden nicht angezeigt),
  bzw. schaltet den Motor zwischen vorwärts und rückwärts um.
- `Run + Prgm`: zeigt die Batteriespannung in Millivolt an.
- `Run + View + Prgm`: stoppt das Programm.

3.2.2 „Echo-Navigation“ mit `ProximityTest`

`ProximityTest` nutzt aus, dass der RCX Infrarot-Licht aussenden kann. Dieses wird von einem Hindernis, dass sich in der Nähe befindet stärker reflektiert als von einem Hindernis, das sich weiter weg befindet. Ein Lichtsensor, der nach vorne zeigt empfängt das reflektierte Licht. Die Stärke des Lichtsignals ist ein Maß für die Entfernung des RCX vom Hindernis. Dieses Prinzip benutzen z. B. die Fledermäuse, wenn sie sich im Flug orientieren.



Weitere Informationen entnimmt man dem Quelltext und der in Anhang aufgeführten Literatur.

Hier das Listung zum `ProximityTest`:

```
1   import josx.platform.rcx.*;
2   import proximity.*;
3   public class ProximityTest implements ProximityListener {
4       public ProximityTest() {
5           Motor.A.forward();
6           Motor.C.forward();
7       }
8       public synchronized void objectDetected(int ping) {
9           Motor.A.backward();
10          Motor.C.backward();
11          try {
12              Thread.sleep(500);
13          } catch(InterruptedException e) {}
14          Motor.A.forward();
15          try {
16              Thread.sleep(700);
17          } catch(InterruptedException e) {}
18          Motor.C.forward();
19      }
20      public static void main(String[] args) {
21          ProximityDetector pd = new ProximityDetector(Sensor.S1);
22          pd.addProximityListener(new ProximityTest());
23          try {
24              synchronized(pd) {
25                  Button.RUN.waitForPressAndRelease();
26                  System.exit(0);
27              }
28          } catch(InterruptedException e) {}
29      }
30  }
```

Weiterhin benötigt Proximity-Test die Klasse ProximityDetector:

```
1   package proximity;
2   import josx.platform.rcx.*;
3   import java.util.*;
4   public class ProximityDetector extends Thread implements
5   SensorConstants {
6       private final byte[] packet = { 127 }; // Ein Bit-Muster
7       private Sensor lightSensor;
8       private Vector proximityListeners;
9       public ProximityDetector(Sensor lightPort) {
10          proximityListeners = new Vector(2, 2);
11          lightSensor = lightPort;
12          lightSensor.setTypeAndMode(
13              SENSOR_TYPE_LIGHT, SENSOR_MODE_RAW);
14          lightSensor.activate();
15          Serial.setRangeLong();
16          this.start
17      }
```

```
18              public void run() {
19                  int oldValue;
20                  int newValue;
21                  while(true) {
22                      oldValue = lightSensor.readValue();
23                      Serial.sendPacket(packet, 0, 1);
24                      try {
25                          Thread.sleep(5);
26                      } catch(InterruptedException e) {}
27                      // Der Lichtsensor empfängt das Bit-Muster
28                      newValue = lightSensor.readValue();
29                      int diff = Math.abs(oldValue - newValue);
30                      if(diff > 80) {
31                          notifyListeners(diff);
32                      }
33                      try {
34                          Thread.sleep(160);
35                      } catch(InterruptedException e) {}
36                      Thread.yield();
37                  }
38              }
39              public void addProximityListener(ProximityListener listener) {
40                  proximityListeners.addElement(listener);
41              }
42              private void notifyListeners(int ping) {
43                  for(int i = 0; i < proximityListeners.size(); i++) {
44                      ProximityListener prox =
45                          (ProximityListener) proximityListeners.elementAt(i);
46                      prox.objectDetected(ping);
47                  }
48              }
49          }
```

Das Interface ProximityListener:

```
1           package proximity;
2           public interface ProximityListener {
3               public void objectDetected(int ping);
4           }
```

4 Programmieren der *Fischer-Technik*-Roboter

4.1 Beispielprogramm für das *Fischer-Technik*-Interface mit *JavaFish*

Um einen Roboter aus Fischertechnik steuern zu können, öffnet man zuerst ein neues Projekt (FILE → NEW... → PROJECTS → FISCHER TECHNIK JAVAFISH APPLICATION). Der Projektname könnte beispielsweise `FtRobot` sein. Wenn CREATE NEW WORKSPACE markiert ist, bekommt man nach einem Klick auf OK auch gleich eine neue Arbeitsumgebung.
In der `import`-Anweisung (Zeile 1) sind schon alle Funktionen geladen, die durch das Paket `JavaFish` für die Fischertechnik Programmierung bereitgestellt werden. Links oben gibt es ein kleines Fenster, in dem alle bereits geladenen Dateien angezeigt werden. Bis jetzt sieht man dort nur WORKSPACE, darunter unseren Projektnamen `FtRobot` und dort die Datei `FtRobot.java`.

Mit einem Doppelklick auf `FtRobot.java` öffnet man diese Datei im Editor und nimmt folgende Änderungen und Ergänzungen vor:

```
1   import ftcomputing.*;
2   public class FtRobot {
3       private JavaFish iinterface;
4       public FtRobot() {
5           iinterface = new JavaFish();
6           iinterface.openInterface("COM4");
7           iinterface.setMotor(1, JavaFish.LEFT);
8           iinterface.pause(3000);
9           iinterface.setMotor(1, JavaFish.OFF);
10          iinterface.closeInterface();
11      }
12      public static void main (String[] args) {
13          new FtRobot();
14      }
15  }
```

Für dieses einfache Beispiel muss ein Motor an den Ausgang `M1` des Fischertechnik-Interfaces angeschlossen sein. Durch Klicken auf die Schaltfläche COMPILE FILE wird das Programm compiliert und durch Klicken auf EXCECUTE FILE wird es gestartet.
Dabei schaltet sich der Motor ein und nach 3 Sekunden wieder aus.

Da unser Programm nur eine Instanz von `JavaFish` benötigt, können wir unsere Klasse `FtRobot` auch von `JavaFish` ableiten. Dann vereinfacht sich der Quelltext ein wenig, die Funktionsweise ist exakt dieselbe.
Die Änderung zeigt das Listing von `FtRobot2`:

```
1   import ftcomputing.*;
2   public class FtRobot2 extends JavaFish {
3       public FtRobot2() {
4           super();
5           openInterface("COM4");
6           setMotor(1, JavaFish.LEFT);
7           pause(3000);
8           setMotor(1, JavaFish.OFF);
9           closeInterface();
10      }
11      public static void main (String[] args) {
12          new FtRobot2();
13      }
14  }
```

Anhang

Der Editor *JavaEdit*

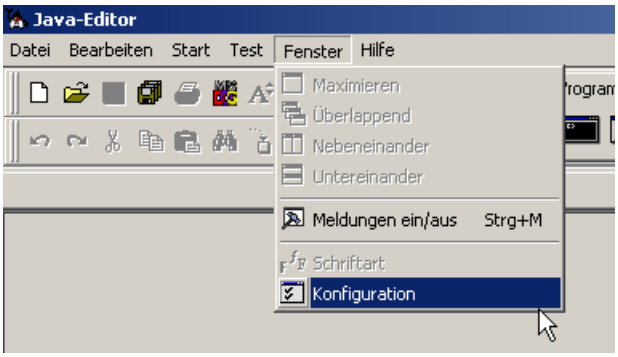
JavaEdit ist ein sehr einfach zu handhabender Editor. Er ist speziell für den Einsatz in Schulen entwickelt worden. In seiner neusten Version 3.11 unterstützt *JavaEdit* auch das Erstellen, Compilieren und Ausführen von *LEGO-Mindstorms*-Programmen mit Hilfe von *leJOS*.

4.1.1.1 *JavaEdit* installieren

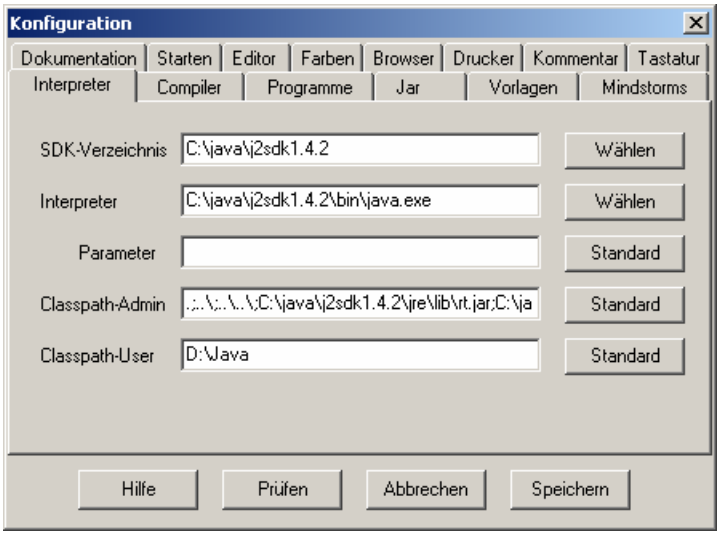
Zur Installation packt man das Archiv `JavaEdit.zip` in einen temporären Ordner aus und startet dann das Programm `setup.exe`. Dabei wählt man als Zielordner am besten `C:\java\JavaEditor`. Die Konfigurationseinstellungen können wahlweise in der Windows-Registrierungsdatei oder in `ini`-Dateien gespeichert werden; wobei die *Windows*-Registrierungsdatei meist eine gute Wahl ist. Wenn man möchte kann man die Vorlagendateien aus der Datei `vorlagen.zip` in den Ordner `C:\JavaEditor\Templates` auspacken.

4.1.1.2 *JavaEdit* konfigurieren

Zunächst startet man *JavaEdit* und wählt dann den Menüpunkt FENSTER → KONFIGURATION.



Es öffnet sich ein Fenster mit einigen Registerkarten. Wir starten mit der Registerkarte INTERPRETER:



Das *JDK*-Verzeichnis, hier SDK-Verzeichnis genannt, findet *JavaEdit* in der Regel selbst. Andernfalls klickt man auf die zugehörige Schaltfläche WÄHLEN und wählt es in dem sich öffnenden Verzeichnisbaum aus. Das Feld INTERPRETER wird automatisch ausgefüllt.
Das Eingabefeld PARAMETER kann man leer lassen. Möglich wäre auch der Eintrag `-d classes`. Er bewirkt, dass der Compiler die compilierten Klassen jeweils in einem Unterordner `classes` abspeichert. Das gilt jedoch nicht für *LEGO-Mindstorm*-Programme, da man `lejosc.exe` mit *JavaEdit* keine weiteren Parameter übergeben kann.
Die Einträge in der Umgebungsvariablen `CLASSPATH` legen fest, wo die Java-Compiler und Interpreter nach den jeweiligen Dateien und Paketen suchen sollen. Aus diesem Grund muss man sicherstellen, dass alle benötigten Pakete in der Variablen `CLASSPATH` eingetragen sind. Dabei werden die einzel-

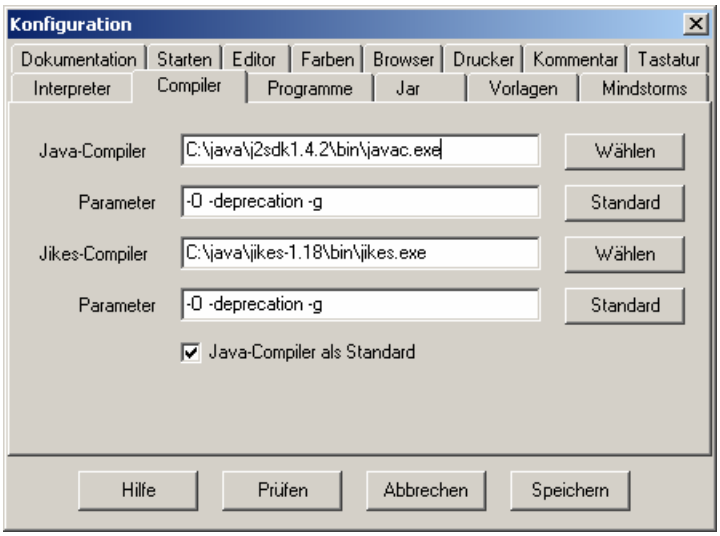
nen Pfade sind durch ein Semikolon voneinander getrennt. *JavaEdit* sieht hier zwei verschiedene Felder vor. Das Eingabefeld CLASSPATH-ADMIN enthält Eintragungen, die für alle Benutzer gleich sind und nicht häufig geändert werden. Zum Änderen dieses Feldes benötigt man Administrator-Rechte. Das Eingabefeld CLASSPATH-USER kann man je nach Bedarf ändern. Die Einträge beider Felder werden aneinander gehängt und der Java-Umgebungsvariablen `CLASSPATH` übergeben.
Das Eingabefeld CLASSPATH-ADMIN füllt man zunächst durch Drücken auf die Schaltfläche STANDARD aus. Dabei wird `.;C:\java\j2sdk1.4.2\jre\lib\rt.jar` eingetragen. Der Punkt „`.`“ am Anfang bedeutet, dass das jeweils aktuelle Verzeichnis in den `CLASSPATH` aufgenommen wird, in `rt.jar` sind alle Klassen der Java-API abgelegt.
Diesen Eintrag ergänzt man wie folgt (es ist eine recht lange Zeile):

```
.;..\;..\..\;C:\java\j2sdk1.4.2\jre\lib\rt.jar; C:\java\user\lib;
C:\java\j2sdk1.4.2\lib\comm.jar;C:\FischerTechnik\ft\comm\ft.comm.jar;
C:\FischerTechnik;C:\FischerTechnik\JavaFish
```

„`..\`“ ist das Verzeichnis, das eine Ebene höher liegt als „`.`“. Das Verzeichnis „`..\..\`“ liegt zwei Ebenen höher als „`.`“, „`..\..\..\`“ liegt drei Ebenen höher usw. Diese Einträge sind hilfreich, wenn man eigene Pakete erstellt, die in einem beliebigen Ordner abgelegt sind und ggf. zwei, drei oder mehr Verzeichnisebenen aufweisen. In `C:\java\user\lib` liegen selbst erstellte Pakete, die man häufig benötigt. Die letzten drei Einträge `C:\java\j2sdk1.4.2\lib\comm.jar` , `C:\FischerTechnik\ft\comm\ft.comm.jar` und `C:\FischerTechnik` sind für die Programmierung der *Fischer-Technik*-Modelle nötig, einmal für das Paket `ft.comm` und dann für das Paket `JavaFish`.
Bei längern Pfad-Eintragungen kann es passieren, dass der Hintergrund des betreffenden Eingabefeldes nach rot wechselt. Wenn man sich sicher ist, dass alle Pfade korrekt eingetragen wurden, sollte man sich davon nicht weiter stören lassen.

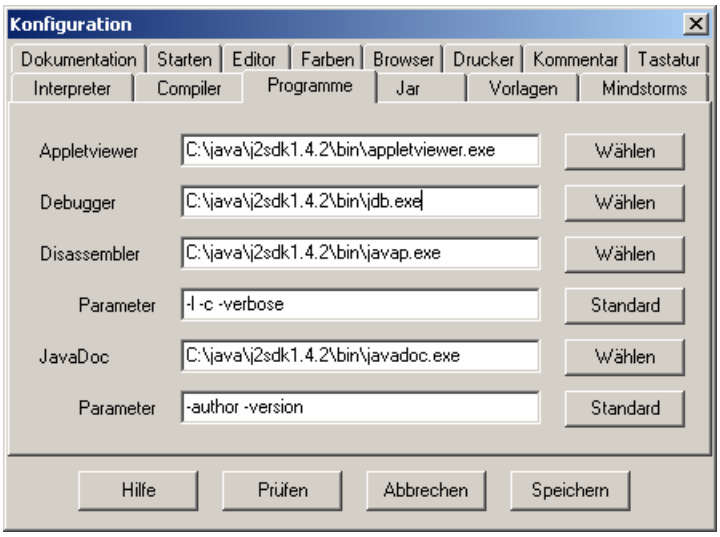
In das Eingabefeld CLASSPATH-USER trägt man z. B. `D:\java` ein. Es ist das Standardverzeichnis, wo man selbst geschriebene Java-Programme ablegt.

Als nächstes muss die Registerkarte COMPILER bearbeitet werden:



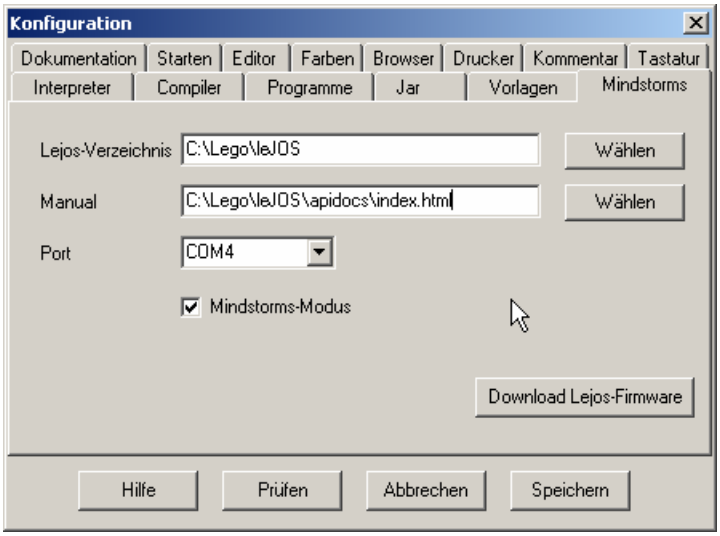
Die meisten Eintragungen werden automatisch vorgenommen. Nur der Pfad zum *Jikes*-Compiler stimmt noch nicht. Man klickt auf WÄHLEN und sucht die Datei `jikes.exe` im sich öffnenden Verzeichnisbaum.
Um die Einstellung zu übernehmen, kann man jetzt auf SPEICHERN klicken. Leider schließt sich dann das Konfigurationsfenster. Dann muss man es für die nächsten Eintragungen erneut öffnen. Man kann aber auch in die anderen Registerkarten wechseln und erst am Ende auf SPEICHERN klicken, nur darf man es dann nicht vergessen.

Unter PROGRAMME ist fast nichts zu tun, einzig im untersten Feld kann man noch einige Parameter ergänzen. `-author` bedeutet, dass der Autorenname und `-version`, dass die Versionseinträge in die JavaDoc-Datei übernommen werden.



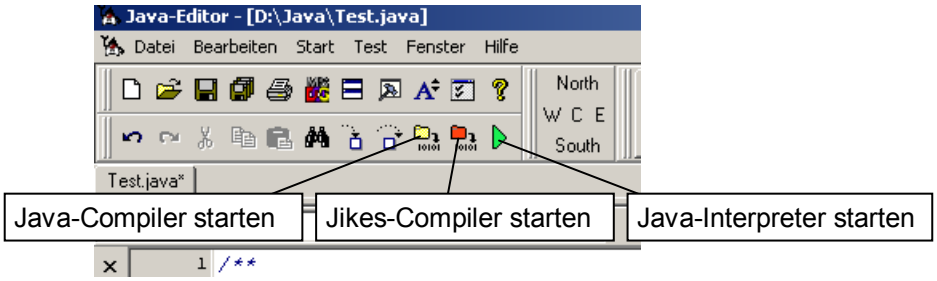
In der Registerkarte JAR ist nichts zu tun.
In der Registerkarte VORLAGEN kann man die unter `C:\java\JavaEdit\Templates` gespeicherten Vorlagen auswählen. Das ist aber nur nötig, wenn man von den Standard-Einstellungen von *JavaEdit* abweichen möchte. Dann muss man die Vorlagendateien nach den eigenen Wünschen anpassen.

In der Registerkarte MINDSTORMS legt man den Pfad zu *leJOS* fest: Dazu klickt man auf WÄHLEN und sucht den *leJOS*-Ordner im sich öffnenden Verzeichnisbaum.

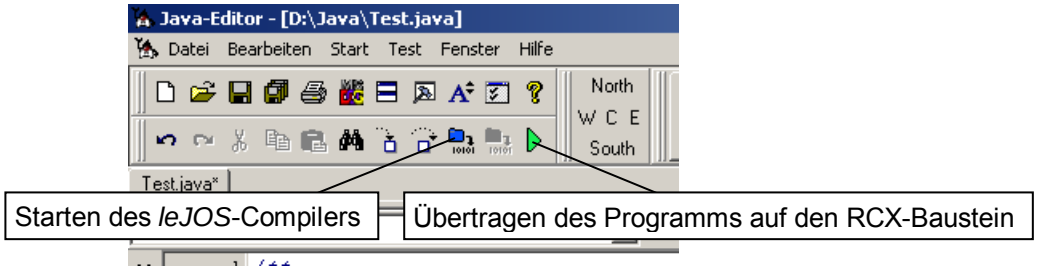


Mit dem Auswahlfeld PORT wählt man diejenige serielle Schnittstelle aus, an der der Infrarot-Sender an den Rechner angeschlossen ist. Standardmäßig erwartet *leJOS* den Eintrag `COM1`.
Die Schaltfläche DOWNLOAD LEJOS-FIRMWARE startet das Programm `lejosfirmdl.exe`, dabei wird die *leJOS* VM auf den RCX-Baustein übertragen. Das muss man nur beim ersten Mal machen, eventuell ist diese Prozedur nach einem Batteriewechsel zu wiederholen.

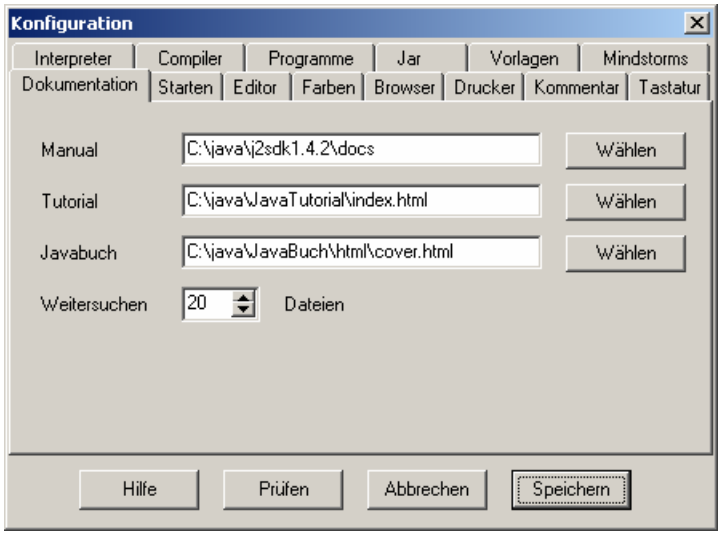
Ist bei MINDSTORM-MODUS kein Häkchen gesetzt, bearbeitet man normale Java-Programme, die entweder mit `javac.exe` oder `jikes.exe` compiliert und anschließend mit `java.exe` ausgeführt werden. Dieser Modus ist für *Fischer-Technik*-Projekte zu wählen.



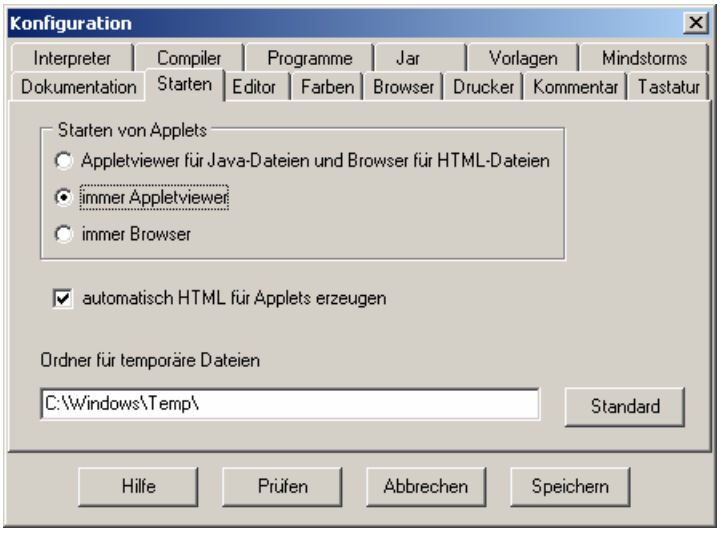
Das Setzten des Häkchens schaltet *JavaEdit* in den Mindstorm-Modus. Das bedeutet, dass als Compiler jetzt `lejosc.exe` und als Interpreter `lejos.exe` verwendet wird. Beim Klicken auf den grünen Pfeil überträgt `lejos.exe` das Programm auf den RCX-Baustein.



Es bleibt noch die andere Reihe der Registerkarten auszufüllen. Unter DOKUMENTATION stellt man ein, wo man die verschiedenen Hilfen finden kann. Das *Java-Manual* ist schon richtig eingetragen. Die Speicherorte von *Java-Tutorial* und *Java-Buch* trägt man entweder direkt von Hand ein, oder man klickt jeweils auf WÄHLEN und sucht die zugehörigen Dateien im Verzeichnisbaum. Dabei muss man darauf achten, dass man auch den richtigen Datei-Typ eingestellt hat.

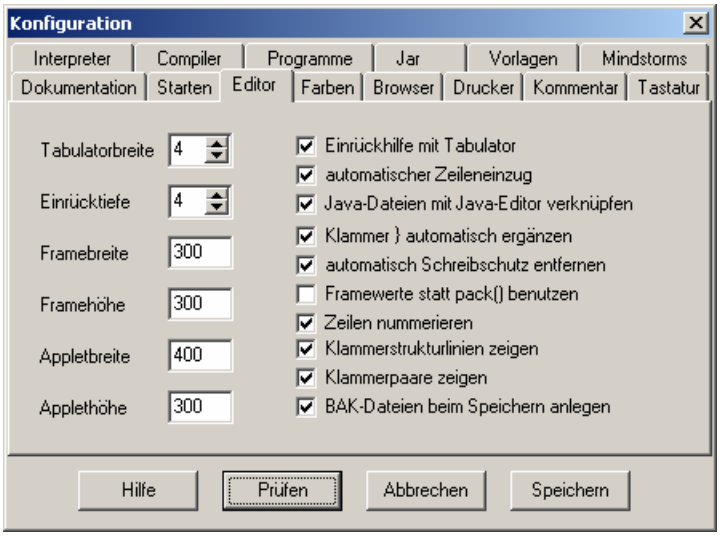


Unter STARTEN wird festgelegt, wie *JavaEdit* beim Ausführen von Applets verfahren soll. Mit dem Auswählen von IMMER APPLET-VIEWER liegt man auf der sicheren Seite.



Im Eingabefeld Ordner für temporäre Dateien kann man festlegen, wo JavaEdit während der Programmausführung temporäre Dateien anlegt. Man kann die Standardeinstellung beibehalten.

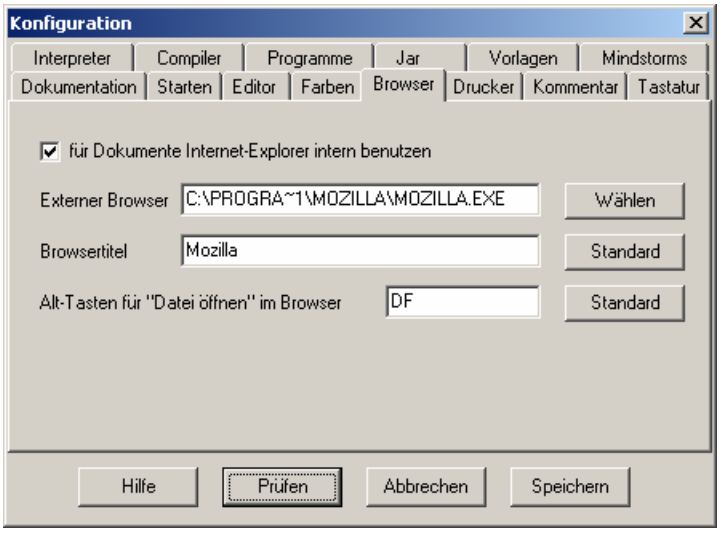
Für die Registerkarte EDITOR haben sich folgende Einstellungen bewährt:



Hier sollte man ruhig ein wenig experimentieren.

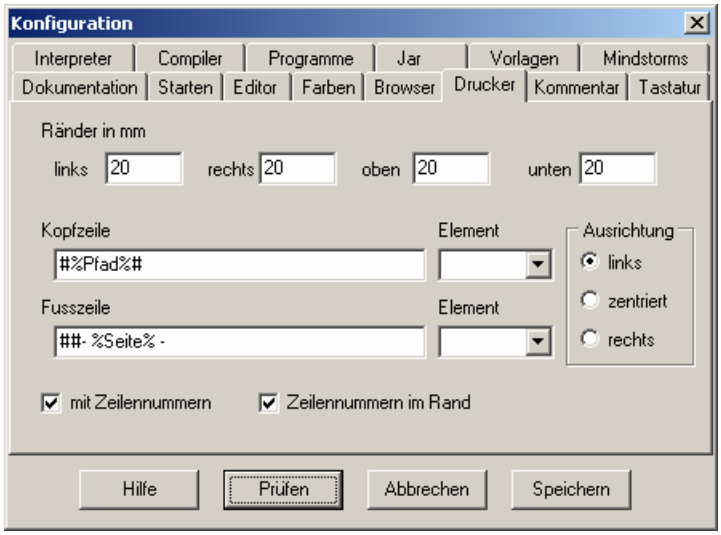
Die Registerkarte FARBEN ist für die Textfarben im Editor verantwortlich. Auch hier ist ausprobieren angesagt, obwohl die Standardeinstellungen schon recht gut sind.

Wir haben *Mozilla* als Browser installiert, das muss man *JavaEdit* noch sagen:

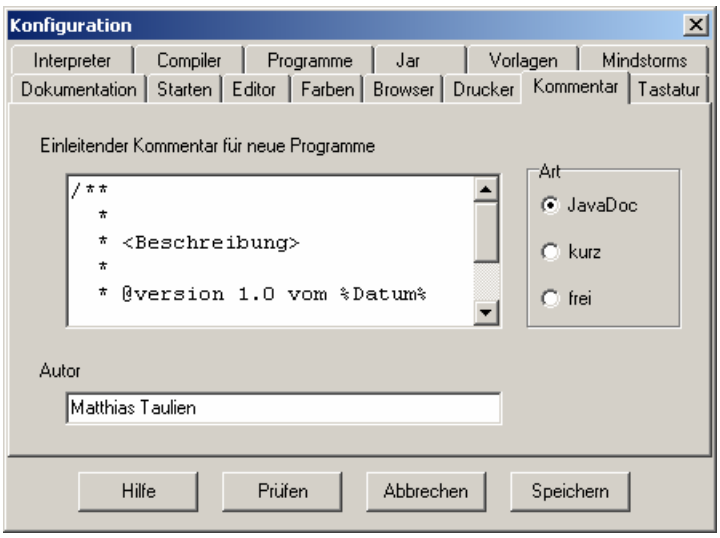


Falls der *Internet-Explorer* Probleme machen sollte, entfernt man das entsprechende Häkchen.

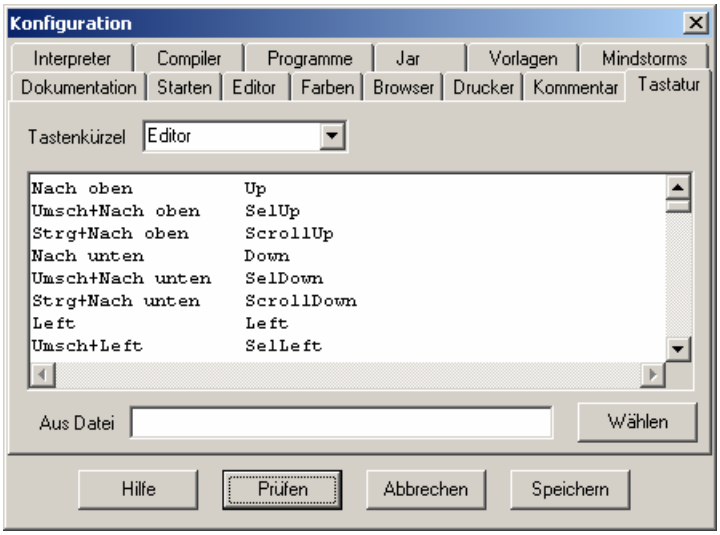
Dann sind noch in der Registerkarte DRUCKER die entsprechenden Einstellungen für das Ausdrucken der Programme vorzunehmen:



Die Registerkarte KOMMENTAR gibt an, auf welche Art Zusatzinformationen am Anfang einer Java-Datei vorgenommen werden. Wann man bei AUTOR seinen Namen einträgt, wird dieser in jedes neu zu erstellende Programm übernommen, sofern dem Programm eine Standardvorlage zugrunde liegt.



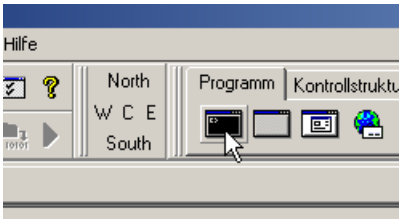
Die Registerkarte TASTATUR regelt die Reaktionen des Editors auf Betätigung von Tastenkürzel. Diese kann man sogar in einer eigenen Datei speichern, die man in das Eingabefeld AUS DATEI eintragen kann.



Wichtig:
Wenn man alle Einstellungen vorgenommen hat, darf man auf keinen Fall vergessen SPEICHERN zu drücken.

4.1.1.3 Arbeiten mit *JavaEdit*

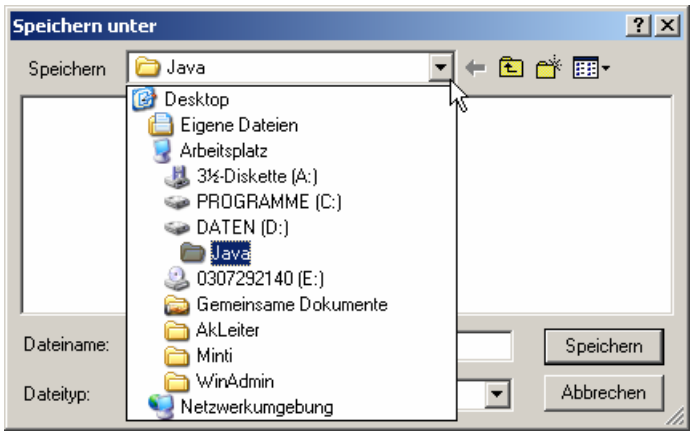
Um neue Java-Applikationen zu erstellen klickt man auf die Schaltfläche PROGRAMM.



Die weiteren Schaltflächen weiter rechts sind für AWT-Applikationen, Swing-Applikationen bzw. Applets gedacht. Diese kommen für die *Mindstorms*-Programmierung weniger in frage.

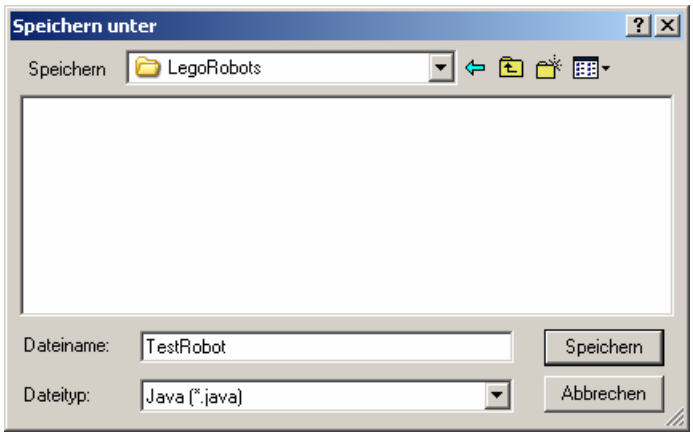
Als Beispiel wollen wir ein Projekt `LegoRobots` mit einer Klasse `TestRobot` erstellen. Alle selbst erstellten Java-Dateien werden in Unterordnern von D:\Java abgelegt. Um die Übersicht bei vielen Pro-

jekten nicht zu verlieren, sollte man für jedes neue Projekt immer einen neuen Ordner in `D:\Java` erstellen, wobei Ordner- und Projekt-Name übereinstimmen können. Dazu klickt man auf die Schaltfläche PROGRAMM. Es öffnet sich ein Fenster. Hier kann man einen Pfad auswählen und ggf. auch neu erstellen.

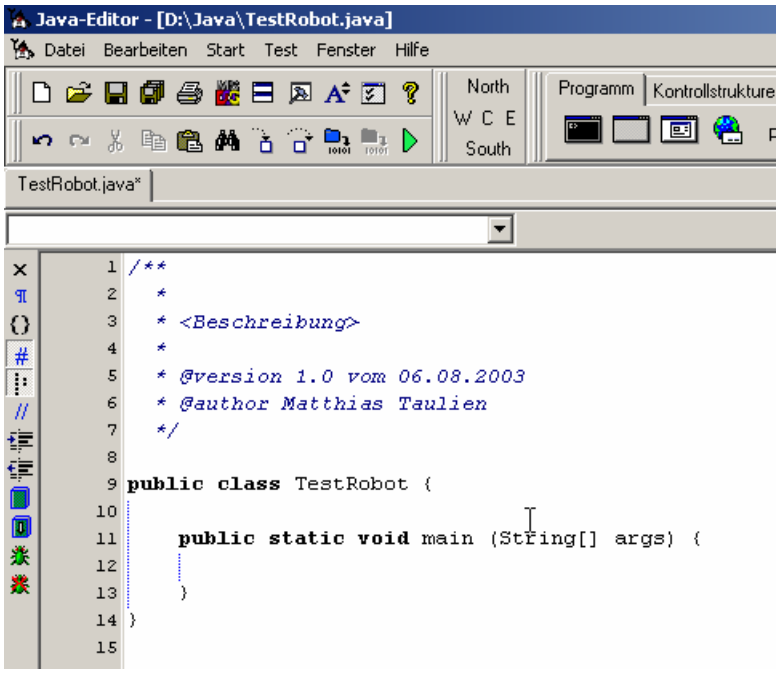


Fall im Auswahlfeld SPEICHERN ein anderer Ordner als `D:\Java` zu sehen ist, klickt man auf die Auswahl-Schaltfläche und wählt den Order `D:\Java`. Anschließend klickt man auf die Schaltfläche NEUEN ORDNER ERSTELLEN und gibt `LegoRobots` ein. Anschließend muss die Eingabetaste für die Namensänderung und ein zweites Mal, um in den neu erstellten Ordner `D:\Java\LegoRobots` zu wechseln, gedrückt werden.
Im Feld DATEINAME gibt man als Klassennamen der zu erstellenden Klasse `TestRobot` ein (die Endung `*.java` wird automatisch ergänzt).

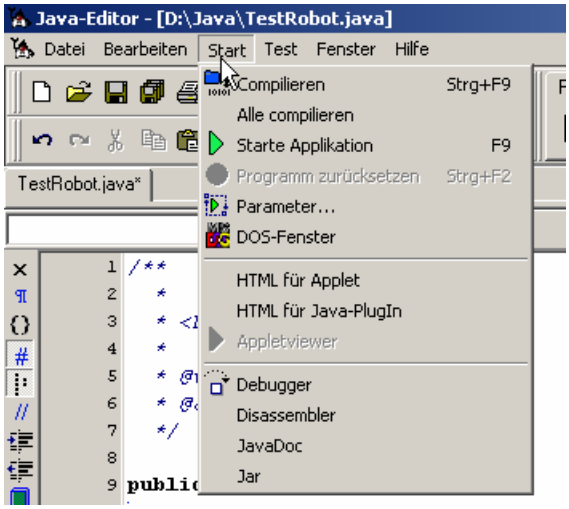
Wichtig:
Die Regeln für die Erstellung von Java-Dateien besagen, dass der Dateiname mit der Endung `*.java` immer mit dem zugehörigen Klassennamen übereinstimmen muss und jede Klasse in einer eigenen Datei gespeichert wird.



Nach dem man SPEICHERN angeklickt hat öffnet sich der Editor, wobei die richtigen Eintragungen schon automatisch erfolgt sind.



Einige wichtige Menüpunkte sind noch erwähnenswert.
Das Menü START enthält einige interessante Menüpunkte:

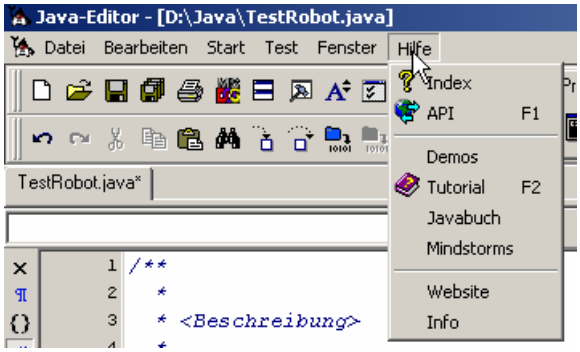


- Die Schaltflächen COMPILIEREN und STARTE APPLIKATION bzw. ÜBERTRAGEN AN DEN RCX wurden schon weiter oben erläutert.
- Mit ALLE COMPILIEREN werden alle geöffneten Dateien compiliert.
- PARAMETER erlaubt die Eingabe von Werten, die man einem Programm beim Starten übergeben kann. Diese werden der Variablen `args` in der Methode `main` übergeben:

```
public static void main(String[] args) {
…
}
```

- HTML FÜR APPLET erzeugt eine HTML-Datei, in die das erstellte Applet eingebunden ist.
- HTML FÜR JAVA-PLUGIN erzeugt eine HTML-Datei, in die das erstellte Swing-Applet eingebunden ist.
- JAVADOC erstellt aufgrund der Dokumentationskommentare in einer Java-Datei die zugehörige Dokumentation. Besteht ein Projekt aus mehreren Dateien, müssen alle geöffnet sein, damit berücksichtigt werden können. Die Hauptdatei sollte dabei im Editor-Fenster zu sehen sein.

Im Menü HILFE findet man die eingebundenen Dokumentationen und Tutorials wieder.



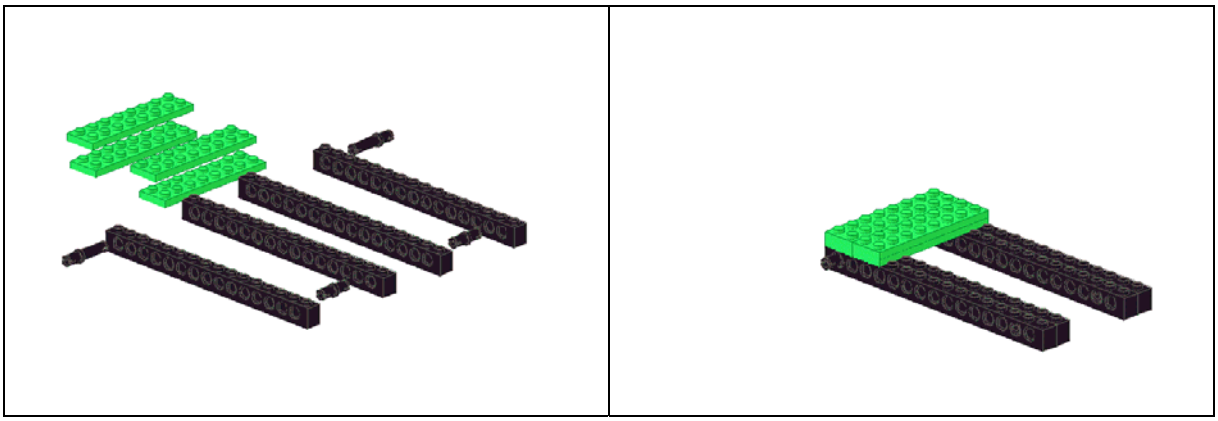
- Unter API findet man die Dokumentation zum Java-API (Application Programming Interface), die man unter `C:\java\j2sdk1.4.2\docs` installiert hat. Sie ist beim Erstellen von Java-Programmen eines der wichtigsten Hilfmittel.
- Die Menüpunkte TURORIAL, JAVABUCH und MINDSTORMS sprechen für sich selbst.

Baupläne für *Lego*-Roboter

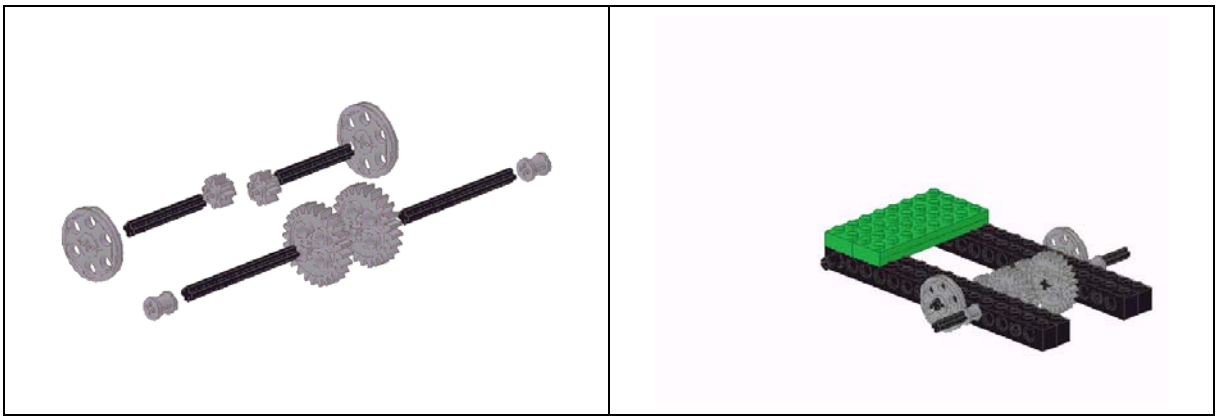
UniversalRobot

Auf der CD findet man diesen Bauplan als *mlCAD*-Datei `Universal_Robot.mpd`.

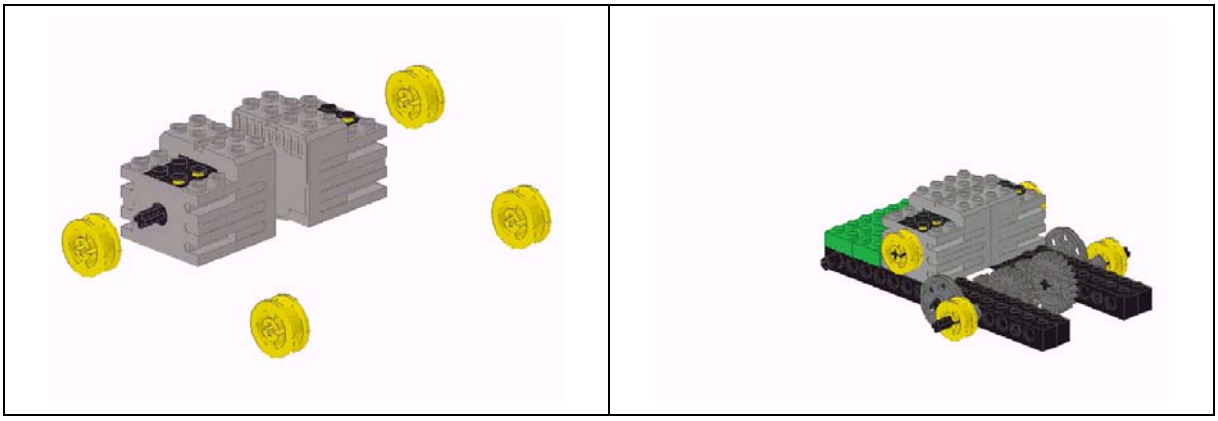
1. Schritt



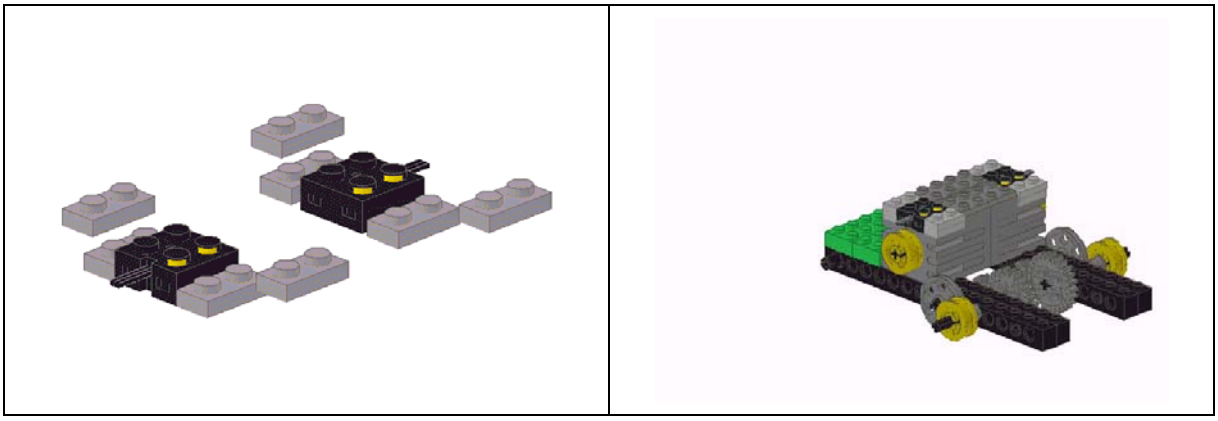
2. Schritt



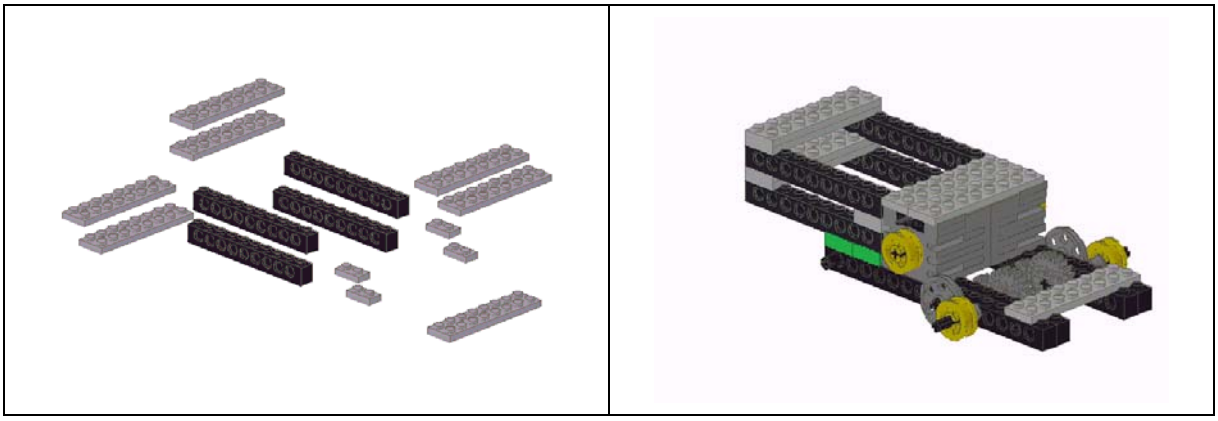
3. Schritt



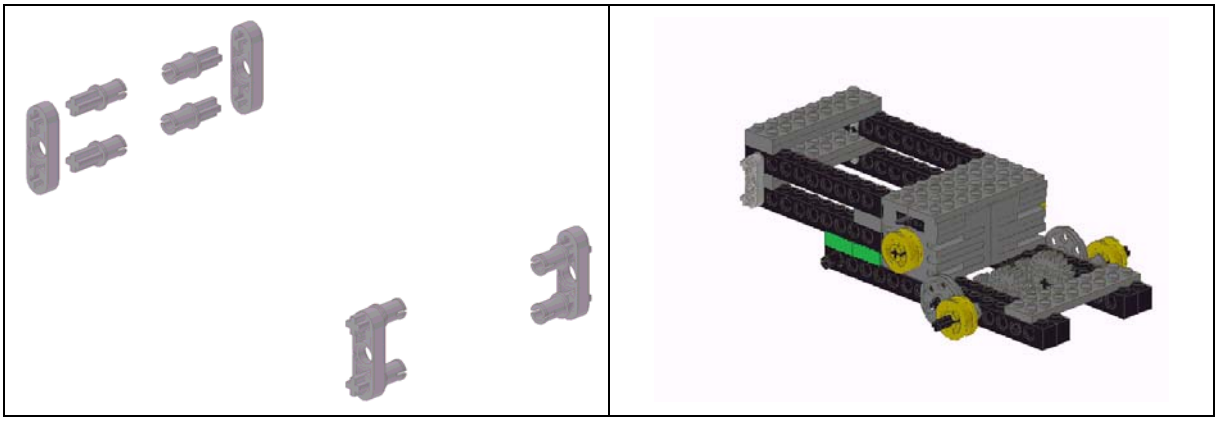
4. Schritt



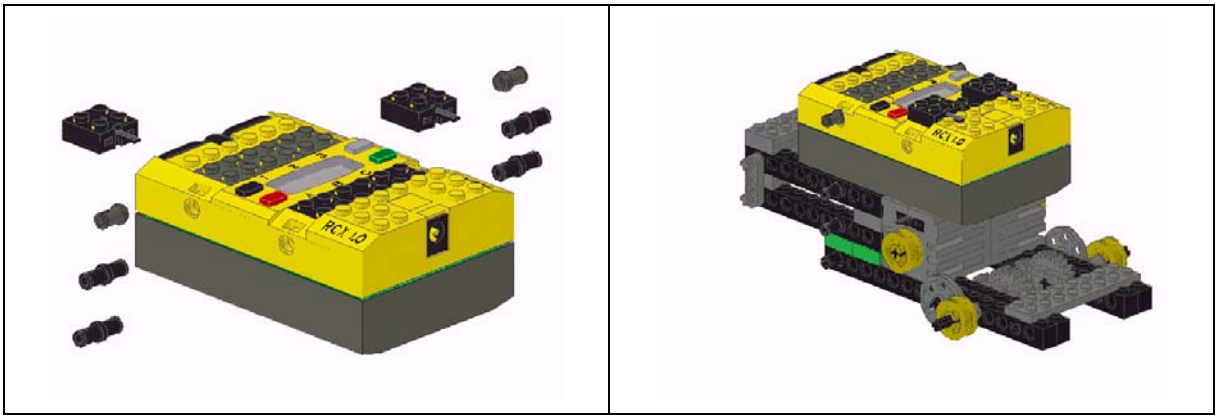
5. Schritt



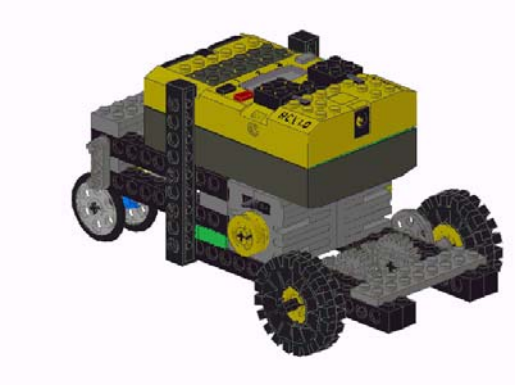
6. Schritt



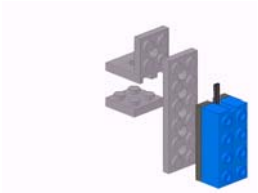
7. Schritt

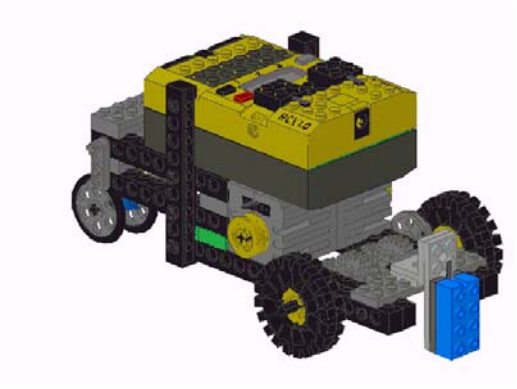


8. Schritt

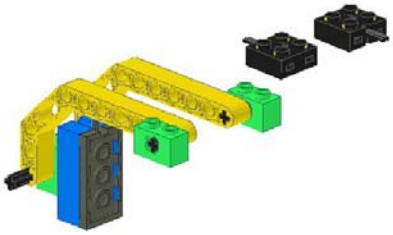


9. Schritt

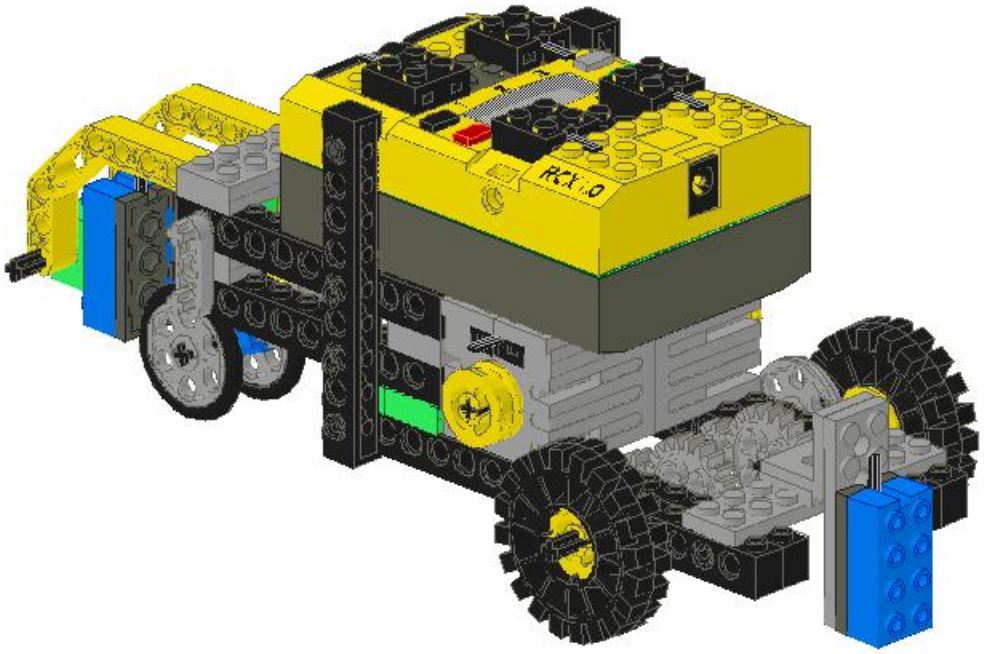




10. Schritt



11. Schritt



Teile-Liste von UniveralRobot

Modell: Universal-Robot (Hector-Seminar)

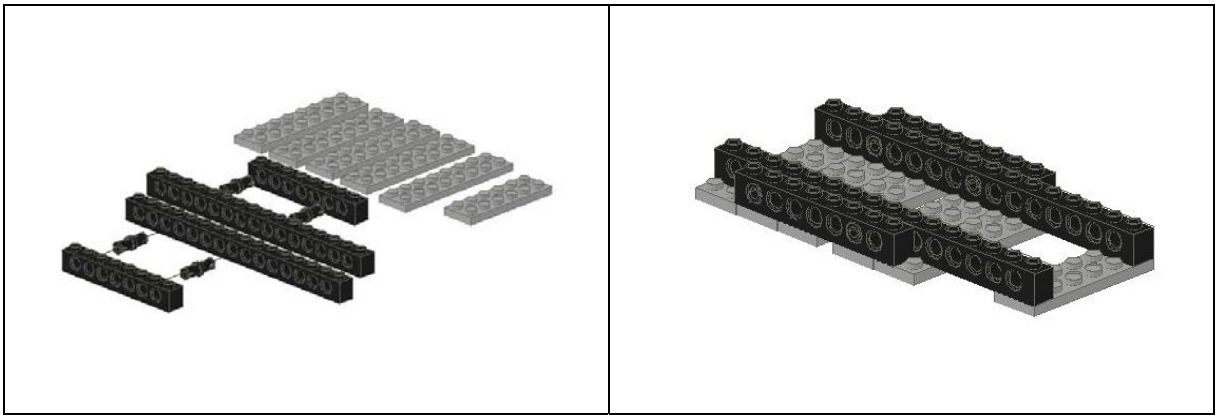
Schritt	Anzahl	Farbe	Bauteil	Beschreibung
1	4	Light-Green	3034.DAT	Plate 2 x 8
1	4	Black	3703.DAT	Technic Brick 1 x 16 with Holes
1	2	Black	6558.DAT	Technic Pin Long with Friction
1	2	Black	4459.DAT	Technic Pin with Friction
2	2	Black	3705.DAT	Technic Axle 4
2	2	Black	3706.DAT	Technic Axle 6
2	2	Light-Gray	3713.DAT	Technic Bush
2	2	Light-Gray	3647.DAT	Technic Gear 8 Tooth
2	2	Light-Gray	3648.DAT	Technic Gear 24 Tooth
2	2	Light-Gray	4185.DAT	Technic Wedge Belt Wheel
3	2	Light-Gray	71427C01.DAT	Electric Technic Mini-Motor 9v
3	4	Yellow	3482.DAT	Wheel Centre Large
4	2	Black	5306.DAT	Electric Brick 2 x 2 x 2/3 with Wire End
4	8	Light-Gray	3023.DAT	Plate 1 x 2
5	4	Light-Gray	3023.DAT	Plate 1 x 2
5	4	Black	2730.DAT	Technic Brick 1 x 10 with Holes
5	7	Light-Gray	3738.DAT	Technic Plate 2 x 8 with Holes
6	4	Light-Gray	3749.DAT	Technic Axle Pin
6	2	Light-Gray	6632.DAT	Technic Liftarm 1 x 3
7	2	Black	5306.DAT	Electric Brick 2 x 2 x 2/3 with Wire End
7	1	Light-Gray	884.DAT	Electric Mindstorms RCX (Complete Assembly Shortcut)
7	2	Dark-Gray	4274.DAT	Technic Pin 1/2
7	4	Black	4459.DAT	Technic Pin with Friction
8	1	Light-Blue	32013.DAT	Technic Angle Connector #1
8	1	Light-Gray	32015.DAT	Technic Angle Connector #5
8	1	Black	32062.DAT	Technic Axle 2 Notched
8	1	Black	4519.DAT	Technic Axle 3
8	1	Black	3706.DAT	Technic Axle 6
8	2	Black	2730.DAT	Technic Brick 1 x 10 with Holes
8	1	Light-Gray	3713.DAT	Technic Bush
8	2	Light-Gray	4185.DAT	Technic Wedge Belt Wheel
8	2	Black	70162.DAT	Technic Wedge Belt Wheel Tyre
8	2	Black	3634.DAT	Tyre 17 x 43
8	1	Yellow	3482.DAT	Wheel Centre Large
9	1	Light-Gray	3956.DAT	Bracket 2 x 2 - 2 x 2
9	1	Light-Blue	2982C01.DAT	Electric Light Sensor (Complete Assembly Shortcut)

9	1	Light-Gray	3022.DAT	Plate 2 x 2
9	1	Light-Gray	32001.DAT	Technic Plate 2 x 6 with Holes
10	2	Black	5306.DAT	Electric Brick 2 x 2 x 2/3 with Wire End
10	1	Light-Blue	2982C01.DAT	Electric Light Sensor (Complete Assembly Shortcut)
10	2	Black	32062.DAT	Technic Axle 2 Notched
10	1	Black	3706.DAT	Technic Axle 6
10	4	Light-Green	3700.DAT	Technic Brick 1 x 2 with Hole
10	2	Yellow	32009.DAT	Technic Liftarm 1 x 11.5 Double Bent

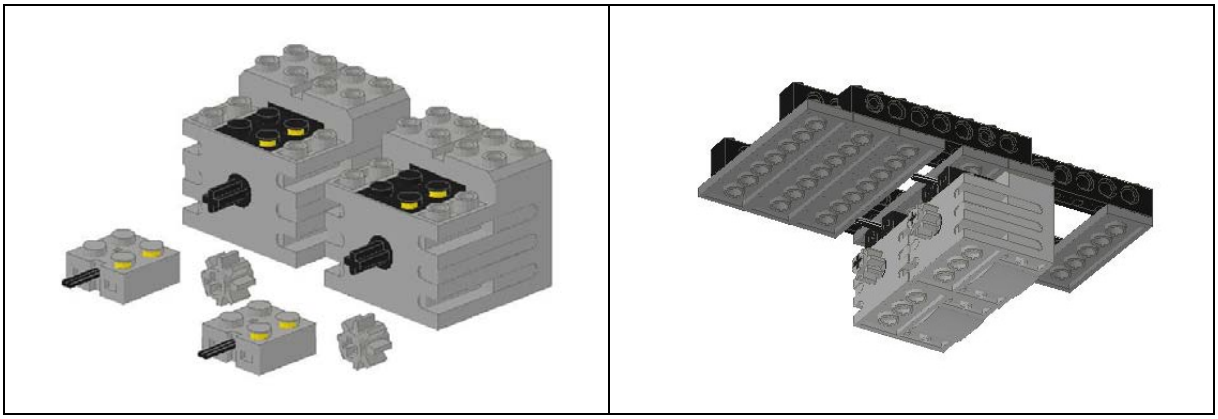
Tippy-Senior

Auf er CD findet man diesen Bauplan als *mlCAD*-Datei `tippy.mpd`.

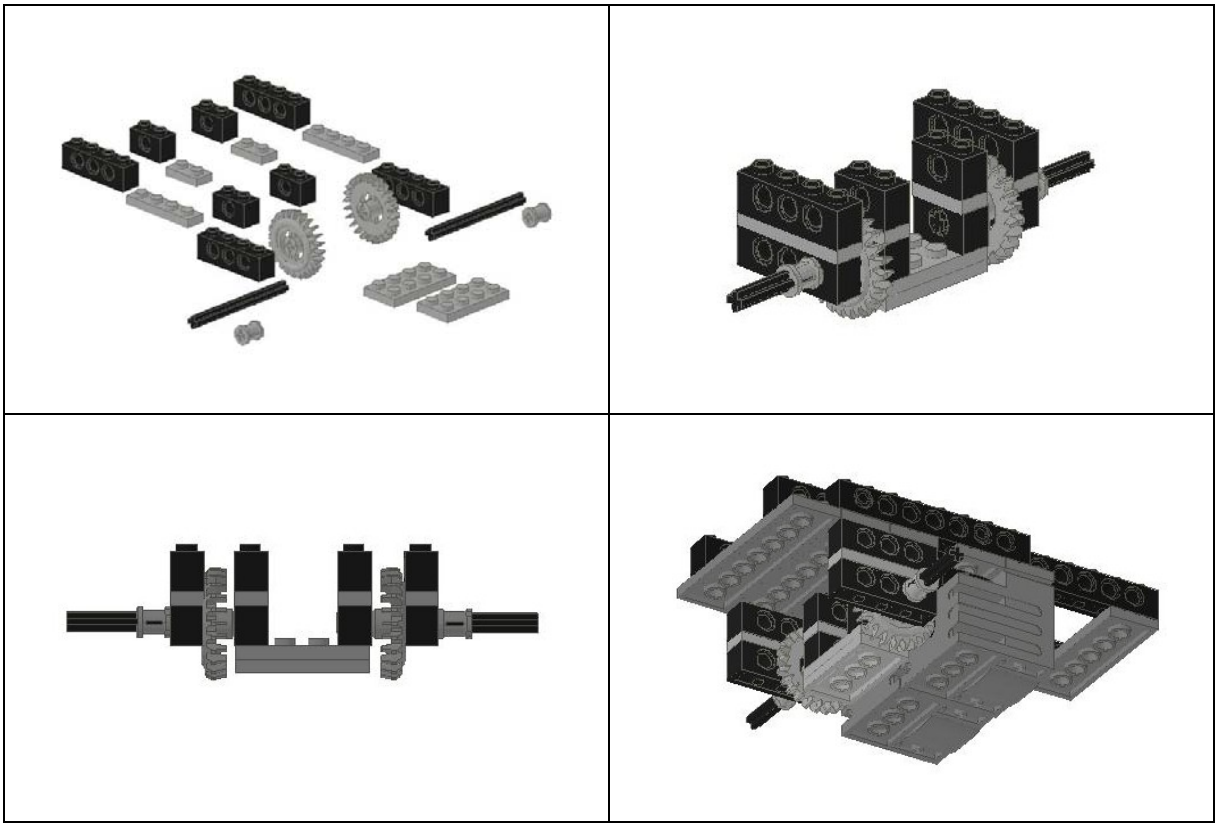
1. Schritt



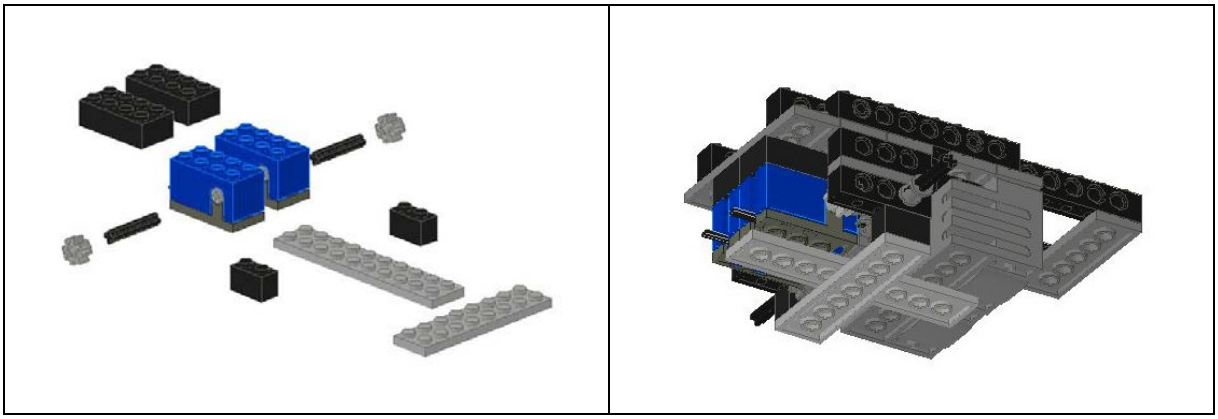
2. Schritt



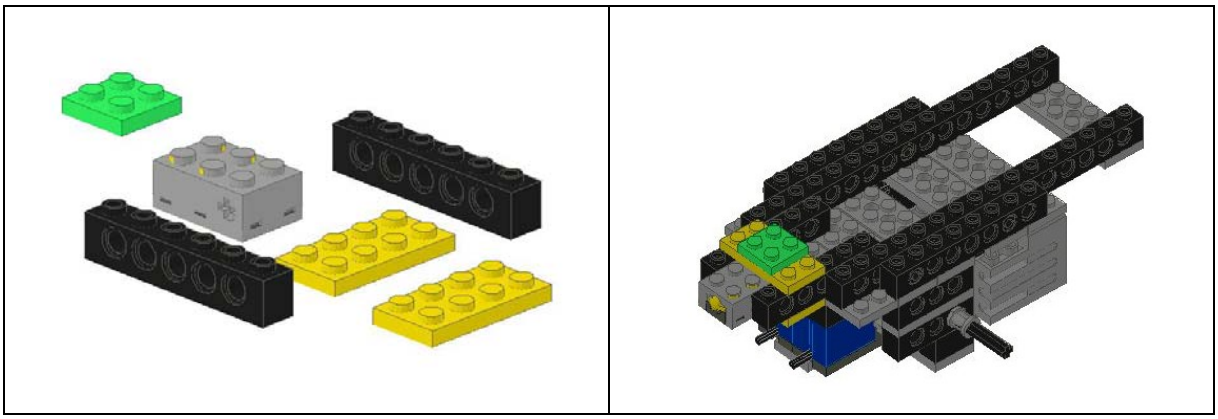
3. Schritt



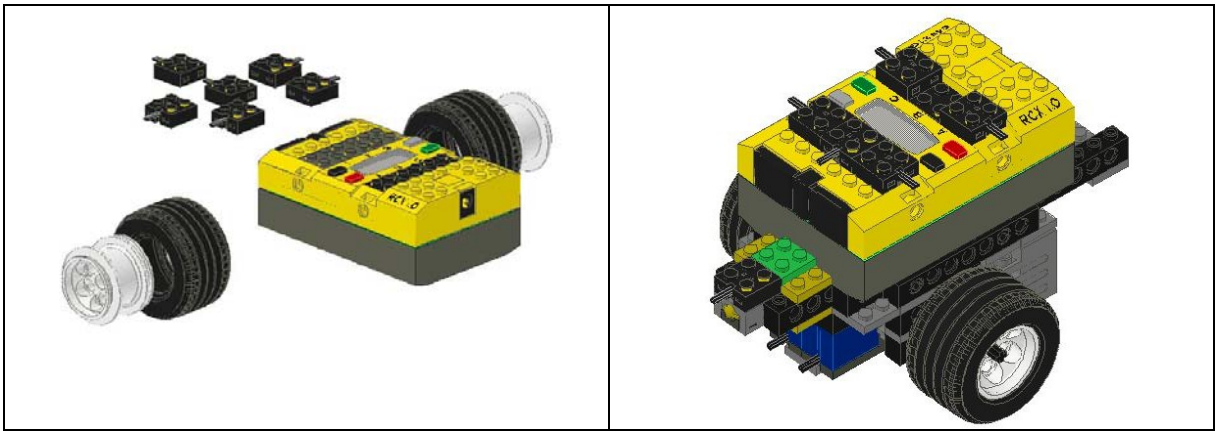
4. Schritt



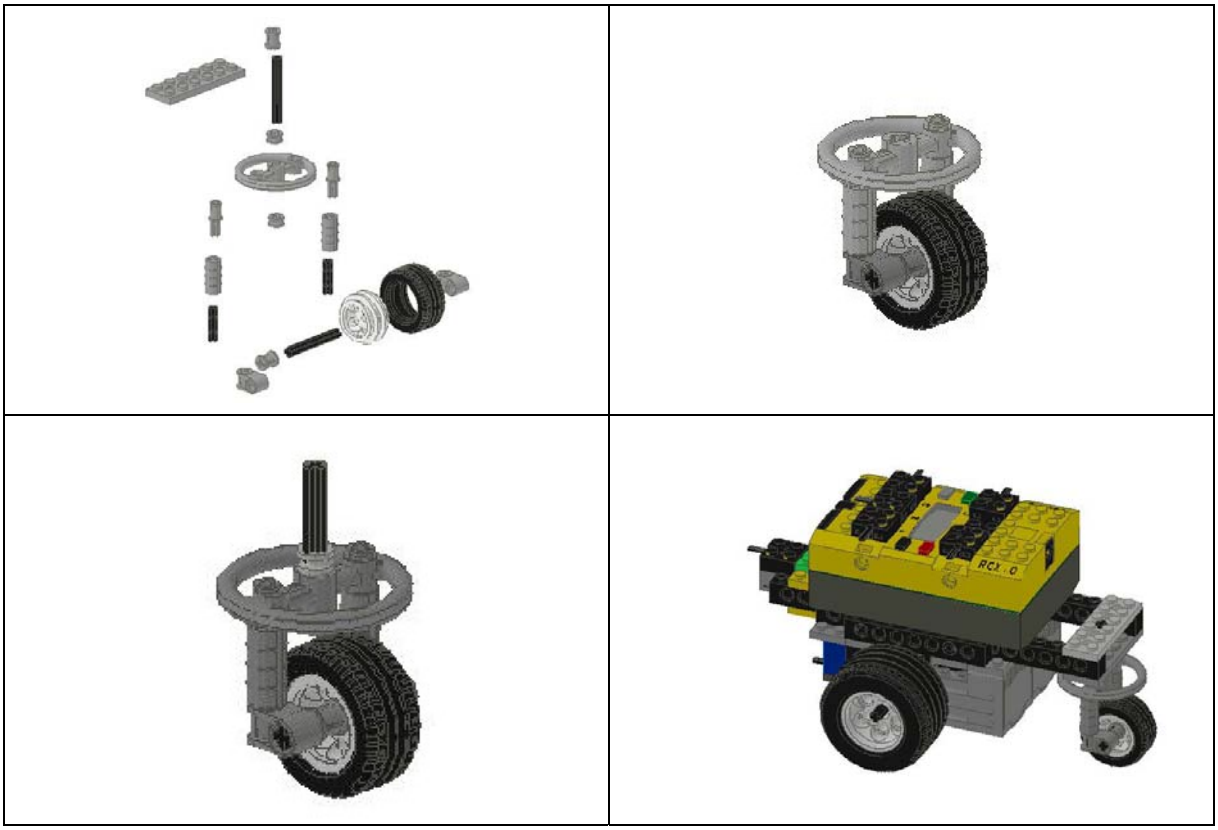
5. Schritt



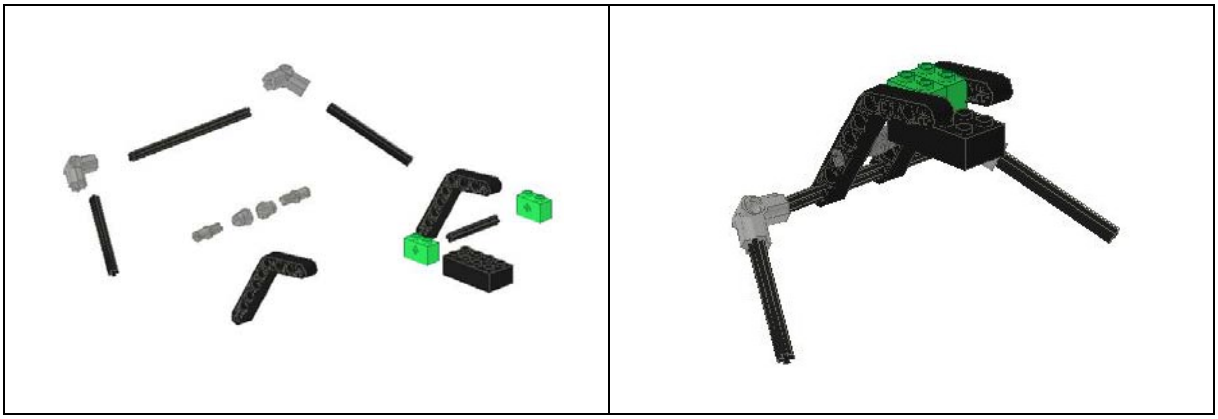
6. Schritt



7. Schritt



8. Schritt



9. Schritt

Teile-Liste von Tippy-Senior

Modell Tippy-Senior (tippy.ldr)

Schr.	Anz.	Farbe	Bauteil	Beschreibung
1	2	Black	3702.DAT	Technic Brick 1 x 8 with Holes
1	2	Black	3703.DAT	Technic Brick 1 x 16 with Holes
1	4	Black	2780.DAT	Technic Pin with Friction and Slots
1	1	Light-Gray	32001.DAT	Technic Plate 2 x 6 with Holes
1	5	Light-Gray	3738.DAT	Technic Plate 2 x 8 with Holes
2	2	Black	5306.DAT	Electric Brick 2 x 2 x 2/3 with Wire End
2	2	Light-Gray	71427C01.DAT	Electric Technic Mini-Motor 9v
2	2	Light-Gray	3647.DAT	Technic Gear 8 Tooth
3	2	Light-Gray	3023.DAT	Plate 1 x 2
3	2	Light-Gray	3710.DAT	Plate 1 x 4
3	2	Light-Gray	3020.DAT	Plate 2 x 4
3	2	Black	3706.DAT	Technic Axle 6
3	4	Black	3700.DAT	Technic Brick 1 x 2 with Hole
3	4	Black	3701.DAT	Technic Brick 1 x 4 with Holes
3	2	Light-Gray	3713.DAT	Technic Bush
3	2	Light-Gray	3650A.DAT	Technic Gear 24 Tooth Crown
4	2	Black	3004.DAT	Brick 1 x 2
4	2	Black	3001.DAT	Brick 2 x 4
4	2	Blue	2977C01.DAT	Electric Rotation Sensor (Complete Assembly Shortcut)
4	1	Light-Gray	3034.DAT	Plate 2 x 8
4	1	Light-Gray	3832.DAT	Plate 2 x 10
4	2	Black	4519.DAT	Technic Axle 3
4	2	Light-Gray	3647.DAT	Technic Gear 8 Tooth
5	1	Light-Gray	879.DAT	Electric Touch Sensor Brick 3 x 2 (Complete Assembly Shortcut)
5	1	Light-Green	3022.DAT	Plate 2 x 2
5	2	Yellow	3020.DAT	Plate 2 x 4
5	2	Black	3894.DAT	Technic Brick 1 x 6 with Holes
6	6	Black	5306.DAT	Electric Brick 2 x 2 x 2/3 with Wire End
6	1	Black	884.DAT	Electric Mindstorms RCX (Complete Assembly Shortcut)
6	2	Black	6594.DAT	Tyre 49.6 x 28 VR
6	2	White	6595.DAT	Wheel 49.6 x 28 VR
7	1	Black	3705.DAT	Technic Axle 4
7	2	Light-Gray	6536.DAT	Technic Axle Joiner Perpendicular
7	1	Light-Gray	3713.DAT	Technic Bush
7	1	Black	6578.DAT	Tyre 30.4 x 14 VR
7	1	White	2994.DAT	Wheel 30.4 x 14 VR
7	2	Black	32062.DAT	Technic Axle 2 Notched
7	2	Light-Gray	6538B.DAT	Technic Axle Joiner Offset
7	2	Light-Gray	3749.DAT	Technic Axle Pin

7	1	Light-Gray	3736.DAT	Technic Pulley Large
7	1	Black	3705.DAT	Technic Axle 4
7	2	Light-Gray	4265C.DAT	Technic Bush 1/2 Smooth
7	1	Light-Gray	3713.DAT	Technic Bush
7	1	Light-Gray	32001.DAT	Technic Plate 2 x 6 with Holes
8	1	Black	3001.DAT	Brick 2 x 4
8	1	Black	3705.DAT	Technic Axle 4
8	2	Light-Green	32064.DAT	Technic Brick 1 x 2 with Axlehole
8	2	Light-Gray	3749.DAT	Technic Axle Pin
8	2	Light-Gray	424.DAT	Technic Handle
8	2	Black	6629.DAT	Technic Liftarm 1 x 9 Bent
8	2	Light-Gray	32015.DAT	Technic Angle Connector #5
8	2	Black	3706.DAT	Technic Axle 6
8	1	Black	3737.DAT	Technic Axle 10

Index der Fachbegriffe

Internet-Adressen

Sun: Java™ 2 Platform, Standard Edition (J2SE™)
http://java.sun.com/j2se/
download: `j2sdk-1_4_2_04-windows-i586-p.exe`
`j2sdk-1_4_2-doc.zip`

Sun: Communications API
http://java.sun.com/products/javacomm/
download: `javacomm20-win32.zip`

Sun: The Java-Tutorial
http://java.sun.com/docs/books/tutorial/
download: `tutorial.zip`

Sun: Tutorials & Short Courses
http://developer.java.sun.com/developer/onlineTraining/

Sun: Java Foundation Classes (JFC), Cross Platform GUIs & Graphics
http://java.sun.com/products/jfc/
- → The Swing Connection
  http://java.sun.com/products/jfc/tsc/
- → Swing Connection, Acticle Index
  http://java.sun.com/products/jfc/tsc/articles
- → Getting Started with Swing
  http://java.sun.com/productsjfc/tsc/articles/getting_started
- → A Swing Architecture Overview
  http://java.sun.com/productsjfc/tsc/articles/architecture
- → Painting in AWT and Swing
  http://java.sun.com/productsjfc/tsc/articles/painting/

IBM Jikes-Compiler
http://oss.software.ibm.com/developerworks/opensource/jikes/
download: `jikes-1.20-1.windows.zip`

JavaEditor
http://www.bildung.hessen.de/abreich/inform/skii/material/java/editor.htm
download: `JavaEdit.zip`
`vorlagen.zip`

JCreator
http://www.jcreator.com/
download: `jcrea310.zip`

Das Java-Buch von Guido Krüger
http://www.javabuch.de/
download: `hjp3html.zip`
`hjp3exam.zip`

Mozilla
http://www.mozilla.org
download: `mozilla-win32-1.6-installer.exe`

Localized builds and language packs
http://www.mozilla.org/projects/l10n/mlp_status.html#moz_1.6
download: `mozilla-1.6-lang-de-AT.xpi`

Das *leJOS*-Projekt
http://lejos.sourceforge.net/
download: `lejos_win32_2_1_0.zip`
`lejos_win32_2_1_0.doc.zip`

rcx-tools
http://rcxtools.sourceforge.net/d_home.html
download: `RCXTools_1_5.zip`
`rcxdirect.zip`

Das LeJOS-Tutorial
http://mp.scholz.bei.t-online.de/lejos/tutorial/index.html
download: `tutorial.zip`

Axel T. Schreiner: Controlling Fischertechnik with Java
http://www.cs.rit.edu/~ats/talks/ft/
download: `code.zip`

Ulrich Müller: ftComputing – Java-Ecke
http://www.ftcomputing.de
download: umfish20setup.exe

fischertechnik
http://www.fischertechnik.com
download: `llw305patch.zip`
`teachin.zip`
`30402Drive.zip`
`indrob.zip`
`drehzahl.zip`

LEGO mindstorms
http://mindstorms.*LEGO*.com
download: `RIS20XPPatch.zip`

LEGO-Modelle konstruieren
→ mlCAD
http://www.lm-software.com/mlcad/
download: `mlcad300.zip`
→ LDraw
http://www.ldraw.org
download: `ldraw027.exe`
`complete.exe`
→ mlCAD-Tutorial
http://www.hpfsc.de/mlcd_tut/index_de.html
download: `tutorial.zip`

MSWLogo für Windows
http://www.ph-ludwigsburg.de/mathematik/personal/klaudt/logo/frame01.html

OpenOffice:
Kostenloses Office-Paket (Textverarbeitung, Tabellenkalkulation, Präsentation u.v.m)
ersetzt Microsoft-Office (Word, Excel und Powerpoint)
http://de.openoffice.de
download: `OOo_1.1.1_Win32Intel_install_de.zip`
`vorlagen_de_sissl.zip`

Literatur

Brian Bagnall, LEGO mindstorms Programming, Prentice Hall, 2002

Giuliou Ferrari et al., LEGO minstorms with Java, Syngress Publishing, 2002

Stefan Middendorf et al., Java Programmierhandbuch und Referenz, dpunkt.verlag, 2003