

# Problemorientiertes Programmieren

## RoboCode

Einführung RoboCode

Eva Eibenberger und Christian Rieß





## Der erste Roboter, Vorüberlegungen

- Auf zu einem praktischen Problem – den Robotern
- Mehr Java kann man unterwegs lernen
- Grundsätzlich kann man den Roboter mit **irgendeinem Editor** schreiben – genauso wie **irgendein anderes Programm**  
**-> Best tool for the job!**
- Editoren müsst ihr euch privat aneignen.
- Wir starten mit einem einfachen Editor, der mit RoboCode mitgeliefert wird (wer will kann jederzeit z.B. auf eclipse umsteigen).



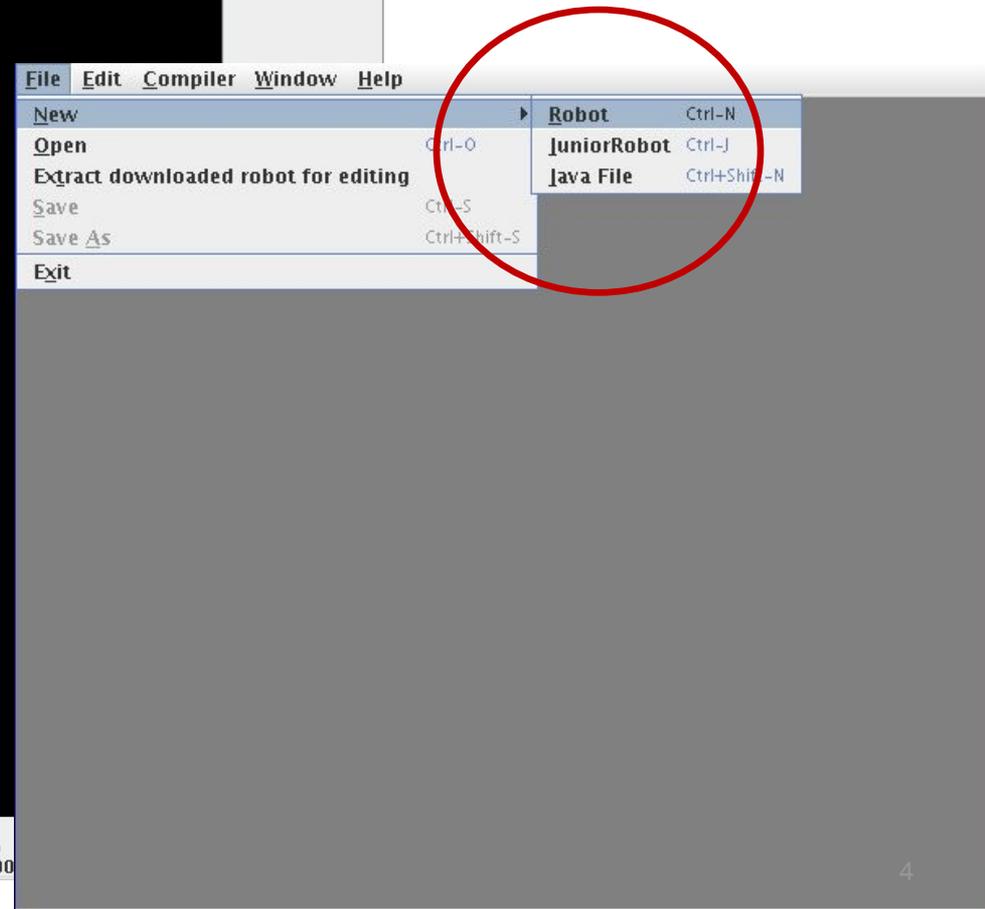
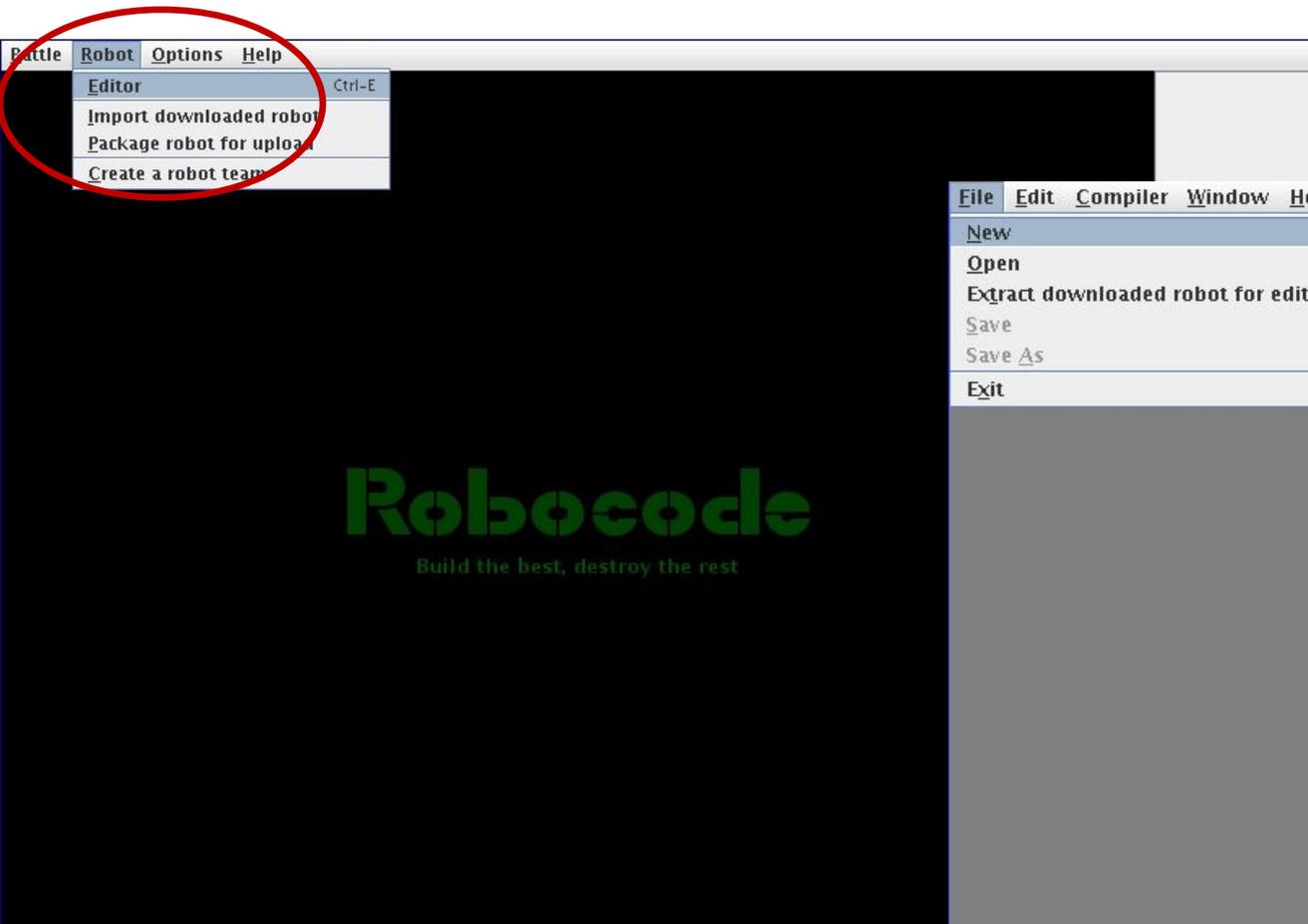
## Einrichten der Umgebung

- 1. RoboCode-Oberfläche herunterladen (.oO google: robocode download)
- 2. Im Home liegt eine Datei robocode-<versionsnummer>.jar
- 3. “.jar” steht für „Java Archive“. Solche Archive können (im Terminal) ausgeführt werden mit der Anweisung  
java -jar robocode-<versionsnummer>.jar  
Klick Dich durch die Installation.
- 4. Starte die Umgebung mit „cd robocode“ und „./robocode.sh“



# Starten der Umgebung

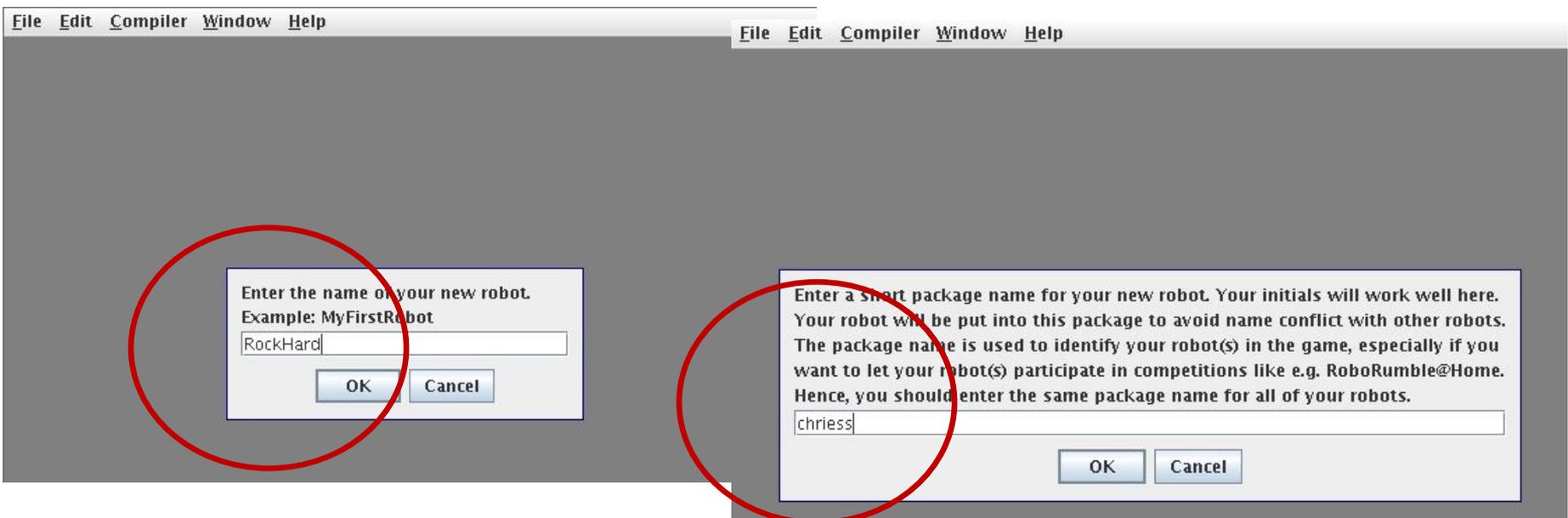
Starte den internen Editor, ein Roboter-Prototyp wird bereitgestellt.





## Wie soll der Prototyp heißen?

- Dein Roboter braucht einen **individuellen** Namen (den Klassennamen), und ein **individuelles** Package.
- Beachte: Keine Umlaute, keine Leerstellen, Klassennamen typischerweise groß schreiben, packages klein<sup>[1]</sup>



```

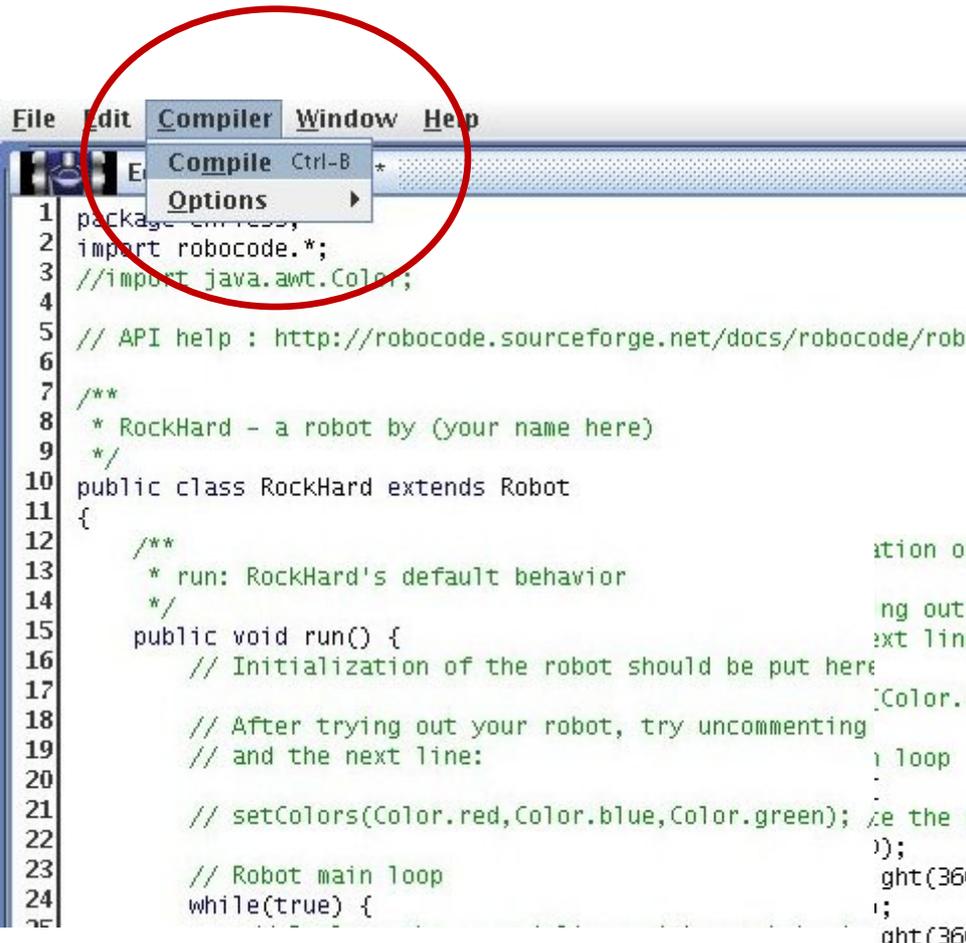
1 package chriess;
2 import robocode.*;
3 //import java.awt.Color;
4
5 // API help : http://robocode.sourceforge.net/docs/robocode/robocode/Robot.html
6
7 /**
8  * RockHard - a robot by (your name here)
9  */
10 public class RockHard extends Robot
11 {
12     /**
13     * run: RockHard's default behavior
14     */
15     public void run() {
16         // Initialization of the robot should be put here
17
18         // After trying out your robot, try uncommenting the import at the top,
19         // and the next line:
20
21         // setColors(Color.red,Color.blue,Color.green); // body,gun,radar
22
23         // Robot main loop
24         while(true) {
25             // Replace the next 4 lines with any behavior you would like
26             ahead(100);
27             turnGunRight(360);
28             back(100);
29             turnGunRight(360);
30         }
31     }
32
33     /**
34     * onScannedRobot: What to do when you see another robot
35     */
36     public void onScannedRobot(ScannedRobotEvent e) {
37         // Replace the next line with any behavior you would like
38         fire(1);
39     }
40
41     /**
42     * onHitByBullet: What to do when you're hit by a bullet
43     */
44     public void onHitByBullet(HitByBulletEvent e) {
45         // Replace the next line with any behavior you would like
46         back(10);
47     }
48
49     /**
50     * onHitWall: What to do when you hit a wall
51     */
52     public void onHitWall(HitWallEvent e) {
53         // Replace the next line with any behavior you would like
54         back(20);
55     }
56 }

```

- In dem mitgelieferten Editor wird der minimal notwendige Code für einen Roboter bereitgestellt.
- Wir lesen den Code gleich; erst wollen wir jedoch testen, ob alles funktioniert:
  - compilieren,
  - in die Arena einbinden.



# Compilieren, speichern (falls nötig):



```
1 package arena;
2 import robocode.*;
3 //import java.awt.Color;
4
5 // API help : http://robocode.sourceforge.net/docs/robocode/rob
6
7 /**
8  * RockHard - a robot by (your name here)
9  */
10 public class RockHard extends Robot
11 {
12     /**
13      * run: RockHard's default behavior
14      */
15     public void run() {
16         // Initialization of the robot should be put here
17
18         // After trying out your robot, try uncommenting
19         // and the next line:
20
21         // setColors(Color.red,Color.blue,Color.green);
22
23         // Robot main loop
24         while(true) {
```

- Achte auf die Ausgabe von „Compile“: Nur ein fehlerfrei übersetzter Roboter kann in der Arena benutzt werden (später mehr...).

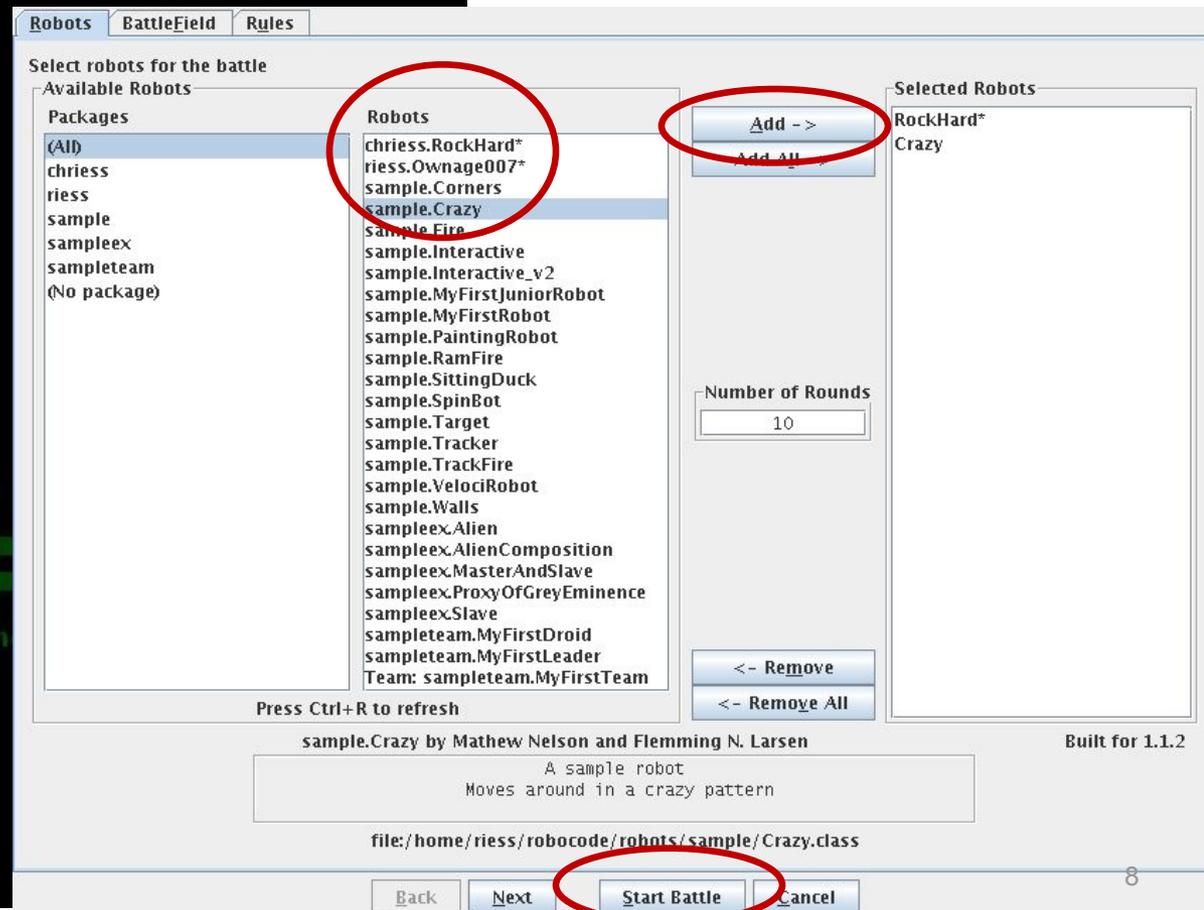
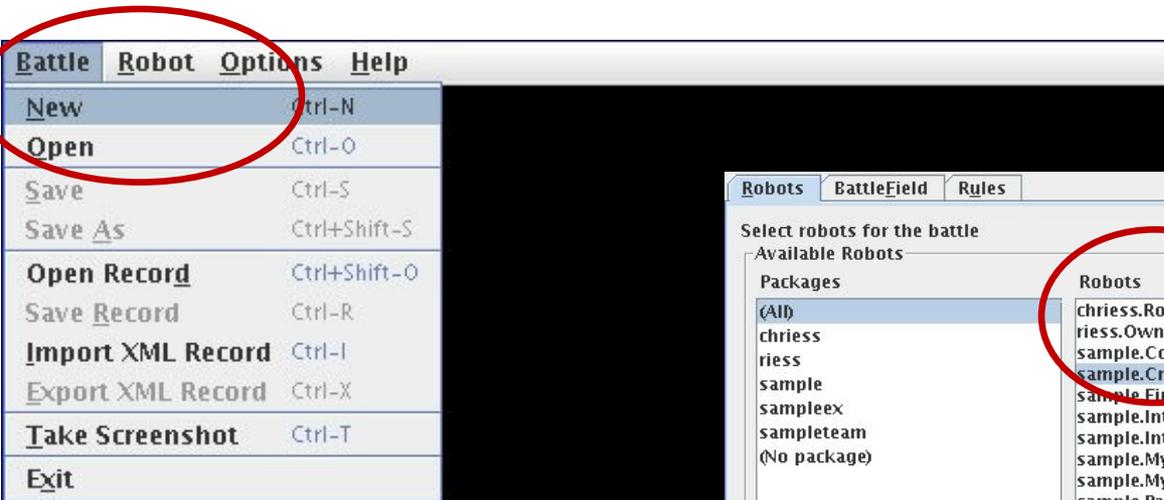


```
:: What to do when you see another robot
annedRobot(ScannedRobotEvent e) {
ie next line with any behavior you would like
```



# Auf in die Arena!

- Der neue Roboter sollte automatisch in der Auswahl erscheinen.





## ...nach dem Kampf...

- Der Prototyp ist natürlich noch recht... suboptimal.
- Bisher haben wir **beobachtet**, dass der Prototyp Schwächen hat.
- Sehen reicht nicht: wir müssen die Aktionen im Code **analysieren**,
- dann Verbesserungsmöglichkeiten in der Arena **recherchieren**,
- eine Idee theoretisch **ausknobeln**,
- und **implementieren**.

- Und danach?

(oO Ab in die Arena,

beten dass die Arbeit sich gelohnt hat,

irgendwas funktioniert nicht

(kann nicht; Dinge funktionieren nie einfach so),

also wieder zum Anfang (beobachten, analysieren...)

```

1 package chriess;
2 import robocode.*;
3 //import java.awt.Color;
4
5 // API help : http://robocode.sourceforge.net/docs/robocode/robocode/Robot.html
6
7 /**
8  * RockHard - a robot by (your name here)
9  */
10 public class RockHard extends Robot
11 {
12     /**
13     * run: RockHard's default behavior
14     */
15     public void run() {
16         // Initialization of the robot should be put here
17
18         // After trying out your robot, try uncommenting the import at the top,
19         // and the next line:
20
21         // setColors(Color.red,Color.blue,Color.green); // body,gun,radar
22
23         // Robot main loop
24         while(true) {
25             // Replace the next 4 lines with any behavior you would like
26             ahead(100);
27             turnGunRight(360);
28             back(100);
29             turnGunRight(360);
30         }
31     }
32
33     /**
34     * onScannedRobot: what to do when you see another robot
35     */
36     public void onScannedRobot(ScannedRobotEvent e) {
37         // Replace the next line with any behavior you would like
38         fire(1);
39     }
40
41     /**
42     * onHitByBullet: What to do when you're hit by a bullet
43     */
44     public void onHitByBullet(HitByBulletEvent e) {
45         // Replace the next line with any behavior you would like
46         back(10);
47     }
48
49     /**
50     * onHitWall: What to do when you hit a wall
51     */
52     public void onHitWall(HitWallEvent e) {
53         // Replace the next line with any behavior you would like
54         back(20);
55     }
56 }

```

„import robocode.\*“ heißt „wir benutzen alles aus dem package robocode“ („\*“ ist ein „Wildcard“, also „alles“)

„extends“ kann man lesen wie „ist ein“, oder „hat die Funktionen von“, in diesem Fall die Funktionen von „Robot“ (siehe Dokumentation, ein paar Folien später)

„run“ ist die Methode, die von der Arena aufgerufen wird, um den Roboter zu starten; er läuft, bis das Match zu Ende ist (siehe Architektur der Arena, ein paar Folien später)

„while(true)“ heißt „für alle Ewigkeit...“  
 ...ahead, turnGunRight, back, turnGunRight liest sich fast wie ein englischer Text.  
 Offenbar fahren wir bisschen vorwärts, drehen die Kanone (mit aufgepflanztem Scanner) fahren bisschen rückwärts, drehen die Kanone bisschen weiter.

„on...“ Methoden, die ein „...Event“ übergeben bekommen, werden von der Arena aufgerufen, wenn etwas passiert.  
 Die Anweisungen in „run“ werden so lange unterbrochen.  
 z.B. die Methode onScannedRobot wird aufgerufen, wenn in unserem Scanner ein anderer Roboter auftaucht. Was der Scanner sieht, kann man aus „ScannedRobotEvent“, hier „e“ genannt, auslesen. Das wird hier nicht gemacht, sondern einfach geschossen.  
 Die anderen beiden Ereignisse muss man analog lesen: Wird der Roboter selbst getroffen, setzt er zurück, rammt er eine Wand, setzt er auch zurück.



## Exkurs: Interne Verarbeitung der Anweisungen

- Ein Roboter wird in einem „Thread“ gestartet ( „pseudo-separates Program“).
- Die Arena lässt jeden Thread abwechselnd ein paar Millisekunden rechnen. Spielanweisungen, z.B. „ahead“, werden gespeichert, bis jeder Thread gerechnet hat.
- Die Arena prüft, ob ein „Ereignis“ eingetreten ist. Falls ja, springt der Thread/Roboter in die Ereignis-Methode.  
z.B. wenn ein Roboter im Scannerbereich liegt, werden dessen Daten in ScannedRobotEvent gespeichert. In der nächsten Rechenphase wird onScannedRobotEvent mit diesen Daten aufgerufen.
- Die Arena führt die Anweisungen aus.  
z.B. „ahead“: Roboter ein kleines Stück vorbewegen und speichern, wie viel Wegstrecke noch verbleibt
- ...und wieder von vorne  
(jeder Roboter darf rechnen; entweder erst in einem Ereignis, oder dort, wo er zuletzt unterbrochen wurde).



## In der Dokumentation recherchieren

- Der Prototyp ist natürlich noch recht... suboptimal.
- Bisher haben wir **beobachtet**, dass der Prototyp Schwächen hat.
- Sehen reicht nicht: wir müssen die Aktionen im Code **analysieren**,
- dann Verbesserungsmöglichkeiten in der Arena **recherchieren**,
- eine Idee theoretisch **ausknobeln**,
- und **implementieren**.

- Und danach?

(oO Ab in die Arena,

beten dass die Arbeit sich gelohnt hat,

irgendwas funktioniert nicht

(kann nicht; Dinge funktionieren nie einfach so),

also wieder zum Anfang (beobachten, analysieren...)



# RoboCode- und Java-Dokumentation lesen

- JavaDoc: Referenzdokumentation in HTML
- Im Installationsverzeichnis unter javadoc/index.html

(.oO HTML: am besten  
im Browser öffnen)

Links unten ist  
eine Liste der  
verfügbaren  
Klassen: am  
Anfang ist die  
Klasse Robot  
wichtig

**Overview** Package Class Tree Deprecated Index Help  
PREV NEXT FRAMES NO FRAMES

## Robocode 1.7.3.0-Beta API

### Packages

<a href="#">robocode</a>	Robot API used for writing robots for Robocode.
<a href="#">robocode.annotation</a>	Contains annotations that can be used with Robocode.
<a href="#">robocode.control</a>	The Robocode Control API is used for controlling the Robocode application from another external application.
<a href="#">robocode.control.events</a>	Battle events that occurs during a game, and which are used for the robocode.control.IBattleListener class.
<a href="#">robocode.control.snapshot</a>	Snapshots of the battle turns, robots, bullets, scores etc.
<a href="#">robocode.robotinterfaces</a>	Robot Interfaces used for creating new robot types, e.g. with other programming languages.
<a href="#">robocode.robotinterfaces.peer</a>	Robot peers available for implementing new robot types based on the Robot Interfaces.
<a href="#">robocode.util</a>	Utility classes that can be used when writing robots.

**Overview** Package Class Tree Deprecated Index Help  
PREV NEXT FRAMES NO FRAMES

Copyright © 2011 [Robocode](#). All Rights Reserved.

Done



# RoboCode- und Java-Dokumentation lesen

Klasse  
 Robot  
 auswählen

Overview Package **Class Tree** Deprecated Index Help

PREV CLASS NEXT CLASS FRAMES NO FRAMES  
 SUMMARY: NESTED | FIELD | CONSTR | METHOD DETAIL: FIELD | CONSTR | METHOD

**robocode**  
**Class Robot**

java.lang.Object  
 ↳ robocode.\_RobotBase  
 ↳ robocode.\_Robot  
 ↳ robocode.Robot

All Implemented Interfaces:  
[Runnable](#), [IBasicEvents](#), [IBasicEvents2](#), [IBasicEvents3](#), [IBasicRobot](#), [IInteractiveEvents](#), [IInteractiveRobot](#), [IPaintEvents](#), [IPaintRobot](#)

Direct Known Subclasses:  
[\\_AdvancedRobot](#)

public class Robot  
 extends [Robot](#)  
 implements [IInteractiveRobot](#), [IPaintRobot](#), [IBasicEvents3](#), [IInteractiveEvents](#), [IPaintEvents](#)

The basic robot class that you will extend to create your own robots.

Please note the following standards will be used:  
 heading - absolute angle in degrees with 0 facing up the screen, positive clockwise. 0 <= heading < 360.  
 bearing - relative angle to some object from your robot's heading, positive clockwise. -180 < bearing <= 180  
 All coordinates are expressed as (x,y).  
 All coordinates are positive.  
 The origin (0,0) is at the bottom left of the screen.  
 Positive x is right.  
 Positive y is up.

Author:  
 Mathew A. Nelson (original), Flemming N. Larsen (contributor), Matthew Reeder (contributor), Stefan Westen (contributor), Pavel Savara (contributor)

See Also:  
[robocode.sourceforge.net](#), [Building your first robot](#), [JuniorRobot](#), [AdvancedRobot](#), [TeamRobot](#), [Droid](#)

**Field Summary**

Fields inherited from class robocode.\_RobotBase  
[out](#)

**Constructor Summary**

[Robot\(\)](#)  
 Constructs a new robot.

Allgemein  
 nützliche  
 Informationen  
 stehen am  
 Anfang der Seite



# RoboCode- und Java-Dokumentation lesen

Danach folgt eine Liste der Methoden, die ein Robot anbietet, zusammen mit einer einzeiligen Beschreibung.

z.B.:  
„double getX()  
Returns the X position of the robot. (0, 0) is at the bottom left of the battlefield.“

Die Methode heißt `getX()`, und wenn man sie aufruft, bekommt man den X-Wert als `double`.

Return Type	Method Name	Description
<code>int</code>	<code>getNumRounds()</code>	Returns the number of rounds in the current battle.
<code>int</code>	<code>getOthers()</code>	Returns how many opponents that are left in the current round.
<code>double</code>	<code>getRadarHeading()</code>	Returns the direction that the robot's radar is facing, in degrees.
<code>Runnable</code>	<code>getRobotRunnable()</code>	This method is called by the game to invoke the <code>run()</code> method of your robot, where the program of your robot is implemented.
<code>int</code>	<code>getRoundNum()</code>	Returns the current round number (0 to <code>getNumRounds() - 1</code> ) of the battle.
<code>long</code>	<code>getTime()</code>	Returns the game time of the current round, where the time is equal to the current turn in the round.
<code>double</code>	<code>getVelocity()</code>	Returns the velocity of the robot measured in pixels/turn.
<code>double</code>	<code>getWidth()</code>	Returns the width of the robot measured in pixels.
<code>double</code>	<code>getX()</code>	Returns the X position of the robot. (0,0) is at the bottom left of the battlefield.
<code>double</code>	<code>getY()</code>	Returns the Y position of the robot. (0,0) is at the bottom left of the battlefield.
<code>void</code>	<code>onBattleEnded(BattleEndedEvent event)</code>	This method is called after the end of the battle, even when the battle is aborted.
<code>void</code>	<code>onBulletHit(BulletHitEvent event)</code>	This method is called when one of your bullets hits another robot.
<code>void</code>	<code>onBulletHitBullet(BulletHitBulletEvent event)</code>	This method is called when one of your bullets hits another bullet.
<code>void</code>	<code>onBulletMissed(BulletMissedEvent event)</code>	This method is called when one of your bullets misses, i.e. hits a wall.
<code>void</code>	<code>onDeath(DeathEvent event)</code>	This method is called if your robot dies.
<code>void</code>	<code>onHitByBullet(HitByBulletEvent event)</code>	This method is called when your robot is hit by a bullet.
<code>void</code>	<code>onHitRobot(HitRobotEvent event)</code>	This method is called when your robot collides with another robot.
<code>void</code>	<code>onHitWall(HitWallEvent event)</code>	This method is called when your robot collides with a wall.



# RoboCode- und Java-Dokumentation lesen

Weiter unten auf der Seite sind die einzelnen Methoden noch genauer beschrieben.

Interessant sind auch die „See Also“-Einträge: z.B. in der Methode `getHeading()` wird auf `getGunHeading()` und `getRadarHeading()` verwiesen.

The screenshot shows a web browser window displaying the Java documentation for the RoboCode API. The browser's address bar shows the file path: `file:///home/riess/robocode/javadoc/index.html`. The page content is organized into sections for different methods. A red circle highlights the `getHeading()` method. The `getHeading()` section includes the following text:

```
public double getHeading()

Returns the direction that the robot's body is facing, in degrees. The value returned will be between 0 and 360 (is excluded).

Note that the heading in Robocode is like a compass, where 0 means North, 90 means East, 180 means South, and 270 means West.

Returns:
    the direction that the robot's body is facing, in degrees.

See Also:
    getGunHeading\(\), getRadarHeading\(\)
```

Below the circled section, other methods are listed:

```
getBattlefieldHeight()
Returns the height of the current battlefield measured in pixels.

Returns:
    the height of the current battlefield measured in pixels.

getHeight()
Returns the height of the robot measured in pixels.

Returns:
    the height of the robot measured in pixels.

See Also:
    getWidth\(\)

getWidth()
Returns the width of the robot measured in pixels.

Returns:
    the width of the robot measured in pixels.

See Also:
    getHeight\(\)

getName()
Returns the name of the robot.

Returns:
    the name of the robot.
```

The left sidebar of the browser shows a list of classes and packages, including `MouseEvent`, `Robot`, and `RobotState`.



## Andere Roboter angucken!

- Zum Anfang kann man auch schauen, was andere Roboter machen:
  - Im Installationsverzeichnis unter `robots/sample` stöbern
  - Im RoboCode-Wiki im Internet
- Beachte: Wir schreiben heute einen `Robot`  
(-> im Code steht `RockHard extends Robot`).  
Einige Beispielroboter sind jedoch `AdvancedRobots` und benutzen leicht unterschiedliche Befehle.  
(-> z.B. `Crazy extends AdvancedRobot`)
- Ein `AdvancedRobot` ist noch bisschen cooler, aber heute werden wir nicht darüber sprechen.



## Ideen ausknobeln!

- Der Prototyp ist natürlich noch recht... suboptimal.
- Bisher haben wir **beobachtet**, dass der Prototyp Schwächen hat.
- Sehen reicht nicht: wir müssen die Aktionen im Code **analysieren**,
- dann Verbesserungsmöglichkeiten in der Arena **recherchieren**,
- eine Idee theoretisch **ausknobeln**,
- und **implementieren**.

- Und danach?

(oO Ab in die Arena,

beten dass die Arbeit sich gelohnt hat,

irgendwas funktioniert nicht

(kann nicht; Dinge funktionieren nie einfach so),

also wieder zum Anfang (beobachten, analysieren...)



## Zeichnungen helfen

- Wenn der Prototyp eine Wand trifft, fährt er ein Stück zurück

```
49  /**
50   * onHitWall: What to do when you hit a wall
51   */
52  public void onHitWall(HitWallEvent e) {
53      // Replace the next line with any behavior you would like
54      back(20);
55  }
56 }
```

[.oO und wenn er die Wand im Rückwärtsgang getroffen hat?]

Au Backe, nicht auszudenken.

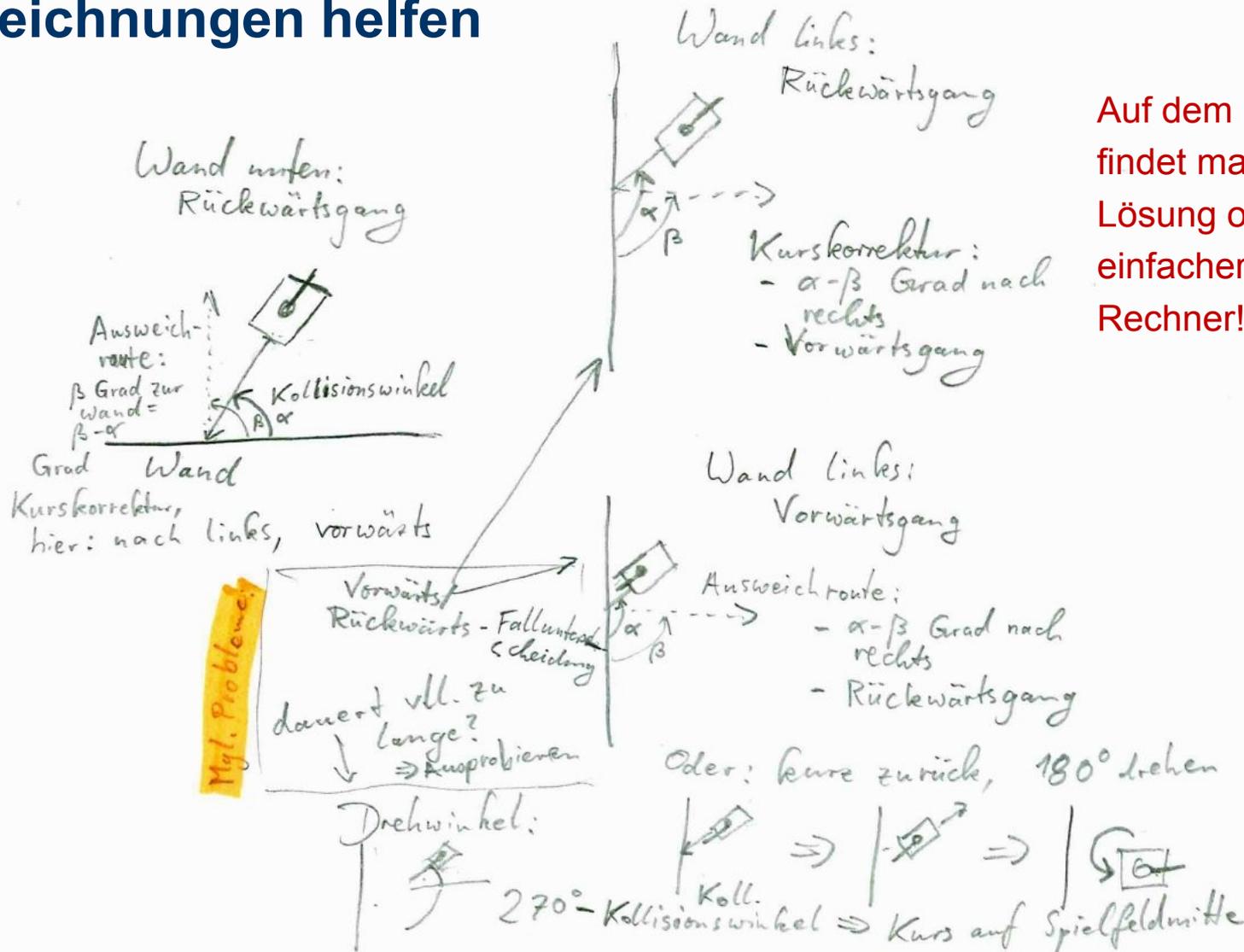


Bild von zazzle.com

- ...und generell: vielleicht wäre es besser, sich nach einer Kollision ein Stück von der Wand zu entfernen!



# Zeichnungen helfen



Auf dem Papier findet man eine Lösung oft einfacher als am Rechner!



## Verbesserungen implementieren

- Der Prototyp ist natürlich noch recht... suboptimal.
- Bisher haben wir **beobachtet**, dass der Prototyp Schwächen hat.
- Sehen reicht nicht: wir müssen die Aktionen im Code **analysieren**,
- dann Verbesserungsmöglichkeiten in der Arena **recherchieren**,
- eine Idee theoretisch **ausknobeln**,
- und **implementieren**.
- Und danach?
  - (oO Ab in die Arena,
  - beten dass die Arbeit sich gelohnt hat,
  - irgendwas funktioniert nicht
  - (kann nicht; Dinge funktionieren nie einfach so),
  - also wieder zum Anfang (beobachten, analysieren...)



## Implementieren: Beispiel in run()

- Zum Beispiel zweite Strategie:  
Mehr Kurven fahren,  
wenn die Energie zu tief  
sinkt

```
15 public void run() {
16     // Initialization of the
17
18     // After trying out your
19     // and the next line:
20
21     // setColors(Color.red, Co
22
23     // Robot main loop
24     while(true) {
25         // Replace the next 4
26         ahead(100);
27         turnGunRight(360);
28         back(100);
29         turnGunRight(360);
30     }
31 }
32
```

```
public void run() {
    double startEnergy = getEnergy(); // Anfangsenergie speichern
    while (getEnergy() > startEnergy/2) { // solange mehr als die Haelfte der Energie
        ahead(100);
        turnGunRight(360);
        back(100);
        turnGunRight(360);
    }
    // wenn wir hier her kommen, hat der andere uns vermutlich ein paar mal getroffen
    while(true) { // vielleicht ist es besser im Kreis zu fahren?
        turnRight(30);
        ahead(40);
        turnGunRight(90);
    }
}
```



## Implementieren: Beispiel Ereignis

- Zur Ereignisbehandlung muss man sich oft den Zustand merken

```
49     /**
50      * onHitWall: What to do when you hit a
51      */
52     public void onHitWall(HitWallEvent e) {
53         // Replace the next line with any be
54         back(20);
55     }
56 }
```

```
public class RockHard extends Robot {
    int fahreVorwaerts; // im Ereignis muessen wir den Zustand kennen, in dem wir
                        // unterbrochen wurden
    public void run() {
        while(true) {
            fahreVorwaerts = 1; // Zustand speichern
            ahead(100);
            turnGunRight(360);
            fahreVorwaerts = 0; // Zustand speichern
            back(100);
        }
    }
    public void onHitWall(HitWallEvent e) {
        if (fahreVorwaerts == 1) // wenn wir hier hineinspringen, haben wir jetzt die
                                // Information, was wir zuletzt gemacht haben.
            back(150);
        else
            ahead(150);
    }
}
```



## Roboter in das EST einstellen

- Roboter = „Übungsabgabe“
- Die „Übung“ ist das ganze Semester offen
- Für EST-Uploads müssen alle Einreichungen den selben Namen haben; daher: Roboter in ein .zip-File verpacken:

```
sichries@fau100j> zip my_robot.zip robocode/robots/chriess/RockHard.java
```

- Jeder Roboter im EST muss einen eindeutigen Namen haben!
- M.a.W.: Wer ihr zu zweit oder dritt arbeitet, und jeder seine Variante hochladen will, muss der Roboter individuell benannt sein, z.B. RockHard2 oder RockHardest.
- Wenn man seinen eigenen Roboter im EST erneut einreicht, kann man den Namen beibehalten



## Turniere spielen

- Ab Montag werden die Roboter jede Nacht aus dem EST heruntergeladen und compiliert.
- Dann wird ein Turnier gespielt.
- Die Turnierergebnisse werden tagesaktuell unter <http://www5.cs.fau.de/fileadmin/lectures/SS12/robocode> veröffentlicht (zugreifbar aus dem Uninetz)



---

# Good to go!

- Viel Erfolg, und viel Spaß!



Bild von 3D Keatins

Let's rock!