



In diesem Programmierkurs sollst du einen Kernkraftwerk-Rescue-Roboter erstellen, der in der Lage ist, bei einem Atomunfall in einem Kernkraftwerk die gefährlichen Arbeiten zu übernehmen. Bis dahin ist aber noch ein weiter Weg ...

Zunächst lernst du, im Robo-Trainingsgebäude den Roboter zu steuern. Dabei experimentieren wir vorerst mit harmlosen Schrauben und nicht mit radioaktiv verseuchten Gegenständen.

ZIEL: Objekte in **Greenfoot** erzeugen können, ihre Fähigkeiten erkennen und nutzen können.

Aufgaben:

- 1 Kopiere die Rohfassung des Robot-Projekts (ReaktorRobot_Szenario_Roh) und benenne den Ordner um (z.B. in ReaktorRobot_Arbeitsfassung). Starte das Szenario nun in Greenfoot (Szenario → Öffnen). Steuere den Roboter **AB1** mit passenden Befehlen aus dem Kontextmenu (Rechtsmausklick auf den Roboter in der Welt) so, dass er gegen eine Wand läuft.
Steuere den Roboter **AB1** nun so, dass er alle drei Schrauben im Raum links oben aufnimmt. Sorge dafür, dass der Roboter anschließend eine dieser Schrauben wieder ablegt.
- 2 Erzeuge nun einen zweiten Roboter vom Typ **AB1** (siehe Bild rechts: Rechte Maustaste auf AB1, dann `new AB1()`), setze ihn den Raum unten links und lasse ihn die dort abgelegten Schrauben aufnehmen. Rufe anschließend bei beiden Robotern nacheinander jeweils die Anweisung `getAnzahlSchrauben()` auf. Was passiert?
- 3 Rufe den Befehl `istAufSchraube()` an jedem Roboter auf. Welche Antworten sind möglich?
- 4 Was geschieht, wenn du einen Roboter aufforderst, eine Schraube abzulegen, obwohl er keine bei sich hat oder eine Schraube abzulegen, wenn schon eine da liegt?
- 5 Wie kannst du neue Wände erzeugen und in die Welt legen? (Was passiert, wenn du nach dem Ablegen die Shift-Taste drückst?) Kannst du sie überall hinbauen?
- 6 Erprobe alle Fähigkeiten der Roboter.
Was musst du tun, damit ein Roboter auf die Anfrage `istVorratLeer()` die Antwort `false` liefert?

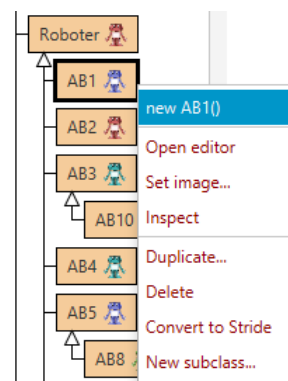


Abb 1: Kontextmenü der Klasse AB1



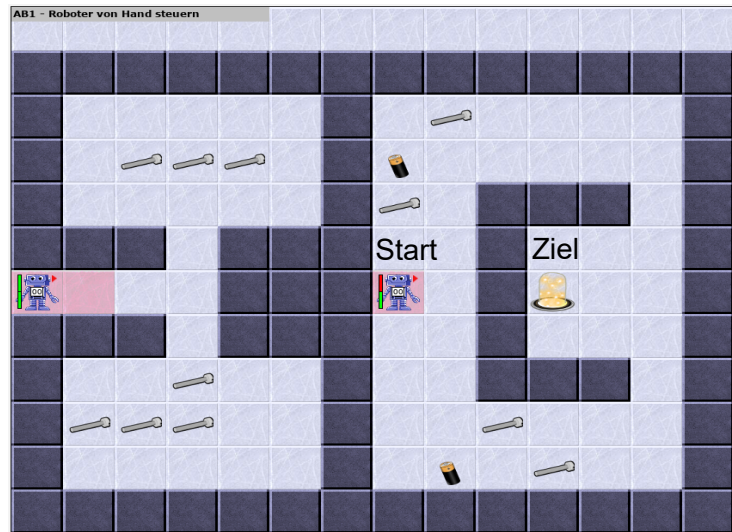
Leveltest: Das erste Training

Drücke auf „Reset“, um das Level neu zu starten.



Nimm mit dem rechten Roboter durch Aufruf der richtigen Befehle im Kontextmenü **zwei** der vier Schrauben auf und laufe zum Portal.

Da es ein weiter Weg ist, wird deine Energie nicht reichen. Sammle deswegen auch Akkus unterwegs auf und benutze sie.



Zusammenfassung: Du kannst Greenfoot starten, ein Szenario laden, Objekte erzeugen und nutzen, einem Roboter über sein Kontextmenü Befehle geben.

Bildquellen: Die verwendeten Bilder des Roboterszenarios sind alle ohne Bildnachweis verwendbar (selbst gezeichnet, Pixabay Lizenz oder Public Domain). Genaue Nachweise: siehe [bildquellen.html](#).



Du hast das erste Level gemeistert. Allerdings hast du den Roboter bisher von Hand gesteuert. Zwischenzeitlich hat sich der erste Störfall ereignet. Möchtest du den Roboter „handgesteuert“ in das Kernkraftwerk leiten??? Sicher nicht. Daher muss der Roboter schnell lernen, sich alleine zu bewegen.

Die Roboter lernen dazu ...

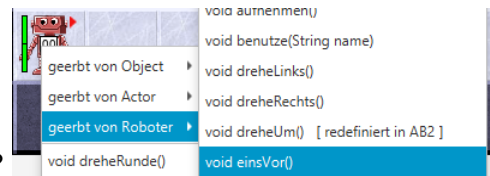
ZIEL: Wissen, dass alles, was die Roboter ausführen können, im Quelltext notiert ist. Vorhandene Quelltexte ergänzen und erweitern können.

Aufgaben:

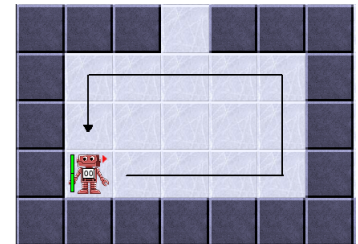
- 1 Welche Befehle bietet dir ein **AB2**-Roboter in seinem Kontextmenü direkt an? Öffne nun den Quelltext; diesen kannst du dir mit einem Rechtsklick auf die AB2-Klasse im rechten Fenster mit „Editor öffnen“ (oder alternativ Doppelklick auf die Klasse AB2) anzeigen lassen. Jede Fähigkeit ist in einer sogenannten Methode im Quelltext beschrieben. Findest du die Befehle des AB2-Roboters?



- 2 Steuere den Roboter noch ein letztes Mal von Hand durch einzelne Befehle so, dass er eine Runde dreht wie im Bild (siehe rechts). Die bekannten Befehle findest du jetzt im Kontextmenu unter „geerbt von Roboter“. Welche Befehle hast du ihm dazu gegeben?

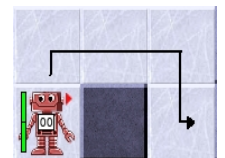


- 3 **Drehe Runde im Raum:** Öffne den Quelltext der Klasse **AB2**. Ergänze die Anweisungen in `dreheRunde()`, damit es eine vollständige Runde wird. Es wird danach auch möglich sein, mehrmals nacheinander den Befehl "dreheRunde" aufzurufen. Nach jedem Befehl musst du einen Strichpunkt setzen. (Hinweis: in rosa und grau findest du sogenannte Kommentare. Das sind Hinweise für dich und haben für den Roboter keine Bedeutung)



Übersetze und erprobe die veränderte Methode.

- 4 **Drehe um:** Schreibe im Quelltext die Anweisungen für `dreheUm()`.
- 5 **Sammele 3 Schrauben:** Bringe dem **AB2** bei, drei Dinge aufzusammeln, die direkt hintereinander liegen. Wir üben wieder im Testgelände mit Schrauben, aber im Kernkraftwerk werden es herumliegende Brennstäbe sein. Mit dem Befehl `aufnehmen()`; kannst du beliebige Dinge einsammeln (sofern sie sich tragen lassen). Dafür ist im Quelltext schon einiges vorbereitet. Ergänze die Zeilen für `sammle3()` an der passenden Stelle. Teste die neue Methode, indem du beim Roboter vor den drei Schrauben die Methode aufrufst.
- 6 **Haken schlagen:** Sorge dafür, dass die **AB2**-Roboter auch frei stehende Wände umlaufen können (s. Bild rechts). Wie nennst du diese Fähigkeit eines **AB2**-Roboters? Das wird auch der Name der Methode, die du im Quelltext beschreibst. Der Methodenname sollte mit einem Kleinbuchstaben beginnen. Neue Methoden müssen immer mit `public void methodenname()` beginnen. Die Befehle der Methode werden dann in `{}`-Klammern eingeschlossen. Schau dieses Konzept bei den bestehenden Methoden ab. Wenn ein **AB2**-Roboter genau vor einer Wand steht und einen Haken schlagen möchte, muss er sich z.B. nach links drehen, einen Schritt vor gehen, nach... Das kriegst du selbst raus. Erprobe deine neue Methode.
- 7 **Vier sammeln:** Bringe den **AB2**-Robotern bei, eine Reihe von vier beliebigen Gegenständen aufzusammeln. Erprobe deine Methode am Roboter oben rechts.





Verändere Deine Methode ggf. so, dass sie den Befehl `sammle3()`; verwendet.

8 Falsche Namensgebung:

a) Genau zwei Namen sind unzulässig. Welche vermutest du? Warum?

`linksUm()`; `vor()`; `vierVor()`; `legeAb()`; `legeSpur()`; `linksum()`; `links um()`; `hebeAuf()`; `einsVor()`; `rechtsUm()`; `dreheUm()`; `lege3Ab()`; `sammle3Blaetter()`; `schiebeBaum()`; `zickzack()`; `1Vor()`;

Einsatz 2: Notfall im Kernkraftwerk, atomare Verseuchung befürchtet!

Der Kraftwerksarbeiter, der als letztes das Kraftwerk verlassen hat, beschreibt die Situation wie folgt:

Vom Eingang muss man drei Schritte vorgehen. Dann dreht man sich nach links und geht nochmals drei Schritte, glaube ich. Dann steht man in einem Raum und direkt vor einer Säule. Vor der Säule liegt ein Akku. Hinter der Säule liegen in einer Reihe ein Akku und zwei Brennstäbe. Aber ich kann mich nicht mehr erinnern in welcher Reihenfolge. Wenn man sich dann nach rechts dreht und zwei Schritte geht, kommt man zum Notausgang. Glaube ich zumindest... Bitte helfen Sie mir. Die Brennstäbe müssen unbedingt eingesammelt werden!!!



Implementiere für diesen Einsatz die Methode `einsatz2()` im Quelltext. Rufe dazu die zuvor erstellten Methoden in der richtigen Reihenfolge auf. Für den Methodenaufruf musst du nur ihren Methodennamen mit der ()-Klammer dahinter hinschreiben (z.B. `dreheUm()`;) Ergänze ggf. weitere Befehle (z.B. `einsVor()`;) (Hinweis: du brauchst insgesamt mindestens 11 Befehle!)

Um den Einsatz durchzuführen, musst du in der Roboter-Welt mit der rechten Maustaste auf einem beliebigen Gangfeld (grauer Hintergrund) die Methode `einsatz_02()` aufrufen (nicht direkt beim Roboter!).

Ich bin mal gespannt, ob du deinen ersten richtigen Einsatz bewältigst!

Falls du mit dem Einsatz Schwierigkeiten hat, kann dir dein Lehrer weiter helfen.

Zusammenfassung: Du kannst nun Programmieren – d.h. Methoden mit Anweisungen füllen. Dadurch kannst du Robotern Befehle geben, die sie dann selbständig ausführen!

Bildquellen: Die verwendeten Bilder des Roboterszenarios sind alle ohne Bildnachweis verwendbar (selbst gezeichnet, Pixabay Lizenz oder Public Domain). Genaue Nachweise: siehe [bildquellen.html](#).



In einem anderen Kraftwerk ist ein Feuer ausgebrochen. Deine ersten Erfolge haben sich rumgesprochen. Daher wird deine RoboRescue-Firma angesprochen, ob sie auch in diesem Fall helfen kann. Natürlich bist du sofort bereit... aber wie löscht man ein Feuer?

Die Roboter machen vieles immer wieder ...

ZIEL: Wiederholungen in Handlungen erkennen, als SOLANGE-Schleife formulieren und in Programmiersprache umsetzen können. Methoden mit Parametern benutzen können.

Aufgaben:

1. **Löschen:** Der Roboter **AB3** ganz oben links soll den vor ihm stehenden Feuerlöscher aufnehmen, damit auf das Feld vor dem Feuer (nicht auf das Feuer!!!) gehen und es löschen. Führe dies zunächst von Hand durch. Via Rechtsklick unter „geerbt von Roboter“ findest du auch die Methode "benutze", um den Feuerlöscher zu benutzen.

Hinweis: Es gibt Methoden, die eine Zusatzinformation (einen sogenannten Parameter) benötigen, um korrekt arbeiten zu können. Die Methode „benutze(String name)“ ist so eine. Es wird also der Name des Gegenstandes erwartet, der benutzt werden soll. Dieser ist ein String (=Text). Strings müssen in Anführungszeichen gesetzt werden. Um den Feuerlöscher zu benutzen, muss man also `benutze("Feuerloescher");` schreiben.

Vervollständige dann die Methode `loesche()` im Quelltext, die dies alles automatisch für den oberen Roboter machen soll.

2. Die weiteren Roboter darunter sollen ihre ersten Feuer in der Reihe auch mit dem Befehl `loesche()` löschen können. Diese Feuer sind aber in einer anderen Entfernung platziert. Überlege dir, welche Zeilen deines Programms überarbeitet werden müssen. Worauf muss der Roboter reagieren können?
3. **VorBisFeuer:** Gib jedem Roboter den Auftrag: `vorBisFeuer()` und beobachte, was sie tun. Lies danach im Quelltext bei der Methode `vorBisFeuer()` die Anweisungen und versuche sie zu verstehen. Hinweis: das Ausrufezeichen bedeutet „nicht“. Setze diese Methode sinnvoll in der `loesche()`-Methode ein, indem du einige Zeilen deines Quelltextes durch den Befehl `vorBisFeuer()`; ersetzt. Teste nun die neue `loesche()`-Methode an allen vier Löschrobotern.
4. Ergänze testweise die Methode `vorBisFeuer` so, dass die Roboter danach noch einen Schritt ins Feuer hinein machen. Was passiert dann? Was passiert, wenn du innerhalb der lila Farbhinterlegung des Quelltextes (also innerhalb der `{ }`-Klammern) noch ein zweites `einsVor()`; einfügst? Entferne die Veränderungen wieder.

Der Auftrag `vorBisFeuer()` muss korrekt ausgeführt werden, egal wie weit das Feuer entfernt ist. Daher muss der Roboter prüfen, wie lange er vorwärts gehen muss. Auf die Anfrage: `istVorne("Feuer")` liefert er die Antwort `true` bzw. `false`. Damit können wir diese bedingte Wiederholung so formulieren:

-- normierte Sprache	-- Programmiersprache	-- Struktogramm
SOLANGE (vorne kein Feuer ist) WDH: ein Schritt vor ENDE WDH	<pre>while (!istVorne("Feuer")) { einsVor(); }</pre>	

Genau so steht es auch im Quelltext. Im runden Klammerpaar hinter dem Schlüsselwort **while** (...) steht die Bedingung, welche die Wiederholungen steuert. Solange diese Ausführungsbedingung gilt, werden alle Anweisungen im Schleifenblock zwischen `{` und `}` wiederholt.



Oder anders ausgedrückt: Die Ausführungsbedingung wird überprüft. Ist sie wahr, so wird jede Anweisung in der Blockklammer zwischen `{` und `}` ausgeführt. Danach wird wieder überprüft, ob die Ausführungsbedingung immer noch wahr ist. Die Anweisungen innerhalb der Blockklammer werden wieder ausgeführt. Und so weiter und so fort. Erst wenn die Ausführungsbedingung falsch ist, wird die Schleife beendet und die Befehle hinter der schließenden Klammer `}` ausgeführt. Innerhalb des wiederholten Vorgangs muss sich die Ausführungsbedingung verändern, damit die Wiederholungen schließlich aufhören und nicht endlos laufen.

5. **Feuersbrunst löschen:** Die Feuer des 3. und 4. Roboters bestehen aus mehreren Flammen hintereinander (ohne Lücke dazwischen). Um diese zu löschen, muss man immer wieder löschen, dann einen Schritt gehen, dann wieder löschen usw..

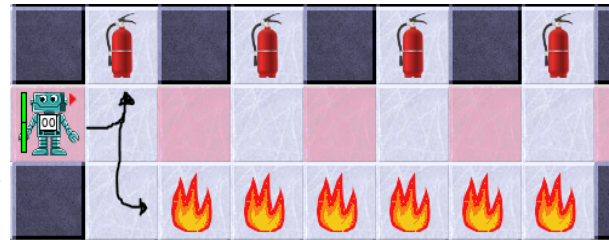


Das wiederholt man so lange, wie nach dem Schritt noch ein Feuer vor dem Roboter ist.

Gehe davon aus, dass der Roboter schon einen Feuerlöscher hat und vor dem ersten Feuer steht. Implementiere eine Methode `loescheReihe()`, die dann alle Feuer löscht. Verwende dazu eine While-Schleife. Teste die Methode, indem du von Hand den Feuerlöscher aufhebst und zum Feuer läufst. Rufe dann die Methode `loescheReihe()` auf.

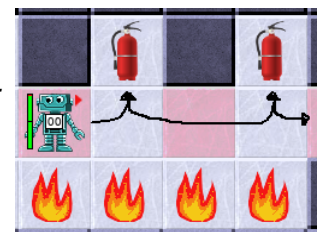
Ersetze anschließend in deiner `loesche`-Methode das Benutzen des Feuerlöschers durch die neue Methode `loescheReihe()`. Teste die Veränderung.

6. **Reichweite testen:** Mit einem Feuerlöscher kommt man nicht weit. Die sind recht schnell leer. Schreibe eine Methode `testeReichweite()`, bei der der Roboter unten links, den ersten Feuerlöscher einsammelt, zur Feuerspur darunter geht und dort so viel Feuer wie möglich löscht. Verwende eine while-Schleife, um zu überprüfen, ob der Feuerlöscher noch nicht leer ist.



Hinweis: Die Bedingung (`getAnzahl("Feuerloescher") >= 1`) testet, ob der Roboter noch mind. 1 Löscher hat. Dies testet automatisch, ob er leer ist, da er einfach aus dem Inventar des Roboters verschwindet, wenn er verbraucht ist.

7. **Feuerlöscher einsammeln:** Für ein großes Feuer braucht man mehrere Feuerlöscher. Vor dem Roboter unten links ist ein Gang mit vielen Feuerlöschern in den Nischen (drücke ggf. unten auf den Reset-Knopf). Er soll nach vorne bis zur Wand laufen und dabei alle Löscher einsammeln. Das soll funktionieren, egal auf welchem orangenen Feld er startet. Er muss aber nur die Löscher einsammeln, die sich nach seiner Startposition befinden.



Implementiere eine Methode `sammleLoescher()`, die zunächst den Roboter bis zur Wand laufen lässt, ohne die Löscher einzusammeln. (Die Methode `istVorneFrei()` testet, ob vor dem Roboter keine Wand ist). Ergänze die Methode dann so, dass er die Löscher einsammelt. Teste verschiedene Startpositionen (irgendwo auf einem roten Feld), indem du neue AB3-Roboter erzeugst und auf einem der roten Startfelder platzierst.

(Tipp: über den Geschwindigkeitsregler unten rechts kannst du die Ausführungsgeschwindigkeit anpassen!)

8. **Homerun:** Lasse den Roboter rechts unten in die Sackgasse („Schneckenhaus“) laufen.
a) Implementiere dazu zunächst die Methode `laufeBisWand()`, die den Roboter bis zur nächsten Wand bewegt.

Tipp: Verwende eine While-Schleife mit `istVorneFrei()` als Bedingung.

b) Implementiere `laufeBisSackgasse()`, indem du zunächst `laufeBisWand()` aufrufst und dann eine while-Schleife verwendest, um zu testen, ob du noch mal bis zur Wand laufen musst oder schon in der Sackgasse angekommen bist. Dies erkennst du, indem du testest, ob links keine Wand ist (`!istWandLinks()` - das ! steht für „nicht“).

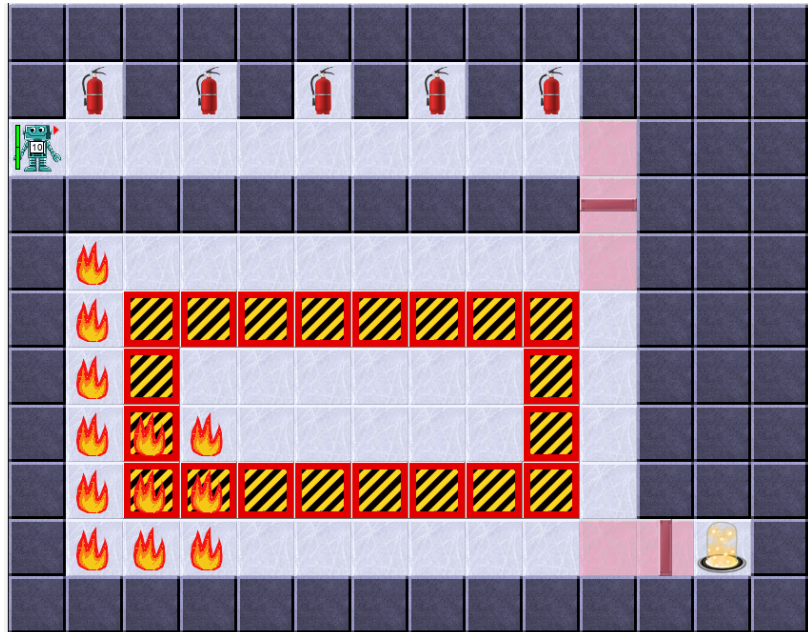
Bildquellen: Die verwendeten Bilder des Roboterszenarios sind alle ohne Bildnachweis verwendbar (selbst gezeichnet, Pixabay Lizenz oder Public Domain). Genaue Nachweise: siehe bildquellen.html.



Einsatz 3: Feuerlöschen im Kernkraftwerk

„Oh mein Gott, in Kernkraftwerk Fessenheim ist ein Feuer ausgebrochen. Sie müssen unbedingt helfen. Wenn das Feuer nicht schnell gelöscht wird, gibt es eine Katastrophe. Unsere Arbeiter können wir nicht mehr rein schicken. Die Gefahr ist einfach zu groß.“

Wenn man reinkommt, gibt es gleich links eine Reihe von Nischen, in der die Feuerlöscher hängen. Sammeln Sie am besten alle ein. Wir wissen nicht, wie groß das Feuer schon ist. Gehen Sie dann in die große Halle. Löschen Sie dort das Feuer. Es ist in der linken unteren Ecke ausgebrochen. Wie weit es sich inzwischen ausgebreitet hat, müssen Sie selbst herausfinden (Hinweis: in jeder Zeile ist direkt vor der Wand mindestens eine Flamme und es gibt keine Lücken im Feuer).



Rechts unten befindet sich ein Schutzraum (mit Portal), in das sich der Roboter am Ende begeben muss. Schaffen Ihre Roboter das?

Ach so, der Weg ist übrigens weit – nutzen Sie die Akkus, die der Roboter für die Mission bei sich hat!“

Tipps:

- Versuche zunächst, nur die oberste Reihe Flammen zu löschen und danach wieder nach rechts zur Wand zu laufen. Drehe den Roboter wieder so, dass er nach unten blickt (Ausgangsstellung).
- Welche der benutzen Befehle müssen wiederholt werden, damit der Roboter die nächste Zeile löscht? Wiederhole diese mit einer While-Schleife, solange das Portal noch nicht erreicht ist (warum erreicht man es automatisch?).


Zusammenfassung: Du kannst Algorithmen formulieren und im Quelltext notieren, die eine oder mehrere Anweisungen solange wiederholen, wie eine Ausführungsbedingung gilt. Als Bedingung eignet sich alles, was wahr oder falsch sein kann. Wir verwenden oft einen Fragebefehl wie `istVorneFrei()`, dessen Ja/Nein-Antwort die Wiederholung steuert.

Fragebefehle wie `istVorratLeer()` oder `istWandVorne()`, die alle Roboter entweder mit wahr oder mit falsch beantworten können, eignen sich als Ausführungsbedingung.

Im Quelltext schreibt man Wiederholungen mit dem Schlüsselwort **while** im Schleifenkopf und dahinter eine Ausführungsbedingung in runder Klammer() sowie einen Schleifenrumpf innerhalb des Blockklammerpaares {...}.

```
while ( Ausführungsbedingung ) {
    // zu wiederholende Anweisung;
    // zu wiederholende Anweisung;
    ...
}
```

Die Ausführungsbedingung muss im Laufe der Wiederholungen einmal falsch werden, damit es keine Endlosschleife gibt.

Falls es dennoch mal eine Endlosschleife gibt, kann man diese mit dem Knopf  unten rechts unterbrechen.



Einlasskontrollen: Bisher hatten Sie Glück. Die Mitarbeiter konnten die Wege recht genau beschreiben. Der Roboter wusste immer, was zu tun ist. So ist das aber nicht immer. Manche Türen gehen von allein auf. Andere müssen mit einem Schlüssel aufgeschlossen werden. Oder es ist gar eine Strombarriere im Weg, die durch einen Schalter deaktiviert werden muss. Gänge knicken unvorhergesehen nach links oder rechts ab. Da muss der Roboter selbstständig Entscheidungen treffen können.

Die Roboter treffen Entscheidungen ...

ZIEL: Alternativen in Handlungen erkennen, als FALLS-DANN-SONST-Entscheidungen formulieren und in Programmiersprache umsetzen können.

In vielen Situationen sind Anweisungen nur unter bestimmten Bedingungen auszuführen. Um z.B. ein Muster zu invertieren, muss ein Roboter dort, wo eine Schraube liegt, diese aufheben und dort, wo keine liegt, eine ablegen. Dazu muss er prüfen, ob eine Schraube da liegt oder nicht und jeweils passend handeln. Die Prüfung erfolgt in einer Prüfbedingung, die nur wahr oder falsch sein kann.

-- normierte Sprache	-- Programmiersprache	-- Struktogramm
FALLS (Schraube ist darunter) DANN aufheben *ENDE DANN SONST ablegen *ENDE SONST	<pre>if (istAufGegenstand("Schraube")) { aufnehmen(); } else { ablegen("Schraube"); }</pre>	<pre> graph TD A[aufSchraube()] -- ja --> B[aufheben] A -- nein --> C[ablegen] </pre>

Vorsicht: statt 'if (Prüfbedingung) then {... ' steht nur: 'if (Prüfbedingung) {... '.

Man nennt solche Entscheidungen in der Programmierung auch **Verzweigungen**.

Aufgaben:

- Invertierer:** Der Roboter links unten steht in einer Reihe mit vielen Schrauben. (Zur Sicherheit üben wir erst mal wieder mit Schrauben, wer weiß, was der Roboter anstellt...). Analysiere die beiden Methoden `tausche()` und `tauscheUndVor()`: Wie verhält sich der Roboter, falls er auf einer Schraube steht? Wie verhält er sich, falls er auf einem leeren Feld steht? Wozu ist in `tauscheUndVor()` die Entscheidung `if(istVorneFrei())`... verwendet worden? Wie unterscheidet sich die Form der Entscheidung in den beiden Methoden?
- Wiederholung der while-Schleife:** Jedes Mal, wenn du auf „Reset“ drückst, entsteht eine neue Schraubenreihe. Implementiere eine Methode `tauscheBisWand()`, die in einer while-Schleife so lange `tauscheUndVor` aufruft, wie die Wand noch nicht erreicht ist. Versichere dich, dass auch direkt vor der Wand getauscht wird.
- Fleckenfrei:** Ölflecken schaden den Robotern. Sie verlieren die Haftung auf dem Boden. Dadurch verlieren sie Energie. Der Roboter soll um die Ölflecken herum laufen. Mit `istVorne("Oelfleck")` kann er überprüfen, ob vor ihm ein Ölfleck ist. Implementiere eine Methode `umgeheOelfleck()`, die den Roboter einen Schritt nach vorne gehen lässt, falls kein Ölfleck vor ihm ist. Ansonsten läuft er um den Ölfleck drumherum. Erweitere diese Methode so, dass der Roboter bis zur Wand läuft.
- Schlüssel aufheben:** Schreibe eine Methode `sammleSchluessel()`, die einen Schlüssel aufnimmt, falls der Roboter auf einem steht. Falls nicht, soll er nichts machen. Teste diese Methode an den beiden Robotern oben links.
- Sesam öffne dich:** Implementiere eine Methode `oeffneTuer()`, die die beiden Roboter oben links aus ihren Gängen heraus führt. Der Roboter soll einen Schlüssel aufheben, falls er an

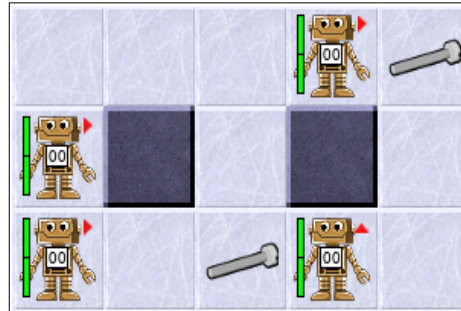


der Startposition liegt. Danach soll er drei Schritte vor gehen, und sich nach links drehen. Falls er vor einem Schloss steht (istVorne("Schloss")), soll er den Schlüssel benutzen (benutze("Schluessel")), andernfalls den Schalter benutzen (benutze("Schalter")). Dann sollte sich die Tür bzw. die Elektrobarriere öffnen. Anschließend soll der Roboter zwei Schritte vor gehen, um hinaus zu treten.

6. Korrigiere die beiden Schreibfehler! Dieser Quelltext wird so nicht übersetzt, sondern mit einer Fehlermeldung zurückgewiesen. Der zweite Fehler führt zwar nicht zu einer Fehlermeldung, ist daher aber noch schwieriger zu finden.
7. Zeichne für jeden der vier AB4-Roboter ein, wie sie sich bewegen, wenn sie diese Anweisungen ausführen:

```
if (aufSchraube() {
    schraubeAufnehmen();
}
if (istVorneFrei()); {
    einsVor();
}
```

```
if (!istVorneFrei()) {
    dreheLinks();
    einsVor();
    dreheRechts();
}
else {
    einsVor();
}
dreheRechts();
```



8. a) Nenne zwei Bewegungsbefehle, die ein **Roboter** ausführen kann und zwei Fragebefehle, die er beantworten kann.
- b) Gib an, welche Ausdrücke von A bis E nicht als Prüfbedingung in einer Entscheidungsanweisung **if (...)** benutzt werden kann. Dabei sind x, alter und anzBlaetter Variablen, die für Zahlen stehen.

A: (!istVorneFrei())

B: (x>5)

C: (alter)

D: (anzBlaetter<=7)

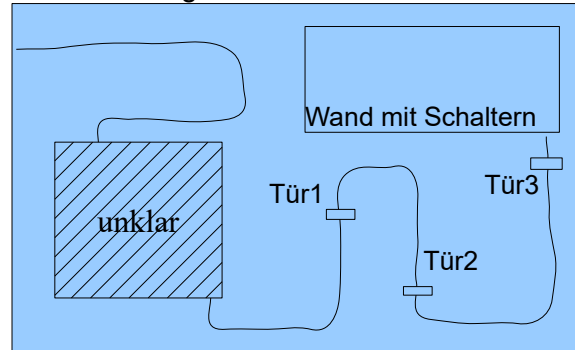
E: (einsVor())

9. **Labyrinth:** Der Roboter in der Mitte soll sich in einem Gang alleine zurecht finden. Der Gang hat keine Verzweigungen, knickt aber immer wieder nach links oder nach rechts ab. Implementiere eine Methode, die den Roboter bis zur Sackgasse laufen lässt:
 - Stelle den Roboter zunächst vor eine Wand. Sorge dafür, dass er sich nach links dreht, falls links frei ist. Falls rechts frei ist, nach rechts. Sonst soll er einfach stehen bleiben.
 - Erweitere deine Methode, indem du den Roboter an einer beliebigen Stelle vor der Wand starten lässt und dann die Methode laufeBisWand() benutzt.
 - Lasse diese beiden Schritte solange wiederholen, wie die Sackgasse noch nicht erreicht ist. Um eine Sackgasse zu erkennen, benötigst du komplexere Bedingungen (siehe dazu auch Präsentation 2_Ablauf_von_Schleifen.odp und dort die letzte Folie). Du kannst mehrere Bedingungen mit „und“ (in Java &&) bzw. „oder“ (in Java ||) verbinden. So kannst du beispielsweise mit istWandVorne() && istWandLinks() && istWandRechts() überprüfen, ob dein Roboter in einer Sackgasse steht. Das Gegenteil – also „keine Sackgasse“ – erreicht man, indem man die ganze Bedingung einklammert und ein „nicht“ davor setzt.



Einsatz 4: Gelangen Sie in den Kontrollraum und schalten Sie das Kernkraftwerk ab.

Die Mannschaft hat das Kernkraftwerk fluchtartig verlassen und leider die elementaren Sicherheitsvorkehrungen vernachlässigt. Sie haben vergessen, das Kernkraftwerk herunterzufahren. Dazu müssen im Kontrollraum alle Schalter umgelegt werden. Der Kontrollraum befindet sich irgendwo versteckt am Ende eines langen Ganges, der durch drei Türen gesichert ist. Jede Tür ist durch einen Schalter oder ein Schloss direkt links neben der Tür zu bedienen. Hier haben Sie die nötigen Schlüssel. Im Kontrollraum gibt es an der Wand links neben dem Eingang zahlreiche Schalter. Wie viele es sind und wo genau sie sich befinden, weiß ich nicht. Schalten Sie alle aus, dann haben Sie Ihren Auftrag erfüllt.



Tipps:

- nutze die Labyrinth-Methode, um zu der Tür 1 zu gelangen. Öffne diese.
- es sind genau drei Türen. Daher kannst du die Befehle einfach 3x kopieren.
- die letzte Tür ist immer genau vor dem Eingang in den Raum. Da reicht ein `einsVor()`-Befehl.
- für die Schalter brauchst du wieder eine Schleife, da nicht klar ist, wie breit die Wand ist.

Zusammenfassung: Du kannst Verzweigungen in Algorithmen nutzen, im Quelltext erkennen und formulieren. Du kennst die Schreibweise dieser Entscheidungs-Anweisung mit dem Schlüsselwort **if** und der Prüfbedingung im runden Klammerpaar dahinter.

```
if ( Prüfbedingung ) {
    // DANN-Teil
    // Anweisungen, die ausgeführt werden
    // falls die Prüfbedingung stimmt.
}
else {
    // SONST-Teil
    // Anweisungen, die ausgeführt werden
    // falls die Prüfbedingung NICHT stimmt.
}
```

Manchmal ist im SONST-Fall nichts zu tun. Dann entfällt der Teil ab **else**. Du kannst die Antworten der Ja/Nein-Abfragen wie `istVorneFrei()` als Prüfbedingung in einer Entscheidung nutzen, auch in ihrer negierten Form wie bei: `if (!istVorneFrei()) {...}`. Dies wird gelesen als: „Falls NICHT vorne frei ist...“ oder „Falls vorne frei falsch ist...“ oder „Falls vorne nicht frei ist...“. Die Verneinung NICHT wird durch das Ausrufezeichen ! geschrieben.

Die Begriffe Verzweigung, Entscheidungsanweisung und auch Alternative werden gleichwertig genutzt.

Bildquellen: Die verwendeten Bilder des Roboterszenarios sind alle ohne Bildnachweis verwendbar (selbst gezeichnet, Pixabay Lizenz oder Public Domain). Genaue Nachweise: siehe [bildquellen.html](#).



Puh, das war knapp! Gut, dass die Roboter schon recht selbstständig Ihren Weg finden. Der kommende Einsatz wird ein wenig „anstrengender“. Ein Upgrade wird benötigt ...

Neue Sensoren ...

ZIEL: Methoden mit boolschen Rückgabewerten erstellen können.

Unsere Roboter haben nur eine beschränkte Anzahl von Methoden, die die Umwelt des Roboters wahrnehmen (z.B. `istVorneFrei()`, `istVorne("Schraube")`). In diesem Arbeitsblatt lernst, du neue (komplexere) Sensoren selbst zu erstellen.

Die Roboter aus dem ersten Übungsblatt hatten Sensoren wie `aufSchraube()`, die die darauffolgenden Roboter nicht mehr hatten, da man das Gleiche auch mit `aufGegenstand("Schraube")` erreichen kann. Trotzdem kann es sinnvoll sein, aus den vorhandenen Sensoren neue zu erschaffen. Schau dir dazu die Methode `istFassVorne()` des **AB5**-Roboters an. Im Gegensatz zu den Methoden, die du bisher implementiert hast, gibt diese eine Antwort zurück.

Aufgaben:

1. a) Teste die Methode `istFassVorne()` in Greenfoot. Wie beantwortet ein Roboter diese Anfrage? Welche Antwortmöglichkeiten gibt es?
b) Ein Roboter erkennt ein Atommüllfass (noch) nicht als Fass. Teste dies an dem Roboter vor dem gelben Fass.
2. Analysiere nun den Quelltext zur Methode `istFassVorne()`: Wo tauchen die Antwortmöglichkeiten (`true` und `false`) im Quelltext auf? Welcher Befehl sorgt dafür, dass eine Antwort zurückgegeben wird?

Alle bisherigen Methoden wurden mit `public void methodName() {...}` programmiert. Void steht dabei für leer/nichts. Void steht an der Stelle, an der der sogenannte Rückgabetyt steht. Es wird also nichts zurückgegeben. Möchte man nun mit `true/false` antworten, muss man den Rückgabewert als `boolean`¹ deklarieren:

```
public boolean istFassVorne() {
    if (istVorne("Fass")) {
        return true;
    } else {
        return false;
    }
}
```

Mit `return true/false;` kann man dann den gewünschten Wert zurückgeben. Return beendet außerdem das Unterprogramm. Es werden also keine Befehle mehr nach dem Ausführen eines Return ausgeführt.

Aufgaben:

3. Füge die Anweisung `dreheUm();` in die Methode `istFassVorne` ein:
 - a) Vor der `if`-Anweisung.
 - b) Jeweils vor dem Return.
 - c) Nach der `if-else`-Anweisung / vor der letzten Klammer.
 Welche Auswirkungen hat dies jeweils? Entferne die `dreheUm()` Anweisung wieder.
4. **Rückspiegel:** Vervollständige die Methode `istFassHinten()`. Diese soll testen, ob hinter dem Roboter ein Fass steht. Dazu muss er sich natürlich kurzfristig umdrehen. Am Ende soll er aber wieder so stehen wie am Anfang.
5. **Out of Power:** Implementiere eine Methode `istEnergieSchwach()`. Diese soll `true` zurückgeben, falls der Roboter nur noch weniger als 20 Energiepunkte hat (Überprüfung mit

¹ Nach George Boole, der sich mit dem mathematischen Teilbereich Logik beschäftigt hat. Die Logik beschäftigt sich u.A. mit Aussagen, die nur die Wahrheitswerte wahr oder falsch annehmen können.

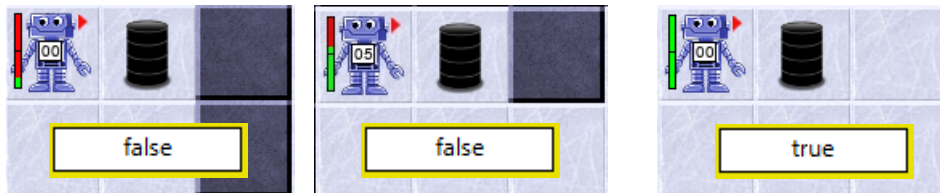


`getEnergie()<20`). Ansonsten gibt sie `false` zurück. Teste deine Methode an den drei Robotern im Raum unten links (nur der unterste hat einen schwachen Ladezustand).

- Heavy Duty:** Implementiere eine Methode `istSchwerBeladen()`, die testet, ob der Roboter fünf oder mehr Gegenstände herumträgt (benutze `getAnzahl()`). Teste deine Methode an verschiedenen Robotern.

Fässer kann man schieben. Aber nur, wenn vor dem Fass Platz ist. Daher ist es hilfreich zu wissen, ob vor dem Fass frei ist oder nicht. Probiere die Methode `einsVor()`, wenn der Roboter vor einem Fass steht.

- Look Ahead:** Implementiere eine Methode `istVorFassFrei()`. Diese soll `false` zurückgeben, falls vor dem Fass eine Wand ist. Dabei können die folgenden Fälle auftreten:



Der Roboter muss rechts um das Fass herumlaufen (dort ist immer Platz) und nachschauen, ob vor dem Fass frei ist. Falls ja, gibt er `true` zurück, sonst `false`. Trifft er schon vorher auf die Wand (Fall 1), dann gibt er auch `false` zurück. In allen Fällen ist er am Ende wieder an seinen Ausgangspunkt.

- Aufräumen:** Implementiere eine Methode `schiebeFassBisWand()`, die ein Fass bis zur nächsten Wand schiebt. Teste deine Methode an dem Roboter unten rechts. Hinweis: Da man `istVorFassFrei` als Bedingung für eine `while`-Schleife verwenden kann, ist der Quelltext dieser Methode nur 3 Zeilen lang. Der Roboter rennt dann aber ganz schön wild durch die Gegend.

Geschickt ist es auch, wenn man zwei Sensoren A und B miteinander verbindet. In Java gibt es dazu die Operatoren

`A || B` → A oder B

`A && B` → A und B

`!(A)` → nicht A

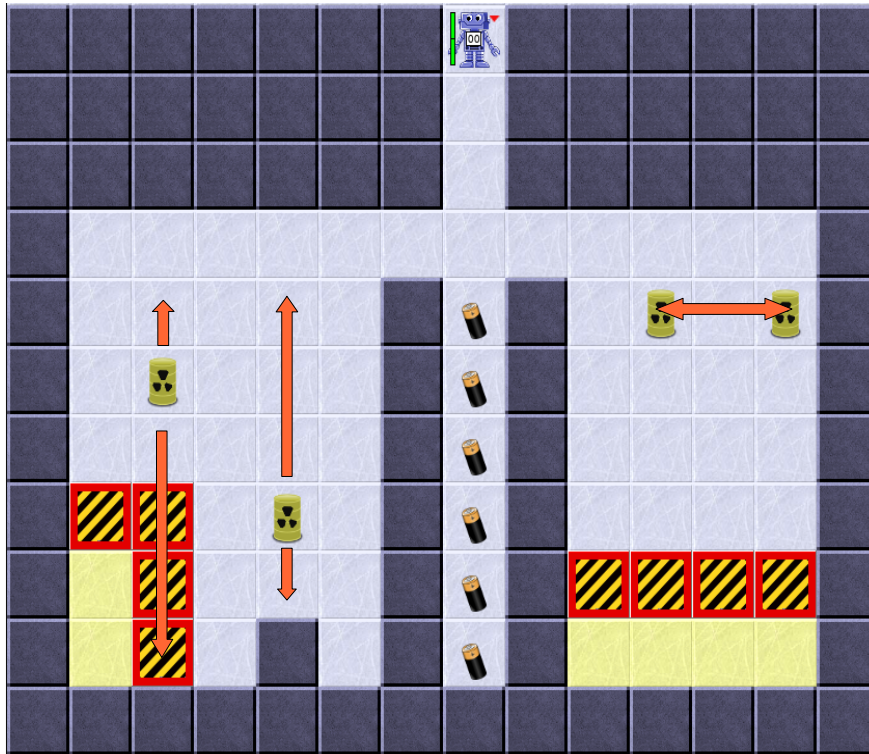
Man kann also beispielsweise mit `!(istWandVorne()) && !(istWandLinks()) && !(istWandRechts())` testen, ob in allen drei Richtung keine Wand ist.

- Führerschein:** Implementiere eine Methode `istKreuzung()`, die testet, ob der Roboter auf einer Kreuzung steht, d.h. ob vorne, links und rechts frei ist. Implementiere eine Methode `geheBisKreuzung()` und teste sie am Roboter oben links.
- Upgrade 1:** Erweitere die Methode `istFassVorne()` so, dass nicht nur die Übungsfässer erkannt werden, sondern auch Atommüllfässer (benutze dazu `istVorne("Atommüll")`). Teste deine Methode an den beiden Robotern im rechten Raum.
- Upgrade 2:** Erweitere die Methode `istVorFassFrei()` so, dass nicht nur Wände erkannt werden, die vor dem Fass sind, sondern auch andere Fässer (normale und Atommüll). Jetzt müsste `schiebeFassBisWand()` auch für den Roboter rechts oben funktionieren.
- Aufräumen:** Implementiere eine Methode, die den Roboter rechts unten das Fass auf das gelbe Feld in der Ecke schieben lässt. Verwende einen sinnvollen Methodennamen.



Einsatz 5: Schaffe Ordnung im Atommüllzwischenlager

Die Arbeiter haben die Atommüllfässer einfach willkürlich in die zwei Räume gestellt. Der Eingang zu den Räumen ist an der Kreuzung. Geht man geradeaus weiter, liegt dort eine Reihe von Akkus. Im Raum links stehen zwei Fässer. Ich weiß nicht genau, wie viele Fässer im anderen Raum stehen. Die Fässer können an den mit den Pfeilen gekennzeichneten Stellen stehen. Sorge für Ordnung. Schiebe dazu die Fässer in die gekennzeichneten gelben Bereiche.



Tipp 1: Kopiere die Methode `laufeBisWand()` von AB3. Sie kann dir gute Dienste leisten.

Tipp 2: Baue in die Methode `istVorFassFrei()` eine Energiekontrolle ein, da sonst die Energie ausgehen kann. Teste dazu am Anfang der Methode mit `istEnergieSchwach()`, ob der Roboter mit `benutze("Akku")` aufgeladen werden muss.

Bildquellen: Die verwendeten Bilder des Roboterszenarios sind alle ohne Bildnachweis verwendbar (selbst gezeichnet, Pixabay Lizenz oder Public Domain). Genaue Nachweise: siehe [bildquellen.html](#).



Ein Notfallprogramm für den Roboter... Wir haben Millionen in die Hardware des Roboters und unendlich viel Zeit in die Implementierung der Software und seiner Verbesserungen gesteckt. Daher soll der Roboter ein Notfallprogramm bekommen mit dem er jederzeit den Ausgang wiederfinden kann, wenn er sich in einem zerstörten Kraftwerk befindet und der normale Rückweg plötzlich versperrt ist...

Die Roboter lernen zu zählen ...

ZIEL: Variablen als Speicher für jeweils einen Wert kennen und einsetzen können. Variablen verwenden, um Roboter das Zählen beizubringen.

Attribute beschreiben Eigenschaften und Zustände

Wir sehen die Roboter in ihrer Welt agieren. Wir sehen auch, wie viel Energie sie noch haben. Mit `getEnergie()` können wir das auch abfragen. Dann erhalten wir von ihm eine Zahl als Antwort. Woher weiß dieser Roboter aber, wie viel Energie er noch hat? Wie merkt er sich diese Zahl?

Wenn mehrere Roboter auf der Welt herum laufen, dann gibt jeder auf Nachfrage die für ihn richtige Antwort. Jeder Roboter führt darüber Buch. Jeder Roboter hat sein Gedächtnis. Dazu verwendet er eine Variable: `energie` – man sagt auch, dass jedes Roboterobjekt eine Eigenschaft `energie` hat. In dieser Variable merkt er sich die Energie, genauer: den aktuellen Wert der Energie, die ihm verbleibt. Dieser Wert kann immer nur eine ganze Zahl sein. Man sagt: Der Typ des Attributs ist `integer` (engl. für Ganzzahl). In Java wird dafür die Kennzeichnung `int` verwendet.

Den Wert kannst du z.B. um 15 verringern, indem du dem Roboter den Befehl gibst:

- `verbraucheEnergie(15)` Damit hat er 15 Energiepunkte weniger, egal wie viele es vorher waren (jedoch unterschreitet er 0 Energiepunkte nicht).

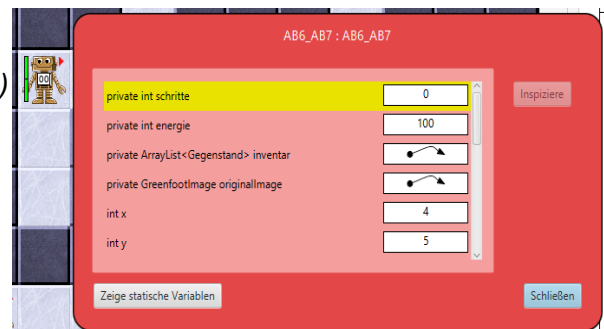
Den aktuellen Wert kannst du erfragen, indem du ihm die Anfrage stellst:

- `getEnergie()` Er wird dir die Restenergie als Antwort nennen.

Die aktuellen Werte aller Attribute bestimmen den **Zustand** eines Roboters.

Aufgaben:

1. Erprobe an zwei Robotern `verbraucheEnergie(...)` bzw. `getEnergie()` und kontrolliere mit **INSPECT** (Rechtsmausklick auf den Roboter → **Inspizieren**) den Zustand der Energie. Verschiebe ggf. das **Inspect-Fenster** so, dass es nicht mehr über der Roboterwelt liegt. Rechts siehst du einen Roboter und sein **INSPECT-Fenster**.



2. Notiere, welche weiteren Eigenschaften des Roboters in Attributen gespeichert sind. Nenne mindestens eine Methode, mit der Du die Werte der ganzzahligen Attribute ändern kannst.

Bei den Integer-Attributen kann man die Werte direkt sehen. Bei komplexeren Attributen (z.B. `inventar`) wird nur ein Pfeil angezeigt. Die Attribute sind komplexer und nicht mit einer einzelnen Zahl auszudrücken. Programmieren musst du das noch nicht. Aber durch Klicken auf den Pfeil kannst du dir anschauen, was dahinter steckt. Du kannst beispielsweise erkennen, wo der Roboter speichert, wie viele Gegenstände er in seiner Inventarliste hat.

Aufgaben:

3. Lasse dir durch einen Klick auf den Pfeil hinter `inventar` anzeigen, wie groß die Inventarliste ist. Versuche herauszufinden, wie groß das Bild (`image`) eines Roboters ist. Dazu musst du mehreren Pfeilen folgen.

Jede Variable, die die Werte für eine Eigenschaft speichert, kannst du dir als beschriftete Kiste



vorstellen. Vorne steht der Name der Variable, damit die Kiste bzw. ihr Speicherplatz wiedergefunden werden kann. Namen von Variablen beginnen mit einem Kleinbuchstaben. Setzt sich der Name aus mehreren Wörtern zusammen, darf man keine Leerzeichen verwenden, sondern beginnt jedes neue Wort mit einem Großbuchstaben. Das bezeichnet man als Camel-Case.

In der Kiste gibt es zu jedem Zeitpunkt genau einen aktuell gültigen Wert. Mit set-Methoden (und auch durch andere Methoden) werden neue Werte in die Kiste gebracht; mit der get-Methode nach dem aktuellen Wert gefragt und dieser Wert dann dem Anfragenden als Ergebnis genannt. Dabei verbleibt der Wert unverändert in der Variable. So wie beim Anhören eines Musikstücks auf deinem Smartphone der Musikinhalt ja weiterhin gespeichert bleibt.

Der Typ der Variable legt die Größe der Kiste fest. In int-Kisten (32 Bit groß) passen z.B. ganze Zahlen zwischen -2.147.483.648 und +2.147.483.647. Für größere Zahlen bräuchte man long-Kisten (64 Bit groß). Dort passen Zahlen zwischen -9.223.372.036.854.775.808 und 9.223.372.036.854.775.807 hinein.

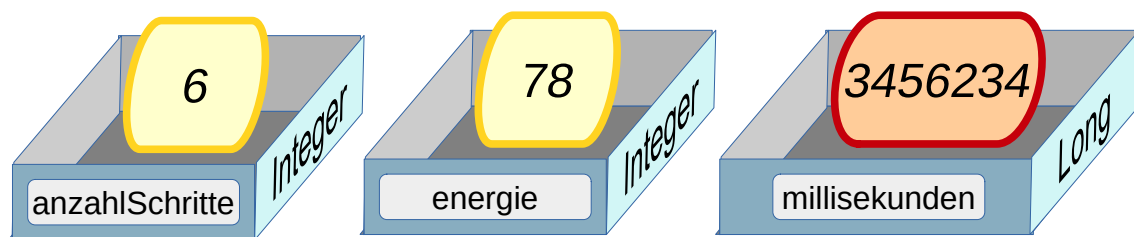


Abb. 1: int und long-Variablen als "Kisten" (Quelle: eigenes Werk)

Die Zeichnungen zeigen zwei Variable vom Datentyp **int**. In solche Variable können nur ganze Zahlen (= Integer; daher **int**) gesteckt werden. Auch wenn die Bilder nur positive Ganzzahlen suggerieren, sind Integer-Zahlen positive und negative Ganzzahlen.

Aufgaben:

4. Denk- und Sprechweisen:

- Wieso kann man sich Variablen als Kisten vorstellen?
- Welche der vier Sprechweisen hältst du für gut?
 - Die Variable *anzahl* hat den Inhalt 7.
 - Die Variable *anzahl* hat den Wert 7.
 - Die Variable *anzahl* hat 7 Werte.
 - anzahl* ist 7.

5. Namensgebung:

Schlage sinnvolle Namen für Attribute vor, die speichern

- wie viele Akkus der Roboter besitzt.
- wie viele Akkus der Roboter schon eingesetzt hat.
- wie viele Drehungen er gemacht hat. (Was genau gibt die gespeicherte Zahl dann an? Wie werden Links- bzw. Rechtsdrehungen gespeichert?)
- wie viele Brennstäbe er schon gefunden hat.



Eigene Attribute

Der Roboter **AB6_AB7** hat eine neue Eigenschaft bekommen: Er hat das Attribut Schritte, das zählt wie viele Schritte er schon gemacht hat.

Dazu müssen drei Dinge erledigt werden:

- ✓ Das Attribut muss deklariert werden (die Kiste muss erschaffen werden): Oberhalb aller Methoden im Quelltext werden die Attribute angegeben. Die Deklaration beginnt mit `private`, dann folgt der Typ und der Name des Attributs.
`private int schritte;`
- ✓ Der Wert des Attributs muss initialisiert werden (die Kiste muss einen sinnvollen Anfangswert erhalten): Dies macht man im sogenannten Konstruktor. Dies ist eine Methode mit dem gleichem Namen wie die Klasse (Achtung: Sie hat keinen Rückgabetyt, auch kein `void`). Sie wird automatisch aufgerufen, wenn das Roboterobjekt erzeugt wird. Dort wird der Wert von `schritte` auf 0 gesetzt. Um zu verdeutlichen, dass dieses Attribut zu dem Roboter gehört, schreibt man statt „`schritte`“ immer „`this.schritte`“:
`this.schritte = 0;`
Anmerkung: Wenn man Strg-Leerzeichen drückt, bekommt man alle verfügbaren Attribute (nur ab Version 3.1.0) und Methoden angezeigt.
- ✓ In der Methode `einsVorMitZaehlen()` wird der Wert des neuen Attributs um 1 erhöht: `this.schritte++`; und dann die normale `einsVor()`-Methode aufgerufen.

Aufgaben:

6. Überprüfe im INSPECT-Fenster, ob der Roboter ordentlich seine Schritte zählt.
7. **Schrittezähler:** Implementiere eine Methode `bisWandMitZaehlen()` unter Verwendung von `einsVorMitZaehlen()`, die den Roboter bis zur Wand laufen lässt und dabei die Schritte zählt.
8. **Drehzähler:** Führe ein neues Attribut `drehungen` ein (deklariere und initialisiere das Attribut). Dieses soll bei jeder Rechtsdrehung um eins erhöht und bei jeder Linksdrehung um 1 erniedrigt werden (es gibt analog zu ++ auch - -). Führe dazu die beiden Methoden `dreheLinksMitZaehlen()` und `dreheRechtsMitZaehlen()` ein.

Get-Methoden

Wie bei den anderen Eigenschaften auch, soll der Roboter auch ohne Inspect-Fenster Auskunft darüber geben können, welchen Wert seine Attribute haben. Dazu brauchen wir get-Methoden. Im letzten Kapitel hast du Methoden kennengelernt, die mit `true` oder `false` antworten. Nun möchten wir mit einer Zahl als Ergebnis antworten. Daher muss der Rückgabetyt der Methode `int` sein:

```
public int getSchritte() {...}
```

Statt `return true` oder `return false`, wollen wir nun den Wert des Attributs zurückgeben. Dazu verwenden wir einfach den Namen des Attributs:

```
return this.schritte;
```

```
/** nennt die gesamten zurück-
 * gelegten Schritte */
public int getSchritte() {
    return this.schritte;
}
```

Aufgaben:

9. **Eine get-Methode:** Verändere die Methode `getSchritte()` im Roboter **AB6_AB7**, so dass die Methode nicht immer 0 zurück gibt, sondern den Wert des Attributs `schritte`. Warum kann die Methode dafür nicht lauten: `public void ...` ?
Warum enthält die Methode `bisWandMitZaehlen()` aus Aufgabe 7 keine Zeile, die mit `return ...` beginnt?
10. **Deine get-Methode:** Implementiere eine get-Methode für die Anzahl der Drehungen. Teste



deine get-Methode.

Einsatz 6: Das Notfallprogramm wird getestet

Der Roboter wird nicht immer so gut informiert, wie sein Einsatzgebiet aussieht. Unsere hohen Entwicklungskosten sollen nicht verloren gehen, nur weil der Roboter irgendwann mal den Ausgang nicht wiederfindet. Daher bekommt der Roboter ein Not-Programm, mit dem er in beliebigen Umgebungen den Ausgang (gekennzeichnet durch das Portal) immer wiederfindet, egal wie viele Ecken und Wände im Weg sind.

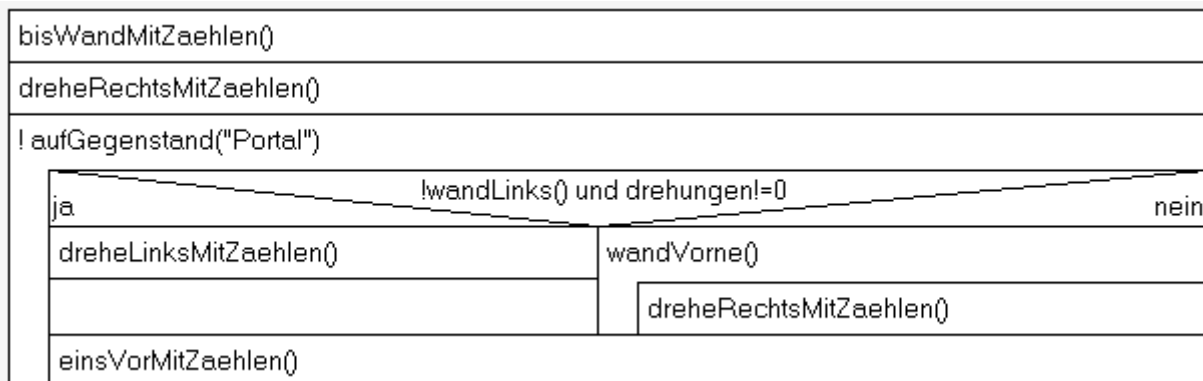
Geht das wirklich? Ja, John Pledge, ein 12-jähriger Junge, hat entdeckt, wie das geht. Wir müssen dazu nur Drehungen zählen können.

Seine Idee war folgendermaßen: Im Prinzip folgt er permanent einer Wand so, dass seine linke Hand die Wand berührt. Da das aber zu Problemen führt, wenn die linke Hand beispielsweise an einer Säule ist, muss man irgendwann loslassen und geradeaus laufen. Das macht Pledge, wenn er sich genauso oft links wie rechtsrum gedreht hat.

Aufgabe:

Implementiere den Pledge-Algorithmus als Not-Programm (Methode `einsatz6()`). Der Roboter wird dann in einer unbekannten Umgebung ausgesetzt und muss das Portal finden.

Hinweis:



Im Bild siehst du ein Struktogramm des Pledge-Algorithmus. Es besteht aus ein paar Anweisungen, die du gerade implementiert hast und nur aufrufen musst und einigen while-Schleifen und if-Anweisungen.

Hinweis: Korrigiere die Befehle (wie beispielsweise `wandLinks()`), falls nötig!

Hinweis: Wenn dein Programm nicht funktioniert, hast du vermutlich das Flussdiagramm nicht korrekt umgesetzt. Kopiere den Abschnitt deines Quelltextes über die Zwischenablage in das Programm "Flussdiagramme.exe" und vergleiche deine Version mit dem Original.

Bildquellen: Die verwendeten Bilder des Roboterszenarios sind alle ohne Bildnachweis verwendbar (selbst gezeichnet, Pixabay Lizenz oder Public Domain). Genaue Nachweise: siehe [bildquellen.html](#).



Der Atom Müll im Endlager rottet vor sich hin... keiner traut sich mehr hinein! Wenigstens sollte eine Aufstellung über die Anzahl der dort lagernden Fässer und der beschädigten Fässer, aus denen die abgebrannten Brennstäbe schon herausgefallen sind, gemacht werden. Eine neue Aufgabe für unseren Rescue-Robot...

Nicht jedes Mal muss man alles neu programmieren ...

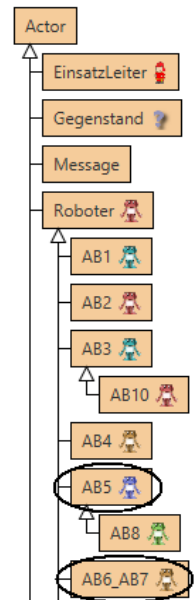
ZIEL: Vererbung einsetzen, um Fähigkeiten eines Roboters in ein neues Modell zu übernehmen..

Im AB5 hast du viele neue Sensoren für den Roboter programmiert, in AB6 hat er gelernt, Schritte und Drehungen zu zählen. Aber dafür sind alle Fähigkeiten des AB5 wieder verloren gegangen. Natürlich könnte man per Copy-Paste die Methode in die neue Klasse übernehmen. Das geht aber auch eleganter.

Du hast ja schon gemerkt, dass deine Roboter alles können, was ganz allgemein ein Roboter kann, ohne dass du jemals dafür eine einzige Methode hast kopieren müssen. Das liegt daran, dass deine ABx-Roboter Unterklassen der Klasse Roboter sind. Sie erben alle Eigenschaften und Fähigkeiten der Oberklasse, können sie dennoch erweitern oder verändern.

Dies erreicht man durch folgende Anweisung am Anfang der Klasse:

```
public class AB6 extends Roboter {...}
```



Aufgaben:

1. **Verändere die Klasse AB6_AB7** so, dass sie nicht mehr eine direkte Unterklasse von Roboter, sondern eine Unterklasse des AB5-Roboters ist. Sie erbt auf diese Weise alle seine Sensoren.
 Compiliere die Klasse neu. Wie verändert sich das Klassendiagramm?
 Überprüfe, dass der AB6_AB7-Roboter jetzt die Fähigkeiten des AB5-Roboters hat. Du findest sie unter „inherited from AB5 >“. (inherited = engl. geerbt).
2. Erkläre, warum dein Roboter weiterhin auch die Fähigkeiten eines normalen Roboters hat.
3. **Kreuzungen zählen:** Implementiere im AB6_AB7 eine Methode `zaehleKreuzungen()`, die einen Gang ans Ende zur Wand läuft und zählt an wie vielen Kreuzungen der Roboter vorbei kam. Benutze dazu einen der bei AB5 programmierten Sensoren. Speichere den Wert in einem geeigneten Attribut. Stelle auch eine `get`-Methode zur Verfügung. Verschiebe zum Testen mit der Maus das Fass vor dem Roboter oben links, damit er einen Gang mit Kreuzungen vor sich hat.

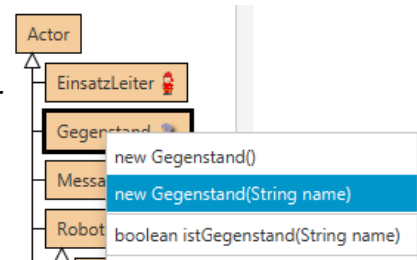


Vererbung kann aber noch mehr. Man kann schon existierende Methoden auch verbessern. Beim letzten Arbeitsblatt wurde eine neue Methode `einsVorMitZaehlen` eingeführt, die eine Verbesserung der Methode `einsVor()` des Standardroboters darstellt. Statt eine Methode mit einem neuen Namen zu erfinden, kann man auch die alte Methode verbessern. Ändere dazu den Namen auf `einsVor()`. Nun gibt es diese Methode zweimal. Einmal beim Roboter und einmal beim AB6_AB7-Roboter. Um die alte Methode aufzurufen, musst du beim AB6_AB7-Roboter `super.einsVor()`; schreiben (lies: rufe die Methode `einsVor()` bei der übergeordneten Klasse auf).



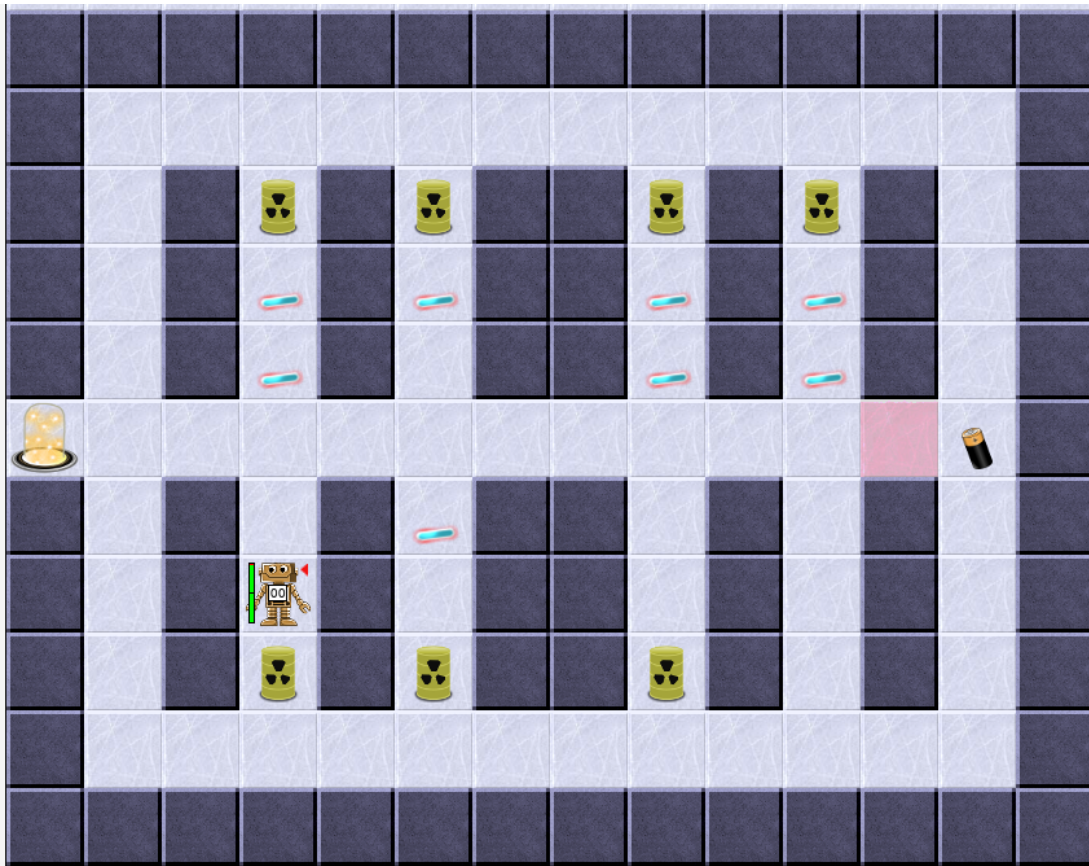
Aufgaben:

4. **Super:** Ändere den Namen der Methode `einsVorMitZaehlen` in `einsVor`. Rufe innerhalb dieser Methode die gleichnamige Methode der Oberklasse mit `super.einsVor()` auf.
Anmerkung: Damit du die Klasse kompilieren kannst, muss du natürlich überall die Namensänderung vornehmen, wo du `einsVorMitZaehlen()` verwendet hast.
Überprüfe die Auswirkungen dieser Änderung:
 - Rufe `einsVor()` beim AB6 auf. Beobachte die Auswirkung im Inspect-Fenster.
 - Rufe die von AB5 geerbte Methode `laufeBisWand()` auf. Beobachte im Inspect-Fenster.
5. **Brennstäbe zählen:** Ändere die überflüssig gewordene Methode `bisWandMitZaehlen()`, so dass sie in einem neuen Attribut `brennstaebe` zählt, wie viele Brennstäbe auf dem Weg zur Wand liegen (wir verwenden auf dem Testgelände natürlich nur ungefährliche Attrappen, die aber täuschend echt aussehen). Verwende dazu `istAufGegenstand("Brennstab")`. Ergänze auch die dazugehörige `get`-Methode. Achtung: Ein direkt vor der Wand liegender Brennstab muss auch erkannt werden.
6. **Standardmethoden überschreiben:** Überschreibe auf die gleiche Weise die Methode `dreheLinks` und `dreheRechts` von `Roboter`, so dass die Drehungen gezählt werden. Überprüfe, ob der Pledge-Algorithmus (`einsatz6()`) immer noch funktioniert.
7. **Fass links:** Implementiere einen neuen Sensor `istFassLinks()`, der erkennt, ob links ein Fass (egal, ob ein normales oder ein Atommüllfass) ist. Verwende dazu den Sensor `istFassVorne()` (Öffne zum Vergleich AB 5).
8. **Überschreiben:** Überschreibe die Methoden `istWandVorne()` und `istWandLinks()` so, dass sie `true` zurückgeben, falls wirklich eine Wand vorne (`super.istWandVorne()`) bzw. links (`super.istWandLinks()`) ist oder falls ein Fass vorne bzw. links ist. Verwende dazu die Sensoren `istFassVorne()` und `istFassLinks()`.
9. Teste die Auswirkungen dieser Änderungen auf den Pledge-Algorithmus, indem du direkt am Roboter (nicht auf einem Bodenfeld) den `einsatz6()` aufrufst, nachdem du einige Fässer in der Welt zusätzlich platziert hast. (Erzeuge dazu mit einem Rechtsklick auf `new Gegenstand(String name)` einen neuen Gegenstand und gib als Name "Atommuell" oder "Fass" ein.)





Einsatz 7:



In einem alten Bergwerk wurde vor etlichen Jahren ein Atommüllendlager eingerichtet. Leider rosten die Fässer dort vor sich hin, so dass einige Brennstäbe in den Gängen herumliegen. Für Menschen ist es zu gefährlich, diese zu zählen, da sie stark strahlen. Außerdem ist das Bergwerk stark einsturzgefährdet. Jeden Moment kann es einen Steinschlag geben. Unser Roboter soll das übernehmen.

Auftrag: Zähle die Brennstäbe in den Gängen, hebe sie auf und kehre zum Portal zurück.

Hinweis:

- Auf dem Rundweg außen herum liegen keine Brennstäbe.
- Mach dir vorher ein Bild von der Situation, indem du den Einsatz7 startest.
- Eventuell reicht die Energie nicht aus, um das Bergwerk zu verlassen. In diesem Fall musst du den Einsatz erneut ausführen.

Bildquellen: Die verwendeten Bilder des Roboterszenarios sind alle ohne Bildnachweis verwendbar (selbst gezeichnet, Pixabay Lizenz oder Public Domain). Genaue Nachweise: siehe [bildquellen.html](#).



Immer mehr Atom Müll. Wie viel passt eigentlich noch in unser Endlager? Einige Räume im Endlager sind noch frei. Dort sollen Fässer eingelagert werden. Vorher muss aber bestimmt werden, wie viel Platz dort noch ist.

Wertzuweisungen

ZIEL: Lokale Variable als Zwischenspeicher für einen Wert kennen und einsetzen können. Lokale Variable von Objektvariablen unterscheiden können. Wertzuweisungen nutzen können.

Lokale Variable als Kurzzeitgedächtnis

Manchmal muss sich der Roboter bestimmte Werte nur vorübergehend merken, um eine Aufgabe zu erfüllen. Wenn er z.B. die Schrauben einer lückenlosen Schraubenspur zählen soll, so muss er sich nur kurzfristig merken, wie viele Schrauben er schon gezählt hat und kann das wieder vergessen, sobald er die Antwort gegeben hat.

Bisher hatten wir Attribute der Objekte als Gedächtnis. Diese sind aber nur für Werte gedacht, die der Roboter sich dauerhaft merken soll, z.B. wie viele Schritte er insgesamt schon gelaufen ist, welche x-Position er im Moment hat, ...

Man könnte auch das „Kurzzeitgedächtnis“ mit Attributen realisieren, würde dann aber sehr viele Attribute bekommen, die immer nur kurzfristig zum Einsatz kämen. Das fördert nicht gerade die Lesbarkeit.

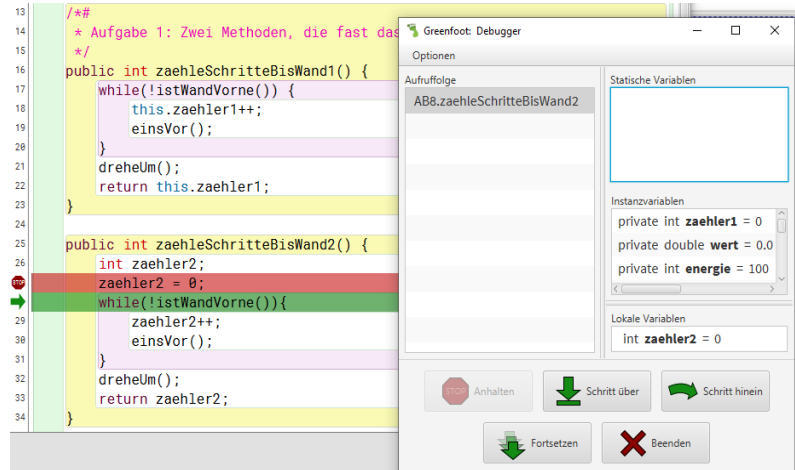
Daher verwenden wir sogenannte lokale Variablen, deren Geltungsbereich nur eine Methode (oder auch nur eine Schleife) ist. Man verwendet sie fast genauso wie Attribute.

	Lokale Variable	Attribut
Deklaration	Innerhalb der Methode: <pre>public class AB8 { ... //Methoden public int berechneXY() { int zaehler2; ... } ... }</pre> → <i>ohne private!</i>	Innerhalb der Klasse, vor allen Methoden: <pre>public class AB8 { private int zaehler1; ... //Methoden ... }</pre> → <i>mit private!</i>
Initialisierung	Direkt nach der Deklaration: <pre>public int berechneXY() { int zaehler2; zaehler2 = 0; ... }</pre>	Im Konstruktor der Klasse: <pre>public AB8() { this.zaehler1 = 0; }</pre>
Verwendung	Ohne this. <pre>zaehler2++;</pre>	Zur Unterscheidung kann/sollte this. verwendet werden <pre>this.zaehler1++;</pre>



Aufgaben:

1. **Finde den Unterschied:** Rufe bei den beiden **AB8-Robotern** links oben die Methoden `zaehleSchritteBisWand1` bzw. `zaehleSchritteBisWand2` auf. Sie sollten das gleiche Ergebnis liefern. Rufe danach die Methoden erneut auf. Erkläre den Unterschied.
2. **Debugging:** Lasse dir mit **INSPECT** die Attribute anzeigen, um den Ablauf der Methoden zu verfolgen. Warum sieht man hier nur den `zaehler1`? Wo sieht man die lokalen Variablen? Dafür benötigt man den Debugger, mit dem man auch bei lokalen Variablen den Verlauf kontrollieren kann. Setze dazu einen Breakpoint, indem du auf die Zeilennummer klickst, an der das Programm unterbrochen werden soll (hier Zeile 27). Wenn du dann die Methode aufrufst, hält das Programm am Stoppschild an und kann mit **Step Schritt** für Schritt ausgeführt werden. Dabei werden Attribute (= Instance variables) und lokale Variablen angezeigt. Du siehst, dass die lokale Variable erst entsteht, wenn sie das erste Mal benutzt wird.
3. Entscheide, ob die folgenden Werte besser als lokale Variable oder als Attribut gespeichert werden sollten:
 - // die Farbe des Roboters,
 - // die Anzahl der insgesamt gelöschten Feuer,
 - // die Anzahl der mit dem aktuell benutzten Feuerlöscher gelöschten Feuer.
4. Entscheide, ob man bei dem Pledge-Algorithmus die Anzahl der Drehungen auch als lokale Variable hätte speichern können.



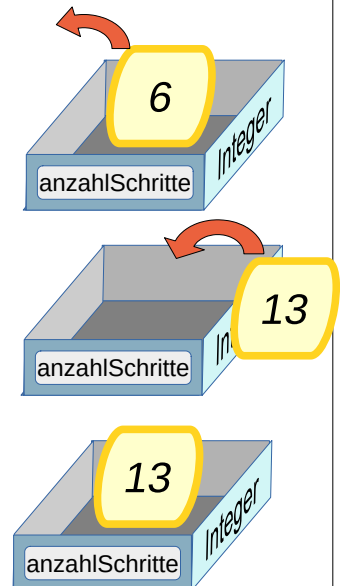
Bisher haben wir Variablenwerte nur initialisiert (z.B. `anzahl = 0`) oder um 1 erhöht/erniedrigt (`anzahl++` bzw. `anzahl--`). Da geht noch mehr:

- Wertzuweisungen:
An beliebigen Stellen im Programm kann der Variable ein Wert zugewiesen werden:
`anzSchritte = 13;`
- Wertzuweisung um Ergebnisse von Methoden zu speichern:
`anzSchritte = zaehleSchritteBisWand1();`
Dabei wird die Methode aufgerufen und das Ergebnis in der Variable `anzSchritte` gespeichert.
- Wertzuweisung mit Rechnungen:
`flaeche = breite * hoehe;`
Dabei wird zunächst die Rechnung auf der rechten Seite ausgeführt und dann das Ergebnis in der Variablen gespeichert.

Ablauf der Wertzuweisung:

`anzSchritte = 13;`

1. Bisherigen Wert entfernen
2. Neuen Wert in die Variable stecken



`ergebnis = anzBlaetter - 7;`

Wertzuweisung

Variablenname = **Wert, Ergebnis einer Berechnung**

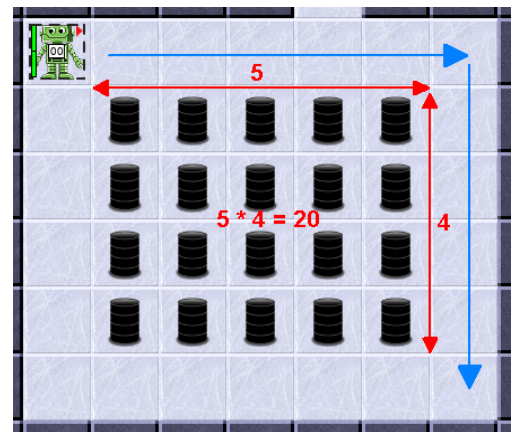
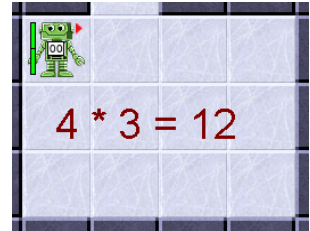
Grundsätzlich gilt: Rechts vor Links! Zuerst wird die Rechnung/Methode auf der rechten Seite ausgeführt und dann der Variable links zugewiesen.



Aufgaben:

5. **Summe:** Rufe die Methode `gibPositionssumme()` an einem Roboter auf. Bewege ihn (per `einsVor-` und `drehen-`Befehle) an eine andere Stelle. Was meldet er jetzt als Antwort auf diesen Fragebefehl? Wie musst du ihn bewegen, damit das Ergebnis gleich bleibt? Begründe dies mit dem Quelltext. Wo findet welche Art von Wertzuweisung statt?
6. **Abstände:** Implementiere die Methode `gibLaenge()` mit `int`-Rückgabewert: Dekлариere drei lokale Variablen `x1`, `x2`, `x_diff`. Weise `x1` den Wert des Aufrufs von `getX()` zu. Lasse den Roboter bis zur nächsten Wand laufen. Weise nun `x2` den Wert des Aufrufs von `getX()` zu. Berechne die Differenz von `x2` und `x1` und speichere sie in `x_diff`. Gib diese Differenz als Ergebnis deiner Methode zurück.
 Teste diese Methode. Welches Ergebnis liefert sie, wenn du den Roboter an eine Wand nach rechts laufen lässt. Was passiert bei Robotern, die nach links/oben/unten laufen? Verbessere deine Methode so, dass sie immer den Abstand zur Wand zurück gibt. Dazu wirst du auch die `y`-Koordinaten speichern müssen.
 Anmerkung: Ok, einfacher als Schritte zählen ist das nicht ...
7. **Fläche:** Implementiere eine Methode, die die Fläche eines Raumes (Breite * Höhe) bestimmt und zurück gibt. Du kannst davon ausgehen, dass der Roboter in einer Ecke des Raumes steht. Du kannst dir aussuchen, in welcher.
8. **Zähle Fässer:** Implementiere eine Methode, die zurück gibt, aus wie vielen Fässern ein rechteckiges Areal von Fässern besteht. Es ist sinnvoll, zunächst einen Sensor `istFassRechts()` zu implementieren. Damit kann man eine Methode `gibLaengeFassreihe()` implementieren, die man dann zum Bestimmen der Anzahl der Fässer benutzt.
 Hinweis 1: Um die Fässer herum ist eine Reihe von freien Feldern.
 Hinweis 2: Der Roboter muss am Ende nicht wieder an seiner Startposition stehen.
9. Gib an, welchen Wert die Variable `summe` hat, wenn folgende fünf Zeilen ausgeführt wurden:

```
int summe;
summe = 10;
summe = summe - 2;
summe = summe * 4;
summe = summe + 6;
```
10. **Sammler:** Implementiere eine Methode, die den Roboter bis zur nächsten Wand laufen lässt und dabei alles aufhebt, was auf dem Boden liegt. Dabei soll der Wert der gesammelten Gegenstände ermittelt werden. Jede Schraube ist dabei 20 Cent Wert, jeder Akku 2,40 Euro, jeder Schlüssel 5 Euro, jeder Feuerlöscher 40 Euro. Er soll dann mittels einer `get-`Methode jederzeit abgefragt werden können.
 Tipp: Überlege dir, ob du eine lokale Variable oder ein Attribut für den Wert verwenden möchtest. Kommazahlen kann man nicht in einer `integer`-Variable speichern. Dafür gibt es den Typ `double`. Kommazahlen werden in Java mit einem Punkt geschrieben (z.B. 2.40).
11. **Finanzamt:** Der Wert aller gesammelten Gegenstände (aus vorheriger Aufgabe) muss besteuert werden, d.h. für jeden gefundenen Gegenstand, der verkauft wird, muss die Mehrwertsteuer hinzugerechnet und ans Finanzamt abgeführt werden. Daher soll der Roboter mit `getMehrwertsteuer()` den Steuerbetrag der gefundenen Gegenstände berechnen können. Implementiere eine Methode, die die Steuer berechnet und zurück gibt.

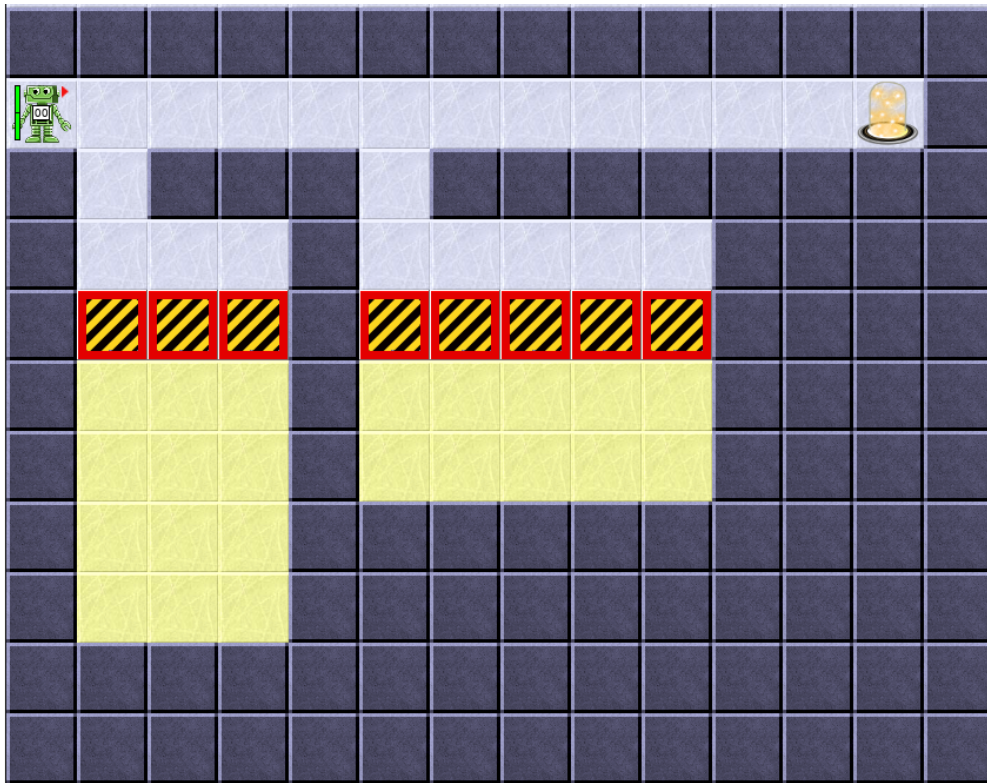




Einsatz 8: Bestimme die Lagerkapazität des Endlagers

Einige Räume im Endlager sind noch frei. Dort sollen Fässer eingelagert werden. Vorher muss aber bestimmt werden, wie viel Platz dort noch ist. Die Räume gehen alle nach rechts von einem Gang ab. Man kommt immer am linken oberen Eck in den Raum. Die ersten zwei Zeilen müssen frei bleiben, damit man mit dem Gabelstapler rangieren kann. Daher können die Fässer nur auf den gelben Feldern platziert werden.

Bestimme die Anzahl der gelben Felder und gehe zum Portal. Der Leveltest (also die Methode `einsatz8()`) muss als Ergebnis die Anzahl der Felder zurückgeben.



Bildquellen: Die verwendeten Bilder des Roboterszenarios sind alle ohne Bildnachweis verwendbar (selbst gezeichnet, Pixabay Lizenz oder Public Domain). Genaue Nachweise: siehe [bildquellen.html](https://www.bildquellen.html).



Ein Platz für ein neues Endlager ist gefunden! In einem Bergwerk soll dies geschaffen werden. Sprengarbeiten müssen vorgenommen und Brennstäbe eingelagert werden. Eine Aufgabe wie gemacht für unsere Roboter... Ein Einsatzleiter wurde bestimmt, die anstehenden heiklen Arbeiten zu koordinieren.

Methoden mit Parametern

ZIEL: Du kannst Probleme mit Hilfe von flexibel einsetzbaren Methoden lösen. Diese Methoden enthalten Parameter, mit denen sich das Verhalten der Roboter flexibel steuern lässt.

Bei manchen Einsätzen müssen Roboter situationsgerecht agieren können. So legt beispielsweise ein Roboter über den Methodenaufruf `ablegen(...)` eine Schraube, ein anderer einen Brennstab ab. Welchen Gegenstand der einzelne Roboter ablegen soll, bekommt er als sogenannten Parameter innerhalb der Parameterklammer () mitgeteilt. So bewirkt `ablegen("Schraube")` etwas anderes als `ablegen("Brennstab")`. Das kennst du schon seit AB3 Aufgabe 1. Als Parameter wird der Methode `ablegen(...)` ein String, also ein Text mitgegeben. Dieser Text muss, wie du schon gelernt hast, in Java immer in Anführungszeichen geschrieben werden.

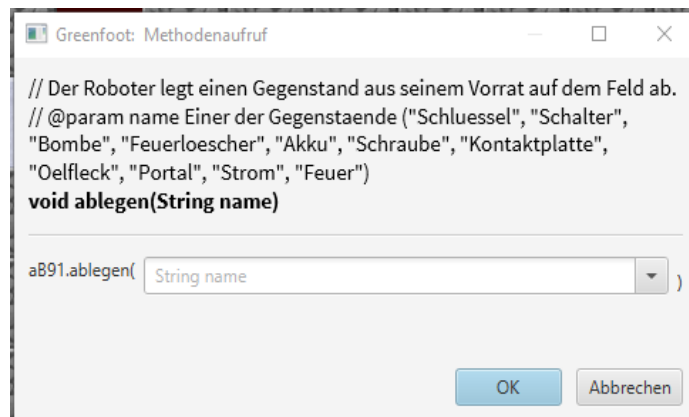
Als weitere Methoden mit Parametern hast du beispielsweise schon diese aufgerufen:

- `benutze("Feuerloescher")`
- `istVorne("Fass")`
- `istAufGegenstand("Brennstab")`

Um den Robotern diese Variabilität beizubringen, muss der Kopf der Methodenbeschreibung (oder auch Signatur der Methode genannt) wie folgt lauten:

```
public void ablegen(String name)
```

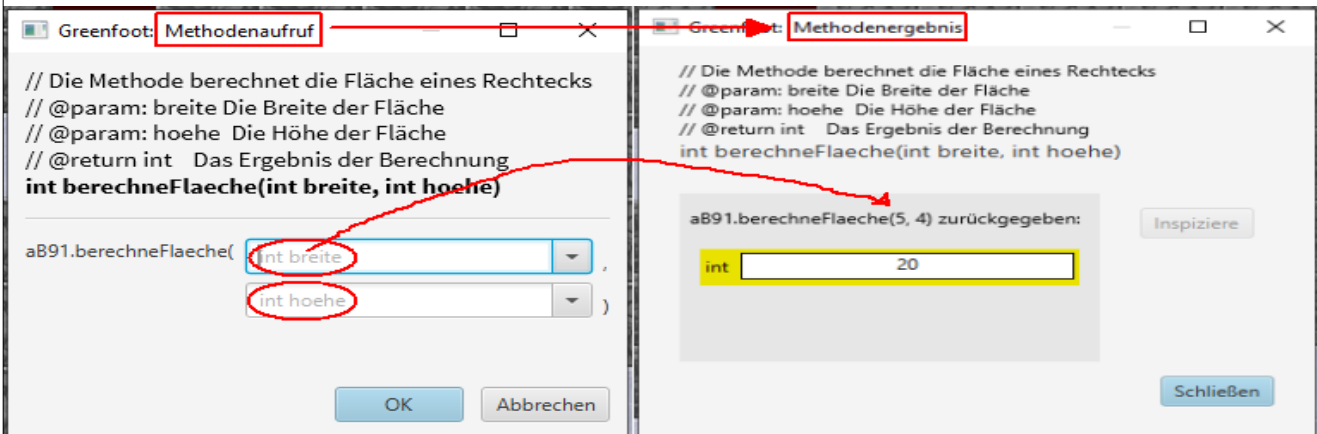
Das Schlüsselwort `String` in der Parameterklammer hinter dem Methodennamen besagt, dass bei Befehlserteilung ein Text anzugeben ist, damit der Befehl ausgeführt werden kann.



Es wird also der Name des Gegenstandes erwartet, der abgelegt werden soll. Wenn du keinen Text eingibst, weiß der Roboter nicht, welchen Gegenstand er ablegen soll. Es erscheint eine Fehlermeldung. Eine Fehlermeldung erscheint ebenfalls, wenn der Eingabewert nicht vom Typ `String`, also z.B. nicht in Anführungszeichen geschrieben ist.

Als Parameter sind alle Variablentypen denkbar. So erwartet beispielsweise die Methode mit der Signatur `public void macheWasXMal(int anzahl)` einen Integer, also eine Ganzzahl. Bei der Befehlserteilung ist also eine Zahl anzugeben, damit der Befehl ausgeführt werden kann. Gültige Methodenaufrufe wären beispielsweise `macheWasXMal(7)` oder `macheWasXMal(1234)`. Sogar `macheWasXMal(-3)` wäre erlaubt, nach einem Sinn müssten wir jedoch suchen ;-).

Manchmal ist es auch erforderlich, mehrere Parameter anzugeben. So hat die Methode `int berechneFlaeche(int breite, int hoehe)` zwei Parameter vom Typ `int`, die zur Flächenberechnung genutzt werden. Der Methodenaufruf `berechneFlaeche(3, 5)` würde somit den Wert 15 als Methodenergebnis liefern (siehe Bilder).



Beim Methodenaufruf der Methode melde(String text, boolean istWichtig) wird an erster Position die Eingabe eines Textes erwartet, an zweiter Position die Eingabe eines Wahrheitswertes true oder false, je nachdem, ob die Meldung besonders wichtig ist oder nicht.

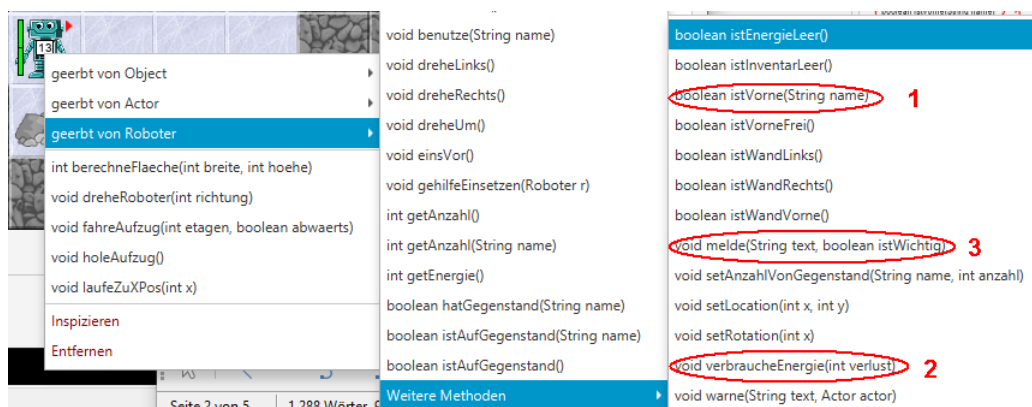
Aufgaben:

1. Markiere im unten stehenden Bild alle Methoden, die beim Aufruf eine Eingabe verlangen. Welche Methode hat die längste Parameterliste? Woran erkennst du, ob eine Methode einen Wert zurück gibt? Welche zwei Methoden haben sowohl Parameter als auch einen Rückgabewert?

```
void ablegen(String name)
int berechneFlaeche(int breite, int hoehe)
float berechneVolumen(float breite, float hoehe, float tiefe)
String gibAntwortsatz()
int gibAnzahlBrennstaebe()
float gibRadioaktivitaet()
void holeBombeUndKehreZurueck(int xPos, int yPos)
boolean istDunkel()
void legeAnzahlBrennstaebe(int anzahl)
void legeBrennstab(int x, int y, boolean aufKontaktplatte)
void legeBrennstaebe(int pos1X, int pos1Y, int pos2X, int pos2Y, int pos3X, int pos3Y)
void spreng(eint startX, int startY)
```

Du bist jetzt in Greenfoot im Level „AB9 – Methoden mit Parametern“. Alle Roboter haben (in der Klasse Roboter) schon einige Methoden mit Parametern implementiert.

2. Teste an einem AB9 die drei unten markierten Methoden (indem du bei einem Roboter rechts klickst und auf geerbt von Roboter → Weitere Methoden gehst). Beschreibe jeweils kurz, welche Aufgabe die jeweilige Methode erfüllt.





Im Bergwerk bringt ein Aufzug die Roboter in das richtige Stockwerk. Durch einen „Schalter“ kann der Aufzug in das oberste Stockwerk gerufen werden. Dafür gibt es schon die Methode „holeAufzug()“.

3. **Drehe Roboter:** Vervollständige die Methode `public void dreheRoboter(int richtung)`, die den Roboter in die angegebene Richtung dreht (0=Blick nach rechts, 90=Blick nach unten, ...). Mit `getRotation()` kann man die aktuelle Richtung erfragen.
Tipp1: Drehe den Roboter so lange, bis "richtung" erreicht ist.

Wie du schon richtig erkannt hast, schauen wir nun nicht mehr auf die Roboterwelt von oben, sondern wir betrachten einen Querschnitt eines Bergwerks. Hier spielt – wie im wirklichen Leben – die **Schwerkraft** eine wichtige Rolle. Pass auf, dass deine Roboter nicht in den Aufzugschacht fallen.



4. **Laufe zu:** Vervollständige die Methode `public void laufeZuXPos(int x)`, die den Roboter zu der angegebenen x-Koordinate laufen lässt. Die y-Position kann sich auch aufgrund der Schwerkraft verändern.

Tipp: Durch Rechtsklick auf die Welt (blauer Hintergrund) kannst du die Methode `zeigeKoordinaten()` aufrufen, somit erkennst du schnell, wie die Koordinaten eines Feldes lauten.

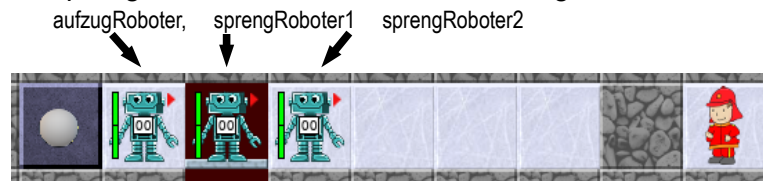
Der alte Grubenaufzug funktioniert etwas eigentümlich und muss noch programmiert werden. Bisher muss ein Roboter nach unten bzw. nach oben schauen und einsVor gehen, damit der Aufzug ein Stockwerk nach unten bzw. nach oben fährt.

5. **Fahre Aufzug:** Vervollständige die Methode `public void fahreAufzug(int stockwerke, boolean abwaerts)`, die einen Roboter, der auf einem Aufzug steht (`istAufGegenstand("Aufzug")`), die angegebene Anzahl von Stockwerken (ein Stockwerk entspricht einem Schritt) abwärts oder aufwärts fahren lässt. Diese Methode soll funktionieren, egal in welche Richtung der Roboter am Anfang schaut. Steht der Roboter nicht auf einem Aufzug, soll nichts passieren.
Bedenke: Die Roboter können in leere Aufzugsschächte stürzen, da man in diesem Level nicht von oben sondern von vorne auf die Welt schaut.
6. **Fahre ins Stockwerk:** Implementiere die Methode `public void fahreInsStockwerk(int stockwerk)`. Dabei werden die Stockwerke vom Boden ab abwärts gezählt (blaue Koordinate).
7. **Bombe sichern:** Implementiere die Methode `public void holeBombeUndKehreZurueck (int xBombe, int yBombe)`, die einen Roboter, der auf dem Aufzug steht, zur angegeben Position laufen lässt und dort eine Bombe einsammelt. Danach soll er zum Aufzug zurück kehren.
8. **Sprengen:** Nun soll ein AB9-Roboter einen Bereich im Bergwerk sprengen. Implementiere dazu die Methode `public void spreng (int xPos, int yPos)`, die einen Roboter, der auf dem Aufzug steht, zur übergebenen (x|y)-Position laufen lässt, mit benutze („Bombe“) die Bombe zündet.
9. **Legen:** Der Roboter soll einen Raum mit einer Zweierreihe von Brennstäben (von denen er ausreichend dabei hat) belegen. Implementiere dazu eine Methode, die als Parameter die Breite des Raumes bekommt. Die Schwerkraft bewirkt, dass ein abgelegter Brennstab nach unten rutscht. D.h. legt man einen Brennstab auf einen Felsen, rutscht dieser runter, wenn kein Fels oder anderer Gegenstand da liegt. Beachte, dass der Roboter sich nicht mehr nach oben bewegen kann.





Nun kommt der Einsatzleiter ins Spiel. Das ist das kleine Männchen oben rechts. Der Einsatzleiter kennt die vier Roboter. Der erste ganz links heißt aufzugRoboter, der rechts daneben sprengRoboter1, dann sprengRoboter2 und unten steht der legeRoboter.



Im Folgenden muss der Einsatzleiter jeweils Methoden bei einem der vier Roboter aufrufen. Im Quelltext schreibt man dazu beispielsweise folgenden Befehl:

```
sprengRoboter2.fahreAufzug(2,true);
```

Man schreibt also den Namen des Roboters, gefolgt von einem Punkt. Danach kommt der Name der Methode und die dazugehörigen Parameter. (Merkhilfe für die Reihenfolge: Wer macht was?) Drückt man die Tastenkombination STRG+Leertaste nach der Eingabe des Punktes, so bekommt man eine Liste aller Methoden, die man bei diesem Roboter aufrufen kann.

10. **Sprengkommando:** Öffne die Klasse *EinsatzLeiter* und vervollständige die Methode *public void holeBombeUndSpreng (int xBombe, int yBombe, int xPos, int yPos)*. Dabei sind (xBombe|yBombe) die Koordinaten, wo sich eine Bombe befindet und (xPos|yPos) die Koordinaten der Sprengposition. Der Einsatzleiter soll dazu dem aufzugRoboter und dem sprengRoboter1 die passenden Befehle geben! (Wie das geht, siehst du im Quelltextbeispiel).

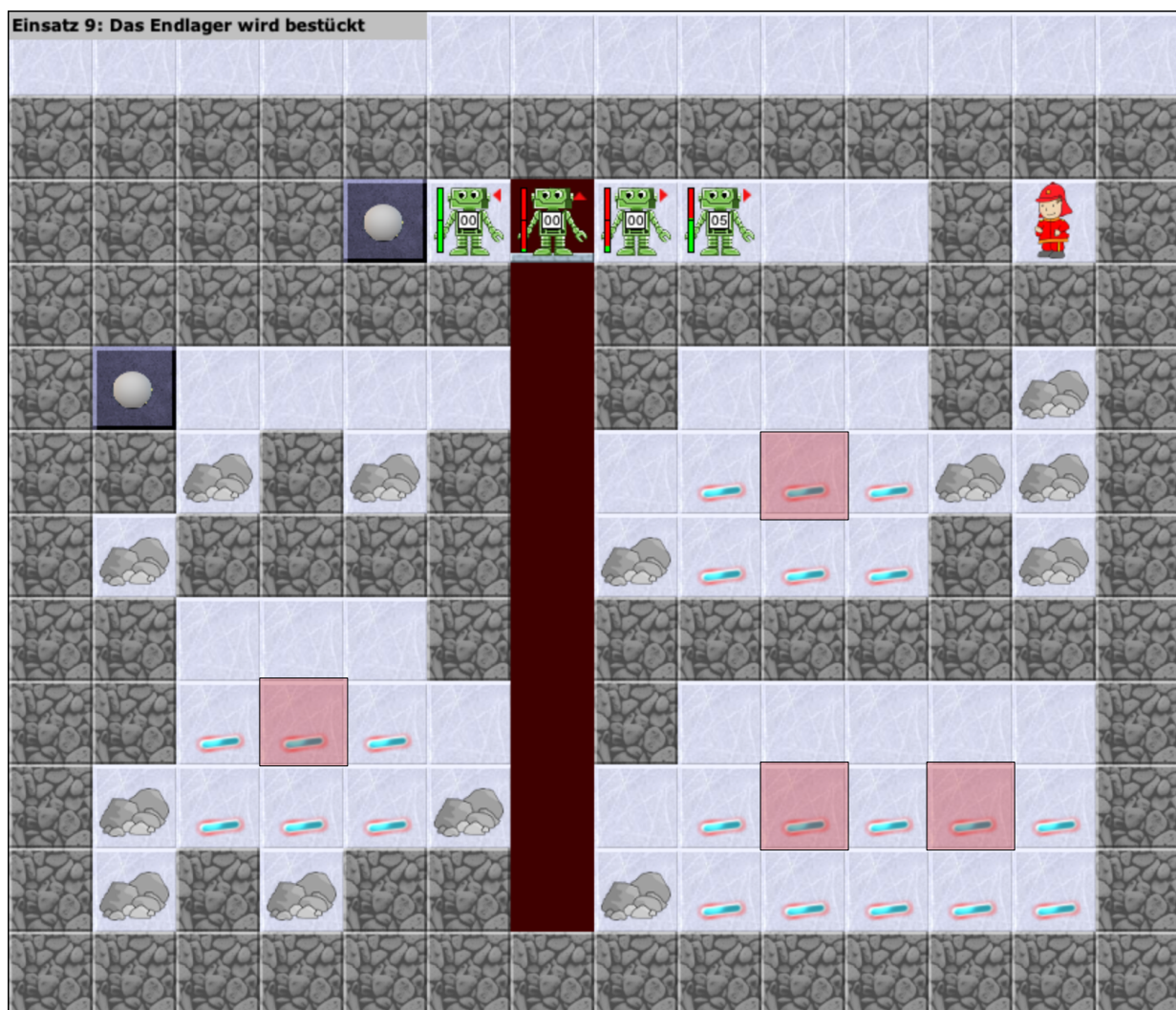


Einsatz 9: Nun übernehmen Sie die Verantwortung – Einsatzleiter!

Im Folgenden soll dein **Einsatzleiter** den Einsatz im Endlager koordinieren. Die AB9-Roboter haben alles gelernt, was sie für diesen Einsatz benötigen:

Sie können in einem Stollen Sprengarbeiten vornehmen und einen freigewordenen Bereich mit Brennstäben belegen. Aber Achtung, die Ressourcen sind knapp – sowohl die Anzahl der Sprengsätze, als auch die Energie der Roboter. Die Arbeit unter Tage ist nicht nur für Menschen anstrengend...

Deine Aufgabe ist es nun, den Einsatzleiter so zu programmieren, dass sie den Robotern klare Anweisungen gibt, um die drei verschütteten Stollen zu erweitern. Der legeRoboter soll dann seine Brennstäbe wie unten abgebildet in den frei gewordenen Hohlräumen ablegen. Hoffentlich geht ihm dabei nicht die Puste aus. Alle Roboter müssen sich am Ende wieder im oberen Eingangsstollen befinden.



Tipp: Im Bild sind die Sprengpositionen mit  markiert.

Bildquellen: Die verwendeten Bilder des Roboterszenarios sind alle ohne Bildnachweis verwendbar (selbst gezeichnet, Pixabay Lizenz oder Public Domain). Genaue Nachweise: siehe bildquellen.html.



Eine Diamantminenfirma hat vom großen Erfolg des ReaktorRobots gehört und überlegt, ob sie nicht auch die gefährlichen Arbeiten in den Diamantenminen durch Roboter ausführen lassen soll, da es immer wieder zu Stolleneinbrüchen kommt oder gefährliche Sprengungen notwendig sind. Eine letzte Aufgabe für unsere Rescue-Robots ... zum Glück kommen diese gerade aus dem Trainingslager. Schau mal an, was sie dort gelernt haben.

Gezählte Wiederholungen

ZIEL: Zählschleifen kennen und (in mindestens einer Art) in Java notieren können.

Die Methode rundeDrehen() in der Klasse **AB10** wurde umständlich notiert:

```
public void rundeDrehen() {
    laufeBisWand();
    dreheLinks();
    laufeBisWand();
    dreheLinks();
    laufeBisWand();
    dreheLinks();
    laufeBisWand();
    dreheLinks();
}
```

```
public void rundeDrehen() {
    int anzGemacht;
    anzGemacht = 0;
    while (anzGemacht < 4) {
        laufeBisWand();
        dreheLinks();
        anzGemacht++;
    }
}
```

Der Roboter umrundet gegen den Uhrzeigersinn ein von Wänden eingeschlossenes rechteckiges Feld.

Dabei macht er vier Mal das Gleiche. Sehr häufig sind einige Befehle zu wiederholen. Daher gibt es in den Programmiersprachen die Möglichkeit, solche Wiederholungen zu notieren.

Wir müssen dem Roboter mitteilen, was er wiederholen soll [laufeBisWand() und dreheLinks()] und wie oft. Dabei muss er selbst mitzählen, wie oft er es schon gemacht hat.

Er muss also SOLANGE die Befehle erneut ausführen wie die Rundenzahl kleiner ist als 4.

Die Wiederholungsrunden muss er zählen. Dafür benötigt er innerhalb der Methode rundeDrehen() eine Variable. Sie wird als ganzzahlige Variable mit dem Namen anzGemacht für die Wiederholung bereitgestellt: **int** anzGemacht; Die Anweisung anzGemacht = 0; legt den Anfangswert der Zählvariable auf Null fest. Bisher hat der Roboter auch noch keine Runde gedreht.

Der Wert dieses Wiederholungszählers muss bei jedem Schleifendurchgang um 1 erhöht werden. Daher findest du am Ende der zu wiederholenden Anweisungen den Erhöhe-Befehl für die Zählvariable: anzGemacht++;

Die Ausführungsbedingung (anzGemacht<4) sorgt dafür, dass die Schleife genau viermal durchlaufen wird: Das erste Mal mit dem Wert 0 für anzGemacht, das zweite Mal mit dem Wert 1, das dritte Mal mit dem Wert 2 sowie das vierte und letzte Mal mit dem Wert 3.

Eine **13**-fache Wiederholung kannst du daher so schreiben:

```
int i=0;
while (i < 13) {
    // Anweisungen, die wiederholt werden
    i++; // Mitzählen!!
}
```

Statt dem Namen i könntest du auch einen Namen wie wdHzaehler benutzen oder sonst einen Namen. Informatiker verwenden gern kurz und knapp i als Zählvariable. Längere Namen blähen die Ausdrücke auf. Daher werden wir meist auch i oder j als Namen für Zählvariablen verwenden.

Wenn von Anfang an feststeht, wie oft etwas wiederholt werden muss, dann kann man Zählschleifen benutzen. Es gibt eine andere Formulierung für Zählschleifen, die wir aber nicht benutzen müssen, denn jede Wiederholung lässt sich wie gesehen mit **while** formulieren. Diese spezielle Formulierung mit dem Schlüsselwort **for** lautet:

```
for (int i=0; i<13; i++) {
    // Anweisungen, die wiederholt werden
}
```

Das Schlüsselwort **for** erinnert an: **für** alle Wiederholungsdurchgänge ist Folgendes zu machen.



Diese drei Schreibweisen sind gleichwertig. Nur die letzten beiden sind Java-Standard:

```
public void vierVor() {
    wdh (4 mal) {
        einsVor();
    }
}
```

```
public void vierVor() {
    for (int i=0; i<4; i++){
        einsVor();
    }
}
```

```
public void vierVor() {
    int i=0;
    while (i < 4) {
        einsVor();
        i++;
    }
}
```

Aufgaben:

- Patrouille:** Teste mit dem Roboter unten links die Methode `rundeDrehen()` und betrachte anschließend den Quelltext. Wie oft läuft er die Strecke ab? Ändere die Methode so, dass er 10x hin und herläuft, d.h. 10x nach rechts und 10x wieder nach links. Welche Zahl darf `anzGemacht` nicht überschreiten, damit der Roboter genau 800x hin und herläuft?
- Warum wird die Variable `anzGemacht` aus der Methode `rundeDrehen()` nicht im Objektinspektor angezeigt?
- Verändere in der Methode `rundeDrehen()` die Zählschleife so, dass du anstatt einer `while`-Schleife eine `for`-Schleife verwendest. Oben stehende Tabelle kann dir als Hilfe dienen. Der Roboter soll nach Methodenaufruf genau 5x hin und herlaufen.
- Zu Befehl:** Vervollständige die Methode `dreheAnzahlRunden(int anz)` so, dass der Roboter `anz`-Runden dreht, je nach übergebenem Parameter für `anz`. So lässt der Aufruf `dreheAnzahlRunden(7)`; den AB10 sieben Mal hin und herlaufen. Blöderweise kann ihm der Strom ausgehen... zum Glück hat er jedoch drei Akkus dabei. Ergänze die Methode `dreheAnzahlRunden(int anz)` so, dass er seine Akkus sinnvoll einsetzt. (Tipp: mit `if(getEnergie())<60` kann er überprüfen, ob sein Ladezustand ausreicht.)
- Laufe x Schritte:** Implementiere eine Methode, die den Roboter genau `x` Schritte nach vorne laufen lässt.
- Was tut es?:** Analysiere das Verhalten der nebenstehenden Methode (Welche Aufgabe erledigt der Roboter?). Entscheide, welche der Schleifen sinnvoll durch eine `For`-Schleife ersetzt werden kann. Implementiere die Methode mit einer `for`-Schleife und benenne die Methode geeignet. Kann man durch einen geeigneten Aufruf der Methode alle Schrauben links unten auf einmal einsammeln?
- Aufräumen:** Implementiere die Methode `aufraeumen()`, die den Roboter rechts oben alle von oben herunterrutschende Fässer in die untere Kammer schieben lässt.
- Aufzug:** Implementiere die Methoden `fahreAufzug` und `fahreInsStockwerk` von AB9 unter Verwendung einer `for`-Schleife. Entscheide in beiden Fällen, ob die `while`-Schleife oder die `for`-Schleife geschickter war.

```
public void wastutes(int anz) {
    int i=0;
    while(i<anz) {
        while(!this.istWandVorne()) {
            aufnehmen();
            einsVor();
        }
        dreheUm();
        i++;
    }
}
```

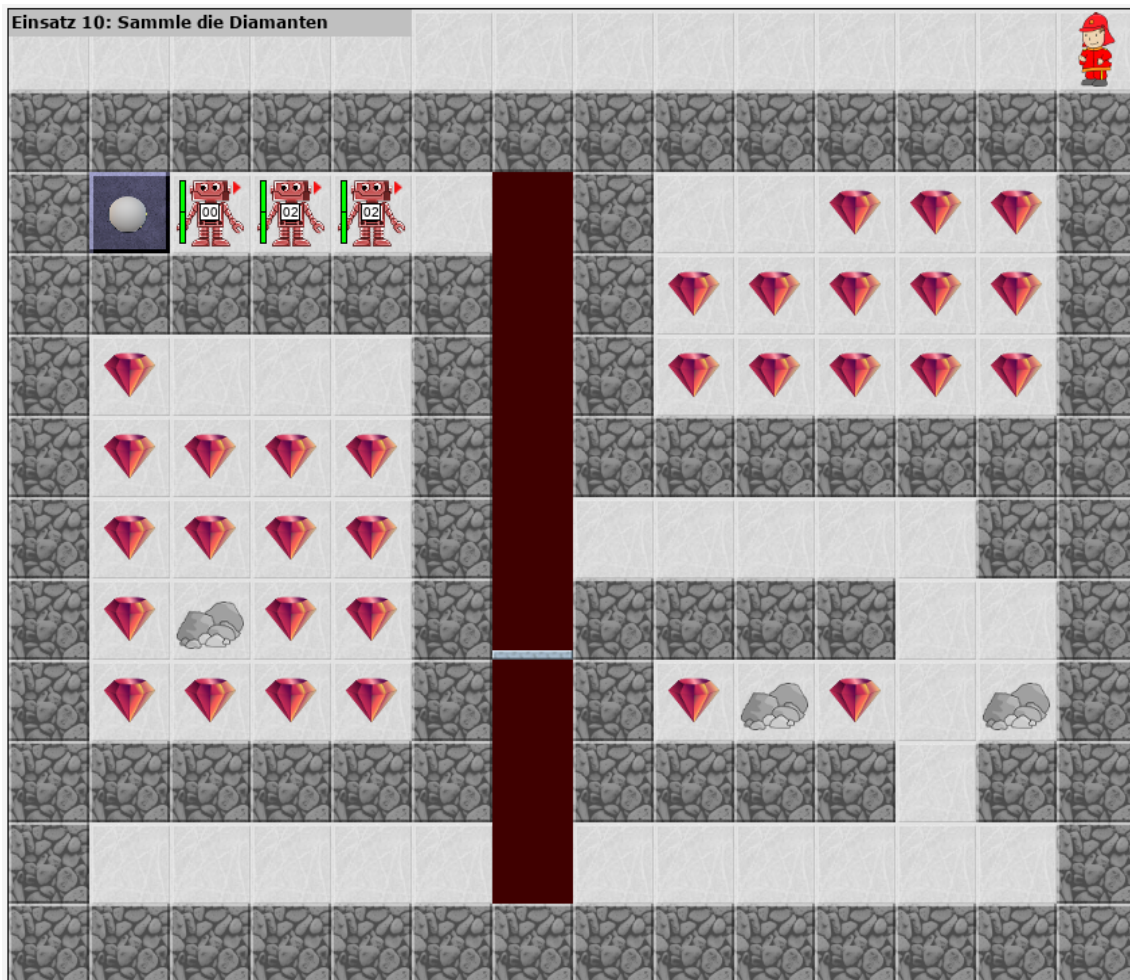


Einsatz 10:

Die Diamantminenfirma möchte nun Ergebnisse sehen. Der ReaktorRobot soll im Bergwerk Diamanten einsammeln. Der ReaktorRobot soll dabei zunächst nur seine grundsätzliche Eignung für diese Aufgabe unter Beweis stellen:

Die Unwägbarkeiten in der Minenarbeit sind riesig. Daher ist es in diesem Szenario nicht notwendig, dass die Roboter die Aufgabe jedes Mal schaffen, sondern es reicht aus, wenn er es einmal bewältigt. Sammeln alle Roboter zusammen mind. 10 Diamanten, überlegt sich die Firma den Einsatz der Roboter, bei mind. 15, werden die Roboter (vielleicht im Fortsetzungsszenario) speziell für diese Aufgabe trainiert, bei mind. 20 Diamanten ist der Roboter schon so als Bergbauarbeiter geeignet und wird sofort gekauft.

Es stehen dir 3 Roboter (roboter1, roboter2 und roboter3) zur Verfügung. Roboter 2 und 3 führen jeweils 2 Bomben mit sich. Der Einsatzleiter muss den Einsatz koordinieren. Wie sie zu ihrem Ziel kommen, ist dabei egal. Aber Achtung: Die Steine und Diamanten sind nicht immer an der gleichen Stelle. Sprengt man Stollenwände weg, fallen die Steine und Diamanten herunter. Stürzt der Roboter runter oder wird von einem fallenden Stein getroffen, verliert er Energie.



Aufgabe:

Implementiere beim Einsatzleiter die Methode `public void einsatz10()`. Implementiere bei den AB10-Robotern notwendige Hilfsmethoden. Verwende dabei – wenn möglich – for-Schleifen.

Bildquellen: Die verwendeten Bilder des Roboterszenarios sind alle ohne Bildnachweis verwendbar (selbst gezeichnet, Pixabay Lizenz oder Public Domain). Genaue Nachweise: siehe [bildquellen.html](#).