



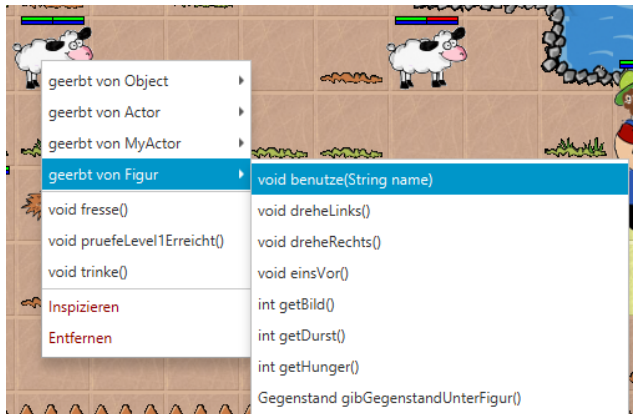
In diesem Programmierkurs sollst du auf einem Bauernhof verschiedene Tiere und den Bauern automatisch steuern und Aufgaben erledigen lassen. Bis dahin ist aber noch ein weiter Weg ...

Zunächst lernst du, ein Schaf zu steuern. Dabei gibst du dem Schaf zunächst einzelne Befehle und erforscht die Eigenschaften und Fähigkeiten der Schafe.

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

## Aufgaben:

1. Kopiere die Rohfassung des Bauernhof-Projekts (Bauernhof\_Szenario\_Roh) und benenne den Ordner um (z.B. in Bauernhof\_Arbeitsfassung). Starte das Szenario nun in Greenfoot (Szenario → Öffnen).
2. Steuere die Schafe mit passenden Befehlen aus dem Kontextmenu (Rechtsmausklick). Einige Befehle findest du direkt beim Schaf, andere bei "geerbt von Figur". Lasse das Schaf gegen einen Zaun laufen. Was passiert?

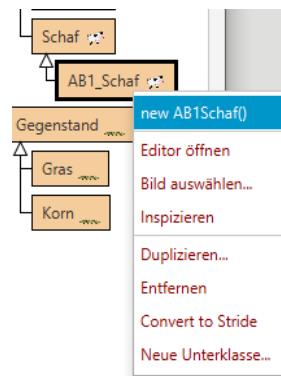


Die rot unterlegten Meldungen sind Fehlermeldungen, die bei korrekter Steuerung der Figuren nicht auftreten sollen. Kommt also eine rote Meldung hast du etwas falsch gemacht und das Programm startet neu.

3. Rufe den Befehl `istVorneFrei()` an jedem Schaf auf. Welche Antworten sind möglich? Später werden wir mit der Hilfe dieses Befehls vermeiden gegen einen Zaun zu laufen.

Die Figuren haben zwei Lebensanzeigen. Eine für die Sättigung und eine dafür, wieviel Durst sie haben. Führt die Figur Aktionen aus, werden diese Werte reduziert.

4. Untersuche, welche Befehle einem ermöglichen den exakten Wert der Lebensanzeigen (Hunger und Durst) herauszufinden. Was passiert, wenn man Gras frisst? Wie viel Energie bekommt man hinzu? Wie kann man den Durst eines Schafes löschen?
5. Rufe bei einem Schaf auf einem Grasfeld den Befehl "`pruefe(String name)`" auf. Gib "Gras" (mit den Anführungszeichen) ein. Was macht der Befehl? Was passiert, wenn man es nicht auf einem Grasfeld aufruft.
6. Was macht der Befehl "`istAufGegenstand()`"? Es gibt auch noch die Befehle "`istAuf(String name)`" und "`istVorne(String name)`". Beide erwarten, dass du sagst, was untersucht werden soll. Die Informationen beim Aufruf des Befehls sagen dir, was du eingeben kannst.
7. Was passiert beim Befehl "`warte()`"? Hast du eine Idee, wozu er gut sein könnte.
8. Erzeuge ein neues Schaf vom Typ **AB1\_Schaf** (siehe Bild rechts: Rechte Maustaste auf **AB1\_Schaf**, dann `new AB1_Schaf()`), setze es auf die Welt. Erzeuge weitere Elemente für die Welt.
9. Sorge dafür, dass alle Schafe mindestens den Wert 50 bei Hunger und Durst haben. Wenn du fertig bist, rufe in der Bauernhof-Welt mit der rechten Maustaste auf einem beliebigen bräunlichen Hintergrundfeld die Methode `checkup_01()` auf (nicht direkt beim Schaf!). Warst du erfolgreich, kannst du in den nächsten Level wechseln, indem du wieder in der Bauernhof-Welt die Methode `geheZuLevel` aufrufst und im Eingabefeld die Zahl 2 eingibst.



**Zusammenfassung:** Du kannst Greenfoot starten, ein Szenario laden, Objekte erzeugen und nutzen, einer Figur über ihr Kontextmenü Befehle geben.

Abb 1: Kontextmenü der Klasse AB1

**Bildquellen:** Die verwendeten Bilder des Bauernhofes sind alle selbst gezeichnet und stehen unter CC BY-SA-NC 3.0 Lizenz)



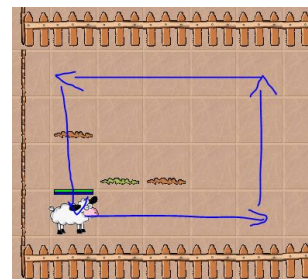
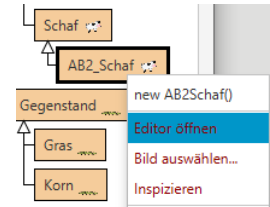
Du hast die ersten Schritte gemeistert. Allerdings hast du die Schafe bisher von Hand gesteuert. Wer kann schon einem Schaf den lieben langen Tag sagen, was es tun soll? Ok, ein Schäferhund vielleicht... Die Figuren sollen nun lernen, sich alleine zu bewegen.

## Die Figuren lernen dazu ...

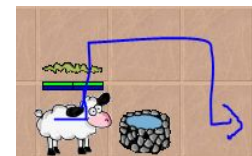
**ZIEL:** Wissen, dass alles, was die Figuren ausführen können, im Quelltext notiert ist. Vorhandene Quelltexte ergänzen und erweitern können.

### Aufgaben:

1. Welche Befehle bietet dir ein **AB2-Schaf** in seinem Kontextmenü direkt an (also nicht „geerbt von“)? 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 AB2Schaf) anzeigen lassen. Jede Fähigkeit ist in einer sogenannten Methode im Quelltext beschrieben.
2. Steuere das Schaf unten links noch ein letztes Mal von Hand durch einzelne Befehle so, dass es eine Runde dreht wie im Bild (siehe rechts). Die bekannten Befehle findest du jetzt im Kontextmenu unter „geerbt von Schaf“ bzw. „geerbt von Figur“. Welche Befehle hast du ihm dazu gegeben?



3. **Drehe Runde im Gatter:** Öffne den Quelltext der Klasse **AB2\_Schaf**. Ergänze die Anweisungen in `dreheRunde()`, damit es eine vollständige Runde wird. Es soll 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 das Schaf keine Bedeutung) Übersetze und erprobe die veränderte Methode `dreheRunde()`.
4. **Drehe um:** Schreibe im Quelltext die Anweisungen für `dreheUm()`.
5. **Fresse dich satt:** Bringe dem **AB2-Schaf** bei, drei Grasstücke abzufressen, die direkt hintereinander liegen. Du kannst davon ausgehen, dass sie schon genügend gewachsen sind. Teste die neue Methode am Schaf rechts unten. Welche Änderung am Code sind nötig, um zu beeinflussen, ob das Schaf mit dem Grasbüschel direkt unter ihm oder mit dem vor ihm beginnt.
6. **Haken schlagen:** Sorge dafür, dass das **AB2-Schaf** frei stehende Brunnen umlaufen kann (s. Bild rechts). Wie nennst du diese Fähigkeit eines **AB2-Schafs**? 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. Schaue dieses Konzept bei den bestehenden Methoden ab. Wenn ein **AB2-Schaf** genau vor einem Brunnen 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.



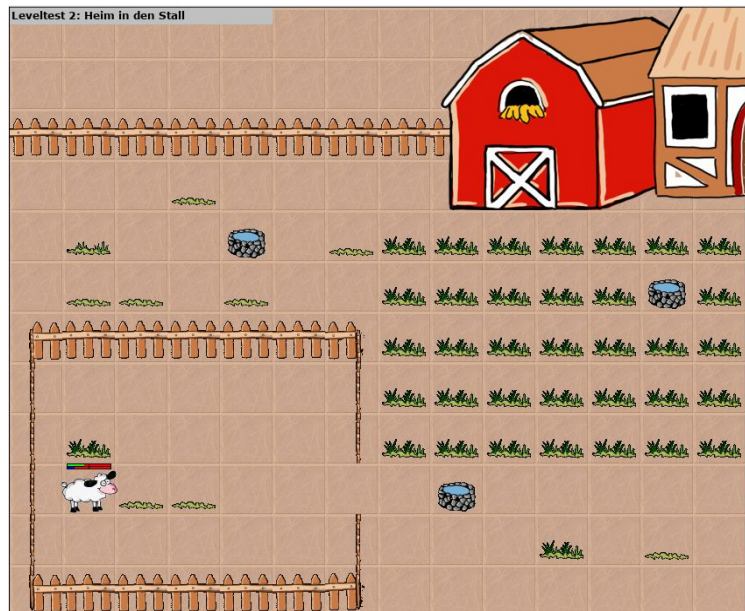


7. **Drehe Runde Variante 2:** Man kann auch die gerade erst selbst programmierten Methoden in eigenen Methoden nutzen. Implementiere dazu nochmal die Methode `dreheRunde`. Kopiere zunächst den ganzen Quelltext der Methode. Ersetze dann, wo immer möglich mehrere Befehle durch die Methode `dreiSchritte()`. Dadurch werden Programme kürzer und besser lesbar.
8. **Falsche Namensgebung:**  
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();`

Teste abschließend deine Implementierungen mit dem Befehl `checkup02()`, den du auf einem freien Feld mit der rechten Maustaste starten kannst. Wenn alles richtig ist, kannst du direkt zum nächsten Level gehen (`geheZuLevel()`) oder dich noch am Leveltest 2 versuchen.

## Leveltest 2: Bring das Schaf in den Stall

Das Schaf muss abends in den Stall. Auf ihrer kargen Weide gab es nicht viel zu fressen und auch keinen Brunnen zum Trinken. Daher wird es sich auf dem Weg zu Stall stärken müssen.  
Es kommt ohnehin an einem Brunnen vorbei und überquert eine saftige Wiese, die schon lange nicht mehr abgegrast wurde.



Implementiere für diesen Einsatz die Methode `leveltest2()` 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 kannst mit unter 10 Befehlen auskommen!)

Um den Einsatz durchzuführen, musst du in der Bauernhof-Welt mit der rechten Maustaste auf einem beliebigen bräunlichen Hintergrundfeld die Methode `leveltest_02()` aufrufen (nicht direkt beim Schaf!).

Ich bin mal gespannt, ob du deine erste richtige Aufgabe 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 Figuren Befehle geben, die sie dann selbständig ausführen!

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



Nicht jede Wiese ist gleich groß, der Stall ist nicht immer gleich weit weg. Trotzdem sollen die Schafe richtig reagieren. Daher müssen sie ihre Umwelt wahrnehmen und darauf reagieren.

## Die Schafe machen etwas immer wieder...

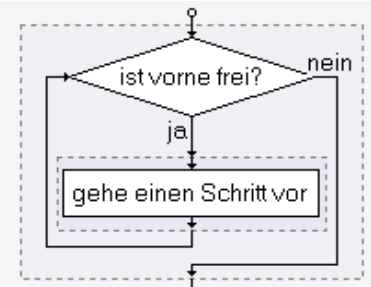
**ZIEL:** Wiederholungen in Handlungen erkennen, als SOLANGE-Schleife formulieren und in Programmiersprache umsetzen können. Methoden mit Parametern benutzen können.

### Aufgaben:

- Wiese erkunden:** Die Schafe sollen eine Runde auf ihrer Wiese drehen, egal wie groß diese Wiese ist. Dazu gibt es eine Methode `vorBisZaun()`. Teste diese Methode und lies dann im Quelltext nach, wie sie implementiert wurde. Setze diese Methode ein, um die Schafe eine Runde entlang des Zauns ihrer Wiese drehen zu lassen. Teste diese Methode an verschiedenen Schafen.

Der Auftrag `vorBisZaun()` muss korrekt ausgeführt werden, egal wie weit der Zaun entfernt ist. Daher muss das Schaf prüfen, wie lange es vorwärts gehen muss. Auf die Anfrage: `istVorneFrei()` liefert es die Antwort `true` bzw. `false`. Damit können wir diese bedingte Wiederholung so formulieren:

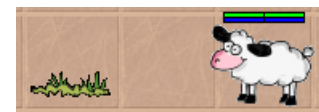
<i>-- normierte Sprache</i> <b>SOLANGE</b> (vorne frei ist) <b>WDH:</b> ein Schritt vor <b>ENDE WDH</b>	<i>-- Programmiersprache</i> <code>while (istVorneFrei())</code> <code>{</code> <code>einsVor();</code> <code>}</code>
---	--



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.

- Zum Gras:** Jedes Schaf muss fressen und dafür muss es einen schönen Grasbüschel finden. Es gibt eine Methode `istAufGegenstand()`, die testet, ob eine Figur auf einem Gegenstand (also z.B. auch Gras) steht. Nutze diese Methode, um die Schafe bis zum nächsten Grasbüschel laufen zu lassen und danach das Gras zu fressen. Hinweis: Es muss also solange laufen, wie es (noch) nicht auf einem Grasbüschel steht. In Java hat `while(!Bedingung)` die Bedeutung "solange die Bedingung nicht erfüllt ist, mache...". Das Ausrufezeichen heißt also "nicht".



Teste die Methode an verschiedenen Schafen. Beschreibe, was passiert, wenn das Schaf schon auf einem Grasbüschel steht oder nirgends vor dem Schaf Gras zu finden ist.

- Scratch:** In Scratch gibt es auch eine vergleichbare Schleife. Analysiere, welchen entscheidenden Unterschied es zwischen dem Scratch-Programm und dem Java-Programm gibt.



```

while (getHunger() > 50)
{
    einsVor();
}
  
```

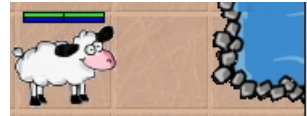


Wir haben bisher Methoden benutzt, die keine zusätzliche Information benötigen, um ausgeführt werden zu können. Viele Methode brauchen aber zusätzliche Informationen. Diese werden als **Parameter** bezeichnet.

Die Methode `istVorne(String name)` testet beispielsweise, ob die Figur vor einem Feld mit "name" steht. Dabei können name verschiedene Dinge (z.B. "Wasser", "Gras", "Zaun") sein. Also z.B. `while(istVorne("Zaun")) ...`

4. **Methode mit Parameter:** Teste diese Methode direkt an einem Schaf. Der Kommentar beim Aufruf gibt an, welche Werte der Parameter haben kann. Untersuche, woran man bei einer Methode erkennen kann, ob sie einen Parameter benötigt.

5. **Zum Wasser:** Jedes Schaf muss mal trinken. Aber der Teich oder der Brunnen sind nicht immer gleich weit weg. Es muss also so lange laufen, wie kein Wasser vorne ist.



Implementiere die Methode und teste sie an allen drei Schafen oben.

Bisher haben wir Methoden für die Bedingungen der Schleifen verwendet, die nur `true` oder `false` zurückgegeben haben. Andere Methoden geben aber Zahlen oder Texte zurück. Möchte man diese für eine Bedingung verwenden, muss man das Ergebnis der Methode mit einem Wert vergleichen. Für die Vergleiche gibt es folgende Zeichen:

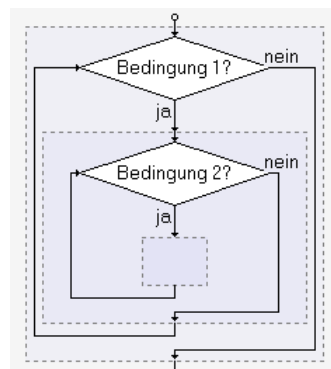
gleich	ungleich	größer	größer oder gleich	kleiner	kleiner oder gleich
<code>==</code>	<code>!=</code>	<code>&gt;</code>	<code>&gt;=</code>	<code>&lt;</code>	<code>&lt;=</code>

z.B. `while(getHunger() < 70)` oder `while(getDurst() != 0)`

6. **Zum Wasser und trinken:** Ergänze `vorBisWasserUndTrinke()` so, dass das Schaf solange trinkt, wie es noch Durst hat (der Wert 100 noch nicht erreicht ist).

7. **Satt fressen:** Das Schaf links unten steht auf einer saftigen Wiese. Implementiere eine Methode, die das Schaf die Wiese so lange abfressen lässt, bis eine Sättigung von mindestens 80 erreicht ist (die vier Grasfelder reichen dazu auf jeden Fall). Teste die Methode mehrmals. Drücke dazwischen den "Reset-Knopf". Prüfe, ob der gewünschte Wert auch wirklich jedes Mal erreicht wird.

8. **Faules Schaf:** Fred ist faul. Statt über die Wiese zu laufen und nach schönen Grasbüscheln zu suchen, wartet er einfach, bis das Gras genügend gewachsen ist. Implementiere eine Methode, die das Schaf so lange warten lässt, bis das Gras auf dem es steht vollständig gewachsen ist und es dann frisst. Teste deine Methode an den drei Schafen in den Gattern. Hinweis: Mit der Methode "pruefe(String name)" kannst du das Gras testen. Wenn der Wert 15 erreicht ist, ist es vollständig gewachsen. Für den Parameter "name" musst du natürlich wieder den richtigen Wert einsetzen.



9. **Faules Schaf 2:** Von einem Grasbüschel wird Fred natürlich nicht satt. Verbessere die Methode so, dass die Befehle von Aufgabe 8 solange immer wieder ausgeführt werden, wie ein Sättigungsgrad von 80 noch nicht erreicht wurde. Hinweis: Man kann Schleifen ineinander schachteln.

10. **Zurück zum Stall:** Die Methoden `getX()` und `getY()` liefern die Koordinaten der aktuellen Position des Schafs. `getRotation()` liefert die Richtung, in die es schaut (0 = rechts, 90 = unten, 180 = links, 270 = oben). Implementiere eine Methode, die das Schaf zunächst solange dreht bis es nach links schaut, dann nach links laufen und am Ende nach oben laufen lässt, bis es im Stall ist. Der Stall hat die Position x=6 und y=3. Teste deine Methode an allen drei Schafen oben.





Überprüfe wie immer deine Implementierungen am Ende des Levels mit der entsprechenden checkup-Methode. Die komplexere Zusatzaufgabe (Leveltest) ist auch dieses Mal eine Bonusaufgabe. Führe diesen Schritt ab sofort am Ende jedes Levels durch.

## Leveltest 3: Karges Land

„Mäh, mäh, mäh, wo seid ihr alle?“ Das arme kleine Schaf hat den Anschluss verloren und steht als einziges noch auf dem Hof des Bauernhofes. Es ist schon ganz ausgehungert, aber auf dem Hof findet man fast nichts zu fressen. Sorge trotzdem dafür, dass es sicher in den Stall kommt.

Irgendwo vor dem Schaf ist ein karges Grasfleckchen, weiter vorne ein Brunnen. Die exakte Position ist jedes Mal etwas anders.

Versuche mit möglichst wenigen Befehlen das Schaf in seinen Stall zu bekommen. Nutze möglichst die zuvor programmierten Methoden.




**Tip:** Zusammengesetzte Bedingungen kann man mit `&&` (= und) bzw. `||` (=oder) formulieren.

**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. Aber auch Vergleiche wie „größer“ können verwendet werden.

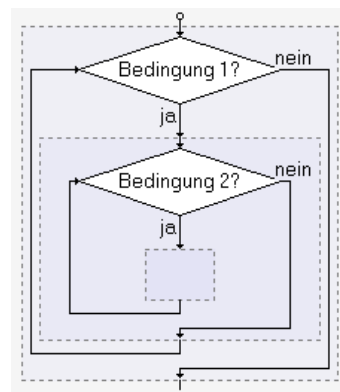
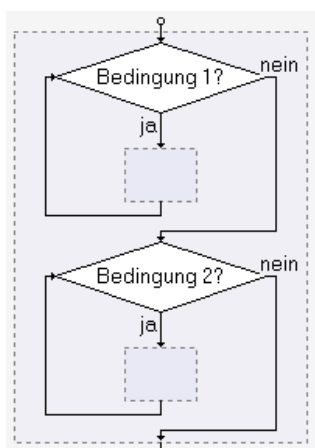
Im Quelltext schreibt man Wiederholungen mit dem Schlüsselwort **while** im Schleifenkopf und dahinter eine Ausführungsbedingung in runder Klammer() sowie einen Schleifenrumpf innerhalb des Blockklammerpaares {...}.

```
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.

Mehrere Schleifen können nacheinander ausgeführt werden oder ineinander verschachtelt werden:





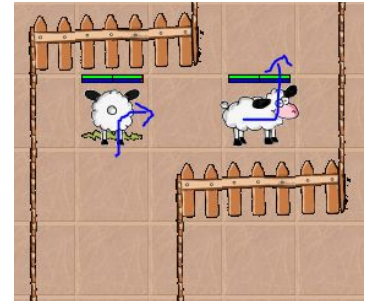
Bisher war es einfach. Alles Grüne konnte man fressen. Es gibt aber auch besondere Leckerbissen - Blümchen. Aber aufgepasst: Die blauen sind giftig! Da muss sich ein Schaf schon überlegen, ob es die einfach so fressen möchte.

## Die Figuren treffen Entscheidungen ...

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

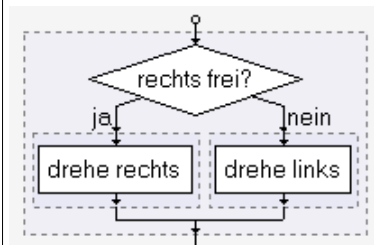
In vielen Situationen sind Anweisungen nur unter bestimmten Bedingungen auszuführen. Folgt ein Schaf einem Gang und trifft auf einen Zaun vor ihm, muss es entscheiden, ob es sich nach links oder rechts drehen soll. Dazu muss es prüfen, ob rechts frei ist. Wenn nicht, muss es links weitergehen.

Derartige Entscheidungen trifft man anhand von Prüfbedingungen, die wie bei den while-Schleifen wahr oder falsch sein können.



```
-- normierte Sprache
FALLS (rechts frei ist)
DANN
    drehe nach rechts
*ENDE DANN
SONST
    drehe nach links
*ENDE SONST
```

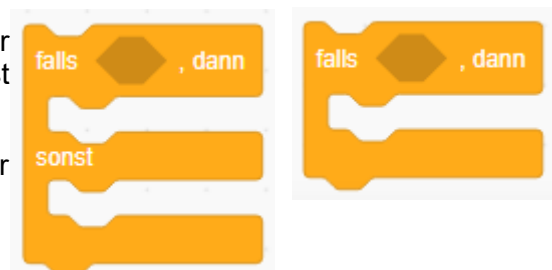
```
-- Programmiersprache
if(istRechtsFrei())
{
    dreheRechts();
}
else {
    dreheLinks();
}
```



Vorsicht:

statt ~~if (Prüfbedingung) then {...}~~  
steht nur ~~if (Prüfbedingung) {...}~~

Man nennt solche Entscheidungen in der Programmierung auch **Verzweigungen**. Du kennst sie schon von Scratch (siehe rechts).



Genauso wie bei Scratch, kann auch in Java der else-Teil weggelassen werden.

### Aufgaben:

- Drehe richtigum:** Implementiere die Methode `dreheRichtigum()`, die das Schaf nach rechts drehen lässt, wenn dort frei ist. Andernfalls soll es sich nach links drehen. Teste die Methode an dem Schaf links unten und am Schaf vor dem Teich.
- Gerade aus oder drehen:** Implementiere eine Methode, die das Schaf einen Schritt nach vorne machen lässt, wenn dies möglich ist. Andernfalls soll es sich in die freie Richtung drehen.  
Hinweis: Du kannst entweder zwei Verzweigungen ineinander verschachteln oder die Methode von Aufgabe 1 nutzen.
- Zurück in den Stall:** Die Schafe sollen gemäß den Regeln von Aufgabe 2 laufen, bis sie im Stall ankommen. Sie sind solange nicht im Stall, wie die x-Koordinate nicht 7 oder die y-Koordinate nicht 3 ist. Teste diese Methode an allen vier Schafen. Eines kommt nicht an. Warum?  
Hinweis: Zusammengesetzte Bedingungen kann man mit `&&` (= und) bzw. `||` (=oder) formulieren.
- Zurück in den Stall 2:** Ergänze die Methode von Aufgabe 3 so, dass die Schafe alles Gras fressen, was unterwegs wächst.



5. **Lecker Blümchen:** Implementiere eine Methode, die das Schaf bis zum nächsten Zaun laufen lässt und dabei alle leckeren Blümchen unterwegs frisst. Achtung: die blauen sind giftig.



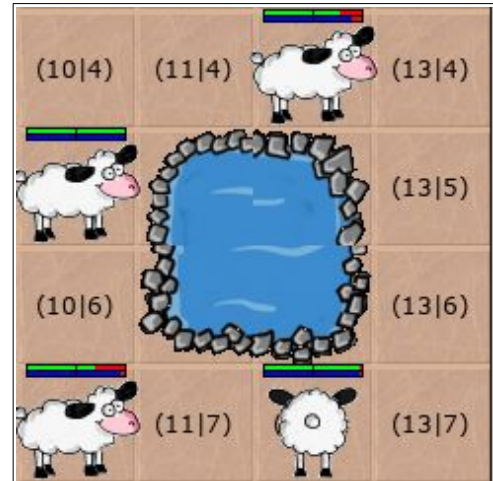
Hinweis: Rufe pruefe("Blume") direkt am Schaf auf, um herauszufinden, wie man giftige Blumen erkennen kann..

```
if (istAuf("Gras")) {
    fresse();
}
if (istVorneFrei()); {
    einsVor();
}
```

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 jedes der vier AB4-Schafe im Bild rechts ein, wie sie sich bewegen, wenn sie diese Anweisungen ausführen:

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

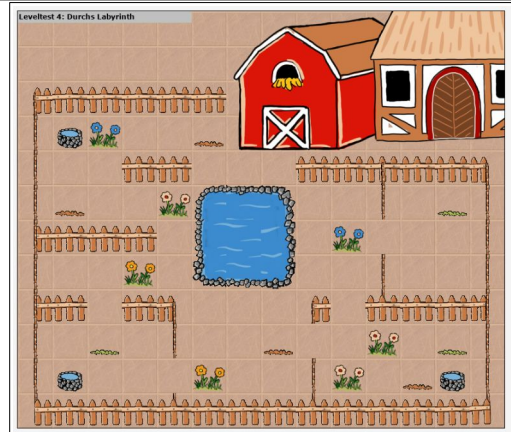


8. Gib an, welche Ausdrücke von A bis E nicht als Prüfbedingung in einer Verzweigungsanweisung **if (...)** oder in einer **while**-Schleife benutzt werden können.
- A: (!istVorneFrei())      B: (getHunger() > 5)      C: (getX())  
 D: (istAuf("Gras") && (getHunger() > 0))      E: (einsVor())



## Leveltest 4: Durchs Labyrinth

Das eine Schaf kommt mit der Strategie von den Aufgaben 1-3 nicht in den Stall, sondern läuft immer im Kreis. Eine Strategie den Ausgang in einem Irrgarten zu finden, ist die linke Hand an eine Wand zu legen und dann immer an dieser Wand so entlang zu laufen, dass die Hand an der Wand bleibt. Ist der Irrgarten so gebaut, dass man nicht im Kreis laufen kann, so findet man auf diese Weise garantiert den Ausgang. Man kann dabei in diese drei Situationen gelangen:



Überlege dir, wie man erkennen kann, in welcher der drei Situationen das Schaf ist. Es gibt neben `istVorneFrei()` auch die Methoden `istRechtsFrei()` und `istLinksFrei()`.

ist links frei	links nicht frei	
	vorne ist frei	vorne ist nicht frei
Man hat die Wand verloren, da sie geendet hat. Dann biegt man links ab und geht eins vor. Damit steht man wieder neben der Wand.	Wenn aktuell links eine Wand ist, geht man entweder nach vorne oder muss rechts abbiegen, weil vorne kein Platz ist. Beachte, dass das Schaf nach dem Rechts-Abbiegen keinen Schritt gehen darf, da es sich manchmal gleich nochmal drehen muss	

Hinweis 1: Am Anfang muss man natürlich erst mal noch einen Zaun finden, also so lange nach vorne laufen, bis man einen gefunden hat und sich dann so drehen, dass er auf der linken Seite ist.

Hinweis 2: Und unterwegs sollte man genügend fressen und trinken.

**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.



Die Schafe können schon eine ganze Menge. Viel mehr lernt so ein Schaf auch nicht. Daher wenden wir uns nun dem Bauern zu und bringen ihm einiges bei...

## Neue Sensoren ...

**ZIEL:** Methoden mit boolschen Rückgabewerten erstellen können. Parameter verwenden können, um Methoden Zusatzinformationen mitzugeben.

Unsere Figuren haben nur eine beschränkte Anzahl von Methoden, die die Umwelt der Figur wahrnehmen (z.B. `istVorneFrei()`, `istVorne("Wasser")`). In diesem Arbeitsblatt lernst, du neue (komplexere) Sensoren selbst zu erstellen. Schau dir dazu die Methode `istAufLeeremAcker()` des `AB5_Bauer` an. Im Gegensatz zu den Methoden, die du bisher implementiert hast, gibt diese eine Antwort zurück (wie z.B. auch `istVorneFrei()`).

### Aufgaben:

1. Teste die Methode `istAufLeeremAcker()` an den verschiedenen Bauern. Wie beantwortet der Bauer diese Anfrage? Welche Antwortmöglichkeiten gibt es?
2. Analysiere nun den Quelltext zur Methode `istAufLeeremAcker()`: Wo tauchen die Antwortmöglichkeiten (`true` und `false`) im Quelltext auf? Welcher Befehl sorgt dafür, dass eine Antwort zurückgegeben wird? Wie unterscheidet sich der Methodenkopf, d.h. die mit `public ...` beginnende Zeile, von den bisherigen Methodenköpfen?

```
public boolean istAufLeeremAcker() {  
    if(istAuf("Acker") && !istAuf("Tomate")) {  
        return true;  
    } else {  
        return false;  
    }  
}
```

Alle bisherigen Methoden wurden mit `public void methodeName() {...}` programmiert. `Void` steht dabei für leer/nichts. `Void` steht an der Stelle, an der der sogenannte Rückgabotyp steht. Es wird also nichts zurückgegeben. Möchte man nun mit `true/false` antworten, muss man den Rückgabewert als `boolean`<sup>1</sup> deklarieren. 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. **Erntebereit:** Reife Tomaten kann man mit `ernte("Tomate")` ernten. Damit dabei kein Fehler passiert, muss man testen, ob man überhaupt auf einer Tomate steht und ob diese reif ist. Den Reifegrad kann man mit `pruefe("Tomate")` bestimmen. Hat er den Wert 100, kann die Tomate geerntet werden. Implementiere die Methode `boolean istAufReiferTomate()`, die diesen Test vornimmt. Hinweis 1: Dieser Test soll die Tomate nicht ernten. Tests sollten eigentlich nie Änderungen vornehmen, da der Benutzer das nicht erwartet. Hinweis 2: In AB4 Aufg. 3 wird erklärt, wie man zwei Bedingungen verknüpfen kann.
4. **Gießen:** Die Tomaten wachsen leider nicht so leicht wie Gras. Tomaten haben einen sehr hohen Wasserbedarf. Daher muss man sie mit `benutze("Giesskanne")` gießen. Vorher sollte man aber prüfen, ob man auf einer Tomate steht und sie Wasser benötigt (`pruefe("Tomate")` liefert dann den Wert -1). Implementiere die Methode `boolean istAufDurstigerTomate()`, die diesen Test vornimmt.
5. **Out of Power:** Implementiere eine Methode `istEnergieSchwach()`. Diese soll `true` zurückgeben, wenn mindestens einer der Energiewerte (Hunger und Durst) des Bauern auf weniger als 40 Energiepunkte gesunken ist. Ansonsten gibt sie `false` zurück. Teste deine Methode an den unterschiedlichen Bauern.

<sup>1</sup> 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.



## Parameter

Viele Methoden benötigen zusätzliche Informationen, um korrekt arbeiten zu können. Du hast z.B. schon die Methoden `pruefe` oder `istAuf` kennen gelernt. Beide erwarten einen Text als Parameter. Ihr Methodenkopf sieht folgendermaßen aus:

```
public boolean istAuf(String name) {...}
```

```
public int pruefe(String name) {...}
```

Man gibt also in der runden Klammer hinter dem Methodennamen die Parameter an. Dabei wird zuerst der Typ genannt (hier `String`) und dann der Name des Parameters (hier `name`). Diesen Parameter kann man dann überall in der Methode nutzen. Dabei wird nur noch der Name angegeben und nicht mehr der Typ.

6. **Besitztümer:** Der Bauer kann Gegenstände mit sich herumtragen. Mit `getAnzahl(String name)` kann man testen, wie viele Gegenstände eines bestimmten Typs er bei sich hat. Untersuche die Methode `hatGegenstand` des `AB5_Bauer`. Welchen Parameter erwartet diese Methode? Teste die Methode an den Bauern. Wie wirkt es sich aus, dass die Methode einen Parameter hat? Wo wird dieser Parameter innerhalb der Methode genutzt?
7. **Gießbereit:** Implementiere eine Methode `boolean kannGiessen()`, die testet, ob der Bauer eine Gießkanne ("Giesskanne") hat und sie gefüllt ist, d.h. das Prüfen der Gießkanne einen Wert größer als 0 ergibt.  
Hinweis: Hier musst du noch keinen eigenen Parameter definieren, sondern siehst wie man eine Methode mit Parameter nutzt.
8. **Blick nach links:** Implementiere eine Methode `boolean istLinks(String name)`, die testet, ob links vom Bauern ein bestimmter Gegenstand ist. Der Name dieses Gegenstandes ist der Parameter der Methode. Da der Bauer nur die Methode `istVorne(String name)` kennt, musst du den Bauern zunächst nach links drehen. Am Ende soll der Bauer wieder so stehen wie am Anfang.  
Hinweis: Befehle nach der `return` Anweisung werden nicht mehr ausgeführt. An welcher Stelle muss man den Bauern wieder zurückdrehen?
9. **Positionslauf:** Implementiere eine Methode `laufeBisY(int y)`, die den Bauern so lange vorwärts gehen lässt, wie die durch den Parameter angegebene y-Koordinate (`getY()`) noch nicht erreicht ist. Du kannst davon ausgehen, dass die angegebene y-Koordinate vor dem Bauern liegt und er sich nicht dafür drehen muss. Teste deine Methode. Welche Art von Parameter erwartet sie?  
Für Experten: Drehe den Bauer zusätzlich zunächst in die richtige Richtung.

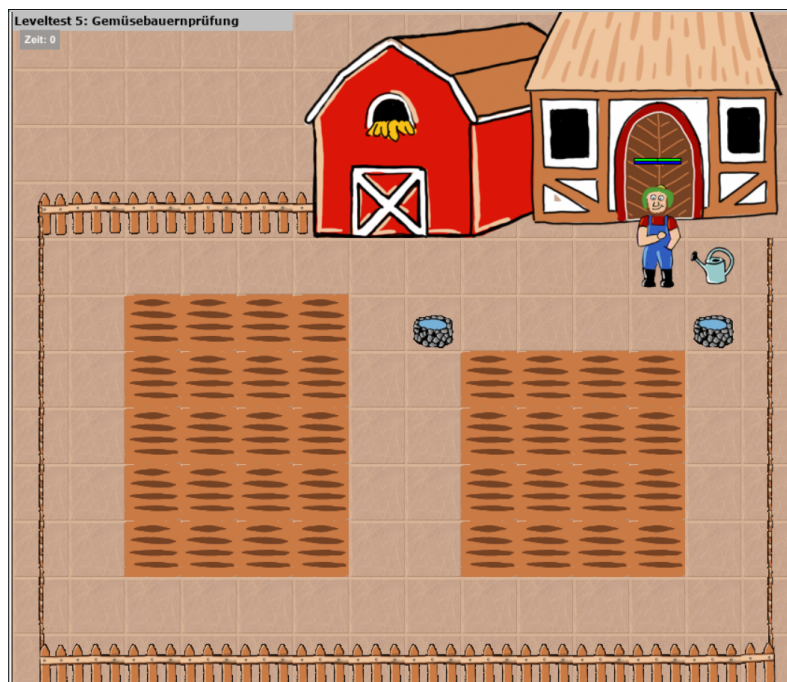


## Leveltest 5: Gemüsebauernprüfung

Dein Bauer ist bereit: Er hat sich für die Gemüsebauerprüfung angemeldet. Seine Aufgabe ist es, Tomaten anzubauen und mindestens 20 Tomaten zu ernten. Je schneller er diese Aufgabe erledigt, desto besser.

Er hat am Anfang vier Tomaten. Diese kann er pflanzen, indem er sie auf einem leeren Ackerfeld ablegt. Danach muss er fleißig gießen, bis die Tomaten herangereift sind. Sind sie schön rot, kann er sie ernten. Tomatenpflanzen produzieren bei ausreichender Wasserversorgung immer wieder neue Tomaten, es müssen keine neuen gepflanzt werden.

So viel Arbeit ist natürlich anstrengend. Daher muss der Bauer immer mal wieder eine Vesperpause in seinem Haus machen, um seine Energiereserven aufzufüllen.



### Hilfreiche Methoden:

<code>void ablegen(String name)</code>	der Bauer legt einen Gegenstand aus seinem Inventar ab.
<code>void aufnehmen()</code>	der Bauer hebt einen Gegenstand vom Boden auf.
<code>void ernte(String name)</code>	der Bauer kann "Tomate" und "Korn" ernten, wenn es reif ist.
<code>void benutze(String name)</code>	der angegebene Gegenstand wird benutzt. Benutzt man eine Gießkanne ( <code>benutze("Giesskanne")</code> ) vor einem Wasserfeld, wird sie gefüllt. Benutzt man sie auf einem Acker, dann gießt man den Boden damit.
<code>void vesperpause()</code>	der Bauer macht eine Pause im Haus. Er muss dazu direkt vor der Tür zum Haus stehen.
<code>int getAnzahl(String name)</code>	gibt die Anzahl der Gegenstände vom Typ name zurück, die der Bauer mit sich herumträgt.



Hühner sind schlauer als viele vermuten. Besonders wenn DU die Hühner programmierst. Dann können sie sich sogar dauerhaft etwas merken und Fragen dazu beantworten...

## Die Figuren merken sich was dauerhaft ...

**ZIEL:** Attribute als Speicher für jeweils einen Wert kennen und einsetzen können. Den Konstruktor einer Klasse kennenlernen und zum Initialisieren von Attributen verwenden können.

## Attribute beschreiben Eigenschaften und Zustände

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

Natürlich mit einer Variablen. Die hast du ja schon in Scratch kennengelernt. Jedes Objekt verwaltet dabei seine Variablen selbst. Wenn also mehrere Figuren auf der Welt herum laufen, dann kann jede einen unterschiedlichen Wert für hunger speichern. Diese Variablen, in denen die Objekte etwas dauerhaft speichern, nennt man Eigenschaften oder Attribute des Objekts. Daneben gibt es auch lokale Variablen, die nur innerhalb einer Methode verwendet werden und danach wieder vergessen werden, und Parameter, die an eine Methode übergebene Werte speichern.

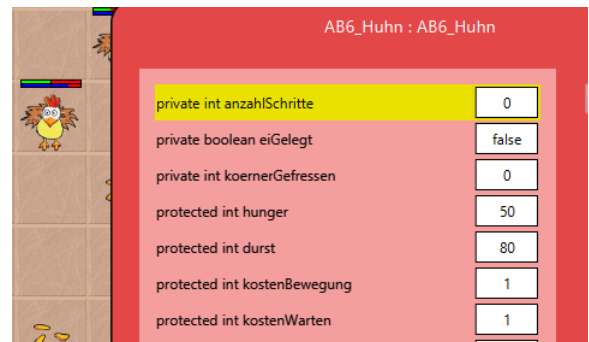
Den aktuellen Wert von Eigenschaften kann man in der Regel mit einer Methode abfragen, deren Name mit "get..." oder "gib..." beginnt, z.B.:

- `getHunger()` bzw. `getDurst()` wird dir die Restenergie als Antwort nennen.

Die aktuellen Werte aller Attribute bestimmen den **Zustand** einer Figur.

### Aufgaben:

1. Rufe bei zwei Hühner das Inspect-Fenster auf (Rechtsmausklick auf das Huhn → Inspizieren) und vergleiche den Wert der Attribute hunger und durst mit der Lebensanzeige des Huhns. Lasse das Huhn laufen und beobachte die Werte der Attribute. Rechts siehst du ein Huhn und sein INSPECT-Fenster.
2. Notiere, welche weiteren Eigenschaften des Huhns in Attributen gespeichert sind. Nenne mindestens eine Methode, mit der Du den Wert eines Attributs ändern kannst.



Du hast gelernt, dass man sich Variablen als beschriftete Kiste vorstellen kann. Vorne auf der Kiste steht der Name der Variable, in der Kiste liegt ihr momentaner Wert. In Java gibt es die Vereinbarung, Namen von Variablen mit einem Kleinbuchstaben beginnen zu lassen. 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.

Außerdem muss man in Java bei der Deklaration der Variablen (=Erschaffen der Kiste) festlegen, was man in der Kiste speichern möchte (=Typ der Variablen). Je nach Typ wird

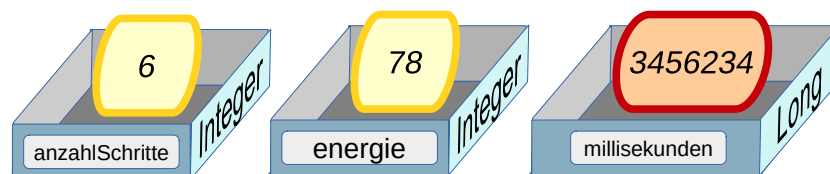


Abb. 1: int und long-Variablen als "Kisten" (Quelle: eigenes Werk)



unterschiedlich viel Speicherplatz reserviert. 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.

Art	ganze Zahl	ganze Zahl	Wahrheitswert	Kommazahl	Buchstabe	Text
Typ	int	long	boolean	double	char	String
Bsp	2837	9288376172	true/false	23.4	'a'	"Beispiel"

## Aufgaben:

### 3. 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.

### 4. Namensgebung:

Schlage sinnvolle Namen für Attribute vor, die speichern,

- wie viele Schritte ein Schaf gegangen ist.
- wie viele Drehungen es gemacht hat. (Was genau gibt die gespeicherte Zahl dann an? Wie werden Links- bzw. Rechtsdrehungen gespeichert?)
- wie viele Blumen es gefressen hat.

## Eigene Attribute

Das **AB6\_Huhn** hat eine neue Eigenschaft: Es hat das Attribut `koernerGefressen`, das zählt wie viele Körner ein Schaf im Laufe seines Lebens schon gefressen hat.

Dazu müssen drei Dinge erledigt werden. Das ist genauso wie bei Scratch:

### 1 Deklaration:

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. In Scratch hat man mit "Neue Variable" diesen Schritt ausgeführt.

```
private int anzahlKoerner;
```



### 2 Initialisierung:

Der Wert des Attributs muss initialisiert werden (die Kiste muss einen sinnvollen Anfangswert erhalten): Dies macht man im sogenannten **Konstruktor**. Er wird automatisch aufgerufen, wenn das Objekt erzeugt wird. Er ist die Methode mit dem gleichem Namen wie die Klasse (Achtung: Sie hat keinen Rückgabtyp, auch kein `void`). Dort wird der Wert von `anzahlKoerner` auf 0 gesetzt. In Scratch hat man das in der Regel direkt nach "Wenn grüne Flagge gedrückt" gemacht.

```
// setze anzahlKoerner auf 0
anzahlKoerner = 0;
```



### 3 Verändern des Attributs:

In der Methode `fresseMitZaehlen()` wird nach dem Fressen der Wert des neuen Attributs um 1 erhöht:

```
// ändere anzahlKoerner um 1
anzahlKoerner = anzahlKoerner + 1;
// neuer Wert = alter Wert + 1;
```



## Aufgaben:

5. Überprüfe im INSPECT-Fenster, ob das Huhn ordentlich seinen Fresserfolg zählt.
6. **Schrittezähler:** Implementiere eine Methode `sucheKoerner()`, die das Huhn bis zum nächsten Feld mit Körnern (`istAuf("Koerner")`) laufen lässt und dabei die Schritte zählt. Anschließend soll das Huhn die gefunden Körner fressen (natürlich mit Zählen). Deklariere dazu ein Attribut `entfernungZumNest`, initialisiere es im Konstruktor und zähle dann die Schritte. Überprüfe im INSPECT-Fenster, ob die Schritte richtig gezählt werden.
7. **Schrittanzeige:** Die Methode `gibAnzahlGefressenerKoerner()` gibt den Wert des Attributs zurück. Implementiere genauso eine Methode `gibEntfernungZumNest()`. Hinweis: Denke daran, dass diese Methode einen Wert zurückgeben soll. Was muss anstelle von `void` hier stehen?
8. **Schrittezähler 2:** Verbessere deine Methode `sucheKoerner()`, so dass sie auch funktioniert, wenn vor dem Huhn gar keine Körner liegen. Das Huhn soll dann so lange laufen, wie vor ihm frei ist.
9. **Heimwärts:** Implementiere die Methode `laufeZurueckZumNest()`, die das Huhn zurück zum eigenen Nest laufen lässt. Damit das Huhn nicht auf dem fremden Nest stehen bleibt, musst du dafür das Attribut `entfernungZumNest` nutzen. Teste auch die Methode `picken()` an allen Hühnern. Sie sollte nun mit Hilfe deiner Methoden ein Huhn alle Körner vor ihm aufpicken und danach zum Nest zurückkehren lassen. Hinweis: Wie sollte sich der Wert der Variable ändern, wenn das Huhn einen Schritt zurück Richtung eigenes Nest macht?
10. **Jedes Huhn legt ein Ei:** Steht ein Huhn auf einem Feld mit einem Nest, kann es versuchen ein Ei zu legen (`versucheEiZulegen()`). Der Erfolg ist umso größer, je besser genährt das Huhn ist. Implementiere die Methode `gackere()`, die das Huhn einen Versuch unternehmen lässt. War er erfolgreich (`istAuf("Ei")`), dann soll sich das Huhn dies dauerhaft merken. Führe dazu ein Attribut `eiGelegt` ein, das `true` ist, wenn das Huhn in seinem Leben schon einmal ein Ei gelegt hat. Andernfalls ist es `false`. Implementiere auch die dazu passende Methode `hatEiGelegt()`, die den Wert des Attributs zurück gibt. Teste deine Methoden.

## Kommentare

Kommentare sind sinnvoll, um ein Programm lesbar zu machen. Man unterscheidet dabei einzeilige Kommentare (`// ...`) und Kommentare, die über mehrere Zeilen gehen (`/* ... */`):

```
// Kommentar einzeilig

int zahl; // auch hinter Quelltext möglich

/* Kommentar mehrzeilig
 * wenn man viel zu sagen hat.
 */
```

## Aufgaben:

11. Füge in die Methode `leveltest6()` der Klasse `AB6_Bauer()` Kommentare ein, die erläutern, in welche Richtung der Bauer gerade läuft. Hinweis: Überlege dir, ob es sinnvoll ist, den Kommentar vor einem einzelnen `einsVor()` zu platzieren, oder ob eine ganze `while`-Schleife für die Bewegung in eine Richtung verantwortlich ist.



## Leveltest 6: Hühnerbauernprüfung

Der AB6\_Bauer soll zeigen, dass er erfolgreich einen Hühnerhof führen kann. Er muss dazu mindestens 30 Eier von seinen freilaufenden Bio-Hühnern einsammeln.

Um diese Aufgabe zu lösen, solltest du zunächst zwei Aufgaben lösen, die du noch in der Standardumgebung testen kannst.

### Aufgabe 1:

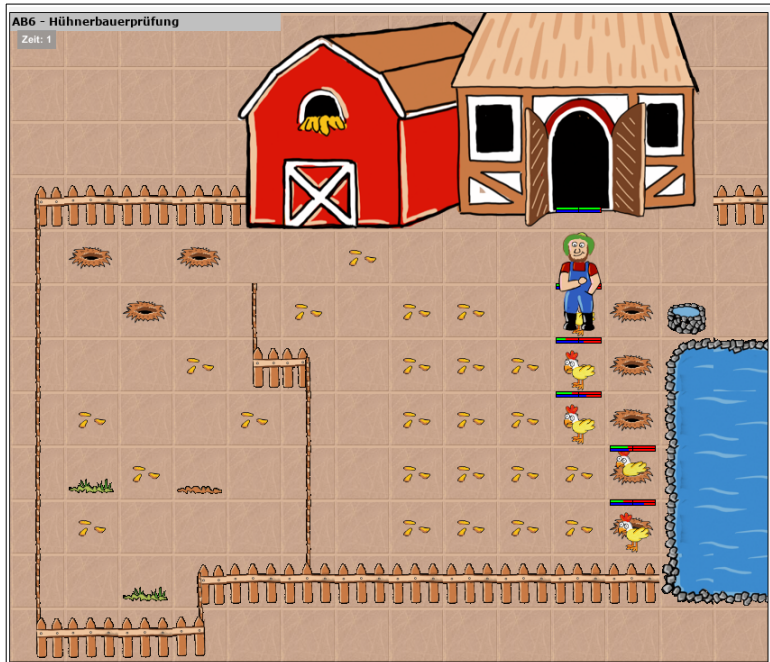
Lasse den Bauer beim Laufen nach links zählen, wie viele Körner in dieser Reihe schon liegen. Beim Rückweg werden weitere Körner auf freien Feldern abgelegt

(`ablegen("Koerner")`), bis die Mindestzahl von 4 Körner pro Reihe erreicht ist.

### Aufgabe 2:

Lasse den Bauer auf dem Rückweg zum Haus die in den Nestern liegenden Eier einsammeln (`aufnehmen();`) und zählen. Die Methode `getAnzahlEier()` muss die Zahl der gesammelten Eier zurückgeben.

Wenn du diese beiden Aufgaben gelöst hast, müsstest du die Hühnerbauernprüfung (`leveltest06`) schon bestehen. Du kannst versuchen, die dafür benötigte Zeit noch zu optimieren.





"Mein Feld ist größer als deins", "meine Tomaten sind schöner als deine!": Schon immer gab es Konkurrenz unter Nachbarn. Auch unsere Bauern sind da nicht besser und wollen ihre Feldergröße vergleichen. Außerdem steht die Kornbauernprüfung an...

## Lokale Variablen und Zählschleifen...

**ZIEL:** Lokale Variablen als Zwischenspeicher für jeweils einen Wert kennen und einsetzen können. Lokale Variable von Objektvariablen unterscheiden können. Zählschleifen einsetzen können.

## Lokale Variable als Kurzzeitgedächtnis

Der AB7\_Bauer hat zwei ähnliche Methoden: `bestimmeAnzahlTomatenAttribut()` und `bestimmeAnzahlTomatenLokaleVariable()`. Beide lassen den Bauern bis zum nächsten Hindernis laufen und dabei Tomatenpflanzen zählen.

### Aufgaben:

1. Rufe beim Bauern auf dem Tomatenfeld zunächst in jeder Spalte die erste Methode auf. Du kannst ihn mit `umkehren(boolean linksrum)` umkehren und in die nächste Spalte laufen lassen.
2. Führe einen Reset durch. Wiederhole die Aufgabe 1 mit der zweiten Methode. Was macht die zweite Methode anders?

Manchmal müssen sich Objekte bestimmte Werte nur vorübergehend merken, um eine Aufgabe zu erfüllen, z.B. wenn die Werte nur innerhalb einer Methode benötigt werden, nach deren Abschluss nicht mehr. Es war beispielsweise im Leveltest von AB6 nicht notwendig, dass sich der Bauer die Zahl der Körner in einer Reihe dauerhaft merkt. Es musste sich das nur merken, bis er die Anzahl auf 4 Körner aufgefüllt hatte.

Bisher hatten wir Attribute der Objekte als Gedächtnis. Diese sind aber nur für Werte gedacht, die die Figuren sich dauerhaft merken sollen. Sie stellen das "Langzeitgedächtnis" dar. Du sollst dir beispielsweise auch dauerhaft merken, dass es keine "if-Schleifen" gibt. Die Koordinaten der Position des Bauernhofs kannst du aber nach Beendigung der Aufgabe wieder vergessen.

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.

### Aufgaben:

3. Vergleiche die Quelltexte der beiden Methoden, die die Tomatenpflanzen zählen. Welche Variable wird jeweils zum Zählen benutzt? Wo wird diese Variable deklariert (erzeugt), initialisiert (auf ihren Startwert gesetzt) und benutzt?

4. **Debugging:** Lasse dir mit INSPECT die Attribute anzeigen. Warum sieht man hier nur den `zaehler1`? Die lokalen Variablen sieht man nur im Debugger. Setze dazu einen Breakpoint, indem du auf die Zeilennummer klickst, an der das Programm unterbrochen werden soll (hier Zeile 32). Wenn du dann die Methode aufrufst, hält das Programm am Stoppschild an und kann mit "Schritt über" Schritt für Schritt ausgeführt werden. Du siehst, dass die lokale Variable erst entsteht, wenn sie das erste Mal benutzt

The screenshot shows the AB7\_Bauer class with two methods. The first method, `bestimmeAnzahlTomatenAttribut`, uses a static attribute `zaehler1`. The second method, `bestimmeAnzahlTomatenLokaleVariable`, uses a local variable `anzahl2`. The variable inspector on the right shows the state of the program. Under 'Attribute', it lists `protected ArrayList<Greenfoot>` and `private ArrayList<String> orig`. Under 'Lokal Variablen', it lists `private int anzahl1 = 0` and `int anzahl2 = 0`. Arrows point from the text 'Attribute' and 'Lokal Variablen' to their respective sections in the inspector.



wird.

## Vergleich Attribut - Lokale Variable:

	Attribut	Lokale Variable
Deklaration	<b>Vor den Methoden:</b> <pre>public class AB7 Bauer {     private int zaehler1;     ...     //Methoden     ... }</pre> → mit private!	<b>Innerhalb der Methode:</b> <pre>public class AB7_Bauer {     ...     public int zaehle() {         int zaehler2;         ...     } }</pre> → ohne private!
Initialisierung	<b>Im Konstruktor der Klasse:</b> <pre>public AB7 Bauer {     zaehler1 = 0;     ... }</pre>	<b>Direkt nach der Deklaration:</b> <pre>public int zaehle {     int zaehler2;     zaehler2 = 0;     ... }</pre>
Verwendung	<pre>public int zaehle {     zaehler1++;     ... }</pre>	<pre>public int zaehle {     zaehler2++;     ... }</pre>

Hinweis: Um Attribute von lokalen Variablen unterscheiden zu können, kann man bei Attributen (außer bei der Deklaration) immer ein `this` voranstellen: z.B. `this.zaehler1 = 0;`

### Aufgaben:

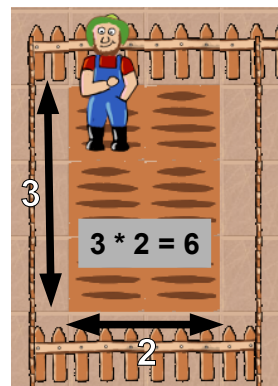
5. **Feldlänge 1:** Implementiere die Methode `int bestimmeLaenge()`, die dem Bauer ermöglicht, die Länge seines Feldes zu bestimmen. Dazu soll er einfach geradeaus bis zum nächsten Hindernis laufen und die Anzahl der Felder in einer lokalen Variable zählen und als Ergebnis zurückgeben.  
Hinweis: beachte, dass es ein Feld mehr ist als man Schritte machen muss.

6. **Feldlänge 2:** Implementiere die Methode `int bestimmeLaenge2()`: Deklariere drei lokale Variablen `y1`, `y2`, `y_diff`. Weise `y1` den Wert des Aufrufs von `getY()` zu. Lasse den Bauern bis zum nächsten Zaun laufen. Weise nun `y2` den Wert des Aufrufs von `getY()` zu. Berechne die Differenz von `y2` und `y1` und speichere sie in `y_diff`. Gib mit Hilfe von `y_diff` die Länge des Feldes zurück.  
Was passiert, wenn der Bauer nach rechts schaut und du die beiden Methoden aufrufst?

7. **Fläche:** Implementiere eine Methode, die die Fläche eines rechteckigen Feldes (Breite \* Höhe) bestimmt und zurück gibt. Du kannst davon ausgehen, dass der Bauer in einer Ecke des Feldes steht und in die passende Richtung schaut. Nutze eine der Methoden aus Aufgabe 5+6 (ggf. auch mehrfach), gerne darfst Du auch noch weitere Methoden formulieren und hier aufrufen, nutze sinnvolle Namen.

8. **Lästiges Unkraut:** Implementiere eine Methode, die den Blumenbauer die Anzahl der Felder mit Gras in seinem einzeiligen Blumenbeet zählen lässt.  
Hinweis: Beachte, dass auch auf dem ersten und letzten Feld Gras wachsen kann.

9. **Blumenbeet:** Implementiere die Methode `int anzahlSchritteBisBlume3()`, die den Blumenbauer die Anzahl der Schritte bis zur dritten Blume zählen lässt und diese Anzahl zurückgibt.  
Hinweis: Du brauchst zwei lokale Variablen. Eine für die Anzahl der Schritte und eine für die Anzahl der schon gefundenen Blumen.





## Zählschleifen

Bisher hast du `while`-Schleifen kennengelernt. Mit ihnen kann man alles programmieren, aber manchmal geht es kompakter. Sehr häufig muss man wissen, in welchem Schleifendurchgang man ist. Macht man es zum ersten, zweiten oder dritten Mal...

Dazu braucht man eine lokale Variable, die zählt, in welchem Durchgang man ist. Dabei beginnen die Informatiker in der Regel mit 0 zu zählen.

While-Schleife	For-Schleife
<pre> 1 public void gehe4 { 2     int zaehler; 3     zaehler = 0; 4     while(zaehler &lt; 4) { 5         einsVor(); 6         zaehler++; 7     } 8 }</pre>	<pre> public void gehe4 {     for(int zaehler=0; zaehler&lt;4; zaehler++) {         einsVor();     } }</pre>

10. Vergleiche die beiden Varianten der Zählschleife. Markiere jeweils, an welcher Stelle die Zählvariable:

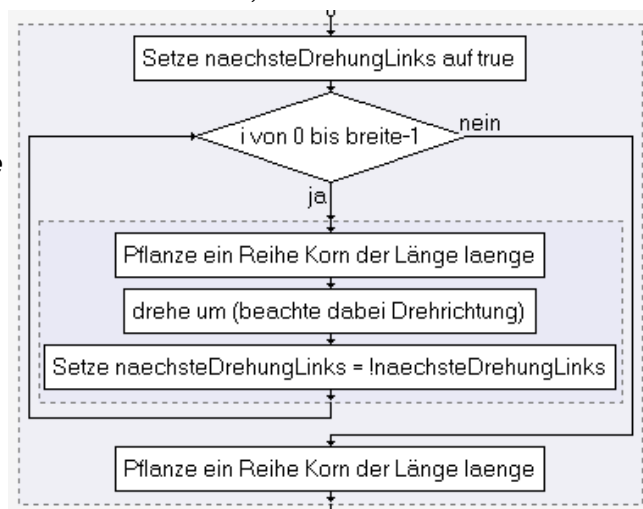
- a) deklariert                      b) initialisiert                      c) erhöht  
wird und  
d) die Schleifenbedingung steht.

11. **Pflanze Korn:** Die Methode `pflanzeKorn(int anzahl)` lässt den Bauern eine bestimmte Anzahl von Getreidepflanzen säen. Implementiere eine Methode `pflanzeKorn2(int anzahl)`, die diese Aufgabe mit einer `for`-Schleife statt einer `while`-Schleife löst.

12. Verbessere beide Methoden so, dass sie auch funktionieren, wenn bis direkt an den Zaun gepflanzt werden soll, d.h. der Bauer steht am Ende auf dem letzten bepflanzten Kornfeld.

13. **Drehwurm:** Implementiere eine Methode `drehwurm(int anzahl)`, die den AB7\_Bauern zunächst links herum und dann rechts herum drehen lässt. Der Parameter `anzahl` gibt dabei an, wie viele komplette 360°-Drehungen er in beide Richtungen machen soll. Verwende eine `for`-Schleife.

14. **Kornfeld:** Implementiere eine Methode `kornfeld(int laenge, int breite)`, das den Bauer eine Fläche von `laenge x breite` mit Korn bepflanzen lässt. Setze dazu das abgebildete Flussdiagramm um. Verwende dabei die Methode `pflanzeKorn`. Mache dir klar, was die Zeile `naechsteDrehungLinks = !naechsteDrehungLinks` macht.

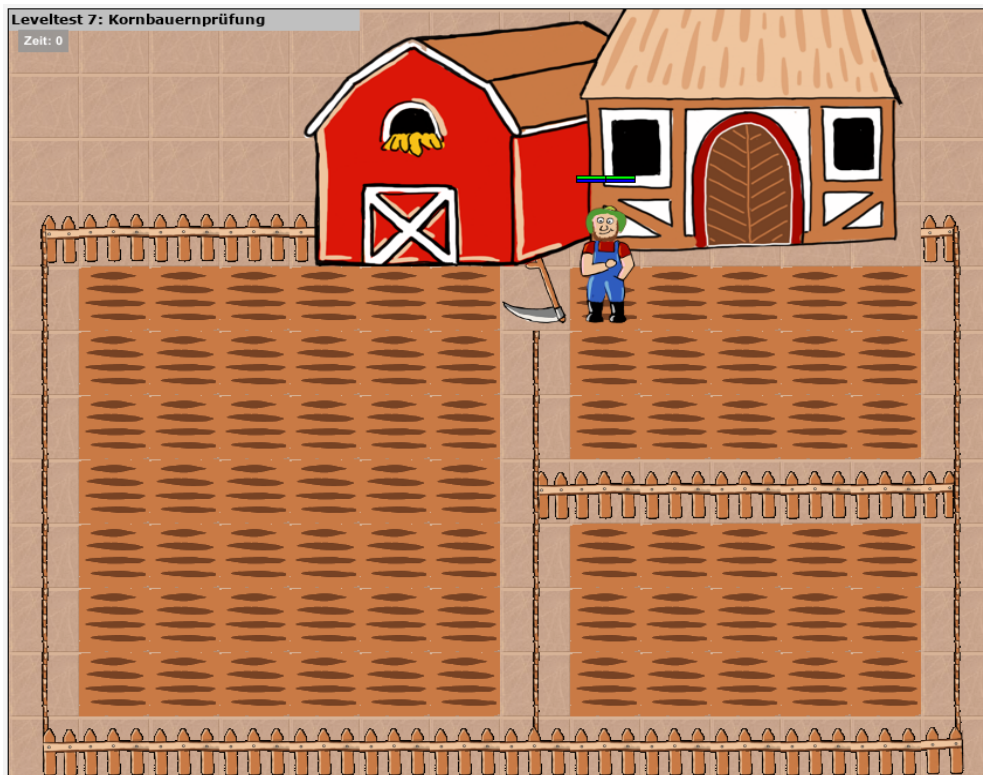




## Leveltest 7: Kornbauernprüfung

Du musst dich als Kornbauer bewähren. Deine Aufgabe ist es 300 Körner (`getAnzahl("Koerner")`) in deinem Inventar zu haben. Du startest mit 70 Körnern, die du auf deinem Feld säen kannst. Sobald die Kornpflanzen reif sind, kannst du sie mit einer Sense wieder ernten (`ernte("Korn")`) und erhältst pro Kornpflanze fünf Körner. Zwischendrin musst du natürlich Vesperpause machen gehen, um nicht zu verhungern und zu verdursten.

Dein Bauer startet in der linken oberen Ecke des Feldes. Die Sense liegt auf dem Feld links daneben. Die Tür zum Hauptgebäude ist immer bei den Koordinaten (10 | 3). Du kannst davon ausgehen, dass das Korn vollständig gereift ist, wenn du das ganze Feld eingesät hast und wieder zum Ausgangspunkt zurückgekehrt bist.



Tipps: Bestimme zunächst die Länge und Breite deines Feldes. Laufe dann zurück zum Ausgangspunkt. Laufe dabei an deinem Haus vorbei und gehe etwas essen (`vesperpause()` auf Feld 10/4 aufrufen).

Verwende dann die Methode `kornfeld`, um Korn zu säen. Laufe wieder zum Startpunkt zurück. Essen nicht vergessen.

Modifiziere die Methode `kornfeld(int laenge, int breite)` so, dass sie entweder Korn pflanzt oder Korn erntet.



Landwirtschaft ist heutzutage zu einer Wissenschaft geworden: Agrarökonomie. Ohne genaue Statistiken über die Erträge wäre die Forschung in diesem Bereich undenkbar. Unser Bauer sollte also für jedes einzelne Huhn zählen können, wie viele Eier es legt. Aber bei einem großen Hühnerhof mit 100 Hühnern bräuchte man ja dann 100 Variablen. Und jetzt kommen noch 10 neue Hühner dazu. Wie soll unser Programm das schaffen? Müssen wir alles umprogrammieren?

## Arrays = Liste vieler gleichartiger Variablen

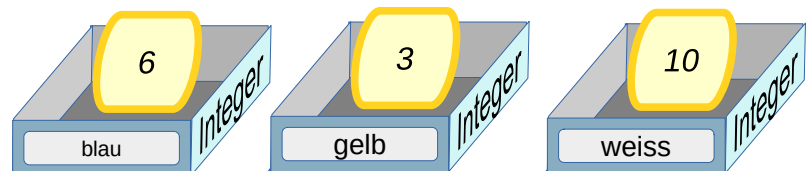
**ZIEL:** Arrays deklarieren und initialisieren und zum Speichern vieler gleichartiger Werte nutzen können. Algorithmen auf Arrays implementieren können.

## Arrays deklarieren, initialisieren und benutzen

Der Bauer links unten verdient sein Geld mit Blumenzucht. Je nach Farbe erhält er einen unterschiedlichen Preis für seine Blumen. Daher möchte er eine Statistik erstellen, wie häufig welche Farbe vorkommt.

Die erste Idee ist drei integer-Variablen mit den Namen `rot`, `blau` und `weiss` zu erstellen:

```
int blau = 6;
int gelb = 3;
int weiss = 10;
```



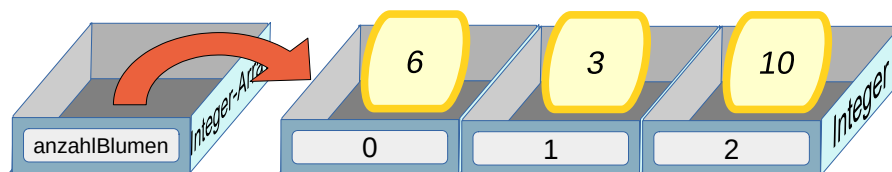
3 Variablen für die Blumenanzahl

Bei drei verschiedenen Farben geht das noch. Aber das Programm soll leicht auf eine große Anzahl verschiedener Blumen erweitert werden können.

Daher erstellt man ein Array:

```
int[] anzahlBlumen;
anzahlBlumen = new int[3];
anzahlBlumen[0] = 6;
anzahlBlumen[1] = 3;
anzahlBlumen[2] = 10;
```

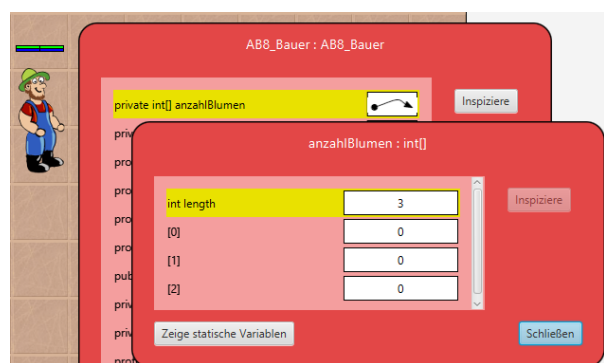
Die einzelnen Speicherplätze spricht man mit ihrer Nummer an. Dabei beginnt die Nummerierung mit 0. Bevor man das Array benutzen kann, muss man mit dem Befehl `new` genügend Speicherplatz reservieren (hier drei `int`-Variablen).



Array mit 3 Speicherplätzen

### Aufgaben:

- Viel Speicherplatz:** Beim `AB8_Bauern` gibt es schon ein Attribut `anzahlBlumen`. Initialisiere dieses Attribut wie oben gezeigt im Konstruktor des `AB8_Bauern`. Setze die Startwerte für alle Blumensorten auf 0. Inspiziere den Bauern. Beim Attribut `anzahlBlumen` siehst du nur einen Pfeil (= Verweis). Diesen kannst du doppelklicken und dadurch inspizieren, ggf. musst Du die roten Fenster an einem Rand verbreitern. Dann solltest du sehen, dass das Array die Länge 3 hat und in jedem Eintrag eine 0 steht.





2. **Blumenzähler:** Die Methode `pflueckeBlumen()` lässt den Blumenzüchter auf seinem Feld herumlaufen und alle Blumen pflücken, bis er keine Energie mehr hat. Ergänze diese Methode so, dass er dabei die Anzahl der Blumen zählt. An Position 0 soll die Anzahl der blauen Blumen gespeichert werden, an Position 1 die Anzahl der gelben und an Position 2 die Anzahl der weißen Blumen.

Hinweis 1: `anzahlBlumen[1]++;` // erhöht die Anzahl an der Position 1

Hinweis 2: Mit `pruefe("Blume")` kann man feststellen, welche Farbe die Blume hat (blau = 40, gelb = 30, weiß = 40).

Teste die Methode. Kontrolliere im Inspect-Fenster, ob die Zählung erfolgreich war.

3. **Gesamtertrag:** Implementiere eine Methode `int gibGesamtzahlBlumen()`, die die Gesamtzahl aller gepflückten Blumen zurückgibt. Addiere dazu die Werte der einzelnen Positionen und speichere sie in einer lokalen Variable `ergebnis`. Gib dieses Ergebnis zurück.

4. **Einzelaufstellung:** Untersuche die Methode `gibAnzahlBlumenart(int art)`. Beschreibe, welche Bedeutung das Ergebnis von `gibAnzahlBlumenart(2)` hat. Wie wird hier ausgewählt, welche Position des Arrays ausgelesen wird?

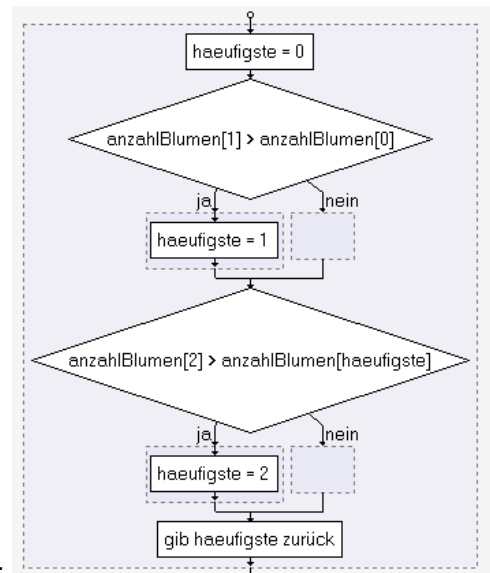
5. **Prozentrechnung:** Implementiere `double gibProzentsatz(int art)`. Diese Methode soll berechnen, wie groß der Prozentsatz der Blumen einer bestimmten Farbe war.

Hinweis 1:  $p\% = \frac{W}{G} * 100\%$

Hinweis 2: Wenn man in Java zwei ganze Zahlen teilt, dann bekommt man wieder eine ganze Zahl. Java schneidet die Nachkommastellen einfach ab. Daher muss man die Anzahl der Blumen in eine Dezimalzahl umwandeln:

```
int i = 3;
double e1 = i / 5;           // ergibt 0.0
double e2 = (double) i / 5;  // ergibt 0.6
```

6. **Wachstumswunder:** Implementiere eine Methode `int gibNrHaeufigsteBlume()`, die bestimmt, welche Blumensorte am häufigsten auf dem Feld gewachsen ist. Als Ergebnis soll eine Zahl zwischen 0 und 2 zurückgegeben werden. Setze dazu nebenstehendes Flussdiagramm in Java-Quelltext um.



7. **Wachstumswunder 2:** Implementiere die Methode `int gibAnzahlHaeufigsteBlume()`, die dieses Mal nicht die Nummer, sondern die Anzahl der häufigsten Blume bestimmt.

Hinweis: Nutze dazu die Methode, die du schon bei Aufgabe 6 implementiert hast.



## Arrays mit Zählschleifen durchlaufen

Aufgabe 4 hat dir gezeigt, dass man die Position in einem Array auch durch eine Variable angeben kann. Dies ist sehr nützlich, wenn man mit allen Elementen eines Arrays etwas machen möchte, z.B. alle auf 0 setzen:

Variante 1:

```
anzahlBlumen[0] = 0;
anzahlBlumen[1] = 0;
anzahlBlumen[2] = 0;
```

Variante 2:

```
for(int i=0; i<3; i++) {
    anzahlBlumen[i] = 0;
}
```

*i* nimmt also die Wert 0 bis 2 an. Der Reihe nach werden die einzelnen Array-Elemente auf 0 gesetzt. Auch die Summenbildung kann man mit einer Schleife realisieren, wenn man die Summenbildung zerlegt und immer nur ein weiteres Array-Element zur Summe addiert:

```
summe = ((0 + anzahlBlumen[0]) + anzahlBlumen[1]) + anzahlBlumen[2];
```

Variante 1:

```
summe = 0;
summe = summe + anzahl[0];
summe = summe + anzahl[1];
summe = summe + anzahl[2];
```

Variante 2:

```
summe = 0;
for(int i=0; i<3; i++) {
    summe = summe + anzahlBlumen[i];
}
```

Ersetzt man die Zahl 3 noch durch `anzahlBlumen.length` kann das Array beliebig groß sein und der Algorithmus funktioniert trotzdem noch.

Der Hühnerbauer möchte untersuchen, welche seiner Nester gut platziert und welche nicht. Dafür zählt er die Eier, die er aus den Nestern nimmt.

### Aufgaben:

8. **Eierstatistik:** Dekлариere ein weiteres Attribut `anzahlEier`. Initialisiere es im Konstruktor des Bauern, so dass es 26 Plätze für `int`-Zahlen bietet. Setze alle Werte mit Hilfe einer Zählschleife auf 0. Kontrolliere deinen Erfolg im Inspect-Fenster.
9. **Eiersammelei 1:** Implementiere die Methode `laufeSchritte(int anzahl)` so, dass der Bauer die gegebene Anzahl von Schritten nach vorne macht. Er muss nicht kontrollieren, dass dort frei ist.  
Teste deine Methode, indem du die Methode `sammleEier()` aufrufst. Wenn du alles richtig programmiert hast, läuft der Bauer zwei Runden außen herum über alle Nester.
10. **Eiersammelei 2:** Ändere die Methode `laufeSchritte` so ab, dass der Bauer die Eier in den Nestern einsammelt. Er soll dabei in dem Attribut `anzahlEier` zählen, wie viele Eier er aus welchem Nest geholt hat. Dazu muss er sich in einem weiteren Attribut `nestNr` merken, beim wievielten Nest er ist. Er testet also nach jedem Schritt, ob er auf einem Nest steht und erhöht die Nummer ggf. um 1. Falls ein Ei im Nest liegt, erhöht er außerdem die entsprechende Speicherstelle des Arrays um 1.  
Hinweis: Wenn er eine Runde gelaufen ist, muss `nestNr` wieder zurückgesetzt werden. Teste deine Methode. Erhöhe die Anzahl der Runden, wenn der Test erfolgreich war.
11. **Einzelaufstellung:** Implementiere die Methode `gibAnzahlEierInNest(int nr)`, die zurückgibt, wieviel Eier im angegebenen Nest eingesammelt wurden.
12. **Gesamtertrag:** Implementiere eine Methode `int gibGesamtzahlEier()`, die die Gesamtzahl aller gesammelten Eier zurückgibt. Verwende dazu eine `for`-Schleife.
13. **Durchschnitt:** Implementiere eine Methode `double gibDurchschnitt()`, die die durchschnittliche Anzahl Eier pro Nest zurückgibt. Denke daran, dass du bei der Division eine Umwandlung der Anzahl in Dezimalzahl benötigst. Die Gesamtanzahl der Nester erhältst du mit `anzahlEier.length`.



14. **Nutzlos:** Implementiere eine Methode `int gibAnzahlLeererNester()`, die zurückgibt, in wievielen der Nester kein einziges Ei gefunden wurde.
15. **Lieblingsnest:** Implementiere eine Methode `int gibNrBestesNest()`, die bestimmt, welches Nest den höchsten Ertrag gebracht hat. Als Ergebnis soll die Nummer des Nestes zurückgegeben werden.  
*Hinweis: Die Methode geht ähnlich wie bei den Blumen. Du musst dir überlegen, was man bei den Blumen ändern müsste, wenn es 4 Sorten gewesen wären. Wie kann man das mit einer for-Schleife vereinfachen? An welcher Stelle muss man den Zähler i einsetzen?*
16. **Lieblingsnest 2:** Implementiere die Methode `int gibAnzahlEierBestesNest()`, die dieses Mal nicht die Nummer, sondern die Anzahl der Eier des besten Nestes bestimmt.  
*Hinweis: Nutze dazu die Methode, die du schon bei Aufgabe 15 implementiert hast.*

## Textvariablen (Typ String)

Deklaration: `String text1;`

Initialisierung: `text1 = "";`

Veränderung: `text1 = "Hallo";`  
`text1 = text1 + "Leute"; // Verlängern des Textes`  
`text1 = text1 + i; // hängt die Zahl i an den Text an`

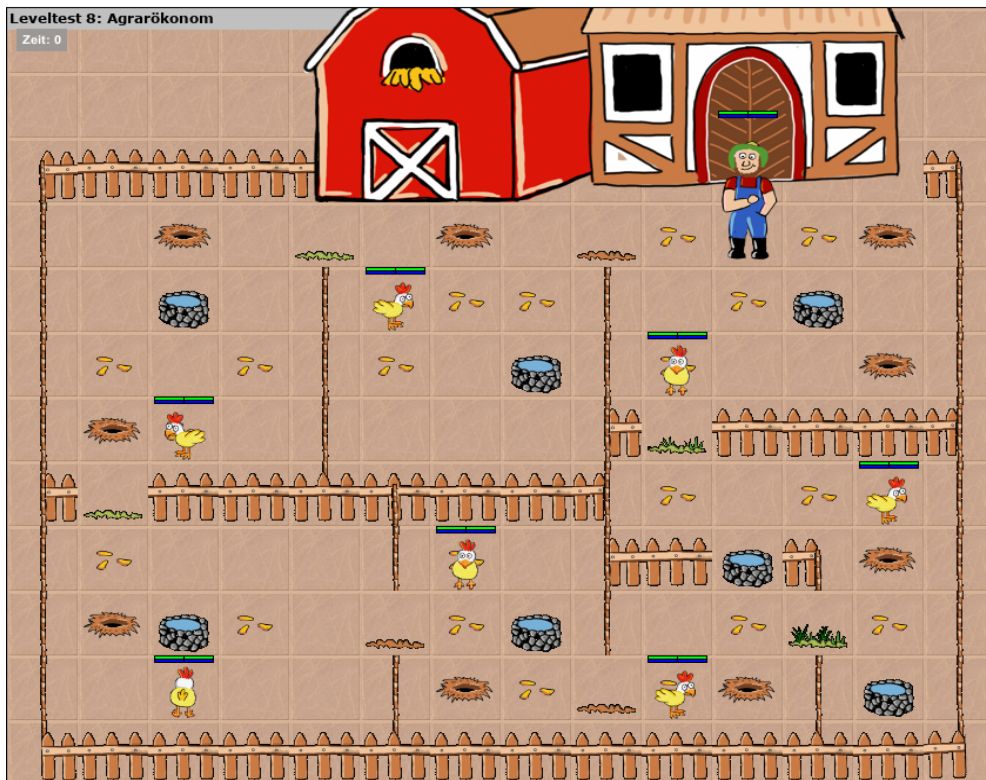
17. **Infotafel:** Implementiere eine Methode die einen Ausgabetext generiert, der für jedes Nest die Anzahl der Eier enthält. Dekлариere dazu eine lokale Variable `text`. Fange mit einem leeren Text an und hänge mit Hilfe einer for-Schleife jede Zahl des Arrays an den Text an. Trenne die Zahlen, indem du dazwischen jeweils ein ", " anhängst.  
 Ausgabe ist dann z.B.: 23, 34, 1, 0, usw.



## Leveltest 8: Agrarökonom

Du hast festgestellt, dass manche Nester besser von den Hühnern angenommen werden als andere. Diesen Aspekt willst du nun weiter untersuchen, um die optimale Form für das Gehege der Hühner zu finden. Daher bekommt jedes Huhn einen eigenen Bereich, der jeweils unterschiedlich gestaltet ist. Sie variieren in der Größe, Form und Anzahl der Nester.

Dein Bauer soll nun einige Runden durch die Gehege laufen und die Eier einsammeln. Im Anschluss soll er eine nach Anzahl der gesammelten Eier sortierte Liste erstellen.

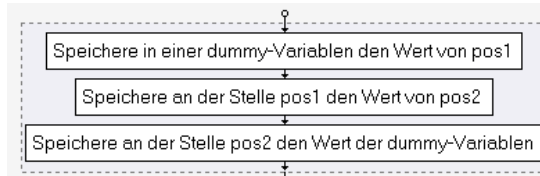


Verändere die Methode `leveltest8()` zunächst so, dass die Anzahl der Eier jedes Huhns gezählt werden. Der Bauer kann an den Grasbüscheln (`istAuf("Gras")`) erkennen, dass er das nächste Gehege betritt.

Deklariere ein Array von Strings und speichere in diesem Array einen Namen für jedes Huhn. Hinweis: Man kann Arrays auch direkt initialisieren:

```
String[] schuelernamen = {"Fred", "Anita", "Martha"};
```

Implementiere eine Methode `vertausche(int pos1, int pos2)`, die die Werte des Anzahl- und des Namensarrays an den beiden übergebenen Positionen vertauscht:



Nutze diese Methode, um das Array der Eieranzahlen und passend dazu das Array der Hühnernamen zu sortieren. Erstelle danach einen aufsteigend sortierten Ausgabetext, der die Hühnernamen und die Anzahl der von diesem Huhn gelegten Eier erstellt (z.B. "Lucmilla: 4, Gacki: 7 usw."). Dieser Text wird vom Leveltest zurückgegeben.