LeJOS:
Mindstorms in Java programmieren

Alexander Koller

Softwareprojekt "Sprechende Roboter"
30. April 2004

Überblick

- Warum Java?
- Was ist LeJOS?
- Motoren steuern
- Auf Sensoreingaben reagieren
- Wie geht's weiter?

Programmiersysteme für Mindstorms

- Lego-Programmierumgebung ist hübsch, aber
  - für Experten zu umständlich
  - eingeschränkte Funktionalität
- Open-Source-Programmiersysteme:
  - BrickOS (früher LegOS)
  - NQC
  - LeJOS

Was ist LeJOS?

- Implementierung eines Teils der Java Virtual Machine für den RCX.
- Initiative von Jose Solorzano, inzwischen von verschiedenen Leuten auf Sourceforge weiterentwickelt.
- Programm wird mit normalem Java-Compiler auf PC kompiliert, dann per IR auf RCX übertragen.

`http://lejos.sourceforge.net/`

Warum LeJOS?

- Standard-Programmiersprache
  (anders als NQC)
- Einfache Installation und Verwendung
  (anders als BrickOS)
- Gute Unterstützung von IR-Kommunikation
- Verwendung von Java erlaubt eine
  Programmier-Infrastruktur "aus einem Guss"
  (Dialogsystem, Erkenner usw. auch in Java)

Speicherorganisation

- Der Speicher eines RCX, auf dem ein LeJOS-
  Programm läuft, enthält:
  - LeJOS-Firmware:
    Implementiert JVM u.ä.
  - Nötige Standardklassen aus der
    LeJOS-API
  - Kompiliertes Programm des Benutzers
    (bis zu ca. 12 KB)

Installation

- Installiert Java-SDK, erhältlich unter
  `http://java.sun.com/`
- Download von LeJOS 2.1 unter
  `http://lejos.sourceforge.net/download.html`
- Dort steht auch die Installationsanweisung. Ihr braucht "set RCXTTY=usb".
- Firmware auf den RCX laden.
- Ein Beispielprogramm ausprobieren.

Ein erstes Beispielprogramm

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

public class MoveForward {
  public static void main(String args[]) {
    Motor.A.forward();

    try {
      Thread.sleep(1000);
    } catch(Exception e) {
    }

    Motor.A.stop();
  }
}
```

Ein erstes Beispielprogramm

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

public class MoveForward {
  public static void main(String args[]) {
    Motor.A.forward();




    Motor.A.stop();
  }
}
```

Hauptprogramm

Packages der LeJOS-API

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

- Die LeJOS-API ist in verschiedenen Packages implementiert. Klassen, die direkt mit der Hardware des RCX zu tun haben, sind in platform.rcx.

Motorsteuerung

```
Motor.A.forward();
Motor.A.stop();
```

- Die Klasse `Motor` hat Methoden zur Steuerung von Motoren. `forward()` startet den Motor, `backward()` startet ihn in der anderen Richtung, `stop()` hält ihn an.
- Motor hat drei statische Members `A`, `B`, `C` vom Typ `Motor`. Diese Variablen stehen für die drei Motor-Anschlüsse am RCX.
- Man kann auch eigene Variablen vom Typ `Motor` deklarieren, z.B.

```
Motor CabinUpDownMotor = Motor.A;
```

Warten

```
try {
      Thread.sleep(1000);
    } catch(Exception e) {
    }
```

- Um den Motor eine Sekunde lang laufen zu lassen, warten wir vor dem Abschalten, indem wir Thread.sleep() aufrufen.
- Diese Methode kann eine Exception werfen, wenn der Thread unterbrochen wird. Diese Exception müssen wir fangen, aber nichts besonderes damit machen.

Ein erstes Beispielprogramm

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

public class MoveForward {
  public static void main(String argv[]) {
    Motor.A.forward();

    try {
      Thread.sleep(1000);
    } catch(Exception e) {
    }

    Motor.A.stop();
  }
}
```

Beispielprogramm ausprobieren

- Das Programm wird kompiliert mit
  `lejosc MoveForward.java`
- Dies erzeugt eine Datei MoveForward.class im aktuellen Verzeichnis.
- Download der Klasse auf den RCX mit
  `lejos MoveForward`
- Ausführen, indem man den Run-Knopf auf dem RCX drückt.

Vorwärts fahren, bis wir anstoßen

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

public class MoveForward {
  public static void main(String argv[]) {
    Motor.A.forward();

    waitUntilWeHitSomething();

    Motor.A.stop();
  }
}
```

Wie merken wir, dass wir anstoßen?

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

public class MoveForward {
  private static void
   waitUntilWeHitSomething() {
    Sensor.S1.activate();
    while( !Sensor.S1.readBooleanValue() ) {
      try {
        Thread.sleep(100);
      } catch (Exception e) {}
    }
  }

// ...

}
```

Umgang mit Sensoren

```
Sensor.S1.activate();
```

- Die Klasse Sensor definiert Methoden zum Auslesen von Sensoren. Sie hat Members `S1`, `S2`, `S3` für die drei verschiedenen Sensoranschlüsse.
- Bevor man einen Sensor ablesen kann, muss man ihn aktivieren. Das ist besonders wichtig für Lichtsensoren, weil erst dann das rote Licht eingeschaltet wird. Dazu dient die Methode `activate()`.

Sensoren ablesen

```
while( !Sensor.S1.readBooleanValue() )
```

- Man kann den Wert eines Sensors auf verschiedenen Skalen ablesen.
- Mit der Methode `readBooleanValue()` bekommt man einen boolean-Wert (also true oder false) zurück. True heißt, dass der Sensor gedrückt ist.
- Mit `readValue()` bekommt man einen int-Wert, der vom Modus des Sensors abhängt. Defaultmäßig ist das `SensorConstants.SENSOR_MODE_PCT`; damit gibt `readValue()` einen Wert von 0 bis 100 zurück.
- Man kann (und sollte) den Modus des Sensors mit `setTypeAndMode()` einstellen, bevor man ihn verwendet.

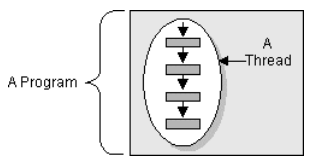
Sensoren: Ist das elegant?

- In jeder Iteration der Hauptschleife muss man jeden Sensor ablesen und entsprechend auf Änderungen reagieren.
- Das wird für viele Sensoren schnell unübersichtlich.
- Wartezeiten zwischen Sensorablesungen erlauben keine schnelle Reaktion.
- Lösung: `SensorListener`-Objekte.
- Ein Thread führt weiter das Hauptprogramm aus, während ein anderer gleichzeitig auf Sensor-Eingaben reagiert.

Exkurs: Nebenläufigkeit in Java

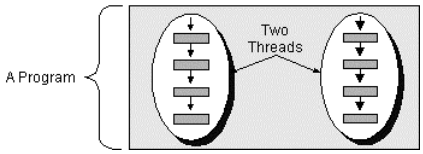
`http://java.sun.com/docs/books/tutorial/essential/threads/`

- Normalerweise werden Programme sequentiell abgearbeitet: Man fängt vorne an, führt einen einzigen Pfad durch das Programm aus, und hört auf.



Threads

- Java unterstützt den Ablauf von nebenläufigen Threads: Konzeptionell laufen verschiedene Berechnungen gleichzeitig ab.



Keine Garantien über die Reihenfolge, in der die einzelnen Operationen in den beiden Threads abgearbeitet werden!

Threads

- Es gibt einen Haupt-Thread, der beim Programmstart erzeugt wird. Wenn der Haupt-Thread endet, endet auch das Programm.
- Neue Threads können explizit erzeugt werden. Man muss dann z.B. ein Stück Code angeben, das der neue Thread ausführen soll.
- Alle Threads teilen sich den gleichen Speicher, können also ihre Variablenwerte gegenseitig lesen und ändern.

Synchronisation von Threads

- Was passiert, wenn zwei Threads gleichzeitig der gleichen Variable etwas zuweisen wollen?
- Was passiert, wenn Thread A den Variablen X und Y konsistente Werte zuweisen will; aber zwischen den beiden Zuweisungen liest Thread B die Werte von beiden Variablen?
- Wie kann ein Thread einem anderen eine Nachricht schicken?

Synchronisation von Threads

- Threads können sich über ein Objekt synchronisieren.

Thread A

```
try {
  synchronized(obj) {
    obj.wait();
  }
} catch(Exception e) { }
```

Thread B

```
synchronized(obj) {
  obj.notifyAll();
}
```

- Thread A wartet, bis Thread B ihm sagt, dass er weitermachen darf.

SensorListeners

- In LeJOS können wir Nebenläufigkeit nutzen, um auf Sensoreingaben zu reagieren.
- Das Interface `SensorListener` stellt einen Thread dar, der auf eine Änderung eines Sensor-Wertes reagiert.
- Typischerweise verbringt der Haupt-Thread die meiste Zeit mit wait(). Die SensorListeners schicken ein notifyAll(), sobald er auf eine Sensor-Änderung reagieren muss.

Anstoßen mit SensorListeners

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

class CollisionDetector implements SensorListener {

  public void stateChanged
        (Sensor s, int oldVal, int newVal) {

    if( (newVal > 80) && (oldVal < 80) ) {
      synchronized(s) {
        s.notifyAll();
      }
    }

  }
}
```

Anstoßen mit SensorListeners

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

class MoveForward {
  private static void waitUntilWeHitSomething() {
    CollisionDetector d = new CollisionDetector();

    Sensor.S1.activate();
    Sensor.S1.addSensorListener(d);

    synchronized(Sensor.S1) {
      try {
        Sensor.S1.wait();
      } catch (Exception e) { }
    }
  }
}
```

Komplexe SensorListeners

- SensorListeners sind für diesen einfachen Fall Overkill.
- Aber SensorListeners sind normale Klassen und können noch viel zusätzliche Funktionalität enthalten!
- Beispiel: Im Fahrstuhl gibt es eine Klasse FloorCounter, die SensorListener implementiert und mitzählt, in welchem Stock wir sind.

Wie geht's weiter?

- Mit elementarem Wissen über Motoren und Sensoren kommt man schon ziemlich weit.
- Vollständige Dokumentation über die Klassen der LeJOS-API findet man unter
  `http://lejos.sourceforge.net/apidocs/`
- Wie immer lernt man Programmieren nur durch Programmieren. Probiert die interessanten API-Funktionen selbst durch!

Zusammenfassung

- LeJOS erlaubt es uns, den RCX in Java zu programmieren.
- Zugriff auf Motoren und Sensoren
- Elegante Überwachung von Sensoren unter Verwendung von Nebenläufigkeit und SensorListeners.
- Jetzt seid Ihr dran!