

Einführung in die Informatik

LeJOS

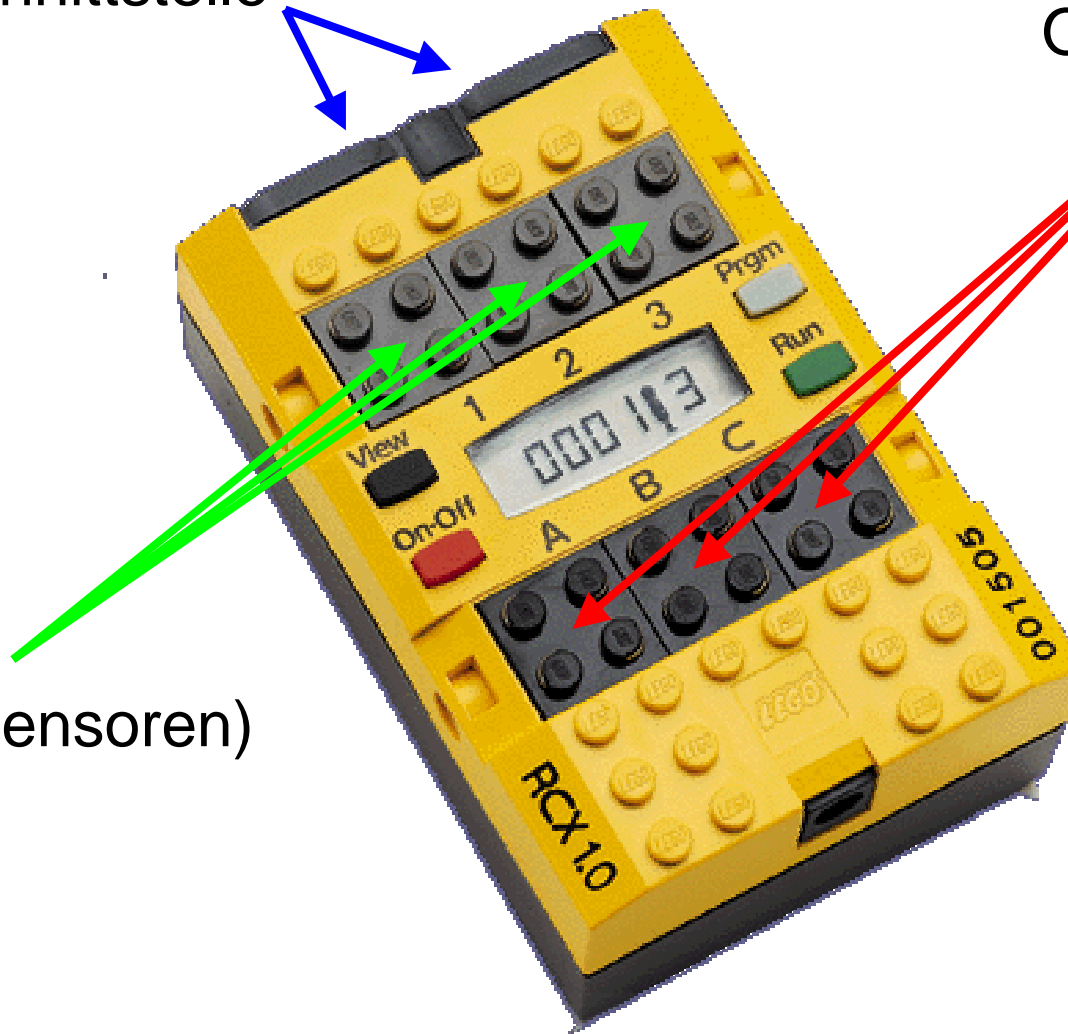
Das Java Operating System für Lego Mindstorms

Wolfram Burgard

Die RCX-Einheit

Infrarot-Schnittstelle

Outputs (Motoren)

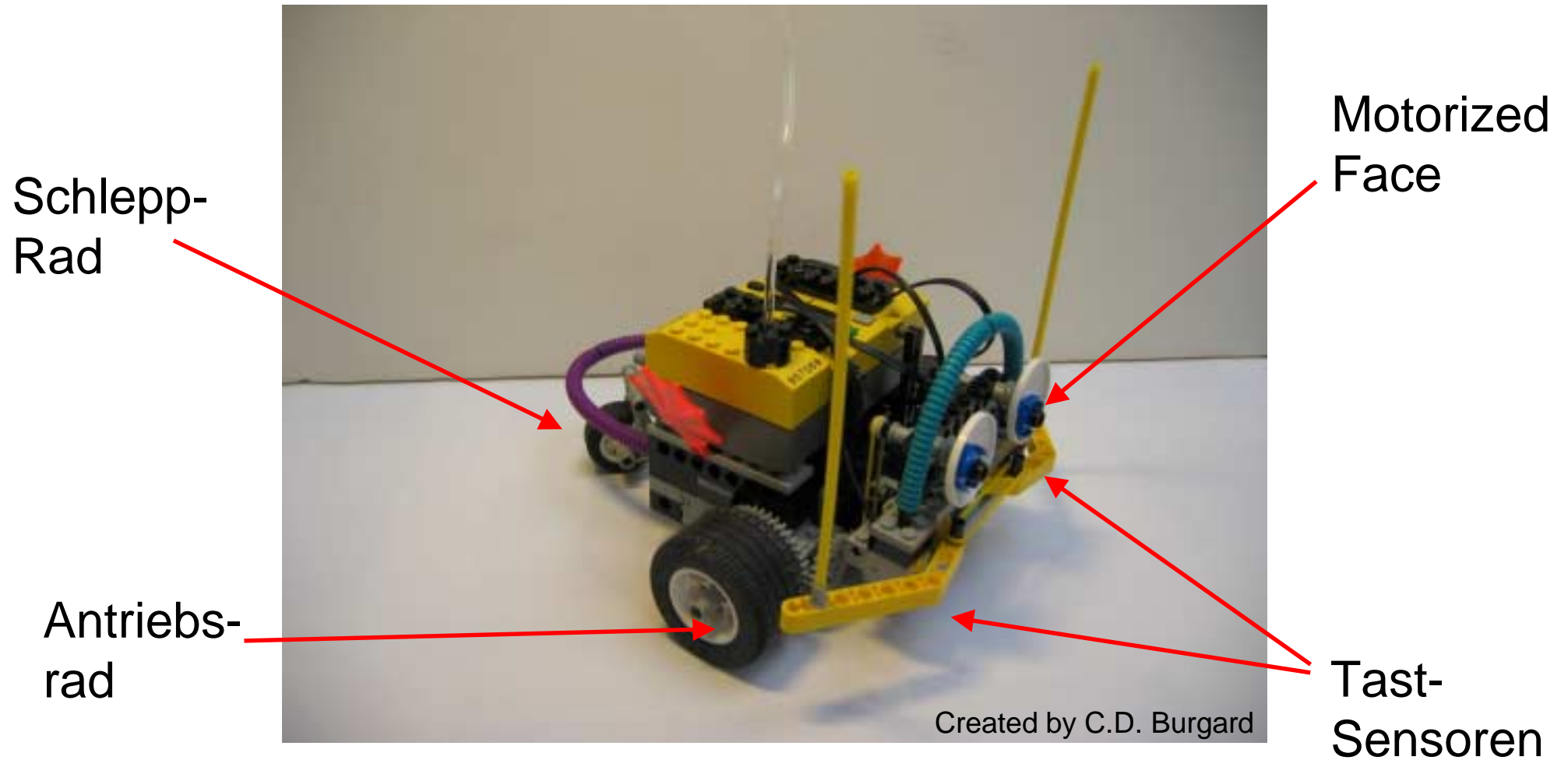


Inputs (Sensoren)

RCX mit Sensoren und Motoren

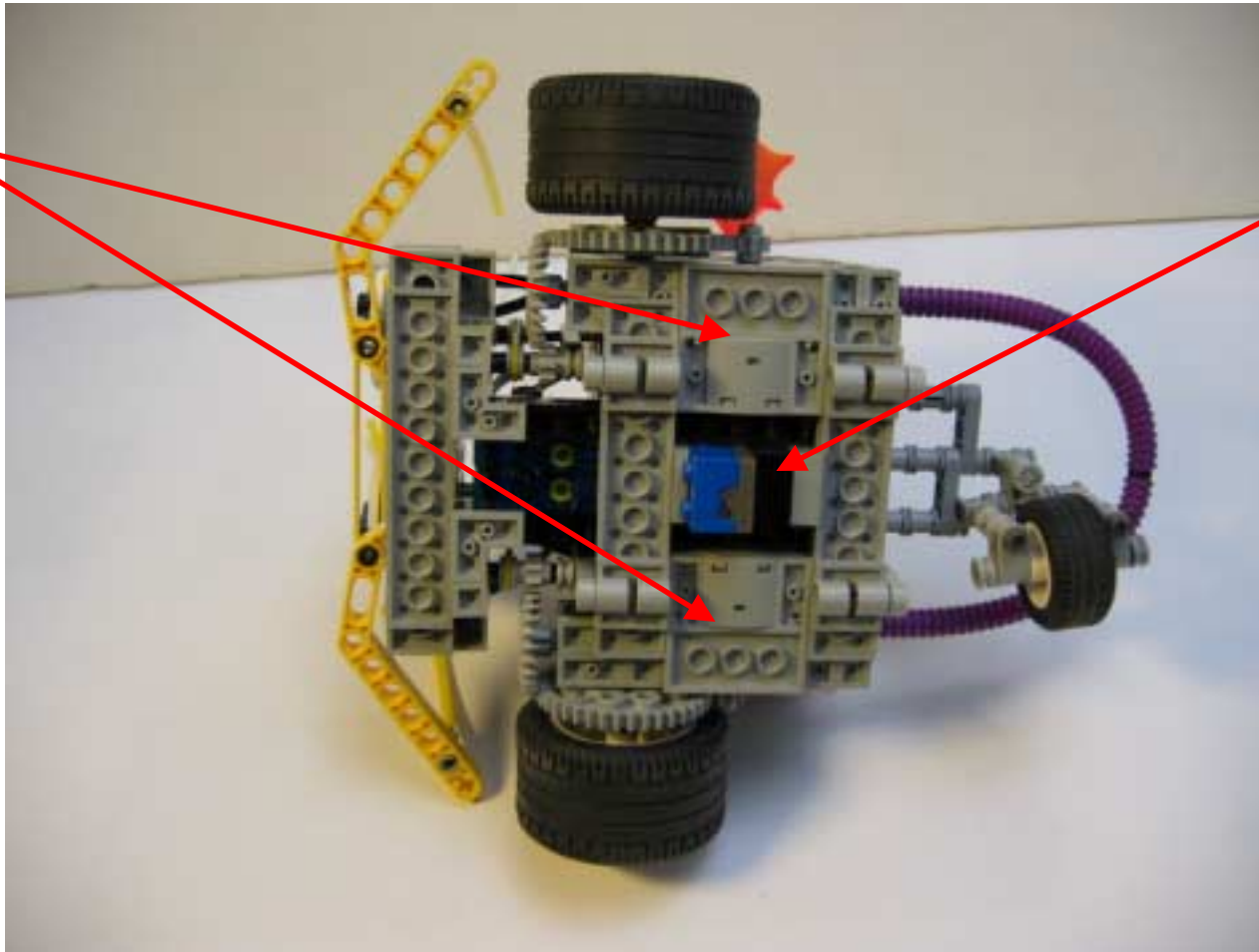


Ein Beispielroboter



Ansicht von unten

Motoren



Licht-
sensor

Quellen für LeJOS

- Sourceforge Project Homepage (LeJOS-Quellen):
<http://lejos.sourceforge.net/>
- Dort sind auch die API-Dokumentation einsehbar (und auch download-bar)
- Tutorial von R. Schiedermeier, FH München, FB 07.
- Web-Page zur Vorlesung.

Eigenschaften von LeJOS

- Objektorientierte Sprache (Java)
- Preemptive threads (tasks)
- Arrays, auch multi-dimensionale
- Rekursion
- Dokumentierte API.
- Fließkommaoperationen (Doubles sind auf 32 Bit eingeschränkt).
- String-Konstanten.
- `java.lang.Math`-Klasse, die `sin`, `cos`, `tan`, `atan`, `pow`, usw. zur Verfügung stellt.
- Casting von longs to ints und umgekehrt.

Darüber hinaus:

- Synchronisierung
- Ausnahmen (Exceptions)
- Eine Windows version, die ohne CygWin läuft.
- Referenzen werden verwaltet, was eine Implementierung einer Garbage Collection ermöglicht.
- Multi-program downloading.

Installation

- Um Programme ausführen zu können, benötigt die RCX-Einheit eine so genannte **Firmware**.
- Diese Firmware **entspricht dem Betriebssystem eines Rechners**.
- Um auf dem **Mindstorms-System Java-Programme** ausführen zu können, **muss** eine **andere Firmware installiert werden**.
- Diese **Firmware für das LeJOS** entspricht im Prinzip einer **Java Virtual Machine für die RCX**.

Hinweise:

1. Auf einer neuen RCX ist keine Firmware installiert.
2. Wenn der On-Button gedrückt wurde, während die Batterien entnommen sind, geht die Firmware verloren.

Installation der LeJOS Firmware unter Linux

- Die Umgebungsvariable `$LEJOS_HOME` muss das Verzeichnis der LeJOS-Installation enthalten
- Das Verzeichnis `$LEJOS_HOME/bin` soll im Pfad enthalten sein.
- Die Variable `RCXTTY` soll die Device enthalten, über die der Infrarot-Tower angesprochen werden kann. Bei einem USB-Tower ist dies eine usb Device und bei einem seriellen Tower üblicherweise `/dev/ttyS1`.

Installation der LeJOS Firmware unter SuSe 10.2

Archiv in ~/lejos auspacken

In .cshrc eintragen:

```
setenv LEJOS_HOME ~/lejos
setenv CLASSPATH .:$LEJOS_HOME/lib/classes.jar:$LEJOS_HOME/lib/pcrcxcomm.jar:$LEJOS_HOME/bin
setenv RCXTTY /dev/legousbtower0
setenv LD_LIBRARY_PATH ${LD_LIBRARY_PATH}:${LEJOS_HOME}/bin
```

Hochladen der Firmware:

```
firmdl
```

Testprogramm starten:

```
cd check
sh checkDistribution.sh
```

Compilieren und hochladen:

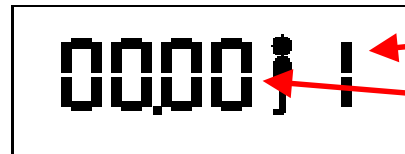
```
lejosjc Test.java
lejos Test
```

Die Firmware und das Display der RCX

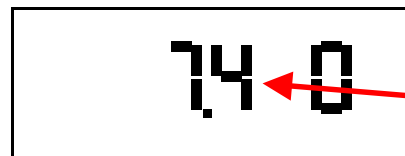
- Ob eine Firmware installiert ist, lässt sich leicht an dem Display der RCX ablesen:



Keine Firmware installiert



Lego Mindstorms Firmware installiert



LeJOS Firmware installiert



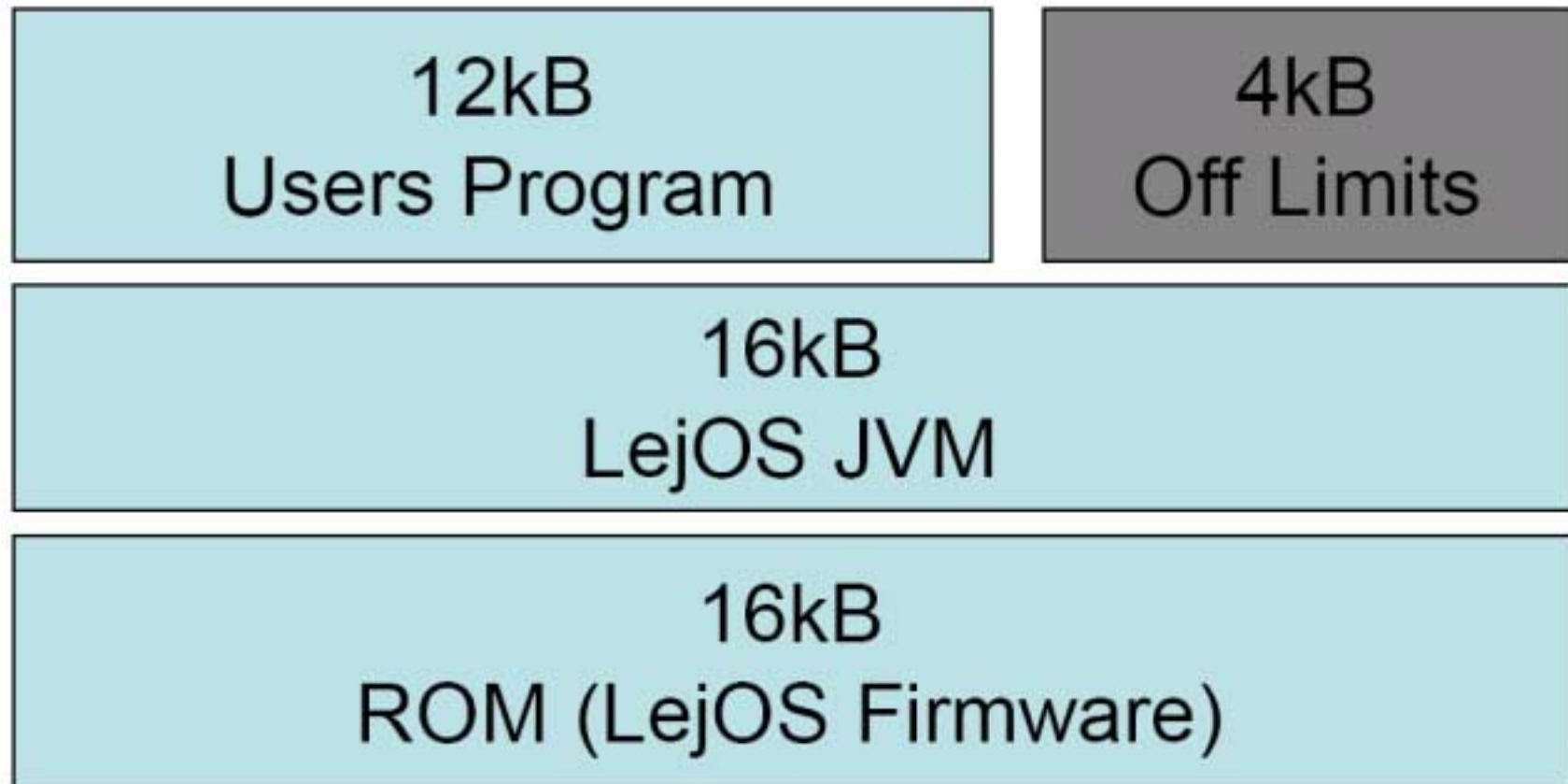
LeJOS und Programm installiert

Programmplatz

Uptime

Spannung

Speicherstruktur der RCX



Jumping into LeJOS: Hello World

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

class HelloWorld {
    public static void main(String[] args) throws
        InterruptedException {
        TextLCD.print("HELLO");
        Button.RUN.waitForPressAndRelease();
        TextLCD.print("WORLD");
        Button.RUN.waitForPressAndRelease();
    }
}
```

Nach dem Kompilieren und Transferieren dieses Programms erscheint auf dem Display nach dem 1. Drücken der Run-Taste die RCX-Darstellung des Strings „HELLO“ und nach dem nächsten Drücken die RCX-Darstellung des Strings „WORLD“.

Übersetzen für die und Ausführen von Java-Programmen auf der RCX

- Die **Übersetzung** von Java-Programmen geschieht unter Linux mit dem Befehl `lejosc` (analog zu `javac`).
- Sollte das **Programm fehlerfrei übersetzt** worden sein, muss es im nächsten Schritt **auf die RCX übertragen** werden.
- Dies geschieht mit dem Befehl `lejos`.
- In unserem HelloWorld-Beispiel lauten die entsprechenden Befehle

```
lejosjc HelloWorld.java  
lejos HelloWorld
```

- War die **Übertragung erfolgreich**, kann das Programm direkt durch **Drücken des Run-Buttons ausgeführt** werden.

Motor-Befehle

- Um die **Motoren zu betreiben**, stellt die **Klasse `Motor` drei Instanzen** zur Verfügung, die den einzelnen **Motoren entsprechen**.
- Dies sind die Instanzen `Motor.A`, `Motor.B` sowie `Motor.C`.
- Jedem dieser Objekte können wir verschiedene Nachrichten schicken (Auswahl):

<code>void forward()</code>	Motor läuft vorwärts
<code>void backward()</code>	Motor läuft rückwärts
<code>void reverse()</code>	dreht die Richtung um
<code>void stop()</code>	Motor stoppt abrupt und lässt sich auch nicht mehr leicht bewegen.
<code>setPower(int aPower)</code>	Setzt die Motor-Geschwindigkeit. Es können die Werte 0 bis 7 gesetzt werden.
<code>void flt()</code>	Der Motor wird nicht mehr angetrieben und befindet sich im Freilauf.
<code>char getId()</code>	Liefert die Nummer des Motors ('A', 'B', oder 'C')
<code>boolean isMoving()</code>	true, wenn der Motor sich bewegt.

Ein Beispiel

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

class TestMotor {
    public static void main(String[] args) throws InterruptedException {
        Motor.A.forward();
        Thread.sleep(1000);
        Motor.A.stop();
        Thread.sleep(1000);
        Motor.A.backward();
        Thread.sleep(1000);
        Motor.A.flt();
    }
}
```

Motor A läuft eine Sekunde vorwärts, stoppt für eine Sekunde, läuft dann eine Sekunde rückwärts und geht dann in den Freilauf.

Einmal im Quadrat fahren

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

class GoSquare{
    public static void main(String[] args) throws
        InterruptedException{

        int forwardTime=3000;    // length of square
        int rotationTime=720;    // time to rotate 90 degrees
        Motor.A.setPower(7);
        Motor.B.setPower(7);
        Motor.A.forward();
        for (int i=0; i < 4; i++){
            Motor.C.forward();
            Thread.sleep(forwardTime);
            Motor.C.reverse();
            Thread.sleep(rotationTime);
        }
        Motor.A.stop();
        Motor.C.stop();
    }
}
```

Und jetzt einmal im Kreis ...

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

class GoCircle{
    public static void main(String[] args) throws InterruptedException{
        int forwardTime=3000;
        Motor.A.setPower(7);
        Motor.C.setPower(1);
        Motor.A.forward();
        Motor.C.forward();
        Thread.sleep(7000);
        Motor.A.stop();
        Motor.C.stop();
    }
}
```

Gleiches Konzept: sehr langsam fahren ...

```
import josx.platform.rcx.*;
class GoSlow {
    public static void main(String[] args) throws
        InterruptedException {
        while(true) {
            Motor.A.forward();
            Motor.C.forward();
            Thread.sleep(1);
            Motor.A.stop();
            Motor.C.stop();
            Thread.sleep(1);
        }
    }
}
```

Engere Kreise ...

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

class GoCircle2{
    public static void main(String[] args) throws InterruptedException{
        Motor.A.setPower(7);
        Motor.C.setPower(1);
        Motor.A.forward();
        for (int i = 0; i < 60; i++){
            Motor.C.forward();
            Thread.sleep(70);
            Motor.C.flt();
            Thread.sleep(150);
        }
        Motor.A.stop();
        Motor.C.stop();
    }
}
```

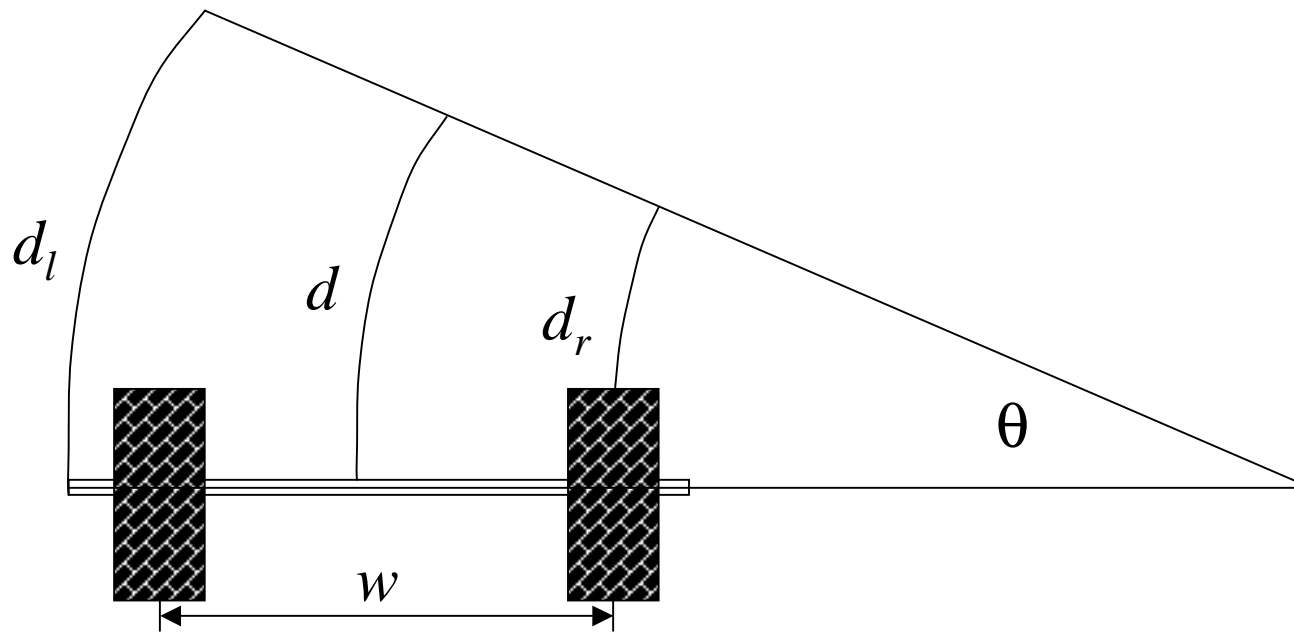
Dead Reckoning

- Dead Reckoning (abgeleitet von deduced reckoning) ist ein aus alten Segelzeiten abgeleiteter Begriff für die Bestimmung der aktuellen Position eines Vehikels.
- Die Grundidee dieses Verfahrens ist das Zählen der Radumdrehungen, um die Bewegungen des Fahrzeugs nachzuvollziehen und so ermitteln zu können, wo sich das System derzeit befindet.
- Leider sind die üblicherweise benötigten Parameter (Raddurchmesser, Abstand der Räder, verstrichene Zeit etc.) nie exakt bekannt.
- Daher ist Dead Reckoning eine sehr gute Hilfe, die aber niemals hinreichend sein kann.

Differentialantriebe

- Differentialantriebe stellen eine der häufigsten Antriebsarten im Bereich mobiler Roboter dar.
- Solche System zeichnen sich dadurch aus, dass zwei unabhängig antreibbare Räder auf jeder Seite der Plattform angebracht werden.
- Die Plattform wird zusätzlich durch ein oder mehrere Schwenkräder abgestützt.

Dead Reckoning mit für Roboter mit Differentialantrieb

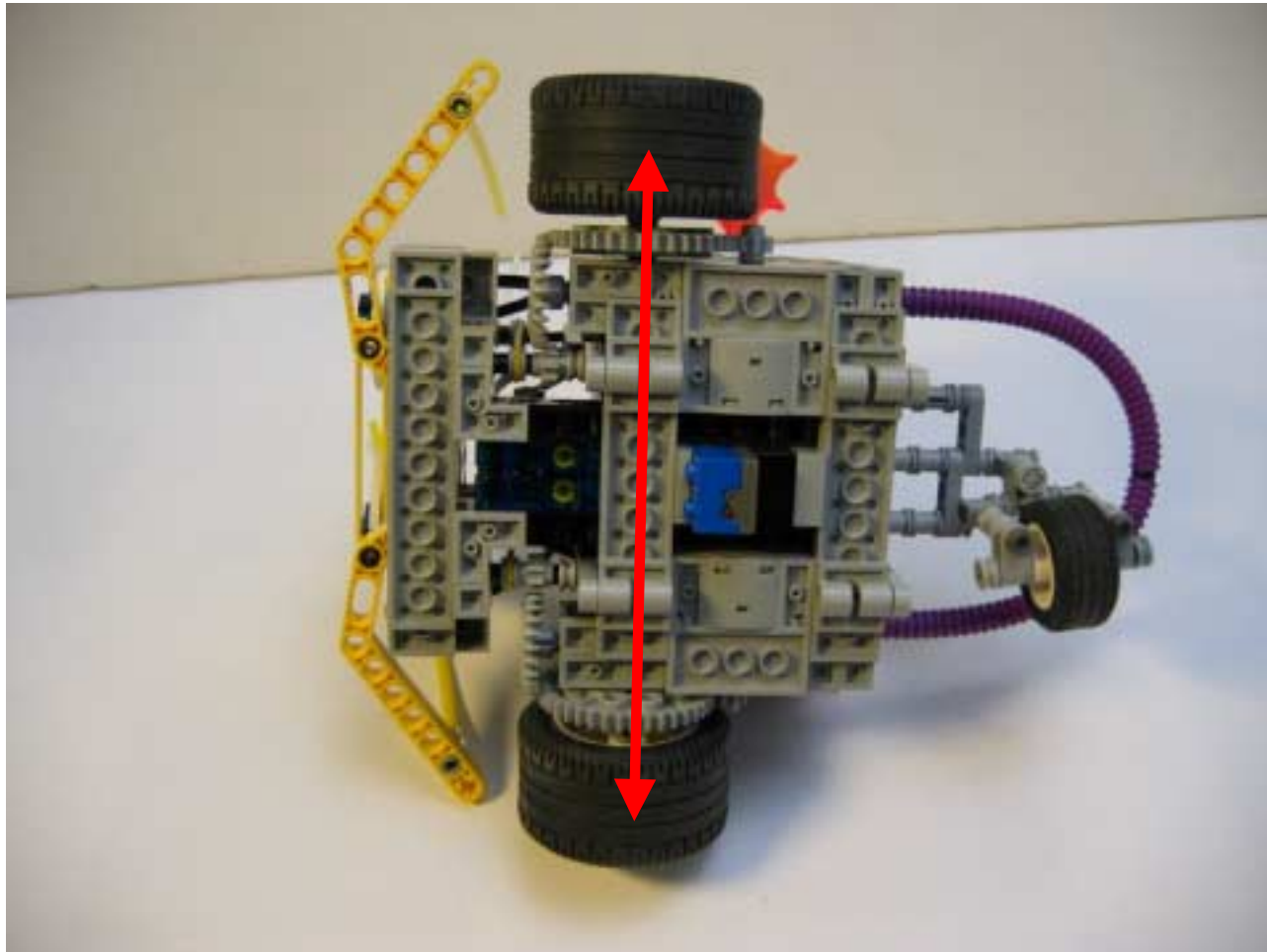


$$v = \frac{v_l + v_r}{2}$$

$$d = \frac{d_l + d_r}{2}$$

$$\theta = \frac{d_l - d_r}{w}$$

Beispiel: unser Roboter



Die Klasse TimingNavigator

- Die Klasse `TimingNavigator` stellt eine komfortable Möglichkeit dar, zu **navigieren** und gleichzeitig ein **Dead Reckoning** zu erhalten.
- Sie stellt **grundlegende Navigationsprimitive** zur Verfügung.
- Gleichzeitig versucht die **Position des Roboters zu berechnen**.
- Dies geschieht auf der Basis der **verwendeten Geschwindigkeiten** und der **verstrichenen Zeit**.
- Daher müssen **alle Navigationsbefehle** (die wir bisher direkt an den Motor gesendet haben) jetzt **an ein Objekt der Klasse `TimingNavigator` geschickt werden**.
- Hinweis: Diese Klasse arbeitet nur korrekt für Roboter, die ein Differentialantrieb besitzen und sich auf der Stelle drehen können.

Methoden der Klasse TimingNavigator (1)

- Um den **Roboter zu bewegen**, stellt die **Klasse** `TimingNavigator` **verschiedene Methoden** zur Verfügung.
- Die **Position** des Roboters wird immer dann **aktualisiert, nachdem der Roboter gestoppt wurde**.

<code>public TimingNavigator(Motor right, Motor left, float timeOneMeter, float timeRotate)</code>	Konstruktor
<code>void forward()</code>	Roboter fährt vorwärts
<code>void backward()</code>	Roboter fährt rückwärts
<code>void stop()</code>	Roboter stoppt, neue Koordinaten werden berechnet
<code>void rotate(float angle)</code>	Rotiert den Roboter um den gegebenen Winkel. Methode endet mit dem Ende der Rotation.
<code>void gotoAngle(float angle)</code>	Rotiert den Roboter, so dass er anschließend in die angegebene Richtung ausgerichtet ist.
<code>void travel(int distance)</code>	Der Roboter wird um die angegebene Distanz vorwärts gefahren

Methoden der Klasse TimingNavigator (2)

<code>public gotoPoint(float x, float y)</code>	Der Roboter fährt zur angegebenen Position
<code>float getX()</code>	Liefert die x-Position des Roboters
<code>float getY()</code>	Liefert die y-Position des Roboters
<code>float getAngle()</code>	Liefert die Orientierung des Roboters
<code>float setMomentumDelay(short delay)</code>	Spezifiziert den zusätzlichen Aufwand für die Überwindung der Trägheit des Roboters bei Rotationen

- Das Argument `timeRotate` des Konstruktors spezifiziert die Zeit, die der Roboter für eine **Drehung um 360 Grad** benötigt.
- Der Wert `timeOneMeter` gibt die Zeit in Sekunden an, die Rotate **für das Zurücklegen von 1m benötigt**.
- Das Argument `delay` von `setMomentumDelay` spezifiziert die **Zeit, die zusätzlich benötigt wird, wenn der Roboter aus dem Stand um 360 Grad** rotiert, verglichen mit einer Rotation mit „fliegendem Start“.

Anwendung der Klasse TimingNavigator

Einmal im Quadrat fahren:

```
class Navigate {
    public static void main(String [] args) throws Exception {
        TimingNavigator n =
            new TimingNavigator(Motor.A, Motor.C, 7.32, 2.92);
        n.gotoPoint(100, 0);
        n.gotoPoint(100, 100);
        n.gotoPoint(0, 100);
        n.gotoPoint(0, 0);
        n.rotate(90);
    }
}
```

Töne abspielen

- Die RCX-Einheit kann ebenfalls **Töne generieren**.
- Benutzer können entweder **vordefinierte Sounds** oder auch **einzelne Töne abspielen**.
- Für das Abspielen von Tönen gibt es in LeJOS die Klasse **Sound**.

<code>void play(int aDuration, int aFrequency)</code>	Spielt dein Ton der entsprechenden Frequenz für die angegebenen Länge ab.
<code>void beep()</code>	Ein Beep
<code>void twoBeeps()</code>	Zwei Beeps
<code>void buzz()</code>	Tiefer Beep
<code>void beepSequence()</code>	Vorgegebene Tonsequenz

Ein Beispiel

```
import javax.platform.rcx.*;

class Duck
{
    public static void main(String[] args) throws InterruptedException {
        while(true){
            for (int i=0; i<duck.length; i++){
                Sound.playTone(duck[i], duration[i]);
                Thread.sleep(500+5*duration[i]);
            }
            Thread.sleep(1000);
        }
    }
    static int duck[] = {262, 294, 330, 349, 392, 392, 440, 440, 440, 440, 392, 440, 440,
                        440, 440, 392, 349, 349, 349, 349, 330, 330, 294, 294, 294, 294, 262};
    static int duration[] = {30, 30, 30, 30, 90, 90, 30, 30, 30, 30, 90, 30, 30, 30,
                            90, 30, 30, 30, 30, 90, 90, 30, 30, 30, 30, 90};
}
```

Ausgabe auf das LCD

- Zur Ausgabe von Texten und Zahlen auf dem Display der RCX gibt es verschiedene Methoden.
- `LCD.clear()` löscht das Display
- `LCD.showNumber(int aValue)` zeigt eine Zahl ohne Vorzeichen an.
- `LCD.showProgramNumber(int aValue)` zeigt eine Ziffer auf dem Programmplatz an.
- Die Methode `TextLCD.print(string aString)` zeigt die ersten fünf Zeichen des `String`-Objektes `aString` an.

Sensoren

- Um die an die RCX anschließbaren Sensoren auszulesen, stellt LeJOS die Klasse `Sensor` zur Verfügung.
- Es gibt 3 Anschlüsse für Sensoren (1, 2, 3, im Gegensatz zu A, B, C bei den Motoren).
- Für jeden Sensoranschluss stellt die Klasse `Sensor` ein `Sensor`-Objekt bereit.
- Diese heißen `Sensor.S1`, `Sensor.S2` bzw. `Sensor.S3`
- Von Lego werden für die RCX Tast-, Licht- und Rotationssensoren angeboten.
- Die Sensoren liefern einen Wert aus dem Bereich 0-1023. Dies ist der Rohwert der Sensoren.

Methoden der Klasse Sensor

<code>void activate()</code>	Aktiviert einen Sensor (u.a. wichtig beim Lichtsensor).
<code>void passivate()</code>	Schaltet den Sensor ab
<code>int readRawValue()</code>	Liest den aktuellen Wert aus
<code>void setTypeAndMode(int aType, int aMode)</code>	Legt den Rückgabewert eines Sensors fest.
<code>boolean readBooleanValue()</code>	Liefert den boolean Wert des Sensors.

- Die möglichen Parameter der Methode `setTypeAndMode` sind in der Klasse `SensorConstants` festgelegt.
- Beispielsweise initialisiert man einen Tastsensor mit der Anweisung

```
Sensor.S1.setTypeAndMode(SensorConstants.SENSOR_TYPE_TOUCH,  
                          SensorConstants.SENSOR_MODE_BOOL);
```

Ein einfaches Beispiel

```
import josx.platform.rcx.*;
class ShowTouch {
    public static void main(String[] args) {
        while(true)
            LCD.showNumber(Sensor.S1.readValue());
    }
}
```

Treads: Parallele Ausführung von Prozessen

- **Normalerweise**, wenn wir programmieren, **denken wir rein sequenziell**.
- Beispielsweise lesen wir häufig erst die Daten erst ein. Dann prozessieren wir sie und anschließend geben wir das Ergebnis aus.
- Selbst wenn **heutige Comuter in der Lage** sind, **mehrere Programme gleichzeitig auszuführen**, haben wir die grundsätzliche **Möglichkeit der Parallelisierung innerhalb von Programmen bisher keine Rücksicht genommen**.
- Wir haben also **sequentielle Programme** geschrieben und der **Computer hat die parallele Ausführung dieser Programme realisiert** (impliziter Parallelismus)?
- **Threads** stellen nun eine **Möglichkeit** dar, **parrallele Lösungen für ein gegebenes Problem zu programmieren**.
- Allerdings müssen wir dann die **Parallelisierung explizit** machen.

Warum Threads?

- **Insbesondere interaktive Programme** sollen die Möglichkeit bieten, **Eingaben vorzunehmen, während Berechnungen durchgeführt werden.**
- Beispielsweise können **Web-Browser in einem Fenster eine Web-Seite laden**, während wir **in einem anderen Fenster ein Formular ausfüllen.**
- Auch können wir beispielsweise ein **Menü öffnen, während eine Seite geladen wird.**
- **Textverarbeitungssysteme führen Rechtschreibprüfungen** im Hintergrund durch, **während wir Tastatureingaben vornehmen.**
- Unser **Roboter** soll beispielsweise **Hindernisse umfahren** können und **gleichzeitig nach der schwarzen Linie suchen.**
- Insbesondere **in Fällen, in denen ein Programm mehrere Aufgaben gleichzeitig erfüllen soll**, ist es **wichtig, dass** eine **parallele Ausführung unterschiedlicher Programmteile möglich ist.**
- **Threads** stellen nun eine sehr **elegante Möglichkeit** dar, **mehrere parallele Prozesse innerhalb eines Programms** zu definieren.

Geht es nicht auch ohne Threads?

- **Im Prinzip kommt man auch vollkommen ohne Threads aus.**
- Beispielsweise kann man die **parallele Ausführung von Prozessen** auch **selbst simulieren**.
- Betrachten wir beispielsweise das folgende Programmstück zum „parallelen“ Einlesen mehrerer Web-Seiten.

```
for (int i=0; i<numberOfPages; i++)  
    if ((line=webPage[i].readNextLine()) != null)  
        page[i].addLine(line);
```

- Wir müssen die **tatsächliche Anzahl aller Web-Seiten vor dem Beginn der Schleife kennen**. **Späteres Hinzufügen oder Entfernen ist nicht möglich**.
- Der **Prozess beginnt für alle Seiten zum gleichen Zeitpunkt**.
- Die **langsamste Verbindung bestimmt das Tempo**.
- **Bei jeder Operation** müssen wir den **Parallelismus explizit machen**.

Threads in Java

- Java stellt eine **Klasse** `Thread` zur Verfügung, die **Threads modelliert**.
- Diese bietet uns die Möglichkeit, **Anwendungen zu realisieren**, in denen **mehrere Prozesse gleichzeitig ablaufen**.
- Mit Hilfe dieser Klasse erzeugen wir **Thread-Objekte**, zu denen **Programm-Code** gehört, den wir **gleichzeitig mit dem Code anderer Threads ausführen lassen**.
- Beispielsweise können wir Threads erzeugen, die Musik abspielen, den Batteriestatus beobachten, auf Hindernisse reagieren, einer Linie folgen etc.
- **Thread-Objekte führen ihren Code simultan aus**.
- Wie diese **simultane Ausführung der unterschiedlichen Threads** stattfindet, ist **nicht festgelegt**.

Gleichzeitigkeit versus Parallelität

- Die **meisten der heutigen Computer besitzen lediglich einen Prozessor**.
- Deswegen können sie **zu jedem Zeitpunkt auch nur eine Anweisung ausführen**.
- Eine **echte parallele Ausführung von Anweisungen verschiedener Threads ist auf solchen Rechnern daher nicht möglich**.
- In der **Java Virtual Machine** wird die **gleichzeitige Ausführung von Threads** jedoch **simuliert**.
- Dies geschieht dadurch dass der **Interpreter zyklisch die einzelnen Threads durchgeht** und dabei **jeweils ein Stück des entsprechenden Codes abarbeitet**.
- Daher sprechen wir von **simultaner/gleichzeitiger Ausführung anstelle von paralleler Ausführung**.
- Für den Anwender ist der Unterschied allerdings nicht merkbar.
- **Auf Mehrprozessoren, kann der Laufzeitgewinn allerdings deutlich spürbar sein**.

Die Thread-Klasse

- Die Klasse `Thread` stellt eine **Methode** `start()` zur Verfügung, welche die **Ausführung des entsprechenden Threads initiiert**.
- **Bevor diese Methode aufgerufen wird, gibt es zwar ein `Thread`-Objekt, der mit ihm assoziierte Code wird allerdings nicht ausgeführt.**
- Ein einfaches Beispiel für die Ausführung von Threads ist das folgende Programm:

```
class DoNothingThreadExample {
    public static void main(String[] a) {
        Thread t1 = new Thread();
        Thread t2 = new Thread();
        t1.start();
        t2.start();
    }
}
```


Verwendung der Thread-Klasse

- Wenn wir das **oben gezeigte Programm ausführen, passiert nichts.**
- Das Programm hält unmittelbar an.
- Grund hierfür ist, dass der **Code, den ein Thread-Objekt beinhaltet per Default leer** ist. Es sind also keine Anweisungen in einem Thread-Objekt enthalten.
- Um ein Thread-Objekt zu erzeugen, welches ein simultan auszuführendes Programmstück beinhaltet, müssen wir eine **Klasse definieren, welche die Thread-Klasse erweitert.**
- Diese Klasse muss die **Methode run() implementieren**, die das **simultan auszuführende Programmstück enthält.**
- Ein **typisches Muster einer solchen Klasse** ist:

```
class DoSomethingThreadExample extends Thread{
    ...                               // Code, e.g., constructor
    public void run() {
        ...                             // Source code of the thread
    }
    ...                               // Code, e.g., instance variables
}
```

Ein Beispiel: Threads die zählen

- Die folgende Klasse erlaubt das Erzeugen von Threads, die eine Variable inkrementieren.
- Der Wert, um den jeweils erhöht wird, wird dem Konstruktor übergeben.

```
class CountingThread extends Thread {
    public CountingThread(int x) {
        this.x = x;
    }
    public void run() {
        int i=0;
        while (i<15){
            System.out.println(i);
            i+=x;
        }
    }
    private int x;
}
```

Anwendung dieses Threads

- Das folgende Programm erzeugt nun zwei solcher Threads:

```
public class CountingThreadExample {  
    public static void main(String a[]) {  
        CountingThread t1 = new CountingThread(2);  
        CountingThread t2 = new CountingThread(3);  
        t1.start();  
        t2.start();  
    }  
}
```

- Die Ausgabe dieses Programms ist üblicherweise:

```
0  
2  
4  
6  
8  
10  
12  
14  
0  
3  
6  
9  
12
```

Suspendieren von Threads

- Die **Ausführung dieser Threads erscheint nicht simultan stattzufinden**.
- Ein Grund hierfür ist, dass die Zeitscheibe, die die Java Virtual Machine den einzelnen Threads zuordnet so lang ist, dass die Threads innerhalb dieser Zeitspanne fertig werden können.
- **Grundsätzlich muss die Ausführung nicht verzahnt stattfinden**.
- Die Java Virtual Machine **wechselt jedoch üblicherweise zu einem anderen Thread, wenn das gerade aktuelle Thread wartet** (beispielsweise weil die Festplatte auf die entsprechende Position fahren muss etc.).
- In Java kann man das **Wechseln erzwingen**, indem man **Threads kurzzeitig „suspended“**.
- Dies geschieht mit der Anweisung `sleep(int milliseconds)`, welches ein **Pausieren** für die entsprechende Zeit bewirkt (kein Busy-Wait wie `for(;;)`)

```
try {  
    sleep(35);           //Suspend for 35 milliseconds  
} catch(Exception e) {}
```

Ausführung von Threads, die suspendiert werden

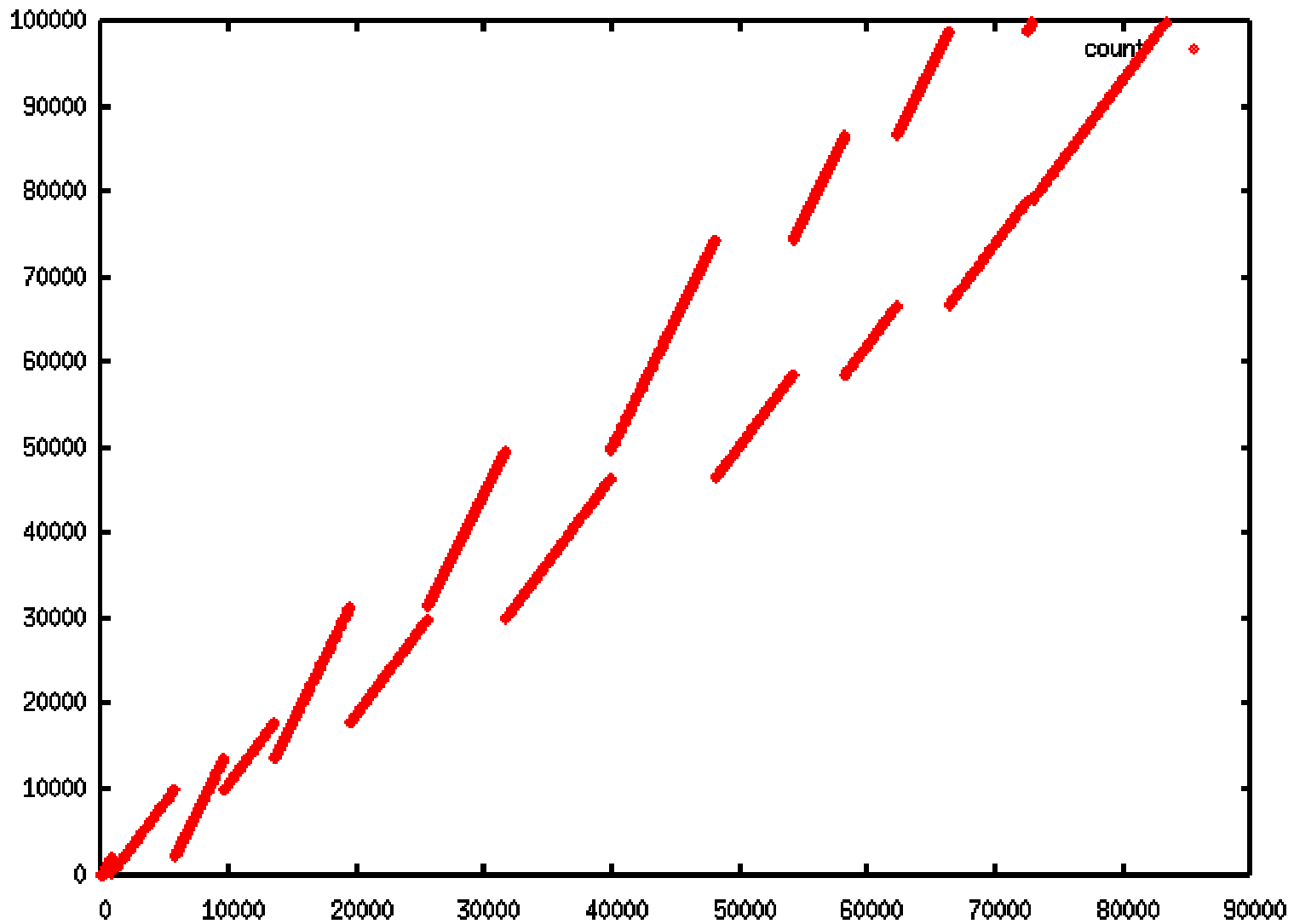
Variante:

```
class CountingThreadSleep extends Thread {
    public CountingThreadSleep(int x) {
        this.x = x;
    }
    public void run() {
        int i=0;
        while (i<15){
            System.out.println(i);
            i+=x;
            try {
                sleep(1);
            } catch(Exception e){}
        }
    }
    private int x;
}
```

Ausgabe:

```
0
0
2
3
4
6
6
9
8
12
10
12
14
```

Automatische Suspendierung ohne `sleep()`, wenn `i` bis 100000 läuft



Anwendung auf der RCX: Hintergrundmusik

```
import josx.platform.rcx.Sound;
class MusicPlayer extends Thread {
    public void run() {
        while(true){
            for(int i=0;i<note.length; i+=2) {
                int duration = note[i+1];
                Sound.playTone(note[i], duration);
                try {
                    Thread.sleep(duration*10);
                } catch (InterruptedException e) {};
            }
        }
    }
    private static int note[] = {
        165,24, 0,12, 175,12, 196,12, 0,12, 262,72, 0,12, 147,24, 0,12, 165,12,
        175,96, 0,24, 196,24, 0,12, 220,12, 247,12, 0,12, 349,72, 0,24, 220,24,
        0,12, 247,12, 262,48, 294,48, 330,48, 165,24, 0,12, 175,12, 196,12, 0,12,
        262,96, 0,24, 294,24, 0,12, 330,12, 349,96, 0,48, 196,24, 0,12, 196,12,
        330,36, 0,12, 294,24, 0,12, 196,12, 330,36, 0,12, 294,24, 0,12, 196,12,
        349,36, 0,12, 330,24, 0,12, 294,12, 262,96, };
    }
}
```

Anwendung von Threads: die Klasse `Arbitrator`

- Typischerweise soll der **Roboter mehrere Aufgaben gleichzeitig erfüllen können**.
- Beispielsweise soll er einerseits geradeaus fahren (explorieren), andererseits soll er bei Berührungen mit Hindernissen diesen ausweichen.
- Wie oben wollen wir die **einzelnen Verhalten** (geradaus fahren und Hindernissen ausweichen) **unabhängig voneinander spezifizieren**.
- Welcher **Code ausgewählt** wird, soll **vom aktuellen Zustand der Sensoren abhängen**.
- In LeJOS kann dies durch die **Klasse `Arbitrator` realisiert** werden.
- Die **Klasse `Arbitrator`** erlaubt auch die **Spezifikation vom Umgang mit Konflikten**, wenn beispielsweise **zwei Verhalten auf das gleiche Betriebsmittel (Motor) zugreifen**.

Die Klasse Arbitrator

- Die Klasse Arbitrator bietet lediglich zwei Methoden.
- Der Konstruktor `public Arbitrator(Behavior[] behaviors)` erwartet als Argument ein **Array von Behaviors**.
- Die Methode `start()` startet den Arbitrator.
- In dem folgenden Beispiel verwenden wir drei verschiedene Verhalten:
 - Das erste Verhalten fährt vorwärts. Es ist normalerweise aktiv.
 - Das zweite Verhalten versucht Hindernissen auszuweichen.
 - Das dritte Verhalten stoppt den Roboter und beendet das Programm, wenn man den RUN-Knopf drückt.

Ein Beispiel für die Anwendung von Arbitrator

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

class Behavior1 {

    public static void main(String [] args) {

        TimingNavigator navigation =
            new TimingNavigator(Motor.A, Motor.C,
                               timeForOneMeter, timeFor360Deg);

        Behavior b1 = new DriveForward(navigation);
        Behavior b2 = new HitWall(navigation, 20, 50);
        Behavior b3 = new QuitProgram(navigation);

        Behavior [] bArray = {b1, b2, b3};

        Arbitrator arby = new Arbitrator(bArray);
        arby.start();
    }

    static final float timeForOneMeter = 7.32f;
    static final float timeFor360Deg   = 2.92f;
}
```

Verhalten: Das Interface Behavior

- Um die **Verhalten zu implementieren, welche von einem Arbitrator-Objekt verwaltet werden**, verwendet man das **Interface Behavior**.
- Dieses Interface verlangt die **Implementierung von drei Methoden**:
 - `void action()`
Der Code in dieser Methode repräsentiert das eigentliche **Verhalten des Roboters**, wenn diese Behavior aktiviert wird.
 - `void suppress()`
Diese Methode beinhaltet die Anweisungen, die ausgeführt werden, wenn ein anderes Behavior aktiviert wird und dieses **Verhalten beendet werden soll**.
 - `boolean takeControl()`
Diese Methode **liefert den Wert true, wenn die Voraussetzungen für dieses Verhalten erfüllt** sind und es aktiviert werden kann.
- **Jede dieser Methoden wird als eigenes Thread realisiert.**
- Deshalb muss man bei Verwendung von Behaviors **darauf achten**, dass **bei der simultanen Ausführung dieser Methoden wichtige Inhalte gemeinsamer Variablen nicht überschrieben werden**.

Beispiel: Das Verhalten `HitWall`

- Der Roboter soll nach dem Zusammenstoß mit einem Hindernis zuerst eine gewisse Distanz rückwärts fahren und dann eine Rotation vom Hindernis weg ausführen.
- Aktiviert wird dieses Behavior, wenn einer der beiden Tastsensoren gedrückt wird.
- Wenn dieses Behavior deaktiviert wird, soll der Roboter anhalten.
- Um die Rotationsrichtung zu ermitteln, muss die Klasse `HitWall` den Zustand der Tastsensoren auswerten und in Instanzvariablen speichern.
- Weiter wollen wir ein `TimingNavigator`-Objekt verwenden, um die Bewegungen des Roboters zu steuern.
- Dadurch kann der Roboter später (nahe) zu seinem Ausgangspunkt zurückkehren.

Der naive Ansatz: HitWall, Version 1

```
class HitWallVer1 extends Behavior {
    public HitWallVer1(TimingNavigator navigation, int dist, float angle) {
        this.navigation = navigation;
        this.backupDistance = dist;
        this.rotationAngle = angle;
    }
    public boolean takeControl() {
        return (Sensor.S1.readBooleanValue() || Sensor.S3.readBooleanValue());
    }
    public void suppress() {
        this.navigation.stop();
    }
    public void action() {
        Sound.beepSequence();
        navigation.travel(-this.backupDistance);
        if (Sensor.S1.readBooleanValue())
            navigation.rotate(this.rotationAngle);
        else
            navigation.rotate(-this.rotationAngle);
    }
    protected int backupDistance;
    protected float rotationAngle;
    protected TimingNavigator navigation;
}
```

Probleme dieser Variante

- Nachdem der Roboter rückwärts gefahren ist, werden nochmals die Sensoren abgefragt, um festzustellen, in welcher Richtung das Hindernis umfahren werden soll.
- Die Sensoren sind aber zu diesem Zeitpunkt innerhalb der Methode `action()` längst nicht mehr in Kontakt mit dem Hindernis.
- Daher wertet sich die Bedingung von

```
if (Sensor.S1.readBooleanValue())
```

immer zu `false` aus.

- Der Roboter weicht daher immer im Uhrzeigersinn aus, wenn die Variable `rotationAngle` einen positiven Wert hat.
- Die Lösung besteht darin, die Werte der Sensoren in Instanzvariablen zu speichern.

HitWall, Version 2

```
public class HitWallVer2 implements Behavior {
    public HitWallVer2(TimingNavigator navigation, int dist, float angle) {
        this.navigation = navigation;
        this.backupDistance = dist;
        this.rotationAngle = angle;
        this.leftSensor = this.rightSensor = false;
    }
    public boolean takeControl() {
        this.leftSensor = Sensor.S1.readBooleanValue();
        this.rightSensor = Sensor.S3.readBooleanValue();
        return (this.leftSensor || this.rightSensor);
    }
    ...
    public void action() {
        Sound.beepSequence();
        navigation.travel(-this.backupDistance);
        if (this.leftSensor)
            navigation.rotate(this.rotationAngle);
        else
            navigation.rotate(-this.rotationAngle);
    }
    ...
    protected boolean leftSensor;
    protected boolean rightSensor;
}
```

Warum sich nichts ändert ...

- Die Frage, warum dieses Program das gleiche Verhalten zeigt, wie Version 1, lässt sich nur erklären, wenn man die **Ausführung der Behaviors durch den Arbitrator berücksichtigt**.
- Wie oben erwähnt, werden die einzelnen **Methoden** `takeControl()` sowie die `action()`-**Methode des aktiven Behaviors als Threads simultan ausgeführt**.
- Da in `takeControl()` die Instanzvariablen `leftSensor` und `rightSensor` **ständig neu gesetzt werden**, sind sie natürlich längst überschrieben, nachdem der Roboter zurückgesetzt hat.
- Da er beim Zurücksetzen aber nicht an ein Hindernis stößt, haben beide den Wert `false`, sobald der test in der Methode `action()` ausgeführt wird.
- Daher **wertet sich die Bedingung** von

```
if (this.leftSensor)
```

ebenfalls zu false aus.

- Die **Lösung** besteht darin, das **Überschreiben der Werte zu verhindern, während das Behavior aktiv ist**.

Der Konstruktor und die Instanzvariablen von HitWall

```
class HitWall extends Behavior {

    public HitWall(TimingNavigator navigation, int dist, float angle) {
        this.navigation = navigation;
        this.backupDistance = dist;
        this.rotationAngle = angle;
        this.leftSensor = false;
        this.rightSensor = false;
    }

    protected int backupDistance;
    protected float rotationAngle;
    protected TimingNavigator navigation;
    protected boolean leftSensor;
    protected boolean rightSensor;
}
```

Die korrekten Methoden `takeControl()` und `action()` von `HitWall`

```
public boolean takeControl() {
    boolean sensor1 = Sensor.S1.readBooleanValue();
    boolean sensor3 = Sensor.S3.readBooleanValue();

    if (sensor1 || sensor3) {
        this.leftSensor = sensor1;
        this.rightSensor = sensor3;
        return(true);
    } else
        return(false);
}

public void action() {
    ...
    this.leftSensor = this.rightSensor = false;
}
```

Die komplette Klasse HitWall

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

public class HitWall implements Behavior {

    public HitWall(TimingNavigator navigation,
                  int dist, float angle) {
        this.navigation = navigation;
        this.backupDistance = dist;
        this.rotationAngle = angle;
        this.leftSensor = false;
        this.rightSensor = false;
    }

    public boolean takeControl() {
        boolean sensor1 = Sensor.S1.readBooleanValue();
        boolean sensor3 = Sensor.S3.readBooleanValue();

        if (sensor1 || sensor3) {
            this.leftSensor = sensor1;
            this.rightSensor = sensor3;
            return(true);
        } else {
            return(false);
        }
    }

    public void suppress() {
        navigation.stop();
    }

    public void action() {
        Sound.beepSequence();
        // Drive backwards
        navigation.travel(-this.backupDistance);
        // Rotate away
        if (this.leftSensor) {
            navigation.rotate(this.rotationAngle);
        } else {
            navigation.rotate(-this.rotationAngle);
        }
        this.leftSensor = this.rightSensor =
            false;
    }

    protected int backupDistance;
    protected float rotationAngle;
    protected TimingNavigator navigation;
    protected boolean leftSensor;
    protected boolean rightSensor;
}
```

Die komplette Klasse DriveForward

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

public class DriveForward implements Behavior {

    public DriveForward (TimingNavigator navigation) {
        this.navigation = navigation;
    }

    public boolean takeControl() {
        return true;
    }

    public void suppress() {
        navigation.stop();
    }

    public void action() {
        navigation.forward();
    }

    protected TimingNavigator navigation;
}
```

Die komplette Klasse QuitProgram

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

public class QuitProgram implements Behavior {

    public QuitProgram(TimingNavigator navigation) {
        this.navigation = navigation;
    }

    public boolean takeControl() {
        return(Button.RUN.isPressed());
    }

    public void suppress() {
        System.exit(0);
    }

    public void action() {
        Sound.buzz();
        navigation.stop();
    }

    protected TimingNavigator navigation;
}
```

Das Programm BumperCar

Unser Arbitrator-Beispiel realisiert somit einen Roboter, der exploriert und im Fall von Hindernissen, versucht, diese zu umfahren. Das Programm kann durch Drücken des Run-Knopfes gestoppt werden.

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

class Behavior1 {

    public static void main(String [] args) {

        TimingNavigator navigation =
            new TimingNavigator(Motor.A, Motor.C,
                               timeForOneMeter, timeFor360Deg);

        Behavior b1 = new DriveForward(navigation);
        Behavior b2 = new HitWall(navigation, 20, 50);
        Behavior b3 = new QuitProgram(navigation);

        Behavior [] bArray = {b1, b2, b3};

        Arbitrator arby = new Arbitrator(bArray);
        arby.start();
    }

    static final float timeForOneMeter = 7.32f;
    static final float timeFor360Deg   = 2.92f;
}
```

Listener

- In unseren Beispielen oben haben wir **stets aktiv die Sensoren abgefragt**, wenn wir den Zustand erfahren wollten.
- Dies geschah beispielsweise mit `Sensor.readValue()`.
- Dieses Verfahren nennt man **Polling** (engl. abfragen).
- LeJOS stellt mit den **Listenern** ein zusätzliche Möglichkeit zur Verfügung.
- Die `Listener` funktionieren so, dass bei Änderungen der Sensorwerte automatisch von uns definierte Methoden aufgerufen werden können.
- Das **Motto der Listener** lässt sich auch mit
Don't call us, we call you.
umschreiben.
- Dies ist vergleichbar mit einem Email-Benachrtigungssystem, bei dem ein Fenster aufgeht, wenn eine neue Email angekommen ist.

Verwendung von Listenern

- Ein **Listener** ist ein Objekt einer **Listener-Klasse**, die wiederum ein `Listener`-Interface implementieren muss.

- Dieses **Interface gewährleistet**, dass die Listener-Klasse die **Methode**

```
stateChanged(Sensor s, int old, int nu)
```

implementiert.

- Ein typisches Beispiel:

```
class TouchViewer implements SensorListener {  
    public void stateChanged(Sensor s, int old, int nu) {...}  
}
```

- Ein Objekt dieser Klasse wird dann beispielsweise mit der **Methode**

```
Sensor.S1.addSensorListener(SensorListener sl)
```

beim Sensor S1 registriert.

Beispiel: Listener für den Lichtsensor

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

class Watcher implements SensorListener
{
    Watcher(Motor m, int val) {
        this.myMotor = m;
        this.threshold = val;
    }

    public void stateChanged (Sensor src, int oldValue, int newValue) {
        // Will be called whenever sensor value changes
        if (newValue>this.threshold) {
            this.myMotor.stop();
        } else {
            this.myMotor.forward();
        }
        try {
            Thread.sleep(100);
        } catch (InterruptedException e) {}
    }

    private Motor myMotor;
    private int threshold;
}
```

Anwendung des Listeners

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

class ShowLight {

    public static void main(String [] args)
        throws InterruptedException {

        Watcher lightWatcher = new Watcher(Motor.B, 50);
        Sensor.S2.setTypeAndMode(
            SensorConstants.SENSOR_TYPE_LIGHT,
            SensorConstants.SENSOR_MODE_DEGC);

        Sensor.S2.activate();
        Sensor.S2.addSensorListener(lightWatcher);

        Button.RUN.waitForPressAndRelease();
    }
}
```

Und jetzt alles zusammen ...

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

class Behavior2 {

    public static void main(String [] args) {

        TimingNavigator navigation =
            new TimingNavigator(Motor.A, Motor.C, timeForOneMeter, timeFor360Deg);

        Watcher lightWatcher = new Watcher(Motor.B, 50);
        Sensor.S2.setTypeAndMode(SensorConstants.SENSOR_TYPE_LIGHT,
                                SensorConstants.SENSOR_MODE_DEGC);

        Sensor.S2.activate();
        Sensor.S2.addSensorListener (lightWatcher);

        MusicPlayer musicThread = new MusicPlayer();
        musicThread.start();

        Behavior b1 = new DriveForward(navigation);
        Behavior b2 = new HitWall(navigation, 20, 50);
        Behavior b3 = new QuitProgram(navigation);
        Behavior [] bArray = {b1, b2, b3};
        Arbitrator arby = new Arbitrator(bArray);
        arby.start();
    }

    static final float timeForOneMeter = 7.32f;
    static final float timeFor360Deg   = 2.92f;
}
```

Mögliche Erweiterungen

- Timer
- zusätzliche Sensoren
(z.B. Rotationssensor, der bessere Navigation ermöglicht)
- Verhalten, die es ermöglichen, einer Linie zu folgen.
- Verhalten, die es ermöglichen, ein Linie zu finden.
- Verhalten, die es ermöglichen, ein Objekt zu umfahren.
- ...

Zusammenfassung

- **LeJOS bietet eine Vielzahl von Funktionen**, die man auch im normalen Java verwenden kann.
- Der **TimingNavigator** bietet eine einfache **Möglichkeit, Navigation zu realisieren**.
- Der **Arbitrator** kann unterschiedliche **Behaviors verwalten**.
- **Listener vermeiden das regelmäßige Abfragen von Sensoren**.
- Wem das alles zu starr ist, kann es natürlich **auch per Hand programmieren** (Mathematische Funktionen, Threads, ...)

- Mit LeJOS hat man mehr Möglichkeiten als mit dem Standard-System von Lego.

- ... und man hat eine Menge Spaß.

Vielen Dank für die Aufmerksamkeit

