Robocode – wir entwickeln den Roboter weiter

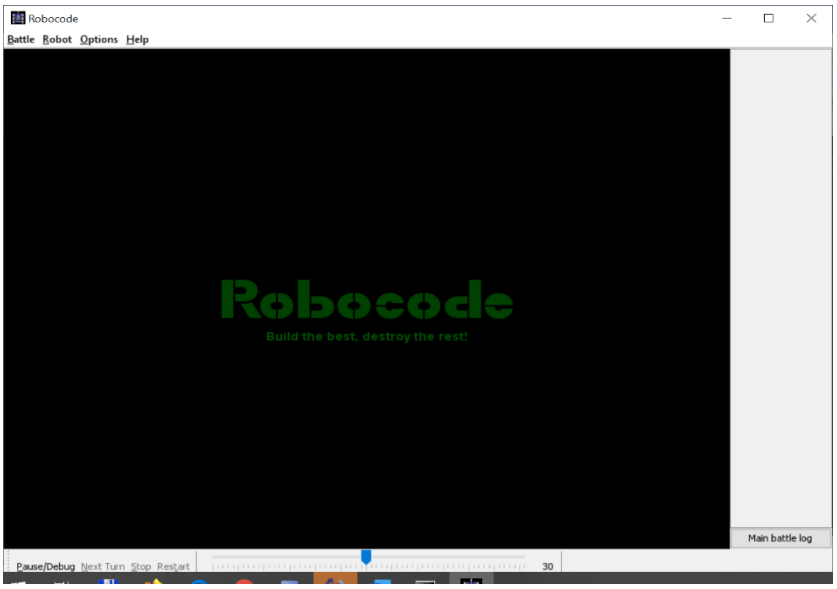
1 Bevor du beginnst

- Stelle sicher, dass Du Java und Robocode wie im Dokument vom ersten Tag **Robocode auf Windows installieren.docx** beschrieben, installiert hast.
- Stelle sicher, dass Du das Dokument **Erste Schritte mit Robocode** durchgearbeitet hast. Wir bauen auf dem dort entwickelten Roboter auf

2 Was ist der Plan für heute?

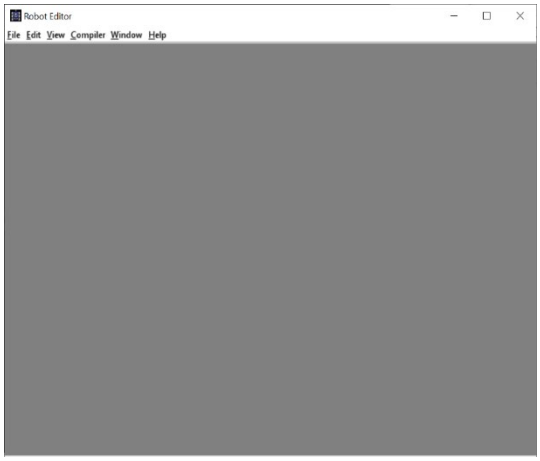
In unserer ersten Lektion haben wir unseren ersten Roboter hergestellt, der leicht einen Target-Roboter besiegen kann. Nun, das war nicht schwer! Der Zielroboter wehrte sich nicht einmal. Heute werden wir unseren ersten Roboter weiterentwickeln und einen zweiten Roboter programmieren. Unser zweiter Roboter wird in der Lage sein, unseren ersten Roboter leicht zu besiegen. ***Nein, warte***... Unser **`Day2`** Roboter wird leicht in der Lage sein, 4 unserer **`Day1`** Roboter gleichzeitig zu besiegen!

3 Starte Robocode und verwende den Code-Editor



- Starte jetzt Robocode wie es im Dokument vom ersten Tag, **Robocode auf Windows installieren.docx**, beschrieben ist.

- Jetzt wähle vom **`Robot Menu`** den **`Source Editor`** (Quellencode Editor) aus, um den Java Editor zu starten. Auch das ist im Dokument vom ersten Tag



beschrieben.

4 Erstelle Deinen Day2 Roboter

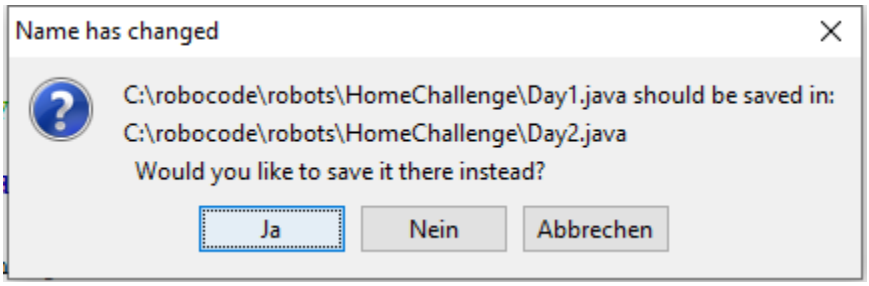
- Wähle **`File > Open`**
- Öffne den **`HomeChallenge`** Ordner und darin öffne die Datei **`Day1.java`**
- Ändere im Editor die Zeile

```
public class Day1 extends JuniorRobot
```

zu

```
public class Day2 extends JuniorRobot
```

- Speichere die Datei, und wähle Ja aus, um die Datei neu als Day2.java zu speichern.



Falls Du interessiert bist hier zur Erklärung noch zwei weitere Teile von Java.

- Eine Klasse (**`class`**) ist eine Sammlung von Methoden und Variablen zum gleichen Thema, bei uns hier ist der Day1 Roboter eine Klasse (class). Wir haben gestern die Methode **`run()`** dieser Klasse verwendet.
- Das Wort **`extends`** das wir auch verwenden (**`public class Day1 extends JuniorRobot`**) heisst, dass unsere **`Day1`** Klasse eine vorhandene Klasse (**`JuniorRobot`**) die von einem Experten geschrieben wurde, erweitert und verändert. Wir brauchen uns aber hier nicht darum zu kümmern wie der Code in dieser anderen Klasse aussieht. Da vertrauen wir dem Experten.

Genug Theorie, weiter mit unserer Anwendung.

5 Schneller zielen, besser zielen

Wir beginnen mit 2 Änderungen. Eine Änderung wird die Kanone schneller drehen lassen. Nur 1 Grad pro Runde zu drehen, ist seeeeeeehr langsam. Wir können die Rate, mit der wir schwingen, verdoppeln, indem wir einfach die Zeile in der **run() -** Methode auf:

```
turnGunRight(2);
```

ändern.

Wir wollen auch den Code reparieren, der versucht, die Waffe auf das Ziel zu richten. Vielleicht hast Du bemerkt, dass jedes Mal, wenn wir einen Gegner im Radar haben während wir die Kanone nach links drehen, drehen wir manchmal die Kanone vom Ziel weg und verfehlen für ein, zwei Schüsse. Um das Problem zu beheben, wollen wir verhindern, dass sich die Waffe dreht, wenn wir einen Feind in unserem Scanner, auch Zielsucher oder Radar genannt, haben. Wir wollen die Kanone nur drehen, wenn kein Feind in unserem Ziel Radar ist.
Dazu brauchen wir Variablen!

Eine Variable ist ein Ort, an dem wir einen Wert speichern können, der von jedem anderen Teil unseres Codes gelesen werden kann. Also ein Stück Speicher. Wenn wir also einen Feind in unserem Scanner haben, können wir einen Wert festlegen, um zu sagen, dass der Feind erkannt wurde. Die **run()** Methode kann dann diesen Wert überprüfen und die Waffe nur drehen, wenn derzeit kein Feind gescannt wird.

Stellt euch das so vor:

Niemand in Sicht – Variable 'false'	**Feind im Scanner – Variable 'true'**
	

Da es keine Methode gibt, die uns sagt, wann ein Feind unseren Scanner verlassen hat (wenn er sich bewegt hat oder wenn er zerstört wurde), müssen wir am Ende jeder Runde davon ausgehen, dass der Feind nicht mehr da ist. Wir verlassen uns dann auf die **onScannedRobot()** Methode, die uns in jeder Runde sagt, dass der Feind immer noch da ist. Klingt schwierig? Das ist es nicht. Hier ist, wie wir das Problem lösen:

Zuerst definieren wir eine Variable. Ein guter Ort in Java eine Variable zu definieren ist direkt nach der öffnenden geschwungenen Klammer **{** mit der die Definition einer Klasse anfängt,

und noch bevor die erste Methode der Klasse beginnt. Füge den unterstrichenen Code unten am Anfang der Klasse ein:

```
public class Day2 extends JuniorRobot
{
	private boolean bScannedEnemy = false;

	public void run() {
```

Hier wieder eine kurze Erklärung, was das alles bedeutet:

- **`private`** – nur unser Roboter kann diese Variable sehen.
- **`boolean`** – Dies teilt dem Computer mit, welchen Typ Variable wir speichern möchten. Für unsere Zwecke ist ein **`boolean`** genau richtig, weil sie nur zwei verschiedene Werte haben kann: **`true`** (wahr) oder **`false`** (falsch).
- **`bScannedEnemy`** – dies ist der Name unserer Variablen. Der Name kann alles sein, was wir wollen, darf aber keine Leerzeichen enthalten. Es ist eine gute Idee, den Namen einer booleschen Variablen mit einem **`b`** zu starten. Wenn wir die Variable verwenden werden wir daran erinnert, dass die Variable vom Typ **`boolean`** ist.
- **`= false`**; mit dem Gleichheitszeichen weisen wir einer Variablen einen Wert zu. Hier weisen wir einen Standardwert (**`default`**) von **`false`** (falsch) zu. Das macht Sinn, denn wenn das Spiel beginnt, haben wir den Gegner noch nicht gescannt und wir wollen, dass sich die Waffe dreht.

Als Nächstes ändern wir sowohl die **run()** als auch die **onScannedRobot()** Methoden, um die neue Variable zu verwenden. Ich habe dem Code Kommentare (beginnend mit einem `/`) hinzugefügt, um zu erklären, was ich tue. Stellt sicher, dass Deine **run()** und **onScannedRobot()** Methoden genau wie folgt aussehen (Achtung: Java unterscheidet zwischen Gross- und Kleinschreibung, dh **`run()`** ist nicht das gleiche wir **`RuN()`**:

```
public void run() {
	// ! means NOT in Java. - bedeutet NICHT in Java
	// Also wenn wir den Feind NICHT sehen,
	// drehen wir die Kanone
	// If we have NOT scanned our enemy, turn the gun.
	// Feind nicht im Radar, wir drehen die Kanone
	if (!bScannedEnemy) {
		turnGunRight(2);
	}
	else {
		// assume we can no longer see our enemy.
		// wir nehmen an, dass wir den Feind nicht mehr sehen
		bScannedEnemy = false;
	}
}

public void onScannedRobot() {
	fire(1);
	// set the variable to true.
	// This will stop the gun from turning
	// Wir setzen die Variable auf true
	// dann stoppen wir die Drehung

	bScannedEnemy = true;
}
```

Und wieder etwas Erklärung zur Fehlersuche: Der Java Editor zeigt mit Farben die verschiedenen teile des Quellen Textes an:

```
public void onScannedRobot() {
        fire(1);
        // set the variable to true. Will stop the gun from turning.
        bScannedEnemy = true;

        }
```

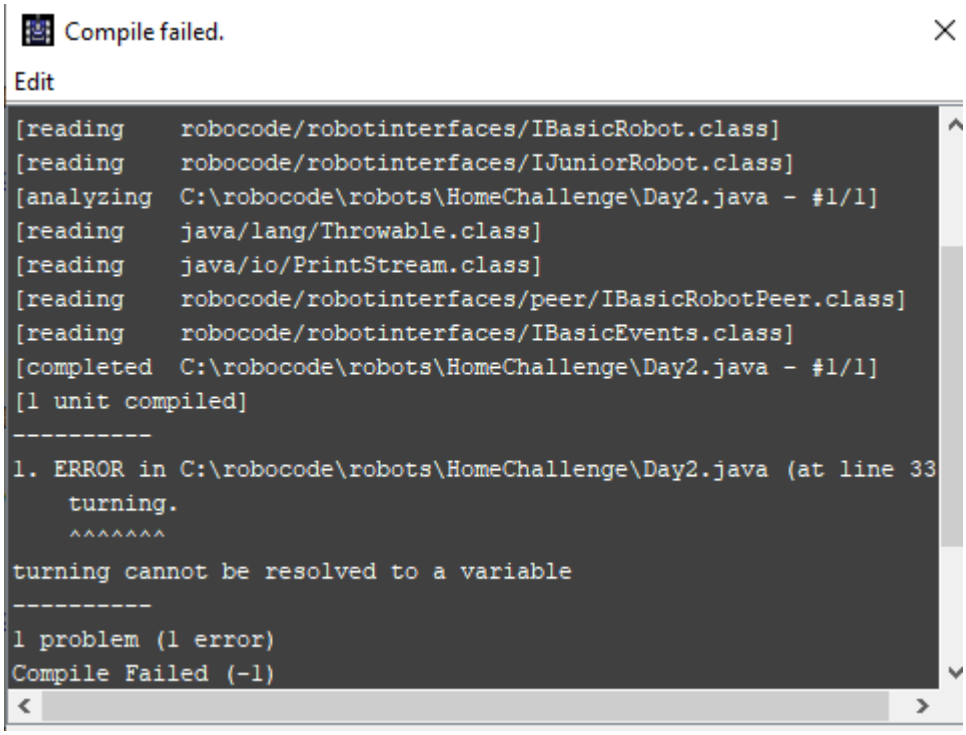
- Blau – Java Schlüssel oder reservierte Wörter
- Grün – Kommentare
- Schwarz – unser Java Code

Wenn wir also unseren Text so sehen

```
public void onScannedRobot() {
        fire(1);
        // set the variable to true. This will stop the gun from
turning.
        bScannedEnemy = true;

        }
```

Dann zeigt es uns, dass wir das Wort **`turning`** aus dem Kommentar auf eine eigene Zeile verschoben haben und jetzt als Teil unseres Java Code angesehen wird. Wenn wir jetzt den Compiler aufrufen erhalten wir eine Fehlermeldung:



Das heisst die Farben im Java Source Editor können uns helfen Fehler zu vermeiden.

In jeder Runde, in denen der Feind in unserem Radar ist, setzen wir die **`variable bScannedEnemy`** auf **`true`**. Am Ende jedes Zuges wird die **run() - Methode** die Variable erneut auf **`false`** setzen. Dies wird so lange weitergehen, bis der Feind nicht mehr erkannt wird, die Variable bleibt **`false`** und jetzt beginnt die Waffe sich wieder zu drehen.

Zeit, unseren Roboter zu testen. Speichere den Code, und kompiliere ihn. Wenn keine Kompilierungsfehler vorliegen, kannst Du zum Robocode-Hauptbildschirm gehen und einen neuen Kampf starten. Stell dieses Mal sicher, dass es einen **`Day1`** Roboter und einen **`Day2`** Roboter in der **`Selected Robots`** Liste gibt und führe den Kampf ein paar Mal aus. Beachte nun:

- Unsere Waffe dreht sich schneller und findet den Feind schneller.
- Die Kanone bleibt auch besser auf dem Ziel, was bedeutet, dass wir mehr Schlachten gewinnen.
- Wenn wir jedoch anfangen, gegen mehr als einen Roboter zu kämpfen, werden wir verlieren, weil wir uns nicht bewegen. Zeit zum....

6 Weglaufen!!

Wenn wir von einer feindlichen Kugel getroffen werden, müssen wir weglaufen, um weiteren Schaden zu vermeiden. Die Strategie, die wir verfolgen werden, besteht darin, so lange zu fahren, bis wir gegen eine Mauer stossen. Wenn wir dann wieder getroffen werden, fahren wir rückwärts, bis wir gegen eine Mauer stossen. Wir werden dies immer wieder tun und hoffen, dass unser schnelleres und genaueres Zielen und unsere Bewegung uns noch besser machen. Zeit für eine neue Variable:

```
private boolean bScannedEnemy = false;
private boolean bGoForward = true;
```

Füge den unterstrichenen Code zu Deinem Roboter hinzu. Dies ist eine weitere boolesche Variable, die unserem Roboter sagt, ob er vorwärts oder rückwärts fahren soll. Wir beginnen damit, die Variable auf **`true`** zu setzen, was den Roboter anweist, vorwärts zu fahren.

Als nächstes ändern wir unsere **`onHitByBullet()`** Methode, sie sollte jetzt so aussehen:

```
public void onHitByBullet() {
	if (bGoForward) {
		ahead(1000);
	}
	else {
		back(1000);
	}
}
```

Dies ist eine einfache Methode. Wenn unsere **bGoForward** Variable **`true`** (wahr) ist, bewegen wir den Roboter um 1000 Pixel vorwärts. Andernfalls bewegen wir ihn um 1000 Pixel zurück. **Einfach**.

Eine weitere Änderung, und die sagt was der Roboter macht, wenn er die Wand berührt. Eine Wandberührung wird mit der Methode **onHitWall()** signalisiert**.** Wir haben uns ja entschieden bei jeder Wandberührung die Richtung zu wechseln.

```
public void onHitWall() {
	bGoForward = !bGoForward;
}
```

Dieser Code ist auch ziemlich einfach. Immer wenn wir die Wand berühren, setzen wir die **variable bGoForward** auf den entgegengesetzten Wert dessen, was sie war. Wenn **bGoForward** wahr (**`true`**) war, dann ist NICHT wahr falsch (**`NOT true`** wird **`false`**). Wenn es falsch (**`false`**) war, dann ist NICHT falsch wahr (also **`NOT true`** ist **`false`**. Zur Erinnerung: **`!`** , das Ausrufezeichen, bedeutet **`NICHT`** in Java.

Zeit für einen anderen Kampf oder zwei. Speichere und kompiliere den Code erneut. Stelle sicher, dass keine Fehler angezeigt wurden. Gehe nun zurück zum Robocode-Hauptfenster und starte eine weitere Schlacht, in der ein **`Day1`**-Roboter gegen deinen neuen **`Day2`**-Roboter antritt. Führe die Schlacht ein paarmal aus.

Wie Du siehst, macht der **`Day2`** Roboter seine Aufgabe ziemlich gut und vermeidet, wenn möglich, den Angriff des **`Day1`** Roboters. Es gibt nur noch eine Sache, die unseren **`Day2`** Roboter wirklich noch stärker machen würde und uns erlauben würde, mehrere **`Day1`** Roboter gleichzeitig zu attackieren...

7 Erkenne die Koordinaten Deines Feindes

Warnung! Der nächste Teil ist knifflig und erfordert einige knifflige Mathematik. Normalerweise lernt Ihr diese Mathematik nur ab der 8. Klasse. Vielleicht können Dir Deine Eltern dies erklären und sonst kannst Du uns einfach glauben, dass die Mathematik so richtig ist.

7.1 Neue Anforderungen an unseren Roboter

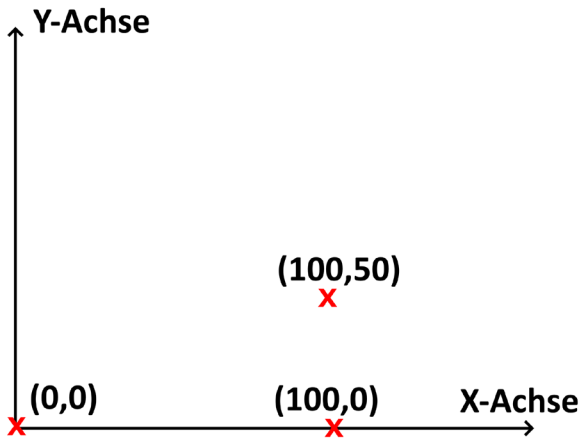
Jedes Mal, wenn wir einen Roboter scannen, wissen wir, dass er da ist. Ausserdem wissen wir, dass sich die **`Day1-`** Roboter nicht bewegen. So scheint es Zeitverschwendung, die Waffe zu drehen, nachdem wir uns bewegt haben. Wäre es also nicht besser, wenn wir unsere Waffe direkt auf die letzte Position des Gegners zeigen könnten, nachdem wir aufgehört haben, uns zu bewegen? Genau das werden wir jetzt tun. Dazu müssen wir zwei Dinge wissen:

1. Wir müssen wissen, wo sich unser Feind befindet (das nennt man die Koordinaten des Feindes) und
2. wir müssen wissen, wie man das Geschütz genau in die Richtung dreht, wo sich der Feind befindet.

Bereit!?

7.2 Etwas Geometrie

X- und Y-Koordinaten: Deine Position auf dem Schlachtfeld wird in X- und Y-Pixeln gemessen. Die untere linke Seite der Karte ist, wo sowohl X als auch Y 0 ist, d. h. die Koordinaten für die untere linke Ecke ist (0,0). Wenn Du Dich nach rechts bewegst, erhöhen



sich die X-Werte, wenn Du Dich also 100 Pixel nach rechts bewegst, sind Deine Koordinaten (100,0). Wenn Du Dich gegen oben bewegst, gegen die obere Kante des Feldes, erhöhen sich Deine Y-Koordinaten, wenn Du also 100 Pixel nach rechts und 50 Pixel nach oben verschiebst, sind Deine neuen Koordinaten (100,50).

`JuniorRobot - Variablen`: Zeit, um mehr über einige Variablen zu erfahren, die wir verwenden können. Diese Variablen sind Teil des **`JuniorRobot-Codes`,** und unser Roboter (**`Day1`**, oder **`Day2`**) erweitert den **`JuniorRobot`**. Das bedeutet, dass wir Zugriff auf alle Methoden und Variablen haben, auf die der **JuniorRobot** Zugriff hat. Hier sind die Variablen, die wir verwenden werden:

- **`heading`** – die Richtung (in Grad wobei Norden 0 ist) in die unser Tank zeigt
- **`scannedBearing`** – der Winkel des Roboters der im Zielradar gefunden wurde, relativ zum Körper unseres Roboters
- **`scannedDistance`** – wie weit (in Pixel) der gescannte Roboter für uns entfernt ist
- **`robotX`** – unsere aktuelle X-Koordinate
- **`robotY`** – unsere aktuelle Y-Koordinate

Alle Variablen und Methoden die **`JuniorRobot`** anbietet findest Du auf der Quellencode Website: https://robocode.sourceforge.io/docs/robocode/robocode/JuniorRobot.html

Jetzt da wir diese Variablen kennen, haben wir alle Zutaten, um unseren Roboter zu verbessern.

Zuerst importieren wir den Code in unseren Roboter. Das Schöne an Java ist, dass du nicht den ganzen Code selbst schreiben musst. Jemand hat wahrscheinlich schon nützlichen Code geschrieben, den wir verwenden können. Wir importieren Code namens **`Point2D`**, der uns dabei hilft, Sachen, wie zum Beispiel die Distanz zwischen zwei Dingen zu berechnen. Füge oben im Code den folgenden unterstrichenen Code hinzu:

```
package HomeChallenge;
import robocode.*;
```

```
import java.awt.geom.Point2D;
```

Nun definieren wir zwei weitere Variablen. Wir verwenden diese Variablen, um die X- und die Y-Koordinate unseres Feindes zu speichern. Wir setzen beide Variable zuerst auf -1. Für unseren Code bedeute das, dass wir noch keinen Feind gescannt haben. Wir verwenden dazu einen neuen Variablentyp namens **int**. Eine **int** (kurz für **integer**, also eine Ganzzahl) wird verwendet, um eine ganze Zahl zu speichern, wie z. B. 0 oder 100 oder -27. Wir beginnen den Variablennamen mit einem i, damit erinnern wir uns, dass es sich um ein **int** handelt. Nun füge den folgenden unterstrichenen Code zu unseren Variablen hinzu:

```
private boolean bScannedEnemy = false;
private boolean bGoForward = true;
private int iEnemyX = -1;
private int iEnemyY = -1;
```

Wir beginnen damit, die X- und Y-Koordinate des gegnerischen Panzers zu berechnen, den wir gescannt haben. Wir müssen dies in unserer **onScannedRobot()** - Methode tun.:

```
public void onScannedRobot() {
        // work out the X and Y coordinates for the enemy
        // Wir berechnen die X und Y Koordinaten unseres Feindes
        double angle = Math.toRadians((heading + scannedBearing)
                                                        % 360);
        iEnemyX = (int)(robotX + Math.sin(angle) *
                                                scannedDistance
                                            );
        iEnemyY = (int)(robotY + Math.cos(angle) *
                                                scannedDistance
                                            );

        fire(1);
        bScannedEnemy = true;
}
```

Wieder müsst Ihr den unterstrichenen Code genauso abtippen und in die **onScannedRobot**() – Methode einfügen.

Was macht der Code?

Jedes Mal, wenn ein anderer Roboter im Radar erkannt wird, wird die Methode **onScannedRobot()** aufgerufen und dann berechnen wir seine X- und Y-Koordinate und speichern diese in den Variablen **iEnemyX** und **iEnemyY!**

Wir haben also jetzt den ersten Teil unseres Ziels codiert.

Für den zweiten Teil, den Code, der die Waffe in die Richtung des Feindes dreht, müssen wir eine ganz neue Methode schreiben. Diese Methode berechnet den Winkel (in Grad) zwischen zwei Koordinaten. Wir berechnen damit den Winkel für das Geschütz basierend auf den Koordinaten unseres Roboters und den Koordinaten des Feindes.

Kopieren die ganze nächste Methode, und füge sie **direkt vor der letzten }** im Code am Ende der Datei ein.

```
// computes the absolute bearing between two points
// berechnet die absolute Richtung zwischen zwei Punkten
double absoluteBearing(double x1, double y1, double x2, double y2) {
	double xo = x2-x1;
	double yo = y2-y1;
	double hyp = Point2D.distance(x1, y1, x2, y2);
	double arcSin = Math.toDegrees(Math.asin(xo / hyp));
	double bearing = 0;

	if (xo > 0 && yo > 0) {		// beide Positionen: unten links
		bearing = arcSin;
	} else if (xo < 0 && yo > 0) { // x neg, y pos: unten rechts
		bearing = 360 + arcSin;  // arcsin ist negative
					 // richtung 360 - winkel
	} else if (xo > 0 && yo < 0) { // x pos, y neg: oben-links
		bearing = 180 - arcSin;
	} else if (xo < 0 && yo < 0) { // beide negativ: oben-rechts
		bearing = 180 - arcSin;  // arcsin ist negativ
					 // Richtung 180 + winkel
	}

	return bearing; // Die Methode liefert die Richtung zum Feind
}
// der nächste und einzige Buchstabe im Code nach der Methode
// sollte die letzte schliessende Klammer } sein!
```

Stelle sicher, dass Du den gesamten Code von oben kopierst und direkt vor der letzten } im unteren Teil des Codes einfügen.

```
47          } else if (xo < 0 && yo < 0) { // beide negativ: oben-rechts
48              bearing = 180 - arcSin;  // arcsin ist negativ
49                               // Richtung 180 + winkel
50          }
51
52          return bearing; // Die Methode liefert die Richtung zum Fein
53      }
54      // der nächste und einzige Buchstabe im Code nach der Methode
55      // sollte die letzte schliessende Klammer } sein!
56
57  }
58
```

7.3 Falls Du interessiert bist noch etwas mehr Erklärung zum Code:

Am Anfang der Methode wird der Unterschied der X und Y Koordinaten berechnet:

```
double xo = x2-x1;
double yo = y2-y1;
```

Was bedeutet unser Code und Kommentar?

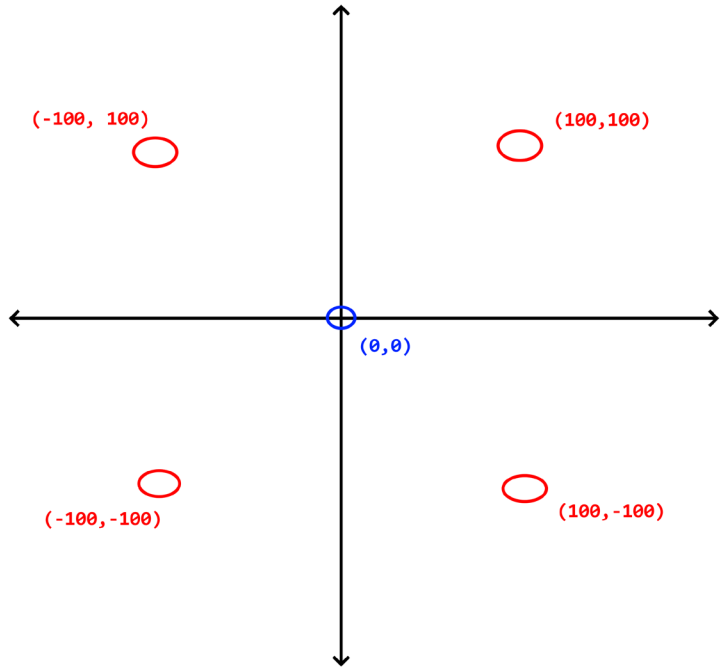
```
if (xo > 0 && yo > 0) {          // beide Positionen: unten links
```

In der folgenden Darstellung des Koordinatensystems stellen wir uns auf Position (x1, y1) und zeigen die 4 verschiedenen möglichen Richtungen von uns in der sich der andere Roboter befinden kann, der sitzt an Position x2, y2.

Zur Vereinfachung nehmt einmal an wir sitzen auf Position (x1,y1) = (0,0) und der Feind auf einer der vier Positionen

Feind Position (x2,y2)	**X0**	**Y0**	**Wo sitzen wir relativ zum Feind? dh der blaue Tank relativ zum roten**
(100,100)	100	100	Unten Links
(-100, 100)	-100	100	Unten Rechts
(100, -100)	100	-100	Open Links
(-100, -100)	-100	-100	Oben Rechts

Oder im Koordinatensystem sieht das so aus:



Alles klar? Schaut den Code nochmals an, was passiert, wenn der Rote Tank auch auf Position (0,0), also auf der gleichen Position wie wir, sitzt. Was ist dann die Richtung (**`bearing`**)?

7.4 Wir verwenden den Code

Jetzt werden wir diese neue Methode verwenden, um die Waffe zu drehen. Wir werden die Waffe drehen, sobald wir eine Wand getroffen haben, also in der **`onHitWall()`** Methode. Suche die Methode **`onHitWall()`**, die sollte eigentlich unmittelbar vor der neuen Methode die wir soeben geschrieben haben zu finden sein. Füge dem Code von **`onHitWall()`** die folgenden unterstrichenen Zeilen hinzu:

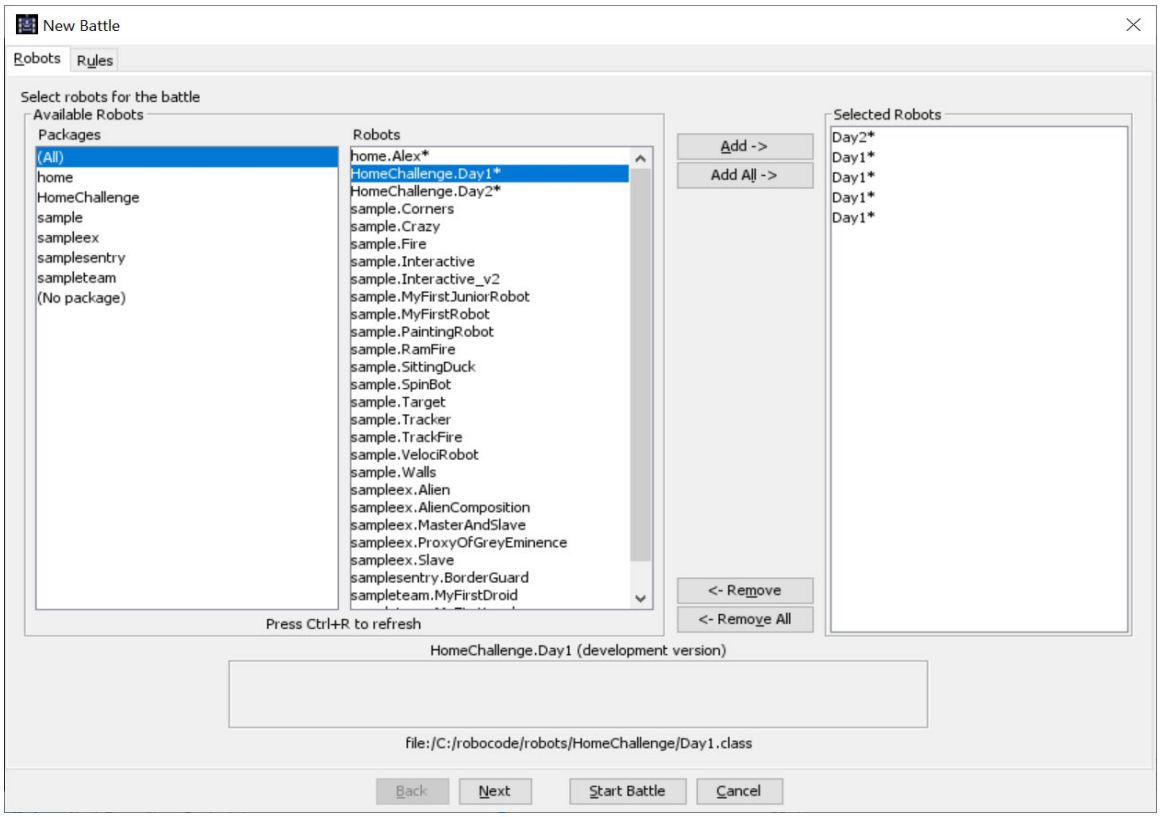
```
public void onHitWall() {
        bGoForward = !bGoForward;
        // if we have a known location for the enemy, point our gun at
it.
        // wir kennen die Position des Feindes
        // richte die Kanone auf ihn
        if (iEnemyX >= 0) {
                turnGunTo((int)absoluteBearing(robotX, robotY,
                                                iEnemyX, iEnemyY));

        }
}
```

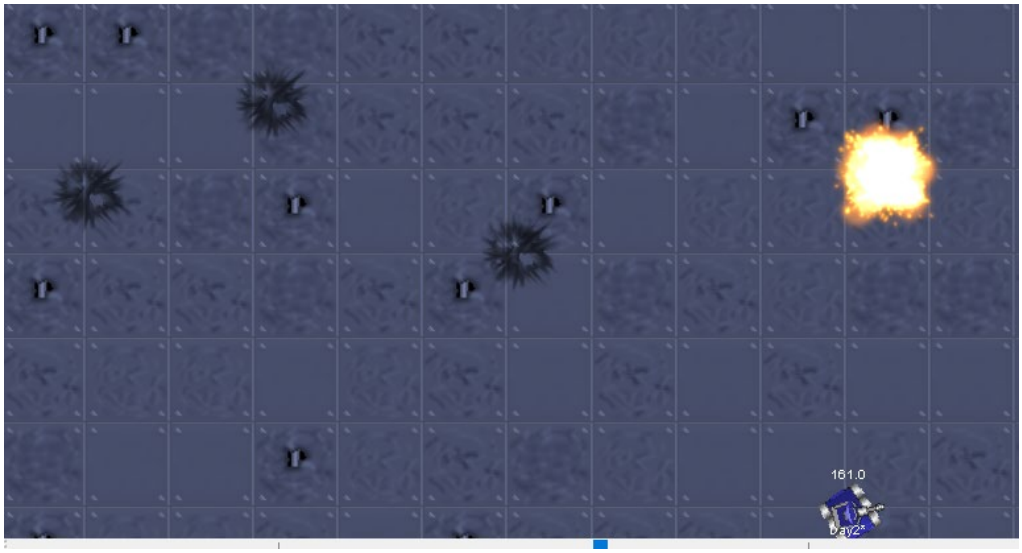
Stelle nochmal sicher, dass Du alles richtig abgetippt hast; hast Du Gross- und Kleinschreibung beachtet! Speichere und kompiliere den Code. Behebe etwelche Fehler. Nun ist es Zeit, den neuen Roboter zu testen. Starte einen neuen Kampf zwischen einem **`Day1`** Roboter und einem **`Day2`** Roboter. Führe den Kampf einige Male aus. Siehst Du wie effizient unser neuer Roboter ist?

8 4 gegen 1 – Was für ein unfairer Kampf!

Jetzt lassen wir unseren **`Day2`** Roboter gegen 4 **`Day1`** Roboter kämpfen. Starte eine neue Schlacht, aber jetzt füge einen **`Day2`** Roboter und 4-mal einen **`Day1`** Roboter zu den gewählten Robotern. Dein Schirm sollte so aussehen:



Starte die Schlacht. Führe sie ein paarmal aus. Wieviel besser ist unser **`Day2`** Roboter? Überlebt er immer?



Drei Krater, eine Explosion und unser **`Day2`** Roboter unten rechts!

Können wir einen Roboter bauen, der unseren **`Day2`** Roboter schlagen kann? Wir werden es beim nächsten Mal herausfinden.

9 Mehr Ressourcen

Untenstehend ein Link zur Robocode-Dokumentation, insbesondere sind hier alle Methoden und Variablen, die Dir für die JuniorRobot-Klasse zur Verfügung stehen dokumentiert:

https://robocode.sourceforge.io/docs/robocode/index.html?robocode/JuniorRobot.html