

# Erste Schritte mit Java unter leJOS

für den LEGO<sup>®</sup>-Roboter-Wettbewerb

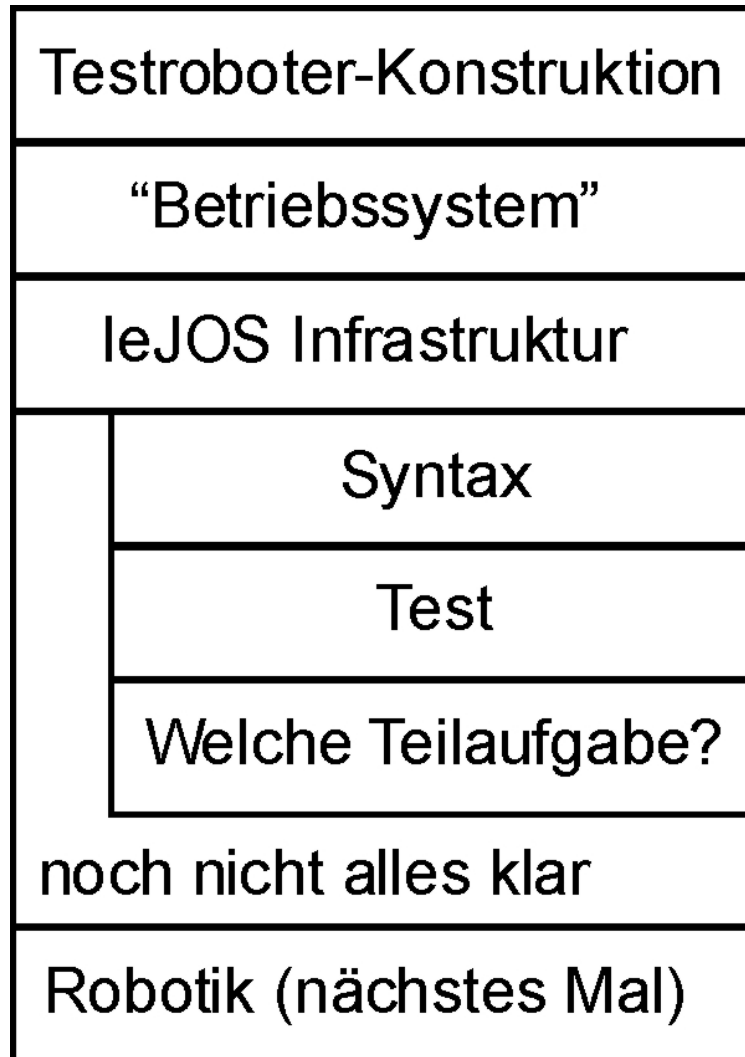
Oktober/November 2002

Prof. Dr. Michael Wülker

Fachhochschule Offenburg

Das Verhalten des Roboters für den Wettbewerb auf der Basis der gewählten mechanischen Konstruktion steuern.

```
do {  
  Wie formuliert man ein Programm (für das Verhalten des Roboters)?  
  -> Syntax  
  Was führt das Programm tatsächlich aus?  
  -> Test  
  Welche Teilaufgabe des Gesamtprojekts ist gelöst?  
  -> modulares Design  
} while (noch nicht alles klar) ;
```

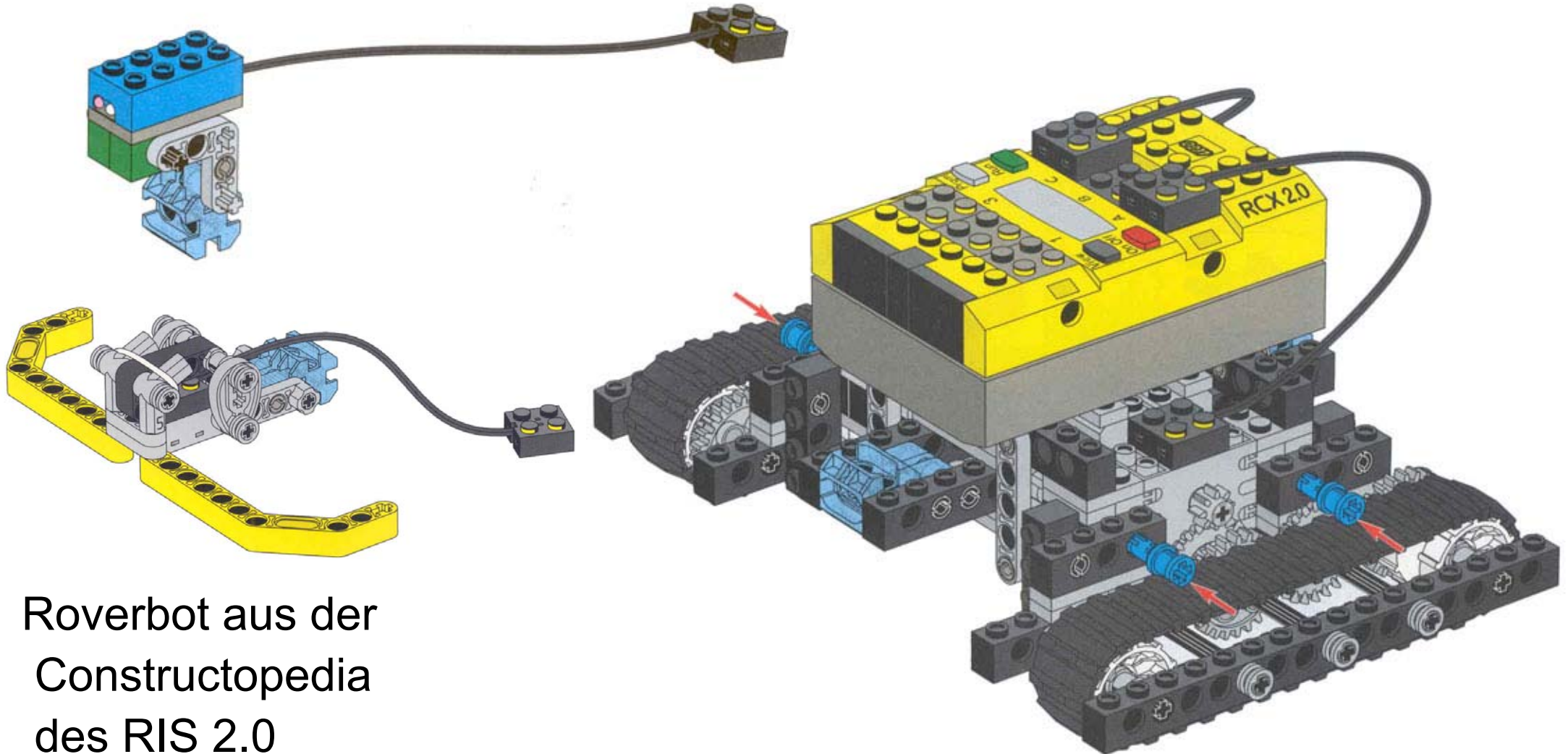


## erste Schritte

- ◆ HelloRobby
- ◆ Variable (Attribute) und Operatoren
- ◆ Klassen, Objekte und Vererbung
- ◆ Kontrollstrukturen
- ◆ Motoren und Sensoren
- ◆ Klassenstruktur von leJOS
- ◆ Schnittstellen
- ◆ Parallelrechner-Betrieb  
Listener, Behavior, Threads
- ◆ Nützliches: Timer, Sound, Display
- ◆ Kommunikation: Datalogging

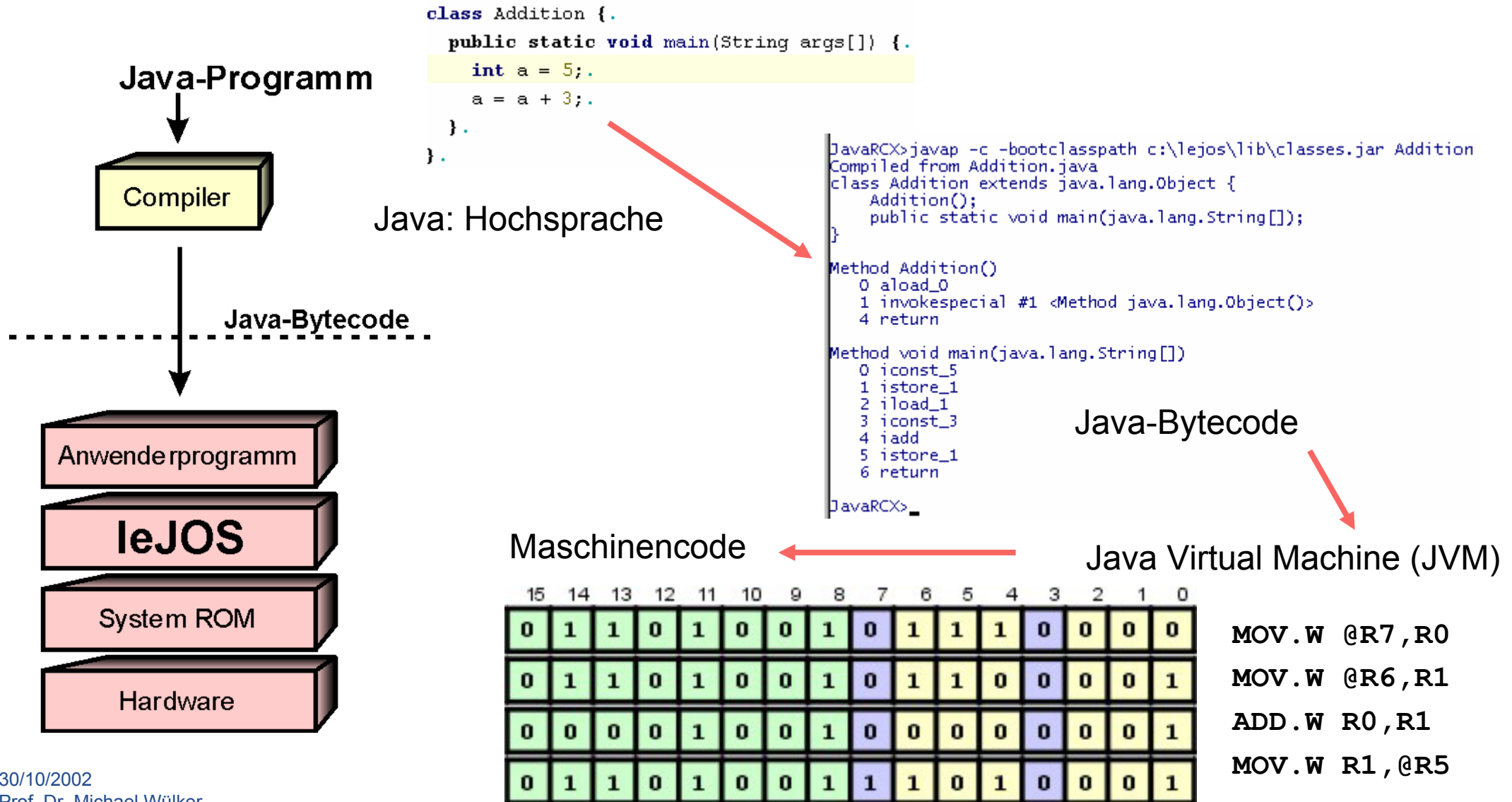
Fortsetzung

# Testroboter-Konstruktion



Roverbot aus der  
Constructopedia  
des RIS 2.0

# Java und leJOS-Firmware ("Betriebssystem")



# Installation und Betrieb der Java-Programmierung unter leJOS

## ◆ Installation

- Java2 SDK installieren (<http://java.sun.com>) und Pfad setzen z. B.  
`set path=%path%;C:\j2sdk1.4.0`
- leJOS herunterladen (<http://www.lejos.org>), entpacken (z. B. nach C:\lejos)  
und Pfad für leJOS setzen: `set path=%path%;C:\lejos\bin`
- wahlweise leJOS-Dokumentation herunterladen und entpacken (z. B. C:\lejos\Dokumente)
- USB-Tower (Win98, Win2000, WinXP) anschließen und Treiber von der Mindstorms-CD installieren.

`set RCXTTY=USB`      sonst      `set RCXTTY=COM1`      für seriellen IR-Tower

- Falls der Brick keinen Batteriespannungswert (links vom Männchen) zeigt,  
Firmware in den RCX laden: `lejosfirmdl -f`

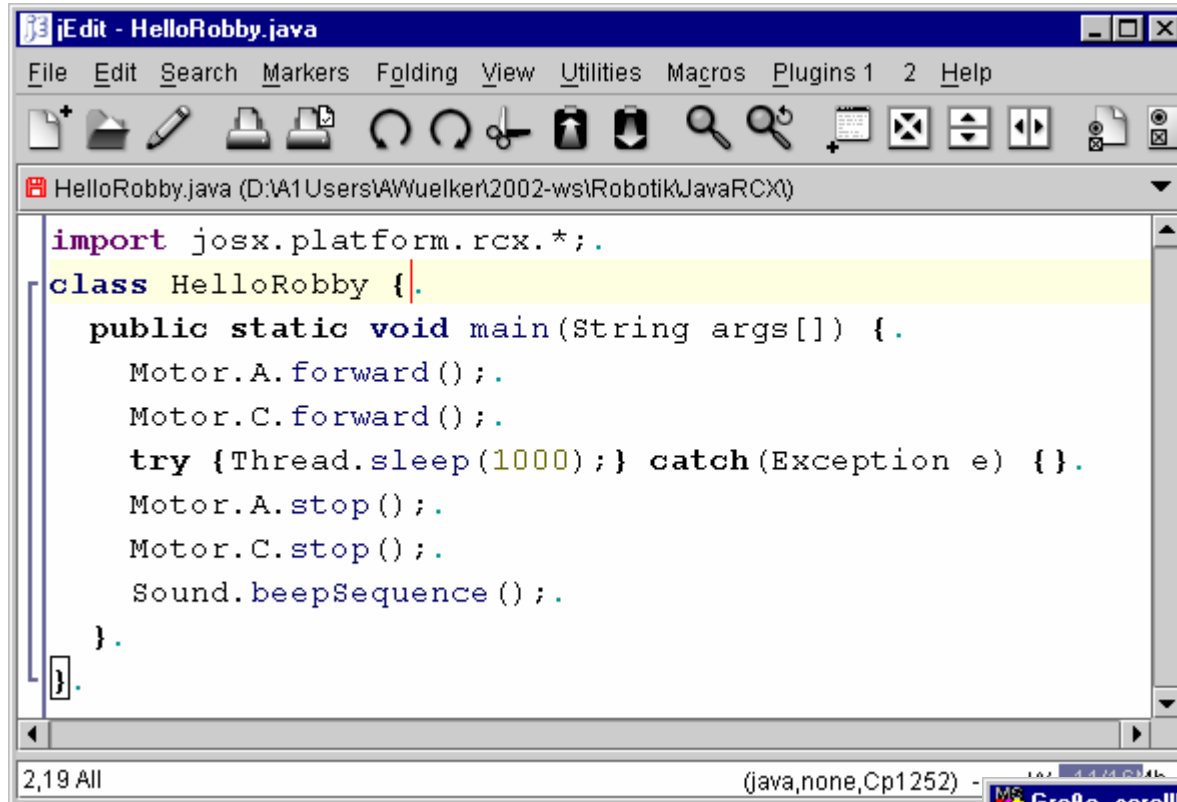
## ◆ Editieren

- Standardeditor (Notepad oder jEdit von <http://www.jedit.org>)

## ◆ Compilieren und Download

- `javac -target 1.1 -bootclasspath c:\lejos\lib\classes.jar HalloRobby.java`
- `lejos HalloRobby`

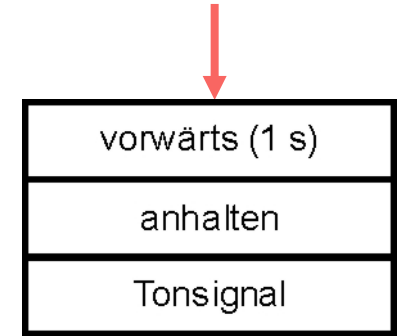
# HelloRobby.java



```
import josx.platform.rcx.*;
class HelloRobby {
    public static void main(String args[]) {
        Motor.A.forward();
        Motor.C.forward();
        try {Thread.sleep(1000);} catch (Exception e) {}
        Motor.A.stop();
        Motor.C.stop();
        Sound.beepSequence();
    }
}
```

neue  
Datei

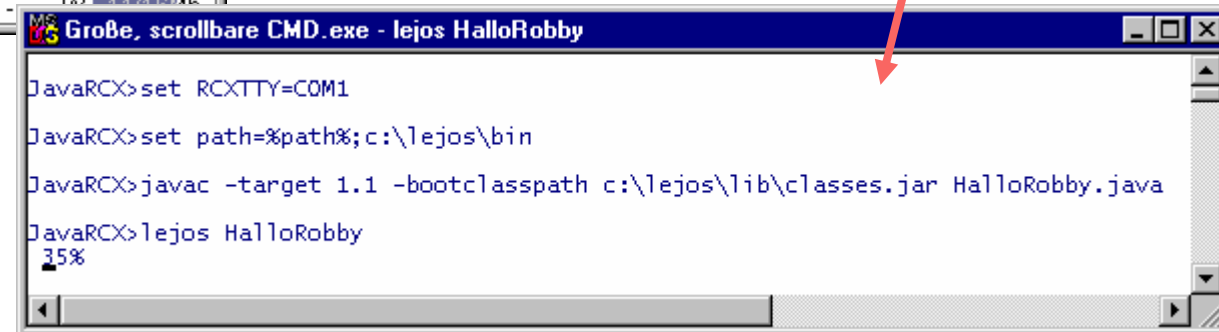
Struktogramm als  
Darstellung des  
Programmablaufs



übersetzen  
und  
übertragen

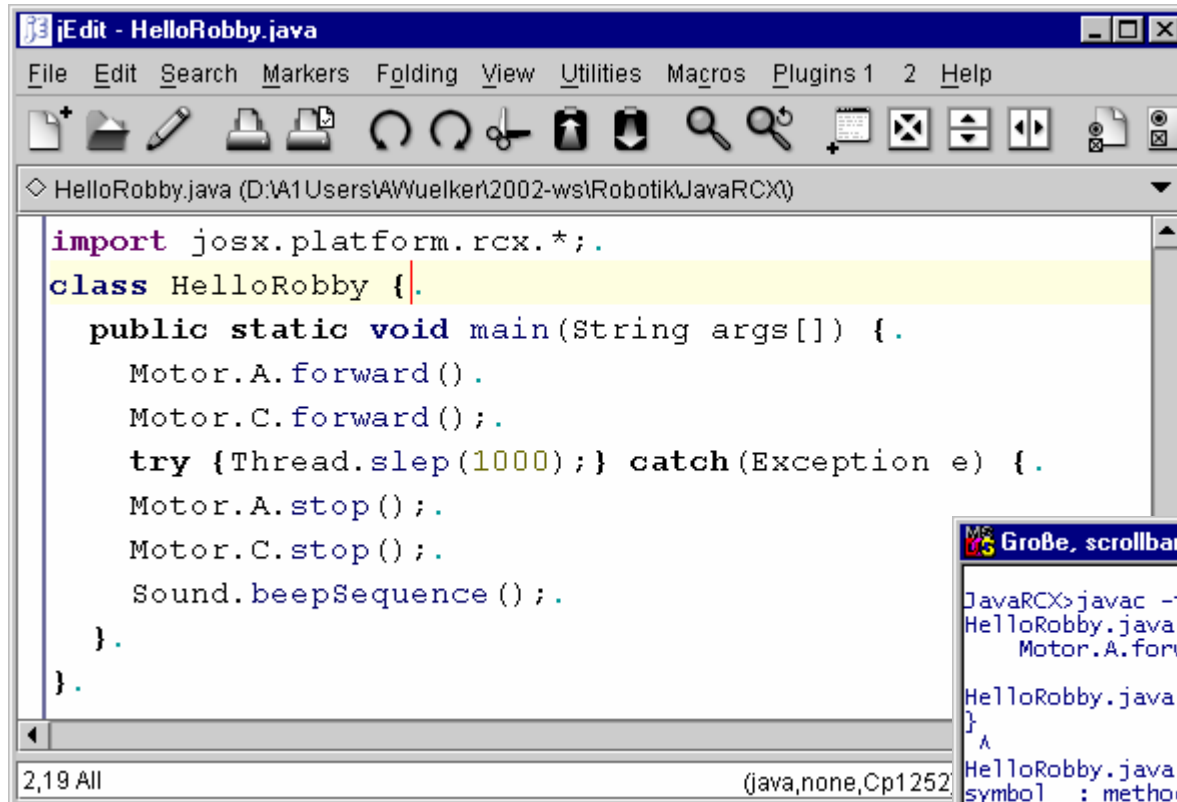
Syntax  
(„Rechtschreibregeln“)  
für Anweisungsfolge

```
{
    Anweisung1;
    Anweisung2;
}
```



```
JavaRCX>set RCXTTY=COM1
JavaRCX>set path=%path%;c:\lejos\bin
JavaRCX>javac -target 1.1 -bootclasspath c:\lejos\lib\classes.jar HalloRobby.java
JavaRCX>lejos HalloRobby
  35%
```

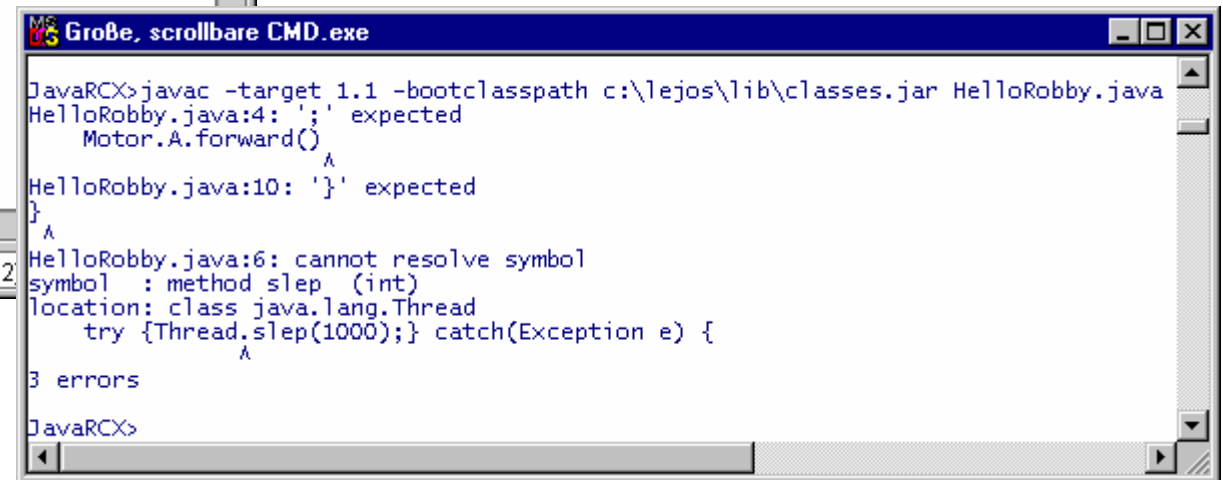
# Fehler sind normal!



```
import josx.platform.rcx.*;.
class HelloRobby {
    public static void main(String args[]) {
        Motor.A.forward().
        Motor.C.forward();.
        try {Thread.slep(1000);} catch(Exception e) {
        Motor.A.stop();.
        Motor.C.stop();.
        Sound.beepSequence();.
    }.
}
```

Was stimmt hier nicht?

Syntaxfehler



```
Große, scrollbare CMD.exe
JavaRCX>javac -target 1.1 -bootclasspath c:\lejos\lib\classes.jar HelloRobby.java
HelloRobby.java:4: ';' expected
    Motor.A.forward()
                    ^
HelloRobby.java:10: '}' expected
}
^
HelloRobby.java:6: cannot resolve symbol
symbol  : method slep (int)
location: class java.lang.Thread
    try {Thread.slep(1000);} catch(Exception e) {
                    ^
3 errors
JavaRCX>
```

Laufzeitfehler: beepSequence () wird nicht immer gespielt!

# Variable (Attribute) und Operationen

**Variable** sind Speicherschubladen mit Namen.

Schubladeninhalte können in Java verschiedenartige Datentypen sein: **byte**, **short**, **int**, (**long**), **float**, **double** (32 bit), **boolean**, **char**

**Namen** müssen mit Buchstaben beginnen und dürfen ansonsten Ziffern und `_` enthalten.

Groß- und Kleinschreibung werden unterschieden!

```
public static void main(String args[]) {  
    aa = 10;  
    bb = 20 * 5;  
    cc = bb;  
    cc /= aa;  
    cc -= 1;  
    aa = 10 * (cc + 3);  
}
```

```
import javax.swing.*;  
class OdometryInARCSandbox {  
    public static void main(String args[]) {  
        double xPosition = 0., yPosition = 0.;  
        int countsLeft = 100, countsRight = 73;  
        float conversionToDisplacement = 0.245f;  
        float baseLine = (float)12.3;  
        float robotDisplacement;  
        double deltaTheta, Orientation = Math.PI/2.;  
        float distanceLeft = conversionToDisplacement * countsLeft;  
        float distanceRight = conversionToDisplacement * countsRight;  
        robotDisplacement = (distanceLeft + distanceRight) / 2.f;  
        deltaTheta = (distanceRight - distanceLeft) / baseLine;  
        Orientation = Orientation + deltaTheta;  
        xPosition = xPosition + robotDisplacement * Math.cos(Orientation);  
        yPosition = yPosition + robotDisplacement * Math.sin(Orientation);  
    }  
}
```

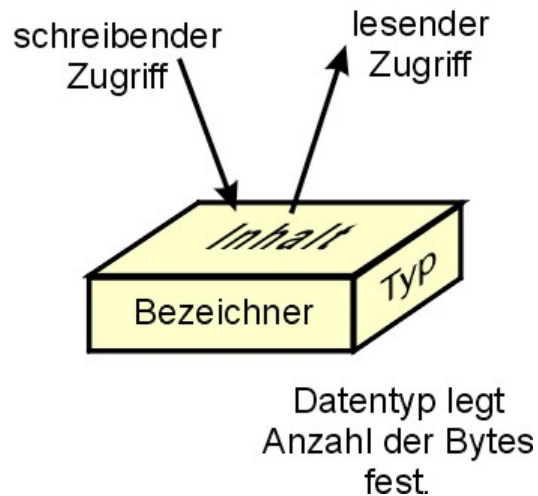
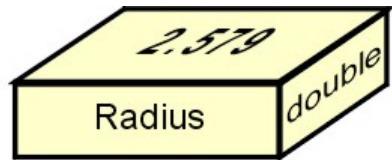
**Operationen** (+, -, \*, /, ... ) verrechnen Konstanten und Variableninhalte.

**Zuweisungen** speichern das Ergebnis in einer Variablen ab.

`cc -= 1;` entspricht `cc = cc - 1;`

`aa++;` entspricht `aa = aa + 1;`

# Eigenschaften von Attributen



- ◆ Name (Bezeichner)
- ◆ Datentyp
  - `int`, `char`, `float`, `double`, `boolean`, Referenztyp, ...
- ◆ Wert (Inhalt, Angabe als Literal)
  - `'A'`, `"Zeichenkette"`, `-10`, `0x3E`, `3.14E-4`, ...
- ◆ Zugriffsart (nur „lesend“ oder „lesend und schreibend“)
  - `private final float PI = 3.141593f;`
- ◆ Initialisierung
  - `public static final int VOLLE_MWST = 16;`
- ◆ Sichtbarkeitsbereich
  - lokale Attribute, `private`, `public`, `protected`, `this`, ...
- ◆ Lebensdauer
  - `persistent` oder `transient`
- ◆ Objekt- oder Klassenattribut (`static`)
- ◆ Restriktionen (z. B. eingeschränkte Zahlbereiche)

# Grundtypen (plattformunabhängig)

Kategorie	Java-Datentyp	Speicher (Bytes)	Wertebereich	Voreinstellung	Beispiel für Literal
logisch	<b>boolean</b>	1	<b>true, false</b>	<b>false</b>	<b>false, true</b>
ganzzahlig	<b>byte</b>	1	0 ... 255	0	105
	<b>short</b>	2	-32768 ... 32767	0	0777
	<b>int</b>	4	-2147483648 ... 2147883647	0	0x3F
	<b>long</b>	8	-9223372036854775808 ... 9223372036854775807	0L	505L
Zeichen	<b>char</b>	2	\u0000 (0) ... \uffff (65535)	\u0000	'\n', '\u103f', 'a'
technisch-wissenschaftl. (Gleitpunktzahlen)	<b>float</b>	4	$-3.4 \cdot 10^{38} \dots +3.4 \cdot 10^{38}$	0.0f	2., .8,
	<b>double</b>	8	$-1.7 \cdot 10^{308} \dots +1.7 \cdot 10^{308}$	0.0d	2.3e-6, 80f, 10d

# Zahlen-Operatoren

## ◆ binäre Operatoren

- arithmetische Operatoren: +, -, \*, /, % (Modulo, also Rest der Division)
- Vergleichsoperatoren: <, <=, >, >=, ==, !=  
Test auf Gleichheit bei Gleitpunktzahlen:  
`abs(x-y) < Epsilon`

## ◆ unäre Operatoren

- -, +
- ++ (Inkrement, z. B. `a = ++b`, `a = b++`)
- -- (Dekrement)

## ◆ bitweise Operatoren

- &, |, ^, <<, >>, >>>, ~
- <<=, >>=, >>>=, &=, |=, ^=

Priorität	Operator
0	[ ] ( ) .
1	++ -- + - ~ ! (type) new
2	* / %
3	+ -
4	<< >> >>>
5	< <= > >= instanceof
6	== !=
7	&
8	^
9	
10	&&
11	
12	? :
13	*= /= %= += -= <<= >>= >>>= &= ^=  =

# Ausdrücke und Typumwandlungen

- ◆ Umwandlung nur zwischen verträglichen Datentypen möglich
  - z. B. nicht zwischen numerischen Datentypen und `boolean` oder Referenztypen
- ◆ implizite Typumwandlungen in einem Ausdruck oder bei einer Zuweisung erfolgen immer vom „schmäleren“ zum „breiteren“ Datentyp
  - Genauigkeitsverlust ist dabei möglich (z. B. von `long` nach `float`)
- ◆ explizite Typumwandlungen werden mit dem `cast`-Operator gemacht
  - z. B. `a = (int) (b/2.3f + 3.53f)`

Ausdruck	Ergebnis
<code>4 / 3</code>	1
<code>7 / 4 .</code>	1,75
<code>2 . * 3 / 4</code>	1,5
<code>1 / 2 * 4</code>	0
<code>7 % 4</code>	3
<code>(-7) % 2</code>	-1
<code>a++ - a</code>	-1
<code>10.0 / 0.0</code>	<b>Infinity</b>
<code>0.0 / 0.0</code>	<b>NaN</b>
<code>(int) 4.1</code>	4

# Eine erste Klasse – Robot.java

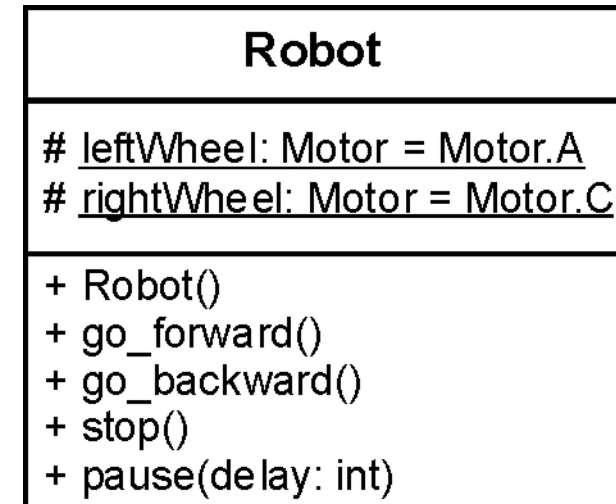
**Klasse** → `public class Robot {.`

**Attribute** → `protected static Motor leftWheel = Motor.A;.`  
→ `protected static Motor rightWheel = Motor.C;.`

**Methoden (Operationen)** → `public void go_forward() {.`  
→ `rightWheel.forward();.`  
→ `leftWheel.forward();.`  
→ `}..`  
→ `public void go_backward() {.`  
→ `rightWheel.backward();.`  
→ `leftWheel.backward();.`  
→ `}..`  
→ `public void stop() {.`  
→ `rightWheel.stop();.`  
→ `leftWheel.stop();.`  
→ `}..`  
→ `public void pause(int delay) {.`  
→ `try {.`  
→ `Thread.sleep(delay);.`  
→ `}..`  
→ `catch (InterruptedException ie) {.`  
→ `TextLCD.print("error");.`  
→ `}..`  
→ `}..`

Eine **Klasse** klassifiziert gleichartige Objekte unserer Umwelt und dient als Schablone für solche Objekte

Robot



# Objekte – wie man Robot.java testet

robby: Robot

## RobotTest.java

```
import josx.platform.rcx.*;
class RobotTest {
    private static Robot robby;
    public static void main(String args[]) {
        robby = new Robot();
        robby.go_forward();
        robby.pause(1000);
        robby.go_backward();
        robby.pause(1000);
        robby.stop();
    }
}
```

Der Programmablauf wird durch die Verwendung der Methoden der `Robot`-Klasse deutlich übersichtlicher.

`main()` -> (einziger) Startpunkt zum Ausführen einer Klasse  
`public` -> sichtbar für die Java Virtual Machine (JVM) zum Starten  
`static` -> "Klassenmethode", die immer existiert und aufgerufen werden kann, bevor überhaupt ein Objekt der Klasse erzeugt wurde.  
`void` -> der JVM wird nichts zurückgegeben  
`args[]` -> eine Zeichenkette ("String"), die beim Aufruf übergeben werden kann. (sinnlos beim RCX)

Hier wird das (eine) neue `Robot`-Objekt erzeugt.  
Bei einer Simulation z. B. eines Roboter-Fußballspiels bräuchte man 22 `Robot`-Objekte.

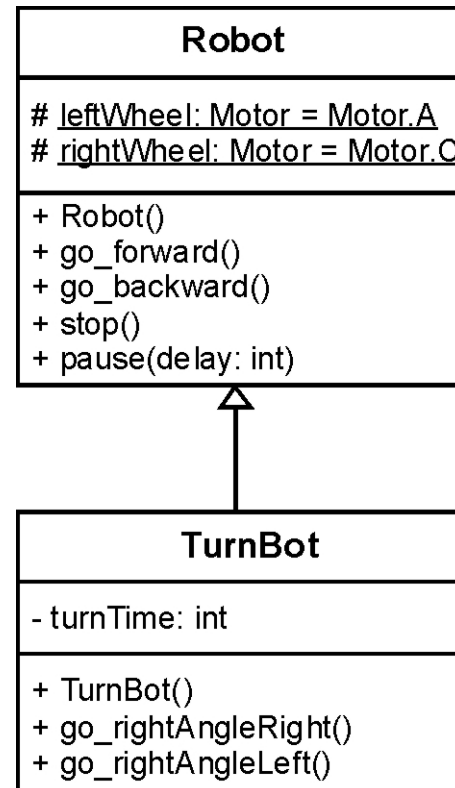
roterTorwart: Robot

gelberTorwart: Robot

# Vererbung – TurnBot.java

**extends** -> TurnBot "erbt" alle Attribute und Methoden von Robot.  
Deshalb sind `leftWheel`, `rightWheel`, `forward()`, ...  
in TurnBot auch bekannt.

```
import josx.platform.rcx.*;
class TurnBot extends Robot {
    private int turnTime;
    public TurnBot() {
        turnTime = 1150;
    }
    public void go_rightAngleRight() {
        leftWheel.forward();
        rightWheel.backward();
        pause(turnTime);
        stop();
    }
    public void go_rightAngleLeft() {
        leftWheel.backward();
        rightWheel.forward();
        this.pause(turnTime);
        this.stop();
    }
}
```



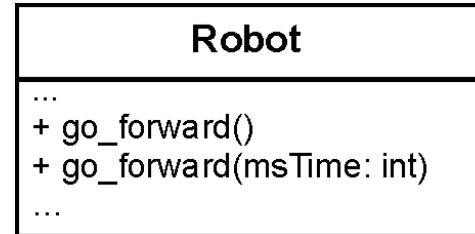
Standardkonstruktor legt Wert für `turnTime` so fest, dass TurnBot möglichst eine 90°-Drehung macht.

## TurnBotTest.java

```
import josx.platform.rcx.*;
class TurnBotTest {
    private static TurnBot roby;
    public static void main(String args[]) {
        roby = new TurnBot();
        roby.go_forward(); roby.pause(500);
        roby.go_rightAngleRight();
        roby.go_forward(); roby.pause(500);
        roby.go_rightAngleLeft();
        roby.go_forward(); roby.pause(500);
        roby.stop();
    }
}
```

# Überladen von Methoden und Konstruktoren

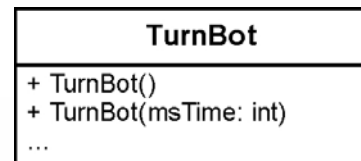
```
public void go_forward() {  
    rightWheel.forward();  
    leftWheel.forward();  
}.  
public void go_forward(int msTime) {  
    rightWheel.forward();  
    leftWheel.forward();  
    pause(msTime);  
    stop();  
}.  
}
```



`go_forward()` in `Robot` wird überladen, so dass `pause()` im Testprogramm entfallen kann.

Standardkonstruktor in `TurnBot` wird so überladen, dass `turnTime` gesetzt werden kann:

```
public TurnBot() {  
    turnTime = 1150;.  
}.  
public TurnBot(int mstime) {  
    turnTime = mstime;.  
}.  
}
```



## TurnBotOverloadedTest.java

```
import josx.platform.rcx.*;  
class TurnBotOverloadedTest {  
    private static TurnBot roby;.  
    public static void main(String args[]) {  
        roby = new TurnBot(750); // adjust turnTime.  
        roby.go_forward(500);.  
        roby.go_rightAngleRight();.  
        roby.go_forward(500);.  
        roby.go_rightAngleLeft();.  
        roby.go_forward(500); .  
        // no stop() since overloaded method stops.  
    }.  
}
```

# Referenztypen

- ◆ Adressen
  - „Hausnummern“ der byteweise organisierten Speicherzellen
- ◆ Referenz
  - Bezug auf ein Objekt **ohne** Kenntnis dessen physikalischer Adresse
  - Einfache Datentypen können in Java nicht referenziert werden.
- ◆ Referenzvariable
  - Definition: `Robot wallFollower, lineFollower;`
  - Zuweisung einer Objektreferenz: `wallFollower = new Robot();`
  - Referenzierung einer Methode: `wallFollower.setPosition(0., -20.);`
  - Wert einer Referenzvariable, die (noch) nicht auf ein Objekt zeigt: `null`
- ◆ Verwechslungsgefahr zwischen Referenz und Eigenschaften eines Objekts!
  - Zuweisung (für Objekte gleichen Typs) kopiert nur die Referenz, aber nicht die Inhalte:
 

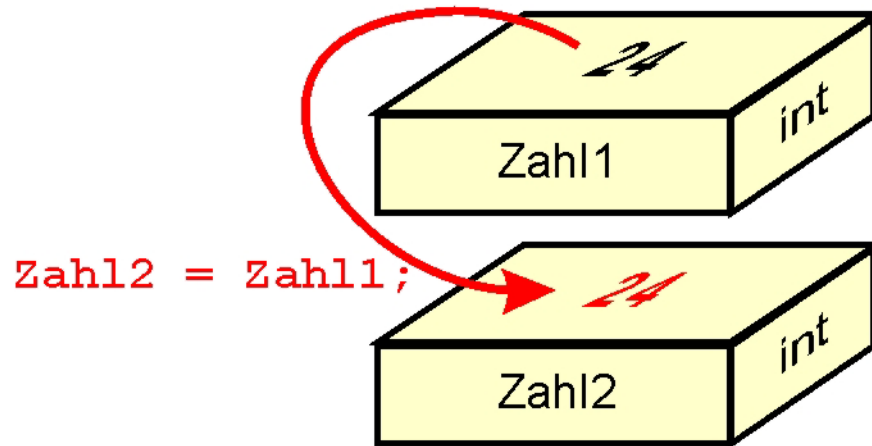
```
wallFollower = lineFollower;
```
  - Vergleich testet nur auf Gleichheit der Referenzen, aber nicht der Inhalte:
 

```
if (wallFollower == lineFollower) { /* ... */ }
```
- ◆ Vergleich von Objektinhalten:
 

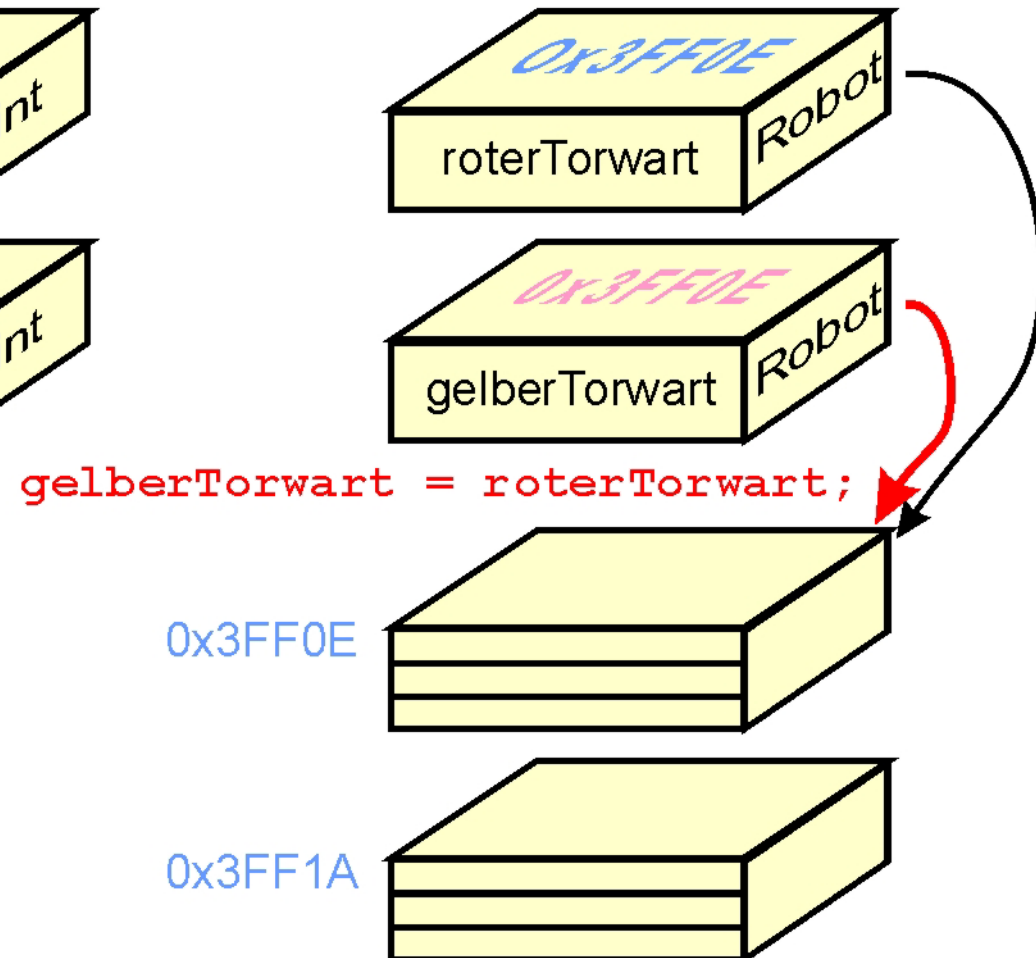
```
if (wallFollower.equals(lineFollower)) { /*...*/ }
```

# Unterschiede zwischen Grunddatentypen und Referenztypen

Grundtyp



Referenztyp

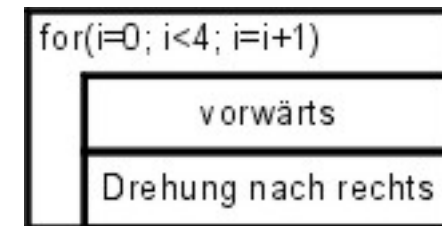


Syntax für for-Schleife:

```
import josx.platform.rcx.*;.
class SquareTest {
    private static TurnBot robbby;.
    public static void main(String args[]) {
        robbby = new TurnBot(1135);
        for (int i=0; i<4; i=i+1) { // loop four times
            robbby.go_forward(); robbby.pause(1000);
            robbby.go_rightAngleRight();
        }.
    }.
}.
```

```
for (Startausdruck; Endeausdruck; Schrittweite)
{
    Wiederholungsanweisung1;
    Wiederholungsanweisung2;
    ...
}
```

Struktogrammdarstellung:



Schnellausstieg aus der Schleife: **break**;

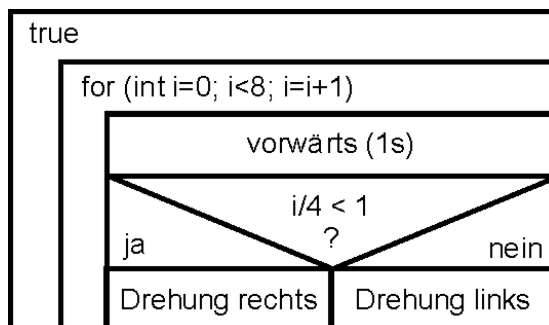
Ausstieg aus Schleifendurchlauf: **continue**;

# Kontrollstrukturen: if-else und while-Schleife

## FigureOfEightTest.java

```
import josx.platform.rcx.*;
class FigureOfEightTest {
    private static TurnBot roby;
    public static void main(String args[]) {
        roby = new TurnBot(1125);
        while (true) {
            for (int i=0; i<8; i=i+1) { // loop eight times.
                roby.go_forward(); roby.pause(1000);
                if (i/4 < 1) // for the first four loops turn right
                    roby.go_rightAngleRight();
                else // turn left for the remaining loops.
                    roby.go_rightAngleLeft();
            }
        }
    }
}
```

## Struktogrammdarstellung:



## Syntax für if-else-Auswahl:

```
if (Bedingung) {
    Anweisung1;
    Anweisung2;
    ...
}
else {
    AnweisungA;
    AnweisungB;
    ...
}
```

## Syntax für while-Schleife:

```
while (Bedingung)
{
    Wiederholungsanweisung1;
    Wiederholungsanweisung2;
    ...
}
```

Schachtelung von  
Kontrollstrukturen  
Endlosschleife

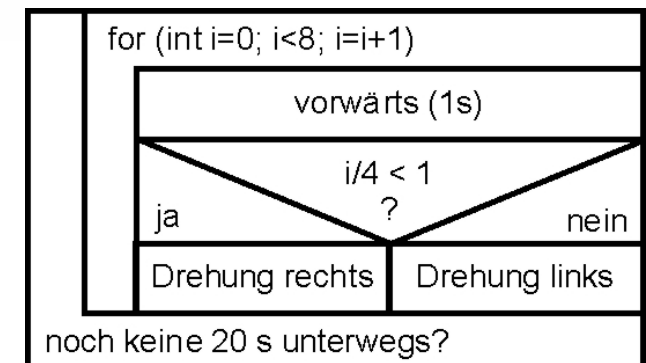
# Kontrollstrukturen: do-while-Schleifen

```
import josx.platform.rcx.*;.
class FigureOfEightStopped {
    private static TurnBot robbby;.
    public static void main(String args[]) {
        robbby = new TurnBot(1125);.
        int startTime = (int)System.currentTimeMillis();.
        do {
            for (int i=0; i<8; i=i+1) { // loop eight times.
                robbby.go_forward(); robbby.pause(1000);.
                if (i/4 < 1) // for the first four loops turn right.
                    robbby.go_rightAngleRight();.
                else // turn left for the remaining loops.
                    robbby.go_rightAngleLeft();.
                LCD.showNumber(((int)System.currentTimeMillis()-startTime)/100);
            }.
        } while ((int)System.currentTimeMillis()-startTime < 20000);.
    }.
}
```

Syntax für do-while-Schleife:

```
do
{
    Wiederholungsanweisung1;
    Wiederholungsanweisung2;
    ...
} while (Bedingung);
```

Struktogrammdarstellung:



# Motoren können bremsen oder auslaufen

## FloatTest.java

```
import josx.platform.rcx.*;.
class FloatTest {
    private static Robot roby;.
    public static void main(String args[]) {
        roby = new Robot();.
        roby.go_forward();.
        roby.pause(1000);.
        roby.stop();.
        roby.pause(1000);.
        roby.go_forward();.
        roby.pause(1000);.
        roby.flt();.
        roby.pause(2000);.
    }.
}.

    /** Lets the robot travel on with motors not energised */.
    public void flt() {
        rightWheel.flt();.
        leftWheel.flt();.
    }.
}
```

`stop()` bremst die Motoren ab – es wird ein Strom von 0 A eingestellt,  
`flt()` hingegen trennt die Motoren aus dem Stromkreis, so dass diese auslaufen.

`flt()`-Methode in `Robot`  
verwendet die `flt()`-Methode  
der Klasse `Motor`

# Verschiedene Leistungsstufen der Motoren

## PowerTest.java

```
import josx.platform.rcx.*;.
class PowerTest {
    private static Robot robbby;.
    public static void main(String args[]) {
        robbby = new Robot ();
        for (int level=7; level>=0; level=level-1) {
            robbby.setPower(level);
            LCD.showNumber(level);
            robbby.go_forward();
            robbby.pause(1000);
            robbby.reverseDirection();
            robbby.pause(1000);
        }
    }
}
```

**setPower ()** hat acht Stufen (0-7) und wirkt sich nicht auf die Drehgeschwindigkeit, sondern auf die Leistung aus. Ein Effekt wird erst unter Last sichtbar.

**setPower ()** -Methode in Robot verwendet die **setPower ()** -Methode der Klasse **Motor**, usw.

```
public void setPower(int power) {
    rightWheel.setPower(power);
    leftWheel.setPower(power);
}
```

# Verwendung von Sensoren

## TouchSensorStopTest.java

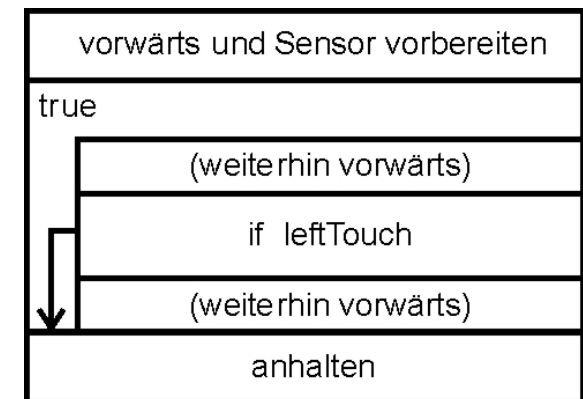
```
import josx.platform.rcx.*;.
class TouchSensorStopTest {
    private static Robot robbby;.
    public static void main(String args[]) {
        robbby = new Robot();.
        robbby.go_forward();.
        Sensor leftTouch = Sensor.S1;.
        while (!leftTouch.readBooleanValue()) {
            // nothing.
        }.
        robbby.stop();.
    }.
}
```

```
Sensor leftTouch = Sensor.S1;.
while (true) {
    // break the endless loop, if sensor touches
    if (leftTouch.readBooleanValue()).
        break;.
}.
robbby.stop();.
```

## Struktogrammdarstellung:



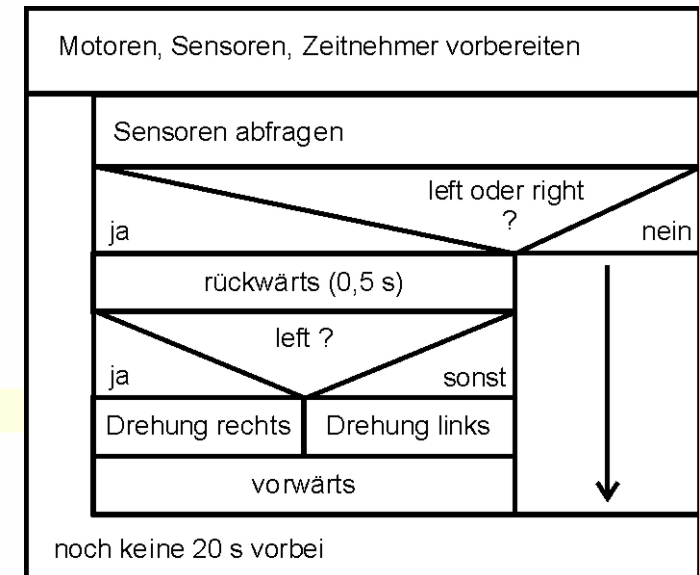
## Struktogrammdarstellung:



# Verknüpfung von Sensoren

```
class TouchSensorAvoidTest {  
    private static TurnBot robbby;.  
    public static void main(String args[]) {  
        robbby = new TurnBot();.  
        robbby.go_forward();.  
        Sensor leftTouch = Sensor.S1;.  
        Sensor rightTouch = Sensor.S3;.  
        int startTime = (int)System.currentTimeMillis();.  
        do {  
            boolean left = leftTouch.readBooleanValue();.  
            boolean right = rightTouch.readBooleanValue();.  
            if (left || right) {  
                robbby.reverseDirection();.  
                robbby.pause(500);.  
                if (left).  
                    robbby.go_rightAngleRight();.  
                else if (right).  
                    robbby.go_rightAngleLeft();.  
                robbby.go_forward();.  
            }.  
            LCD.showNumber(((int)System.currentTimeMillis()-startTime)/1000);  
        } while ((int)System.currentTimeMillis()-startTime < 20000);.  
        robbby.stop();.  
    }.  
}
```

## Struktogrammdarstellung:



Roboter weicht Hindernissen aus, wobei die Richtung davon abhängt, welcher Tastsensor berührt hat.

# Logische Operatoren

## ◆ binäre Operatoren

- Vergleichsoperatoren Gleichheit (==) und Ungleichheit (!=)
- logisches UND (& oder verkürzte Auswertung &&)
- logisches ODER (| oder verkürzte Auswertung ||)
- logisches XOR (^)

## ◆ unäre Operation

- boolesche Negation (!)

x	y	x&y	x y	x^y	!x
false	false	false	false	false	true
false	true	false	true	true	true
true	false	false	true	true	false
true	true	true	true	false	false

Kommutativgesetz:

$$x|y = y|x, x&y = y&x$$

Assoziativgesetz:

$$(x&y)&z = x&(y&z), (x&y)|z = x&(y|z)$$

Distributivgesetz:

$$(x&y)|z = (x|z)&(y|z), (x|y)&z = (x&z)|(y&z)$$

de Morgansche Gesetze:

$$!(x|y) = !x & !y, !(x&y) = !x | !y$$

# Verwendung des Lichtsensor - Folgen einer Linie

## LineFollowingTest.java:

```
import josx.platform.rcx.*;.
class LineFollowingTest {
    private static TurnBot robbby;.
    public static void main(String args[]) {
        robbby = new TurnBot();.
        int threshold; // threshold of light level.
        Sensor lightSensor = Sensor.S2;.
        lightSensor.activate();.
        robbby.pause(300);.
        // robot is assumed to start on black line.
        threshold = lightSensor.readValue() + 10;.
        robbby.go_forward();.
        while (true) {
            if (lightSensor.readValue() > threshold) {
                robbby.go_backward(); robbby.pause(150);.
                robbby.veer_left(); robbby.pause(120);.
                robbby.go_forward();.
            }.
        }.
    }.
}
```

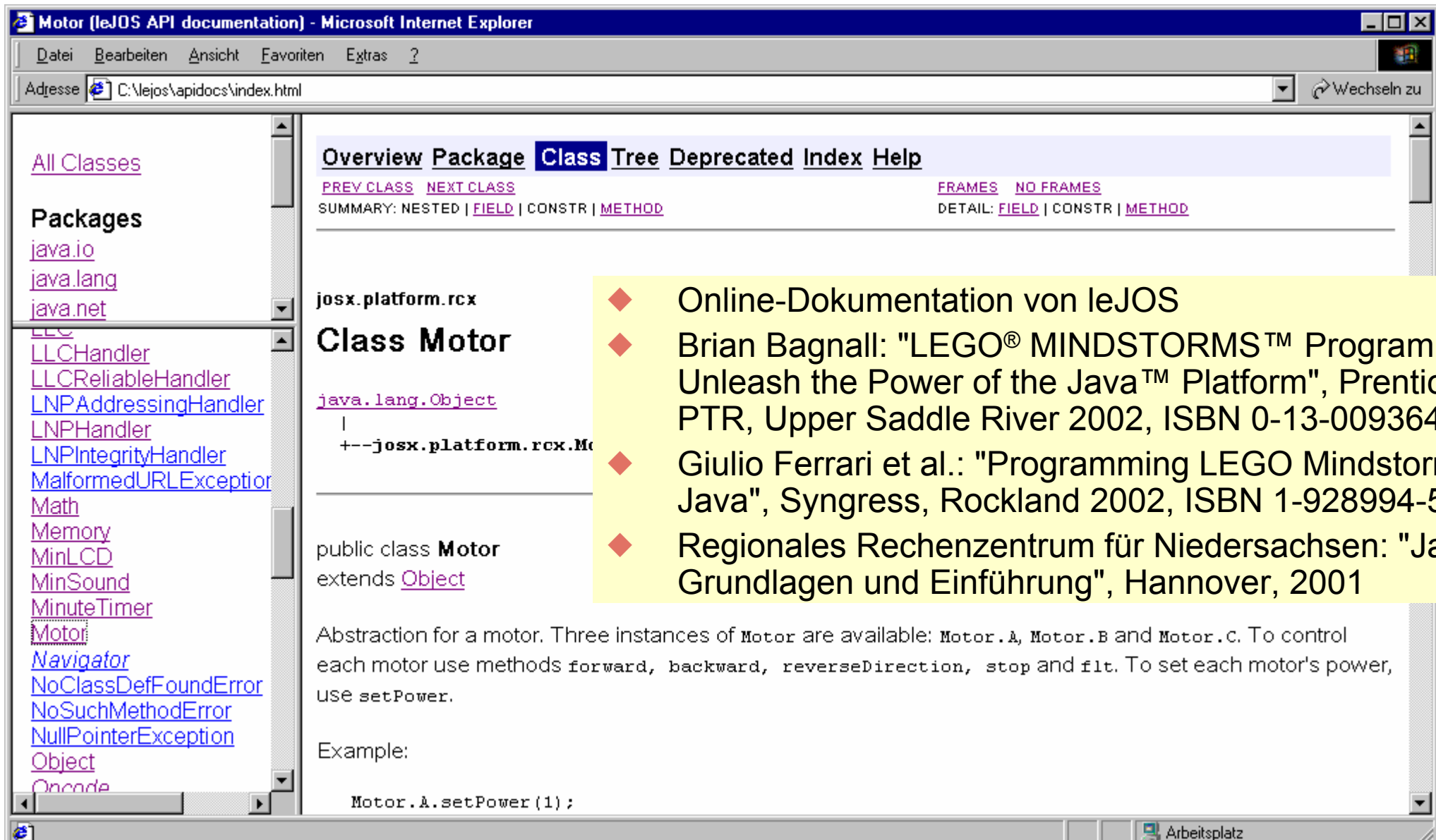
Der Lichtsensor muss mit `activate()` aktiviert werden. Im Fall des Lichtsensors kann auf den Aufruf von `setTypeAndMode()` verzichtet werden, da er voreingestellt ist. `readValue()` gibt dann einen Prozentwert zurück.

## veer\_left()-Methode in Robot:

```
public void veer_left() {
    leftWheel.backward();.
    rightWheel.forward();.
}
```

Roboter muss gegen den Uhrzeigersinn fahren und der Lichtsensor auf die schwarze Linie gesetzt werden.

- ◆ Vorbereitungen (Roboter, leJOS, Java2) erledigt!
- ◆ Einfache Klassen mit Attributen und Methoden entworfen!
- ◆ Kontrollstrukturen (Folge, Verzweigungen, Schleifen) für Abläufe erlernt!
- ◆ Motoren und Sensoren in Betrieb genommen!
- ◆ Eine erste Platzrunde gefahren!
- ◆ Klassenhierarchie von leJOS kennenlernen.
- ◆ Schnittstellen implementieren.
- ◆ Programme "parallel" laufen lassen.
- ◆ Allerlei Nützliches zum Debuggen probieren (Sound, Display, Timer, Datalogging).



Motor (leJOS API documentation) - Microsoft Internet Explorer

Adresse <C:\lejos\apidocs\index.html> Wechseln zu

[All Classes](#)

**Packages**

- [java.io](#)
- [java.lang](#)
- [java.net](#)

**Classes**

- [LLCHandler](#)
- [LLCReliableHandler](#)
- [LNPAAddressingHandler](#)
- [LNPHandler](#)
- [LNPIegrityHandler](#)
- [MalformedURLException](#)
- [Math](#)
- [Memory](#)
- [MinLCD](#)
- [MinSound](#)
- [MinuteTimer](#)
- [Motor](#)
- [Navigator](#)
- [NoClassDefFoundError](#)
- [NoSuchMethodError](#)
- [NullPointerException](#)
- [Object](#)
- [Oncode](#)

**Overview Package Class Tree Deprecated Index Help**

[PREV CLASS](#) [NEXT CLASS](#) [FRAMES](#) [NO FRAMES](#)

SUMMARY: [NESTED](#) | [FIELD](#) | [CONSTR](#) | [METHOD](#) [DETAIL: FIELD](#) | [CONSTR](#) | [METHOD](#)

**josx.platform.rcx**

## Class Motor

[java.lang.Object](#)

|--josx.platform.rcx.Mo

public class **Motor**  
extends [Object](#)

Abstraction for a motor. Three instances of `Motor` are available: `Motor.A`, `Motor.B` and `Motor.C`. To control each motor use methods `forward`, `backward`, `reverseDirection`, `stop` and `flt`. To set each motor's power, use `setPower`.

Example:

```
Motor.A.setPower(1);
```

- ◆ Online-Dokumentation von leJOS
- ◆ Brian Bagnall: "LEGO® MINDSTORMS™ Programming – Unleash the Power of the Java™ Platform", Prentice Hall PTR, Upper Saddle River 2002, ISBN 0-13-009364-5
- ◆ Giulio Ferrari et al.: "Programming LEGO Mindstorms with Java", Syngress, Rockland 2002, ISBN 1-928994-55-5
- ◆ Regionales Rechenzentrum für Niedersachsen: "Java 2 – Grundlagen und Einführung", Hannover, 2001