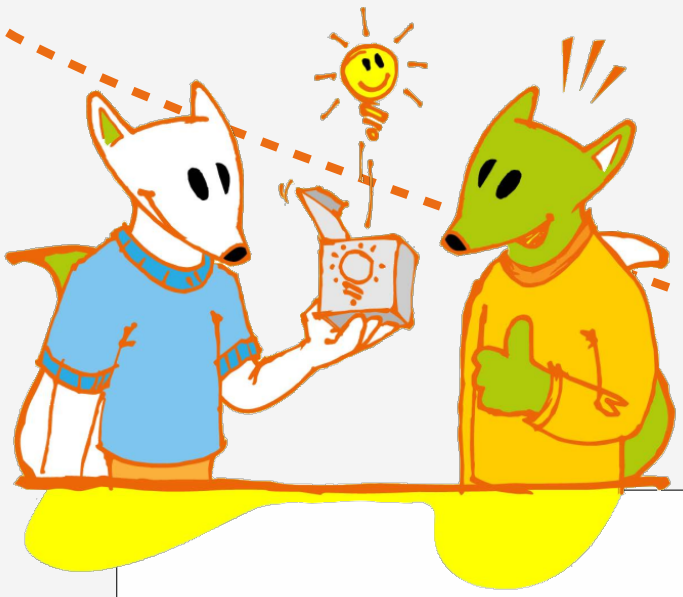
BUNDES WEIT INFORMATIK NACHWUCHS FÖRDERN

Bundesweite Informatikwettbewerbe



Arbeitsheft Algorithmen

Ein Entdeckerpfad durch die Welt der Algorithmen

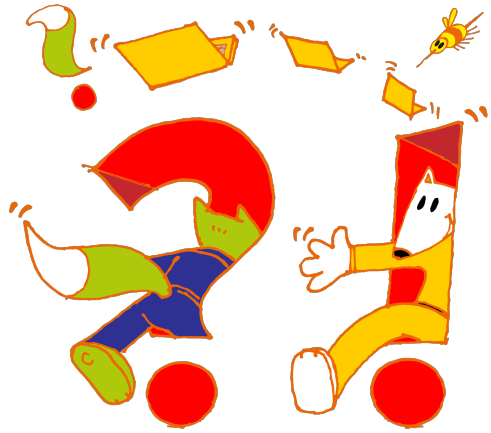
SEK I · Informatik

Dieses Heft gehört:

Konzeption:

Lenka Senajová
✉ lenka.senajova@schule.bayern.de

Hannah Rauterberg
BWINF
✉ jugendwettbewerb@bwinf.de



Herausgegeben von den Bundesweiten Informatikwettbewerben (BWINF)

Dieses Arbeitsheft verwendet die Onlineplattform jwinf.de des Jugendwettbewerb Informatik. Der Jugendwettbewerb Informatik ist einer der Bundesweiten Informatikwettbewerbe.

BWINF ist eine Initiative der Gesellschaft für Informatik (GI), des Fraunhofer-Verbunds IUK-Technologie und des Max-Planck-Instituts für Informatik.

BWINF wird vom Bundesministerium für Bildung, Familie, Senioren, Frauen und Jugend gefördert. Die Bundesweiten Informatikwettbewerbe gehören zu den von den Kultusministerien empfohlenen Schülerwettbewerben und stehen unter der Schirmherrschaft des Bundespräsidenten.

Inhaltsverzeichnis

Willkommen in der Welt des Programmierens!

Hast du dich jemals gefragt, wie eine App auf deinem Smartphone funktioniert, wie ein Computerspiel auf deine Eingaben reagiert oder wie eine Webseite genau das anzeigt, was du suchst? Hinter all diesen digitalen Anwendungen steckt eine gemeinsame Fähigkeit: das Programmieren. Und dieses Arbeitsheft ist deine Eintrittskarte, um die Geheimnisse dieser spannenden Fähigkeit zu lüften.

Keine Sorge, du musst dafür kein Mathe-Genie sein! Programmieren ist wie eine neue Sprache lernen, nur dass dein Gesprächspartner ein Computer ist. Und das Beste: Du lernst hier, wie du ihm ganz klare Anweisungen gibst, damit er genau das tut, was du willst.

Wie funktioniert dieses Arbeitsheft?

Dieses Heft ist dein persönlicher Reiseführer. Es erklärt dir die grundlegenden Konzepte und Ideen, die hinter jedem Programm stecken. Du lernst die wichtigsten Bausteine kennen, aus denen jede Software der Welt gebaut ist.

Aber graue Theorie ist langweilig, oder? Deshalb passiert die eigentliche Magie auf der Online-Plattform `jwinf.de`. Dort kannst du sofort loslegen und wirst vom Aufgabenlöser zum Code-Tüftler. Zu jedem Kapitel findest du auf `jwinf.de` dafür Aufgaben. Manchmal klappt es nicht sofort – aber genau das ist das Spannende am Programmieren: das geniale Gefühl, wenn du eine knifflige Nuss knackst!

Hinweis

Damit du auf der Plattform `jwinf.de` alle Aufgaben lösen und deinen Fortschritt speichern kannst, erhältst du von deiner Lehrkraft einen Code. Dies ist entweder direkt dein persönlicher **Logincode** (beginnt mit `u`) oder ein **Gruppencode** (beginnt mit `g`). Mit dem Gruppencode kannst du dir auf der Webseite selbst deinen Logincode erstellen. Trage dir deinen **Logincode** sorgfältig ein. Dieser Code ist deine Eintrittskarte zur Welt des Programmierens!

Mein Logincode:

_____

Warum ist dieser Code so wichtig?

- Dein Fortschritt wird gespeichert: Jedes Mal, wenn du dich mit deinem Code anmeldest, weiß die Plattform genau, wer du bist. Sie speichert, welche Aufgaben du bereits gelöst hast. Du siehst also immer, welche Aufgabe als Nächstes dran ist und kannst dort weitermachen, wo du aufgehört hast.
- Ein Code für alles: Mit diesem einen Code kannst du nicht nur die Aufgaben aus diesem Arbeitsheft lösen, sondern auch alle weiteren Trainings- und Wettbewerbsaufgaben.

Bewahre dieses Arbeitsheft gut auf, damit du deinen Code immer wiederfindest!

1 Hallo Welt! Dein Start in die Programmierung

Hallo, zukünftige Code-Entdecker!

Willkommen zu deiner ersten Mission in der Welt der Programmierung. Bevor wir uns an komplexere Aufgaben wagen, beginnen wir mit einer weltweiten Tradition, die alle Programmiererinnen und Programmierer vereint: dem `"Hallo Welt"`-Programm.

Es ist wie das erste Händeschütteln mit dem Computer. Mit diesem einfachen Programm zeigst du, dass du mit ihm kommunizieren kannst und die Leitung für alle zukünftigen Abenteuer steht. Es ist der erste kleine Schritt in die Welt des Programmierens.

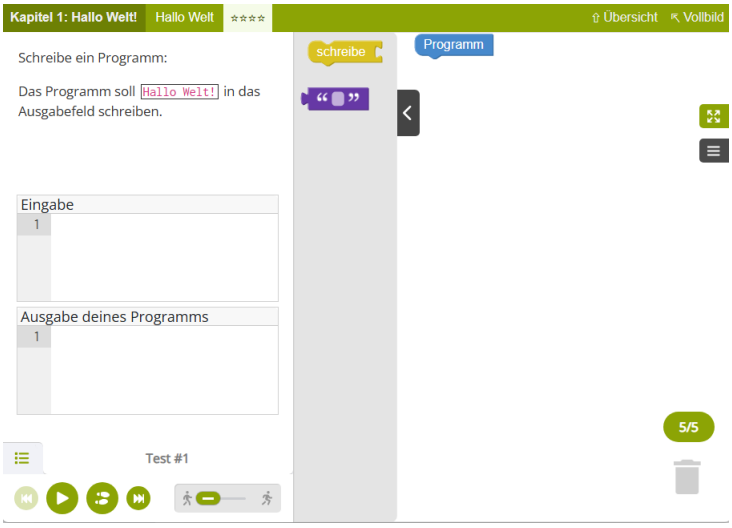
Wir verwenden dafür die visuelle Programmiersprache Blockly auf der Plattform des Jugendwettbewerbs Informatik (`jwinf.de`). Hierbei musst du keine komplizierten Codes tippen, sondern schiebst bunte Bausteine wie digitale LEGO-Steine zusammen. Deine Mission ist es nun, der Welt „“Hallo Welt““ zu sagen und zu zeigen, dass eine neue Programmier-Heldin oder ein neuer Programmier-Held die Bühne betreten hat.

Merke

Das `"Hallo Welt"`-Programm ist ein einfaches Testprogramm, das traditionell als Erstes geschrieben wird, um die grundlegende Funktionsfähigkeit einer Programmierumgebung zu bestätigen.

Bist du bereit? Dann legen wir los! Deine erste Mission: Begrüße die Welt!

Hallo Welt



Dein erster Code: Schritt für Schritt zum Erfolg
Lass uns diese erste Mission gemeinsam meistern.

1. Gehe auf `jwinf.de` und logge dich mit deinem Logincode ein.
2. Klicke dann auf die Kachel „Arbeitsheft Algorithmen“

3. Klicke auf „Kapitel 1 - Hallo Welt!“ und dann auf die Aufgabe „Hallo Welt“
4. Schau dir die Baustein-Palette in der Mitte der Seite an. Für diese Aufgabe brauchst du die Bausteine 

und  .
5. Ziehe den Baustein 

mit der Maus nach rechts auf die weiße Arbeitsfläche. Verbinde ihn mit dem Baustein 

.
6. Nimm nun den leeren `Text`-Baustein  und verbinde ihn mit der dafür vorgesehenen Lücke im `schreibe`-Baustein.
7. Schreibe in den Baustein  `Hallo Welt!`
8. Teste dein Programm: Klicke jetzt unter dem Ausgabefeld auf den Play-Knopf  .

Herzlichen Glückwunsch! Du hast dein erstes Programm erstellt!

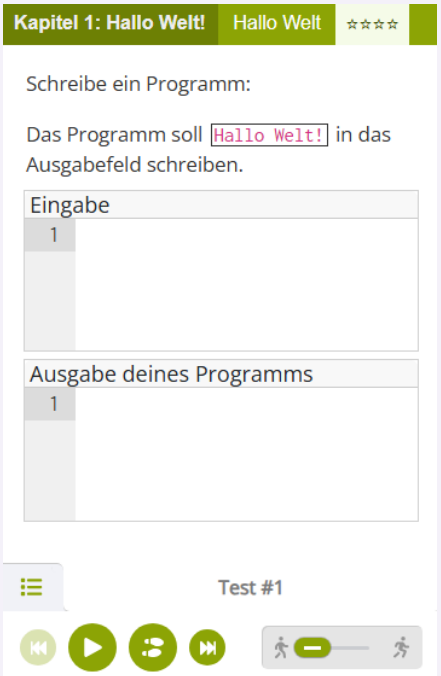
Info

So löst du eine Aufgabe auf jwinf.de

Du hast bereits dein erstes Programm erstellt: Damit du für die kommenden, spannenderen Aufgaben bestens gerüstet bist, schauen wir uns jetzt alle Werkzeuge und Anzeigen auf `jwinf.de` im Detail an. Jede Aufgabe ist dabei in zwei Hauptbereiche aufgeteilt.

Der Aufgabenbereich

Auf der linken Seite findest du alles, was du über die Aufgabe wissen musst.



Die Aufgabenstellung: Lies sie dir immer zuerst vollständig durch, um genau zu verstehen, was von dir gefordert wird.

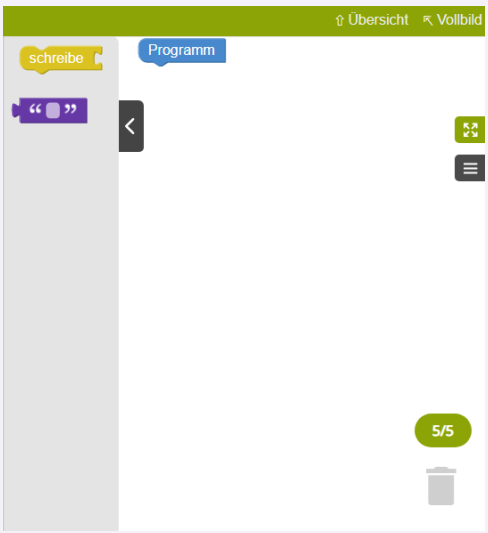
Der Ausgabebereich: Unter dem Text siehst du je nach Aufgabe ein Spielfeld, eine Zeichenfläche oder ein Fenster für die Ein- und Ausgabe von Text. Hier siehst du, was dein Programm bewirkt.

Schwierigkeitsstufen: Für die meisten Aufgaben auf `jwinf.de` gibt es die Versionen ★★ bis ★★★★. Die Schwierigkeiten bauen in der Regel aufeinander auf und wir empfehlen bei Version ★★ anzufangen und dich dann Schritt für Schritt zu steigern.

Testfälle: Für viele Aufgaben gibt es mehr als einen Testfall. Dein Programm muss immer alle Testfälle gleichzeitig lösen!

Der Programmierbereich

Auf der rechten Seite baust du deine Lösung.



Die Baustein-Palette: In der grauen Spalte findest du alle Bausteine, die dir für die Aufgabe zur Verfügung stehen.

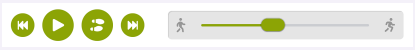
Deine Arbeitsfläche: Ziehe die benötigten Bausteine mit der Maus auf die weiße Fläche und verbinde sie mit dem `Programm`-Baustein.

Aufräumen: Wenn du einen Baustein nicht mehr benötigst, ziehe ihn einfach in den Papierkorb unten rechts. Du kannst dein Programm jederzeit umbauen und die Reihenfolge der Bausteine verändern.

Das Baustein-Limit: Bei vielen Aufgaben darfst du nur eine bestimmte Anzahl von Bausteinen verwenden. Die grüne Anzeige über dem Papierkorb hilft dir dabei. Die Anzeige `5/5` bedeutet: Du hast noch 5 von 5 erlaubten Bausteinen übrig. Wenn du einen Baustein nutzt, steht dort `4/5`.

Nur Bausteine, die mit dem `Programm`-Baustein verbunden sind, werden ausgeführt.

Starte und beobachte dein Programm

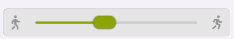
Mit den Knöpfen unter dem Spielfeld steuerst du den Ablauf: 


führt dein Programm von Anfang bis Ende aus und überprüft, ob deine Lösung korrekt ist.


setzt alles auf die Startposition zurück. Dein gebautes Programm bleibt dabei erhalten.


Führt nur den nächsten Baustein aus. Ideal zur Fehlersuche.


führt die noch fehlenden Schritte bis zum Ende aus.


bestimmt, wie schnell dein Programm bei einem Klick auf Play ausgeführt wird.

Verbessere deine Lösung

Funktioniert nicht? Kein Problem! Beobachten, ändern, neu testen – das ist Programmieren. Auch Profis schaffen es selten im ersten Versuch.

Hinweis

Bausteine kopieren und Änderungen rückgängig machen

Du musst gleiche Bausteine oder Bausteingruppen nicht jedes Mal neu zusammensetzen. Mit Tastenkombinationen kannst du sie schnell kopieren und wieder einfügen.

	Windows und Linux		macOS
Kopieren	`Strg + C`	Kopieren	`cmd + C`
Einfügen	`Strg + V`	Einfügen	`cmd + V`
Rückgängig	`Strg + Z`	Rückgängig	`cmd + Z`
Wiederholen	`Strg + Y`	Wiederholen	`cmd + Shift + Z`

Wenn du ohne Tastatur arbeitest, findest du diese Funktionen auch im Menü hinter den drei waagerechten Strichen am rechten Rand der Aufgabenansicht.

Info

In diesem Arbeitsheft gibt es zwei Arten von Aufgaben:

Heft-Aufgaben
Diese Aufgaben bearbeitest du direkt im Arbeitsheft.

Kombi-Aufgaben
Du überlegst im Heft und prüfst deine Lösung am Computer.

Auf `jwinf.de` findest du eine Kachel **Arbeitsheft Algorithmen**. In diesem Bereich gibt es zu jedem Kapitel eine Sammlung an Aufgaben, teils aus dem Heft (siehe Kombi-Aufgaben) und teils weitere Übungsaufgaben.

2 Anweisung, Sequenz, Programm: Die Bausteine des Codes

Super, du hast deine erste Nachricht an die digitale Welt gesendet! Aber was genau hast du da eigentlich getan, als du `"Hallo Welt"` auf dem Bildschirm ausgegeben hast? Du hast dein erstes kleines Programm geschrieben. Lass uns nun die Bausteine genauer ansehen, aus denen jedes Programm besteht.

Die Anweisung: Ein einzelner Handlungsschritt

Jeder einzelne Schritt, den der Computer ausführen soll, ist eine **Anweisung**, z.B. 

.

Du hast sicher bemerkt, dass manche Anweisungen noch zusätzliche Informationen benötigen. Die Anweisung 

allein weiß ja nicht, was sie schreiben soll. Deshalb hast du ihr den Text `"Hallo Welt"` in einem weiteren Baustein mitgegeben. Eine Anweisung kann also aus einem oder mehreren Bausteinen bestehen.

Die Sequenz: Wenn die Reihenfolge zählt

Selten besteht eine Aufgabe nur aus einem einzigen Schritt. Wenn du mehrere Anweisungen untereinander reihst, um eine komplexere Aufgabe zu erledigen, nennst du das eine **Sequenz**.

Das Wichtigste an einer Sequenz ist die Reihenfolge. Der Computer arbeitet die Anweisungen stur von oben nach unten ab, eine nach der anderen. Genau wie beim Anziehen am Morgen: Zuerst die Socken, dann die Schuhe. Die verkehrte Reihenfolge führt zu einem ziemlich komischen Ergebnis! In Blockly baust du eine Sequenz, indem du die Anweisungs-Bausteine einfach untereinander klickst.

Das Programm: Dein fertiges Werk

Und was ist nun ein **Programm**? Ganz einfach: Die gesamte Abfolge von Anweisungen – also deine Sequenz –, die du erstellst, um ein bestimmtes Ziel zu erreichen, ist dein Programm. Dein `"Hallo Welt"`-Programm war also bereits dein erstes Programm!

Merke

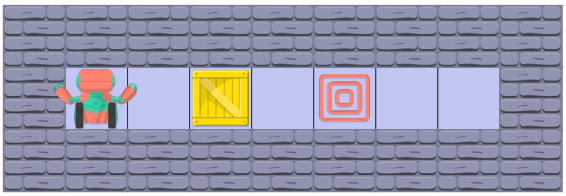
Die Anweisung ist ein einzelner Handlungsschritt, den ein Objekt (wie ein Roboter) ausführen soll. **Die Sequenz** ist eine geordnete Abfolge von Anweisungen. Die Reihenfolge bestimmt das Ergebnis. **Das Programm** fasst alle Anweisungen in Sequenzen zusammen, um ein bestimmtes Ziel zu erreichen oder ein Problem zu lösen.

Aufgabe 2.1 – Der Reihenfolge-Check

Beurteile, ob die folgende Aussage wahr oder falsch ist, und begründe deine Entscheidung in einem Satz:

„In einer Sequenz ist die Reihenfolge der Anweisungen unwichtig, solange alle notwendigen Anweisungen vorhanden sind."

Aufgabe 2.2 – Der Code-Sortierer



Roboter-Programm ☐

schiebe Kiste ☐

gehe vorwärts ☐

drehe nach rechts ☐

schiebe Kiste ☐

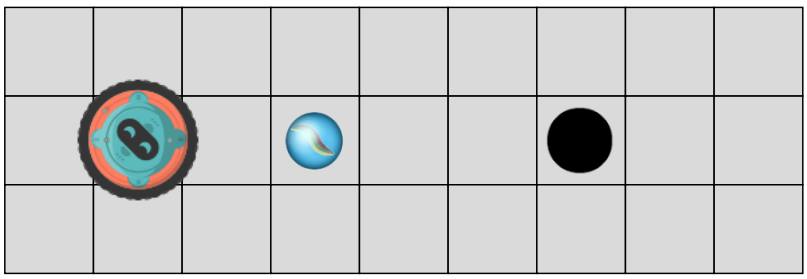
Der Roboter soll die Kiste auf die markierte Position bringen. Seine Anweisungen sind jedoch durcheinandergeraten. Ordne die Bausteine in der korrekten Reihenfolge, indem du die Schrittnummern (1, 2, 3, ...) in die dafür vorgesehenen Kästchen einträgst.

Aufgabe 2.3 – Die Alltags-Analogie

Erläutere den Zusammenhang zwischen den Begriffen Anweisung, Sequenz und Programm. Nutze für deine Erklärung ein passendes Beispiel aus dem Alltag (z.B. ein Kochrezept, eine Bastelanleitung oder eine Wegbeschreibung).

Aufgabe 2.4 – Der Planungs-Profi

Programmiere den Roboter:
Der Roboter soll die Murmel im Loch ablegen.



gehe nach rechts
gehe nach links
hebe Murmel auf
lege Murmel ab

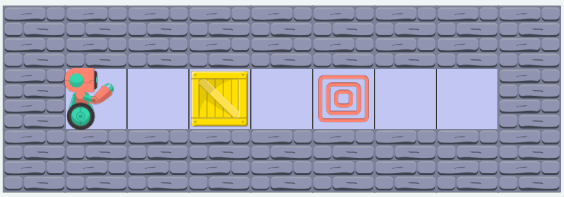
Schätze ein, wie viele Bausteine zur Lösung der Aufgabe benötigt werden und notiere deine Schätzung: ☐
Löse jetzt die Aufgabe auf dem Computer. War deine Schätzung richtig? (✓ / ×)

Exkurs

Die Aufgabentypen auf der jwinf.de-Plattform

Auf der Plattform des Jugendwettbewerbs Informatik (`jwinf.de`) gibt es vier verschiedene Aufgabentypen. Hier lernst du, sie zu unterscheiden.

1. Der Roboter aus der Seitenansicht

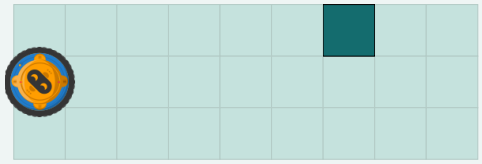


Stell dir vor, du stehst direkt neben dem Roboter. Dieser Roboter hat eine klare „Nase“ und schaut immer in eine bestimmte Richtung. Seine Bewegungen stehen in Abhängigkeit zu seiner eigenen Ausrichtung. Das nennt man **relativ**.

- **Blickrichtung:** Zu Beginn ist festgelegt, wohin der Roboter schaut.
- **Bewegung:** Um die Richtung zu ändern, muss er sich erst drehen.
- **Wichtige Bausteine:** gehe vorwärts, drehe nach links, drehe nach rechts.

Merke dir: Bei diesem Roboter denkst du immer aus seiner eigenen Perspektive: „Drehe ich mich nach links oder rechts? Und wie viele Schritte gehe ich dann vorwärts?“

2. Der Roboter aus der Vogelperspektive

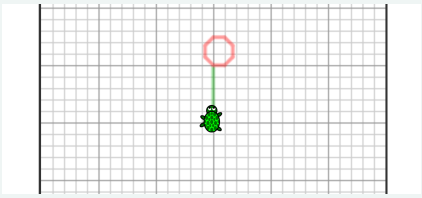


Stell dir vor, du schwebst wie eine Drohne direkt über dem Spielfeld. Dieser Roboter bewegt sich auf dem Spielfeld wie auf einem Schachbrett. Seine Bewegungen sind **absolut** und nicht von einer eigenen Blickrichtung abhängig.

- **Blickrichtung:** Dieser Roboter braucht keine Blickrichtung.
- **Bewegung:** Du sagst ihm direkt, in welche Himmelsrichtung er einen Schritt machen soll.
- **Wichtige Bausteine:** gehe nach oben, gehe nach rechts, gehe nach unten und gehe nach links.

Merke dir: Bei diesem Roboter denkst du immer aus der Draufsicht: „Soll der Roboter auf dem Spielfeld einen Schritt nach oben, unten, links oder rechts machen?“

3. Die Turtle – Dein digitaler Zeichenstift



Bei diesen Aufgaben steuerst du eine Turtle (Schildkröte), die einen Stift an ihrem Panzer hat. Deine Aufgabe ist es, geometrische Formen, Muster oder Bilder zu zeichnen.

- **Konzept:** Die Turtle verhält sich sehr ähnlich wie der Roboter aus der Seitenansicht. Sie hat eine Position und eine Blickrichtung.
- **Der Stift:** Der entscheidende Unterschied zum Roboter ist der Stift. Du kannst diesen aufsetzen, um eine Linie zu zeichnen, oder ihn hochheben, um die Turtle zu einer anderen Stelle zu bewegen, ohne dabei zu zeichnen.
- **Wichtige Bausteine:** gehe, setze Stift auf, hebe Stift ab, drehe um 90° nach links ↺ und drehe um 90° nach rechts ↻.

Merke dir: Hier bist du ein Künstler. Du denkst darüber nach, wie du einen Stift über ein Blatt Papier führen würdest, um eine bestimmte Zeichnung zu erstellen. Das macht dann die Turtle für dich.

4. Text-Aufgaben – Reine Textausgabe

```
Eingabe
1
```

```
Ausgabe deines Programms
1
```

Nicht alle Programmieraufgaben sind visuell. Manchmal ist das Ziel, eine ganz bestimmte Textnachricht auszugeben. Daran erinnerst du dich vielleicht aus dem `"Hallo Welt"`-Programm.

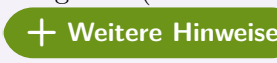
- **Konzept:** Es gibt keine Roboter- oder Turtle-Welt, sondern nur ein Ein- und ein Ausgabefenster.
- **Ziel:** Dein Programm muss exakt den geforderten Text erzeugen – jedes Zeichen, jedes Leerzeichen und jeder Zeilenumbruch zählt.
- **Wichtige Bausteine:** 

und 

.

Merke dir: Bei diesen Aufgaben bist du ein Autor. Deine Aufgabe ist es, präzise Nachrichten zu formulieren und auszugeben.

Info

Auf `jwinf.de` findest du bei vielen Aufgaben (insbesondere im Wettbewerb) unter dem Aufgabentext einen Button: 

. Auf der rechten Seite erscheint dann auch ein grünes Fragezeichen.
Hier befinden sich meist wichtige Hinweise zu dieser Aufgabe, manchmal auch Tipps zum Lösen.

3 Algorithmen: Das Rezept zum Problemlösen

Herzlichen Glückwunsch! Du hast bereits deine ersten kleinen Programme geschrieben und gelernt, wie man aus einzelnen Anweisungen eine Sequenz baut. Aber was hast du dabei eigentlich erschaffen? Du hast deine ersten **Algorithmen** entworfen!

Ein Algorithmus ist eine eindeutige, schrittweise Handlungsanweisung zur Lösung eines Problems – vergleichbar mit einem Kochrezept oder einer Aufbauanleitung. Dein Algorithmus ist das Rezept, und der Computer ist der Koch, der die Anweisungen stur ausführt. Algorithmen begegnen dir auch ständig im Alltag: beim Binden deiner Schuhe, beim Zähneputzen oder beim Weg zur Schule.

Damit eine Anleitung in der Informatik jedoch als echter Algorithmus gilt, muss sie **vier goldene Regeln** erfüllen:

1. Eindeutigkeit

Jeder Schritt muss absolut klar und unmissverständlich sein. Es darf keinen Raum für Vermutungen geben.

Beispiel

Eindeutig: Nimm 200 Gramm Mehl.
Nicht eindeutig: Nimm etwas Mehl. (Wie viel ist „etwas“?)

2. Endlichkeit

Ein Algorithmus muss nach einer bestimmten Anzahl von Schritten zu einem Ergebnis kommen und aufhören. Er darf nicht ewig weiterlaufen.

Beispiel

Endlich: Rühre den Teig 2 Minuten lang. (Hat ein klares Ende)
Nicht endlich: Rühre immer weiter. (Wann soll man aufhören?)

3. Ausführbarkeit

Jeder einzelne Schritt muss tatsächlich praktisch machbar sein.

Beispiel

Ausführbar: Heize den Ofen auf 180°C vor. (Das kann ein normaler Ofen.)
Nicht ausführbar: Heize den Ofen auf 1000°C vor. (Das kann kein normaler Ofen.)

4. Allgemeingültigkeit

Ein Algorithmus sollte nicht nur ein einziges, spezielles Problem lösen, sondern eine ganze Klasse von ähnlichen Aufgaben.

Beispiel

Allgemeingültig: Ein Rezept für einen Apfelkuchen funktioniert mit jeder Sorte von Äpfeln.
Nicht allgemeingültig: Ein Rezept, das nur mit den drei Äpfeln funktioniert, die am Dienstag von einem bestimmten Baum gefallen sind.

Was bedeutet das für deine Programme?

Bevor du am Computer Bausteine zusammenklickst, überlegst du dir zuerst den logischen Ablauf – das ist dein **Algorithmus**. Das eigentliche Zusammenbauen der Bausteine nennt man dann **Implementierung** (Umsetzung). Ein guter Programmierer plant immer erst den Algorithmus, bevor er den Code schreibt!

Merke

Ein **Algorithmus** ist eine präzise und endliche Handlungsanweisung zur Lösung eines Problems.
Er muss immer **eindeutig**, **endlich**, **ausführbar** und **allgemeingültig** sein.

Aufgabe 3.1 – Die vier goldenen Regeln

Ein Algorithmus ist wie ein gutes Rezept. Er muss vier „goldene Regeln“ erfüllen. Setze die vier grundlegenden Eigenschaften von Algorithmen in die passende Lücke ein.

(a) Die Eigenschaft der _____ bedeutet, dass jeder Schritt klar und unmissverständlich sein muss, ohne Raum für Interpretationen.

(b) Ein Algorithmus muss nach einer bestimmten Anzahl von Schritten fertig sein und darf nicht ewig weiterlaufen. Diese Eigenschaft nennt man _____ .

(c) Wenn jeder einzelne Schritt tatsächlich umsetzbar ist (z.B. ein Ofen nicht auf 1000°C geheizt werden muss), erfüllt der Algorithmus die Regel der _____ .

(d) Die _____ sorgt dafür, dass ein Algorithmus eine ganze Klasse von ähnlichen Problemen lösen kann (z.B. jeden Apfelkuchen) und nicht nur ein einziges, spezielles Problem.

Aufgabe 3.2 – Der Algorithmus-Inspektor

Bei den folgenden Anweisungen wurde gegen eine der vier goldenen Regeln verstoßen. Analysiere jede Anweisung und erkläre, welche Regel am deutlichsten verletzt wird.

(a) Gib Salz in die Suppe.

(b) Gehe so lange geradeaus, wie du Lust hast.

(c) Ein Programm, das nur die Rechnung 8 + 4 ausführen kann.

(d) Notiere dir alle Sterne am Himmel.

Aufgabe 3.3 – Rezept für den Alltag

Algorithmen sind überall. Deine Aufgabe ist es, selbst einen zu schreiben. Entwirf einen Algorithmus für die Aufgabe: Zähneputzen. Schreibe eine klare Schritt-für-Schritt-Anleitung in nummerierter Reihenfolge. Achte darauf, dass deine Anleitung alle vier goldenen Regeln erfüllt.

Aufgabe 3.4 – Der Algorithmen-TÜV

Ein Freund hat versucht, einen Algorithmus zu schreiben, um die höchste Zahl aus einem Stapel von 10 Spielkarten zu finden. Sein Plan ist aber noch nicht perfekt. Bewerte seinen Algorithmus, indem du die Tabelle ausfüllst. Begründe, warum eine Regel erfüllt ist oder nicht, und mache einen Verbesserungsvorschlag, falls nötig.

Der Algorithmus:

1. Nimm eine beliebige Karte vom Stapel. Merke dir ihre Zahl als `Höchste Zahl bisher`.
2. Vergleiche sie mit einer anderen Karte.
3. Wenn die neue Karte größer ist, merke dir die neue Zahl.
4. Wiederhole das ein paar Mal.
5. Die letzte gemerkte Zahl ist das Ergebnis.

Regel	**Erfüllt?** (✓ / ×)	**Begründung / Verbesserungsvorschlag**
Eindeutigkeit		
Endlichkeit		
Ausführbarkeit		
Allgemeingültigkeit		

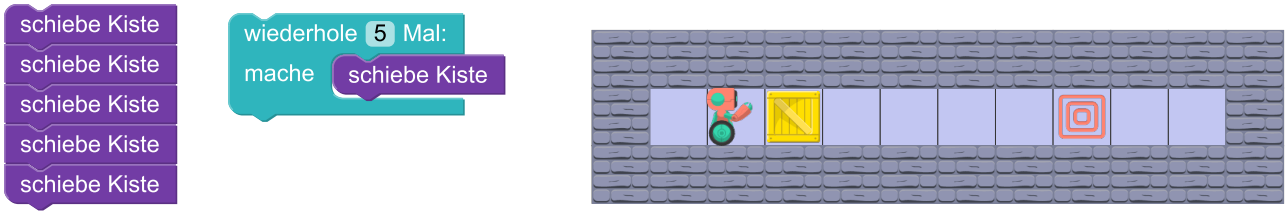
4 Wiederholungen mit fester Anzahl: Die schlaue Abkürzung im Code

Du weißt jetzt, was ein Algorithmus ist und wie du aus einzelnen Anweisungen eine Sequenz baust, um Probleme zu lösen. Aber was ist, wenn dein Algorithmus eine Sequenz nicht nur einmal, sondern fünfmal, zehnmal oder sogar hundertmal ausführen soll?

Anstatt eine Sequenz immer wieder zu kopieren, gibt es eine schlaue Abkürzung: **die Wiederholungsanweisung**. Sie ist ein sehr wirkungsvoller Baustein, der dir unglaublich viel Arbeit erspart.

Stell dir einen magischen Rahmen vor. Du packst die Anweisungen, die wiederholt werden sollen, einfach in diesen Rahmen und sagst ihm dann, wie oft er seinen Inhalt ausführen soll. Statt also zehn Anweisungen untereinander zu setzen, nimmst du nur eine einzige und packst sie in eine Wiederholungsanweisung, der du die Zahl 10 gibst. Das Ergebnis ist dasselbe, aber dein Programm ist viel kürzer und übersichtlicher!

Der Unterschied ist beeindruckend, wie du in der folgenden Abbildung siehst.



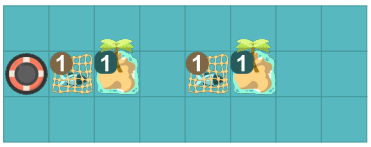
Das Beste daran: Die Anzahl der Wiederholungen kannst du ganz einfach selbst festlegen. Du brauchst nur eine Zahl im `Wiederholungs`-Baustein zu ändern, und schon wird dein Programm zehn-, zwanzig- oder hundertmal ausgeführt, ohne dass du eine einzige Anweisung mehr hinzufügen musst.

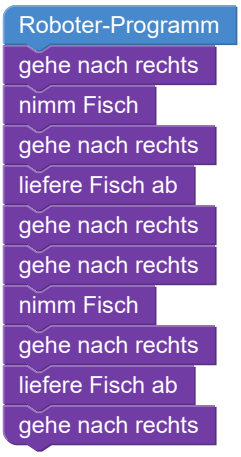
Merke

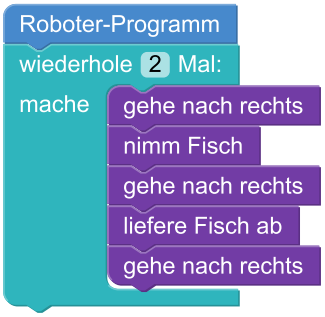
Eine **Wiederholung mit fester Anzahl** führt eine Anweisungssequenz exakt so oft aus, wie es vorab durch eine Zahl festgelegt wurde.

Aufgabe 4.1 – Der Effizienz-Check

Juna und Juno stehen vor der gleichen Aufgabe: Ihr Roboter soll auf dem Meer zwei Fische aus zwei Netzen fischen und sie auf den nahegelegenen Inseln ablegen. Beide haben eine funktionierende Lösung programmiert, aber ihre Programme, die du in der Abbildung unten siehst, sind unterschiedlich lang.







(a) Zähle die Anzahl der Bausteine, die Juno (durch Kopieren) und Juna (mit der Wiederholung) jeweils für ihre Lösung benötigen.

Junos Lösung: _____ Bausteine Junas Lösung: _____ Bausteine

(b) Die Aufgabe wird erweitert. Es gibt nicht nur zwei, sondern zehn Netze und zehn Inseln, die genau wie in der Abbildung oben aufgebaut sind. Ermittle, wie viele Bausteine Juno und Juna jetzt für ihre jeweilige Lösungsstrategie benötigen würden.

Junos Lösung: _____ Bausteine Junas Lösung: _____ Bausteine

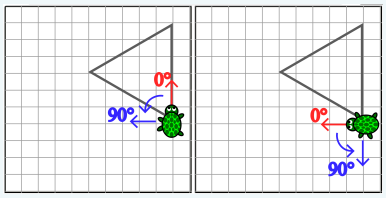
Erkläre, warum Junas Ansatz bei dieser größeren Aufgabe einen entscheidenden Vorteil hat.

(c) Formuliere eine allgemeine Regel, ab wann es sich lohnt, eine Wiederholungsanweisung zu verwenden, anstatt die gleichen Anweisungen untereinander zu reihen.

Hinweis

Wie dreht sich die Schildkröte?

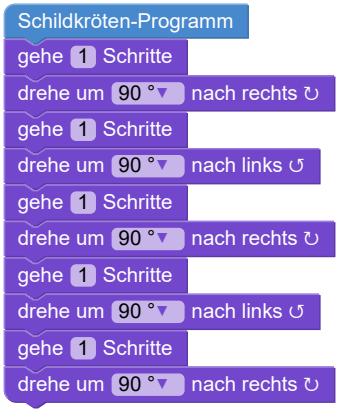
Die Schildkröte hat eine Blickrichtung, genau wie du. Die Drehanweisung dreht ihre Nase um die angegebene Gradzahl nach rechts oder nach links.
Eine volle Drehung zurück zum Startpunkt sind immer 360°.



Aufgabe 4.2 – Der Code-Inspektor

In der Abbildung unten siehst du ein Schildkröten-Programm.

(a) Zeichne das Ergebnis des Programms in das karierte Feld. Die Turtle startet unten links mit Blick nach oben. Beachte dabei: *1 Schritt = 1 Kästchen.*



(b) Analysiere das gezeichnete Muster und den dazugehörigen Code. Finde die genaue Sequenz von Anweisungen, die sich ständig wiederholt und notiere sie.

(c) Optimiere das Programm nun mit einer Wiederholungsanweisung für eine Treppe mit 8 Stufen. Notiere den neuen Code.

Info

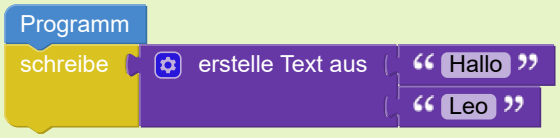
Für einige Textaufgaben benötigen wir noch andere Anweisungen:

Dieser Baustein liest zeilenweise den Text aus einem Eingabefeld. Dort können beliebig viele Namen stehen.

Dieser Baustein ist wie ein Klebestift für Wörter. Er nimmt zwei Textteile und verbindet sie zu einem neuen, längeren Text.

Beispiel

Achte hier auf das Leerzeichen nach `"Hallo "`. Der neue Text ist: `"Hallo Leo"`.



Aufgabe 4.3 – Junos Begrüßungsautomat

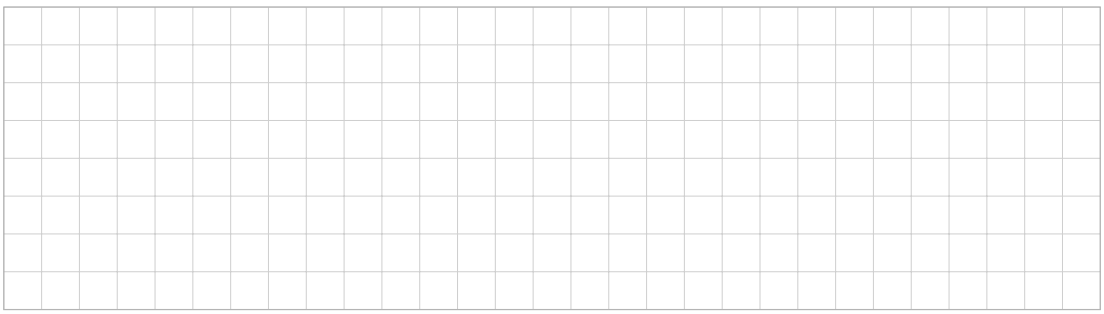
Erinnerst du dich an dein erstes `"Hallo Welt"`-Programm aus dem Kapitel 1? Juno möchte dieses Programm nun Schritt für Schritt erweitern, um seine Familie und Freunde automatisch zu begrüßen.

(a) Zuerst soll ein einfaches Programm seine Schwester Juna begrüßen. Die Ausgabe soll `"Hallo Juna"` lauten. Ergänze den fehlenden Text im Programm:

(b) Bisher begrüßt das Programm nur Juna. Juno möchte, dass der Automat künftig jeden eingegebenen Namen verwendet. Seine Lösung ist wie folgt:

Erkläre, warum Junos Lösung nicht richtig ist und korrigiere sie.

(c) **Herausforderung**: Vier Namen der Freunde stehen im Eingabefeld. Entwirf ein Programm, das alle vier nacheinander begrüßt. Du darfst dabei nur 5 Bausteine verwenden.



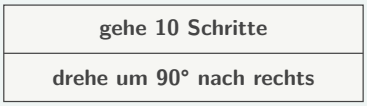
Exkurs

Algorithmen darstellen mit Struktogrammen

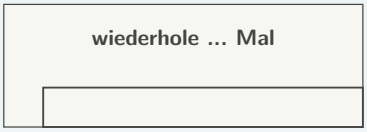
Wie stellt man die Logik eines Algorithmus dar, ohne sich um die genauen Bausteine oder den Code einer bestimmten Programmierumgebung zu kümmern? Genau dafür gibt es die **Struktogramme**.
Ein Struktogramm ist eine grafische, sprachunabhängige Darstellung der logischen Struktur eines Algorithmus. Dabei wird der gesamte Algorithmus in einem großen Rechteck dargestellt, das in kleinere Blöcke für die einzelnen Schritte unterteilt wird.

Die Grundbausteine:

Die Anweisung und die Sequenz: Eine einfache Anweisung wird in ein Rechteck geschrieben. Mehrere Rechtecke untereinander bilden eine Sequenz, die von oben nach unten gelesen wird.

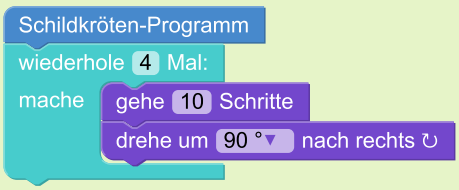


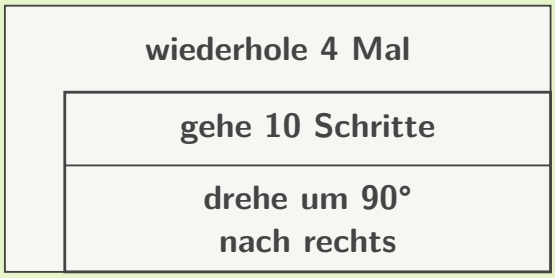
Die Wiederholung (mit fester Anzahl): Die Wiederholung umschließt die Anweisungen, die wiederholt werden sollen. Dabei wird angegeben, wie oft die Wiederholung durchlaufen wird.



Beispiel

Das Programm und das dazugehörige Struktogramm, um ein Quadrat zu zeichnen:





Die Vorteile der Struktogramme:

- Klare, strukturierte Darstellung der Logik.
- Verständlich ohne Programmierkenntnisse.
- Geschlossener Block, alles auf einen Blick.
- Universell: In jede Programmiersprache umsetzbar.

In den späteren Kapiteln wirst du noch weitere Blöcke für bedingte Anweisungen und andere Wiederholungsarten kennenlernen.

Merke

Ein **Struktogramm** ist der grafische Bauplan eines Algorithmus.

Aufgabe 4.4 – Deine Morgenroutine

Jeden Tag folgst du nach dem Aufwachen einer bestimmten Routine. Erstelle ein Struktogramm für deine Morgenroutine (vom Aufwachen bis zum Verlassen des Hauses) mit mindestens 5 Schritten.

Aufgabe 4.5 – Junas Pausenbrot

Juna möchte, dass ihr Roboter ihr Pausenbrot zubereitet. Sie hat ihm dafür Anweisungen in einem Struktogramm gegeben. Juna wünscht sich ein einfaches Käsesandwich: Eine Scheibe Brot, darauf Butter, darauf Käse, eine Scheibe Tomate und zum Schluss die zweite Scheibe Brot obendrauf.
Als sie das Ergebnis probiert, stellt sie fest: Da stimmt etwas nicht! Der Roboter hat sich genau an ihr Struktogramm gehalten.

nimm zwei Scheiben Brot
lege Käse darauf
lege Tomate darauf
bestreiche vorher das Brot mit Butter
lege eine Scheibe Brot darauf

(a) Beschreibe genau, wie das fertige Sandwich aussieht. Warum führen Junas Anweisungen nicht zum gewünschten Ergebnis?

(b) Korrigiere Junas Fehler. Zeichne dazu ein neues, korrektes Struktogramm, das den richtigen Ablauf für die Zubereitung des Käsesandwiches darstellt.

(c) Der Roboter soll 4 identische Sandwiches für Juna und ihre drei Geschwister machen. Erweitere dein korrektes Struktogramm aus (b) effizient mit einer Wiederholung.

5 Verschachtelte Wiederholungen: Die Wiederholung in der Wiederholung

Du bist jetzt schon ein Profi darin, dem Computer zu sagen, dass er eine Sequenz von Anweisungen mehrfach wiederholen soll. Wir gehen noch einen Schritt weiter: Wir verschachteln Wiederholungen ineinander. Klingt kompliziert? Ist es aber nicht!

Das Prinzip lässt sich leicht mit einem Sporttraining erklären: Ein Sportler soll 10 Runden um den Platz laufen und in jeder Runde 5 Liegestütze machen.

Hier haben wir zwei Wiederholungen ineinander verschachtelt:

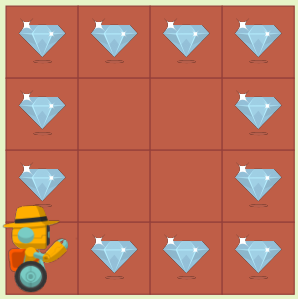
Die 10 Runden sind die **äußere Wiederholung**.
Die 5 Liegestütze pro Runde sind die **innere Wiederholung**.

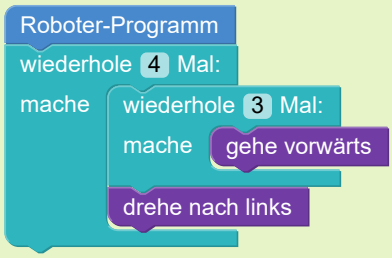
1. Der Sportler läuft die erste Runde. Dann macht er 5 Liegestütze (die innere Wiederholung läuft komplett durch).
2. Er läuft die zweite Runde. Dann macht er wieder 5 Liegestütze.
3. ...und so weiter, bis er alle 10 Runden geschafft hat.

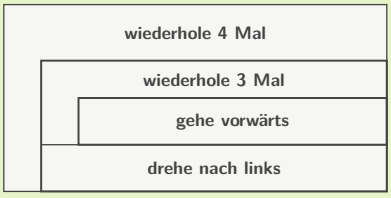
Am Ende ist er 10 Runden gelaufen und hat insgesamt `10 x 5 = 50` Liegestütze gemacht.

Genau dieses Prinzip nennen wir in der Programmierung eine **verschachtelte Wiederholung**. Du nimmst einfach einen `Wiederholungs`-Baustein und steckst ihn in einen anderen `Wiederholungs`-Baustein.

Beispiel







Mit dieser Technik kannst du komplexe Muster und Strukturen mit sehr wenigen Bausteinen erzeugen.

Merke

Bei einer **verschachtelten Wiederholung** wird die **innere Wiederholungsanweisung** bei jedem einzelnen Durchlauf der **äußeren Wiederholungsanweisung** vollständig durchlaufen.

Aufgabe 5.1 – Die Stempel-Meisterschaft

Juna und Juno haben Besuch von ihrem Cousin Fino. Fino prahlt damit, dass er der schnellste Programmierer von allen ist. Um das zu beweisen, rufen sie die große „Stempel-Meisterschaft“ aus.

Die Aufgabe: Eine Stempelmaschine soll so programmiert werden, dass sie 5 Blätter Papier nacheinander verarbeitet. Auf jedes einzelne Blatt sollen genau 4 Namensstempel gedruckt werden.

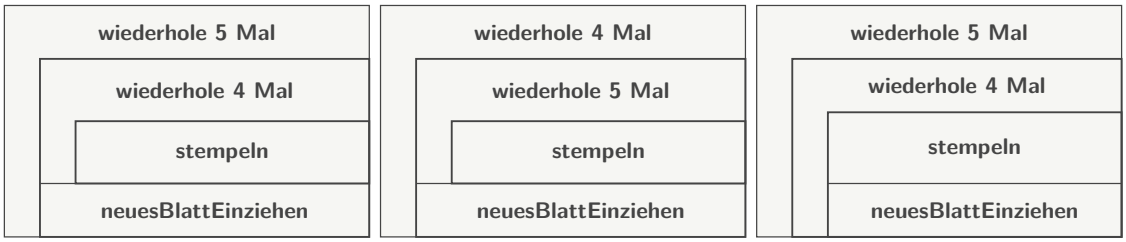
Die Maschine kennt nur drei Anweisungen:

`stempeln`: Druckt einen Stempel auf das aktuelle Blatt und verschiebt das Blatt zur Position des neuen Stempels.

`neuesBlattEinziehen`: Wirft das aktuelle Blatt aus und zieht ein frisches Blatt ein.

`wiederhole ... Mal`: Wiederholt die Anweisungen.

Juna, Juno und Fino haben drei verschiedene Lösungen programmiert. Aber nur eine davon funktioniert genau richtig!



Junas Lösung **Finos Lösung** **Junos Lösung**

(a) Welche Lösung ist die richtige? Begründe deine Antwort.

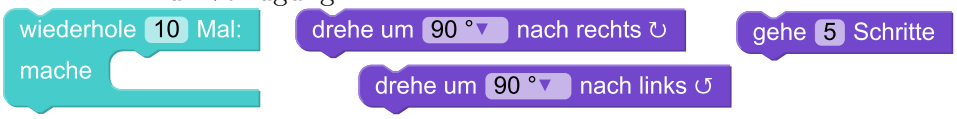
(b) Finde die Fehler! Erkläre, was bei den beiden anderen Lösungen schiefläuft.

_____ Lösung:

_____ Lösung:

Aufgabe 5.2 – Die Quadrat-Reihe

In Kapitel 4 hast du gelernt, wie man in einem Schildkröten-Programm mit einer Wiederholung ein Quadrat zeichnet. Folgende Bausteine stehen der Schildkröte zur Verfügung:



(a) Skizziere das Programm aus den Bausteinen oder das Struktogramm, das mit einer Wiederholungsanweisung ein einzelnes Quadrat mit der Seitenlänge 5 zeichnet.

(b) Jetzt wollen wir nicht nur ein, sondern fünf Quadrate lückenlos nebeneinander zeichnen. Probiere es aus. Nimm einen Stift und zeichne fünf identische Quadrate direkt nebeneinander. Wichtig dabei ist: Du darfst den Stift nicht absetzen! Verfolge den Weg deines Stiftes genau.

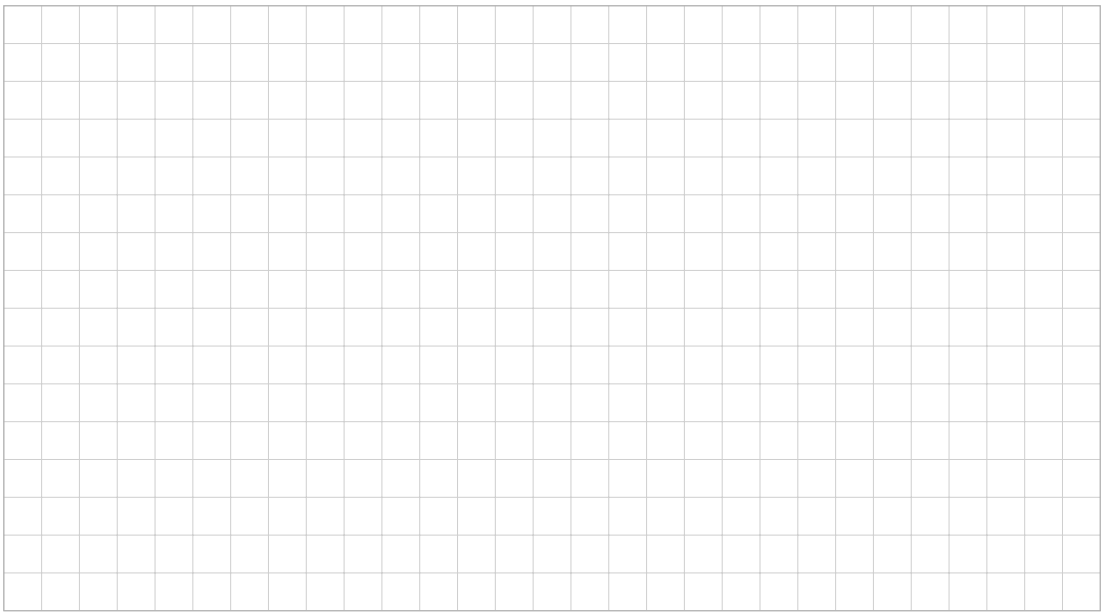
(c) Wie du beim Zeichnen gemerkt hast, wiederholt sich nicht nur das Zeichnen des Quadrats, sondern auch das Weiterbewegen zum nächsten Startpunkt.

Überlege, welche Anweisungen in die innere Wiederholung und welche Anweisungen in die äußere Wiederholung gehören.

Die innere Wiederholung:

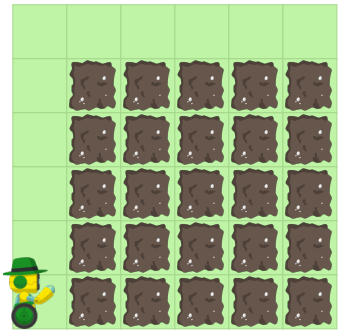
Die äußere Wiederholung:

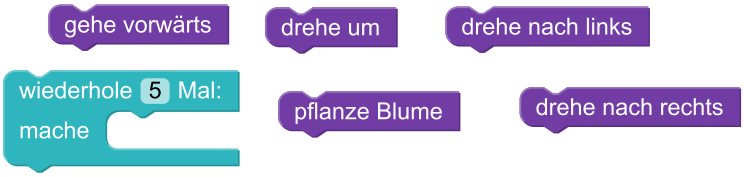
Entwirf ein Programm oder ein vollständiges Struktogramm mit einer verschachtelten Wiederholung, das die Aufgabe elegant löst.



Aufgabe 5.3 – Der Roboter-Garten

Ein Gartenroboter soll ein quadratisches Beet der Größe 5x5 Felder anlegen. In jedes einzelne Feld soll er eine Blume pflanzen.
Der Roboter startet in der Ecke unten links. Er kann die folgenden Anweisungen benutzen:

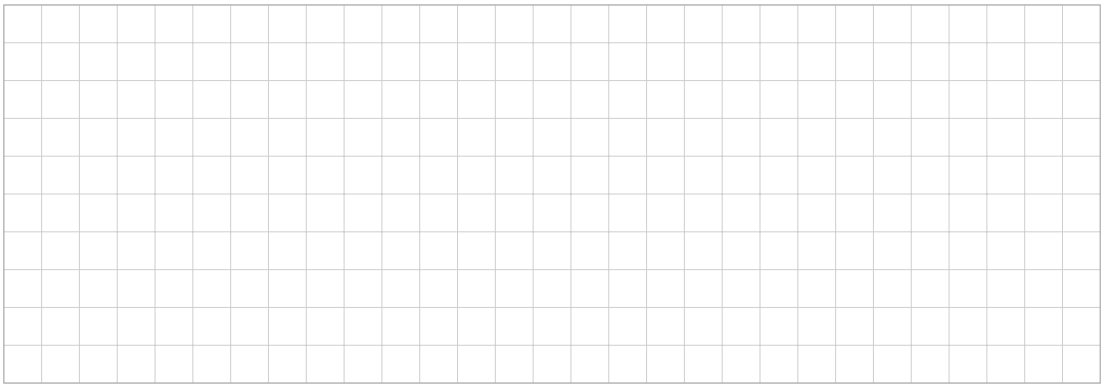




(a) Beschreibe in Worten, was die innere und was die äußere Wiederholung in diesem Fall tun.

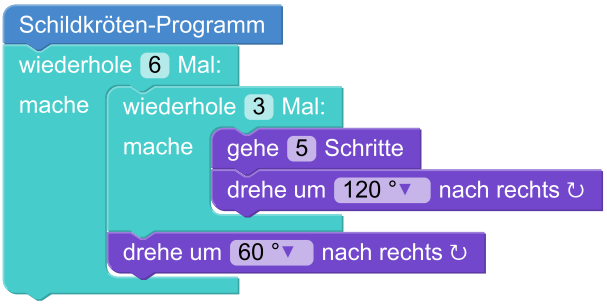
(b) Beschreibe das Programm oder erstelle ein Struktogramm, das den kompletten Ablauf für den Roboter darstellt.

Hinweis: Der Roboter kann nicht durch bereits gepflanzte Blumen laufen. Wie kommt er zum Start der nächsten Reihe?

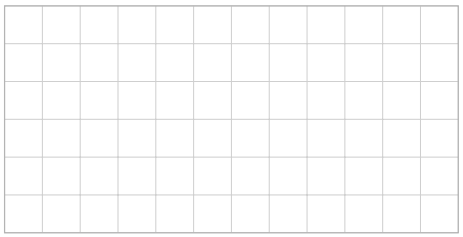


Aufgabe 5.4 – Der Code-Inspektor

Dein Auftrag als Code-Inspektor ist es, ein mysteriöses Programm für die Schildkröte zu analysieren. Du sollst vorhersagen, welche Zeichnung am Ende auf dem Bildschirm erscheint, ohne das Programm tatsächlich auszuführen.

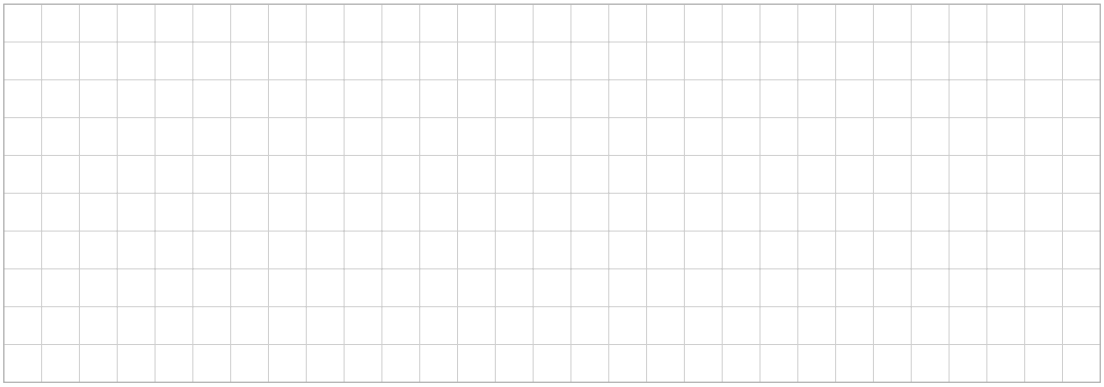


(a) Analysiere zuerst nur die innere Wiederholungsanweisung des Programms. Was für eine geometrische Figur zeichnet die Schildkröte, wenn sie diesen Code-Teil genau einmal ausführt? Zeichne oder beschreibe die Form.



(b) Analysiere nun die äußere Wiederholung. Was passiert nach dem Zeichnen der Figur?

(c) Jetzt bist du bereit, das gesamte Programm zu entschlüsseln. Zeichne eine Skizze des Musters, das die Schildkröte nach der Ausführung des gesamten Programms erstellt hat. Teste anschließend das Programm am Computer. War deine Lösung richtig?



(d) **Expertenfrage**: Warum sind die 60° entscheidend für die Symmetrie der gesamten Zeichnung? Erkläre kurz.

6 Bedingte Anweisungen: Programme, die Entscheidungen treffen

Deine Programme können schon eine ganze Menge. Sie folgen aber bisher nur einem festen Plan. Jetzt lernen sie, Entscheidungen zu treffen – genau wie du im Alltag:

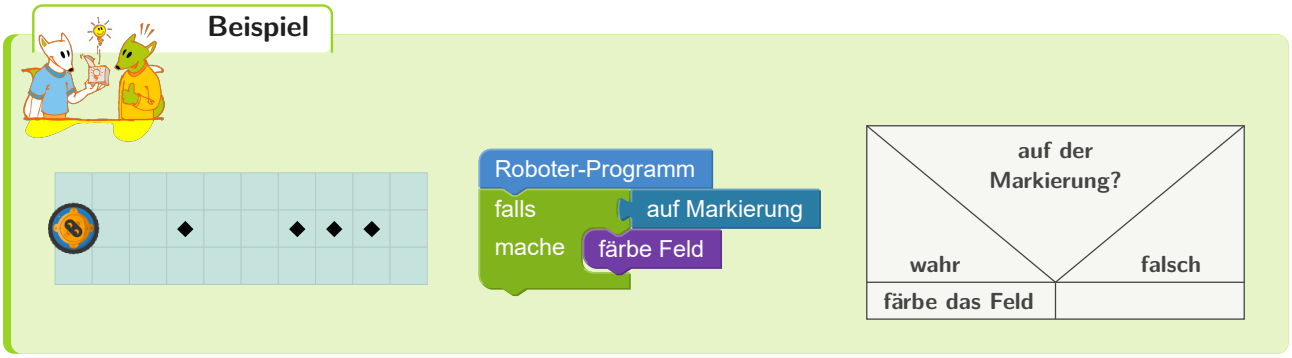
- **Falls** es regnet, **dann** nimmst du einen Schirm mit.
- **Falls** du Hunger hast, **dann** machst du dir ein Brot.

Genau diese Fähigkeit, auf eine Situation zu reagieren, können wir auch unseren Programmen beibringen. Dafür nutzen wir die **bedingte Anweisung**. Sie macht Programme flexibel und intelligent.

Die Falls-Dann-Anweisung (Einseitige bedingte Anweisung)

Die einfachste bedingte Anweisung funktioniert nach dem **Falls-Dann-Prinzip**. Du gibst dem Computer eine Bedingung, die er überprüfen soll. Eine Bedingung ist wie eine Ja/Nein-Frage, die nur mit `wahr` oder `falsch` beantwortet werden kann.

- Ist die Antwort **`wahr`** (also „Ja“), dann wird die Anweisung oder die Sequenz ausgeführt, die im Dann-Teil steckt.
- Ist die Antwort **`falsch`** (also „Nein“), wird der Dann-Teil einfach übersprungen und das Programm macht mit der nächsten Anweisung weiter.



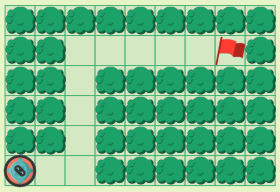
Die Erweiterung: Falls-Dann-Sonst (Zweiseitige bedingte Anweisung)

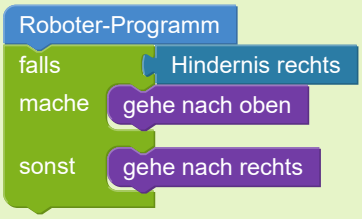
Manchmal möchten wir aber auch festlegen, was passieren soll, wenn die Bedingung nicht erfüllt ist.

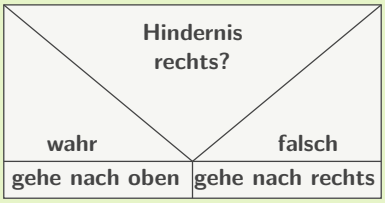
- **Falls** die Ampel grün ist, **dann** gehe über die Straße. **Sonst** warte.

Dafür gibt es die **Falls-Dann-Sonst-Anweisung**. Sie bietet einen zweiten Weg an. Ist die Bedingung `wahr`, wird der **Dann-Teil** ausgeführt. Ist sie aber `falsch`, wird der **Sonst-Teil** ausgeführt. So oder so, das Programm weiß immer, was zu tun ist.

Beispiel







Mit bedingten Anweisungen können deine Programme auf Eingaben reagieren, Fehler vermeiden und sich an unterschiedliche Situationen anpassen.

Merke

Eine **bedingte Anweisung** (Verzweigung) prüft eine Bedingung und wählt dann, je nach Ergebnis (`wahr` oder `falsch`), den weiteren Weg des Programms.

Aufgabe 6.1 – Gute Bedingung, schlechte Bedingung?

Beurteile die folgenden Alltags-Bedingungen. Schreibe auf, ob es sich um eine gute (eindeutige) Bedingung für ein Programm handelt, und begründe deine Entscheidung.

(a) Ist das Licht im Zimmer an?

(b) Ist das Zimmer schön aufgeräumt?

(c) Ist die Zahl auf dem Würfel eine 6?

(d) Ist es draußen warm?

(e) Ist der Rucksack schwerer als 5 kg?

_____

_____

Aufgabe 6.2 – Deine eigenen Entscheidungen

In der letzten Aufgabe hast du fertige Bedingungen beurteilt. Jetzt bist du dran! Deine Aufgabe ist es, eigene Alltagsentscheidungen zu planen und als Struktogramm, Programmcode oder Textbeschreibung darzustellen.

(a) Eine einseitige Entscheidung

Finde ein Beispiel für eine einseitige Entscheidung (`Falls-Dann`) aus deinem Alltag.

Bedingung (`Falls`): _____

Aktion (`Dann`): _____

Zeichne das vollständige Struktogramm oder den Code für deine Entscheidung.

(b) Eine zweiseitige Entscheidung

Finde nun ein Beispiel für eine zweiseitige Entscheidung (`Falls-Dann-Sonst`) aus deinem Alltag.

Bedingung (`Falls`): _____

Aktion A (`Dann`): _____

Aktion B (`Sonst`): _____

Zeichne das vollständige Struktogramm oder den Code für deine Entscheidung.

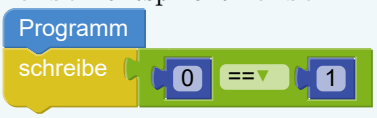
Hinweis

In Blockly nutzen wir für `wahr` und `falsch` die englischen Begriffe:

`wahr` entspricht `true`

Ausgabe deines Programms
1 true

`falsch` entspricht `false`

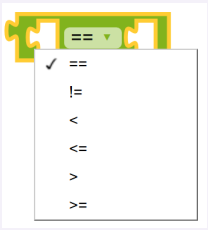


Ausgabe deines Programms
1 false

Info

Bedingungen lassen sich oft durch Vergleiche formulieren – dafür gibt es einen speziellen `Vergleichs`-Baustein. Vergleiche liefern `wahr` oder `falsch`.

Operator	Bedeutung
==	gleich
!=	ungleich
<	kleiner als
<=	kleiner oder gleich
>	größer als
>=	größer oder gleich



Aufgabe 6.3 – Das Geheimwort für den Fuchsbau

Der Eingang zum Fuchsbau wird durch ein kleines Programm geschützt. Wer hinein möchte, muss das Geheimwort eingeben. Das Programm prüft die Eingabe und entscheidet, was geschehen soll.

(a) **Nur im richtigen Fall reagieren**

Zunächst kennt das Programm nur das Geheimwort „FUCHSBAU“:

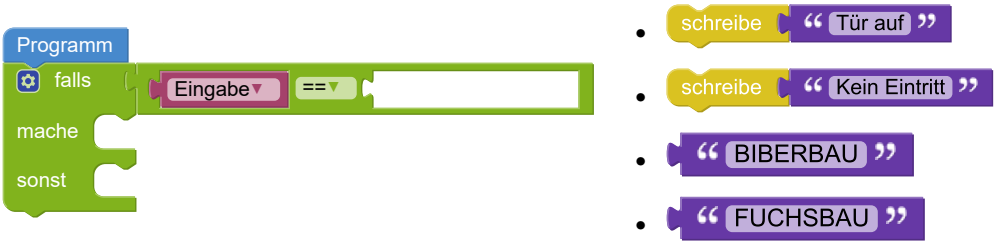
Beschreibe kurz, welche Aktion das Programm ausführt, wenn ein falsches Geheimwort (z.B. „BIBERBAU“) eingegeben wird.

(b) **In jedem Fall reagieren**

(i) Das Programm soll nun immer eine Rückmeldung ausgeben: Lautet die Eingabe „FUCHSBAU“, soll das Programm `"Tür auf"` ausgeben. Lautet die Eingabe anders, soll das Programm `"Kein Eintritt"` ausgeben.

Verbinde die Bausteine rechts mit ihrer korrekten Position im Programm.

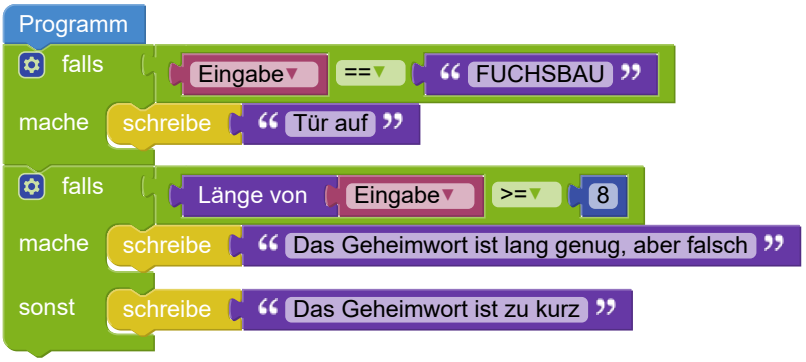
Achtung! Nicht alle Bausteine werden benötigt.



(ii) Jetzt soll das Programm bei jeder Eingabe genau eine der folgenden Rückmeldungen geben:

- Ist das Geheimwort richtig: `"Tür auf"`
- Ist das Geheimwort falsch, aber mindestens acht Zeichen lang: `"Das Geheimwort ist lang genug, aber falsch"`
- Ist das Geheimwort kürzer als acht Zeichen: `"Das Geheimwort ist zu kurz"`

Juno versucht es zunächst mit diesem Programm:



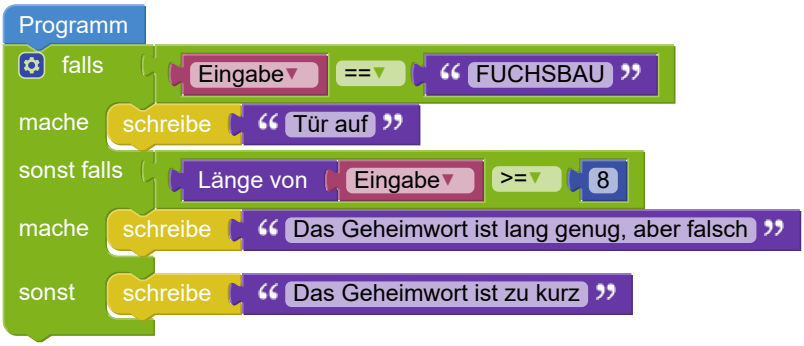
Überlege dir, was Junos Programm für die Eingabe „FUCHSBAU“ ausgibt. Beantworte folgende Fragen:

1. Ist die 1. Bedingung `wahr` oder `falsch`? _____
2. Ist die 2. Bedingung `wahr` oder `falsch`? _____
3. Welche Ausgabe(n) erzeugt das Programm?

   _____

   _____

(iii) Juno verbessert natürlich sofort sein Programm. Er hat das Programm so geändert, dass jetzt die Abfolge der Bedingungen folgende Form hat:



Prüfe mithilfe der Tabelle, ob Junos korrigiertes Programm für die drei Eingaben die gewünschte Ausgabe erzeugt. Trage bei den Bedingungen `wahr` (✓) oder `falsch` (×) ein. Wird eine Bedingung gar nicht mehr überprüft, trage einfach einen Strich ein.

Eingabe	1. Bedingung	2. Bedingung	Ausgabe
`FUCHSBAU`			
`BIBERBAU`			
`FUCHS`			

(c) Erkläre jeweils in einem kurzen Satz (mit je einem praktischen Beispiel), wann du die folgenden Formen verwenden würdest.

`Falls-Dann:`

`Falls-Dann-Sonst:`

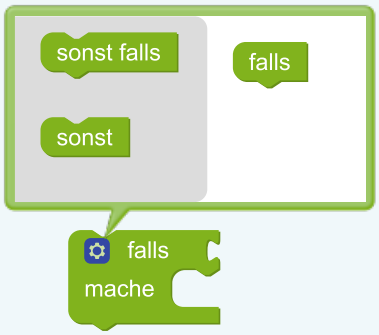
Mehrere Fälle mit **`Sonst Falls:`**

Hinweis

Wenn du auf das blaue Zahnrad an der linken Seite des Bausteins klickst, öffnet sich eine Box:

Das ist wie ein eigener kleiner Programmierbereich nur für die Bedingte Anweisung. Du kannst mehr Fallunterscheidungen und Bedingungen hinzufügen, indem du sie von der grauen Baustein-Palette nach rechts ziehst und unter das 

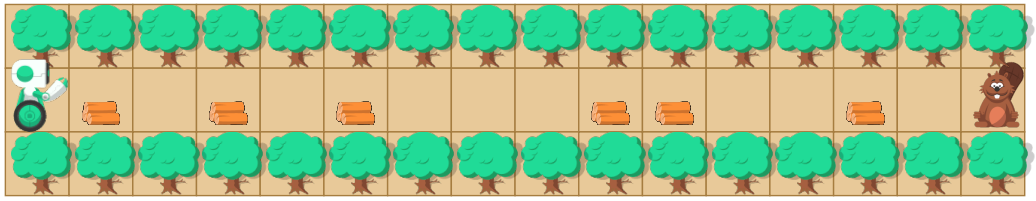
hängst.

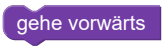


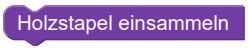
Aufgabe 6.4 – Der fleißige Biber-Helfer

Ein Roboter soll für einen Biber Holz sammeln. Deine Aufgabe ist es, den Roboter so zu programmieren, dass er seine Aufgabe zuverlässig erledigt.

(a) Der Roboter startet am Anfang eines 15 Schritte langen, geraden Waldweges. Am Ende des Weges wartet der Biber. Auf dem Weg können an beliebigen Stellen Holzstücke liegen, der Roboter weiß aber nicht, wo genau sich diese befinden.



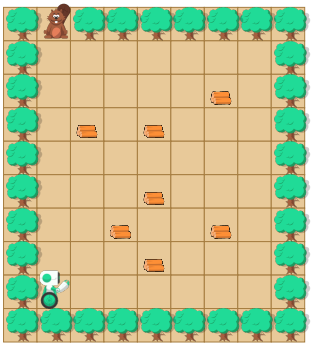




Entwirf das Programm, mit dem der Roboter den gesamten Weg abläuft, jedes gefundene Holzstück aufnimmt und am Ende beim Biber das gesamte gesammelte Holz abgibt.

(b) Die Aufgabe wird schwieriger:

Der Roboter muss nun ein ganzes Gebiet der Größe 8x7 Felder nach Holz absuchen. Der Biber wartet immer in der Ecke zwischen den Bäumen oben links. Beschreibe deine Überlegungen, wie du deine Lösung aus Aufgabe (a) anpassen und erweitern musst, um dieses neue Problem zu lösen. Überlege dir auch, welche Bausteine du zusätzlich zu den vorhandenen noch benötigst.
Tipp: Denke an die Aufgabe 5.3, in der du die verschachtelten Wiederholungen genutzt hast.



(c) Die größte Herausforderung:

Nicht nur das Holz ist zufällig im 8x7 Feld verteilt, sondern auch der Biber kann sich auf irgendeinem der 56 Felder aufhalten. Die Startposition des Roboters in den Testfällen ist immer in der Ecke unten links.

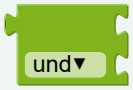
Beschreibe die notwendigen Phasen und die logische Reihenfolge der Aktionen, die der Roboter ausführen muss, um dieses Problem zu lösen.

Exkurs

Super-Bedingungen mit UND, ODER und NICHT

Manchmal reicht eine einzige Bedingung nicht aus, um eine komplexe Entscheidung zu treffen. Stell dir vor, deine Eltern sagen: Du kannst zu deinem Freund gehen, falls du dein Zimmer aufgeräumt hast und deine Hausaufgaben gemacht hast. Hier müssen zwei Bedingungen gleichzeitig erfüllt sein.
Dafür gibt es drei **logische Verknüpfungen**: `UND`, `ODER` und `NICHT`.

Die UND-Verknüpfung (beides muss stimmen)



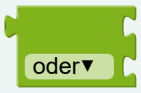
Eine **`UND`-Verknüpfung** ist streng: Sie ist nur dann `wahr`, wenn alle verbundenen Bedingungen `wahr` sind. Fällt auch nur eine einzige Bedingung weg, ist das Gesamtergebnis `falsch`.

Beispiel

Ein Roboter soll nur losfahren, falls sein Akku voll ist `UND` der Weg vor ihm frei ist.

Akku voll?	Weg frei?	Roboter fährt los?
wahr	wahr	wahr
wahr	falsch	falsch
falsch	wahr	falsch
falsch	falsch	falsch

Die ODER-Verknüpfung (eins von beiden reicht)



Die **`ODER`-Verknüpfung** ist viel entspannter: Sie ist schon dann `wahr`, wenn mindestens eine der verbundenen Bedingungen `wahr` ist. Nur wenn absolut keine der Bedingungen zutrifft, ist das Gesamtergebnis `falsch`.

Beispiel

Ein Warnlicht soll leuchten, falls die Temperatur zu hoch ist `ODER` der Druck zu niedrig ist.

Temperatur zu hoch?	Druck zu niedrig?	Licht leuchtet?
wahr	wahr	wahr
wahr	falsch	wahr
falsch	wahr	wahr
falsch	falsch	falsch

Die Verneinung: NICHT (das genaue Gegenteil)

Neben UND und ODER gibt es noch eine dritte logische Operation: die Verneinung. Der NICHT-Operator ist ein kleiner Rebell. Er wird vor eine einzelne Bedingung gesetzt und kehrt deren Ergebnis einfach um. Aus wahr wird falsch und aus falsch wird wahr.

Beispiel

Du darfst draußen spielen, falls es NICHT regnet.

Es regnet?	NICHT Es regnet?	Draußen spielen?
wahr	falsch	falsch
falsch	wahr	wahr

Merke

Mit den **logischen Verknüpfungen UND** (alle Bedingungen müssen wahr sein) und ODER (mindestens eine Bedingung muss wahr sein) werden mehrere Bedingungen kombiniert, während **NICHT** das Ergebnis einer einzelnen Bedingung umkehrt.

Aufgabe 6.5 – wahr oder falsch?

Überprüfe folgende Aussagen. Ist die Aussage wahr, markiere sie mit ✓ . Ist sie falsch, korrigiere sie zu einer wahren Aussage.

(a) Eine UND-Verknüpfung ist wahr, wenn mindestens eine der Bedingungen wahr ist.

(b) Eine ODER-Verknüpfung ist nur dann falsch, wenn alle verbundenen Bedingungen falsch sind.

(c) Der NICHT-Operator verbindet immer zwei Bedingungen miteinander.

Aufgabe 6.6 – Entscheidungen im Alltag

In den folgenden Sätzen fehlt die entscheidende logische Verknüpfung. Setze an der markierten Stelle `UND`, `ODER` oder `NICHT` ein, damit die Aussage logisch sinnvoll ist.

(a) Um ein Fahrradschloss zu öffnen, musst du den richtigen Schlüssel haben _____ den richtigen Zahlencode kennen.

(b) Du bekommst im Kino eine Ermäßigung, falls du ein Schüler bist _____ ein Rentner bist.

(c) Ein automatisches Licht im Flur geht an, falls es dunkel ist _____ eine Bewegung registriert wird.

(d) Die automatische Bildschirmsperre deines Smartphones wird aktiviert, falls der Bildschirm für 30 Sekunden _____ berührt wurde.

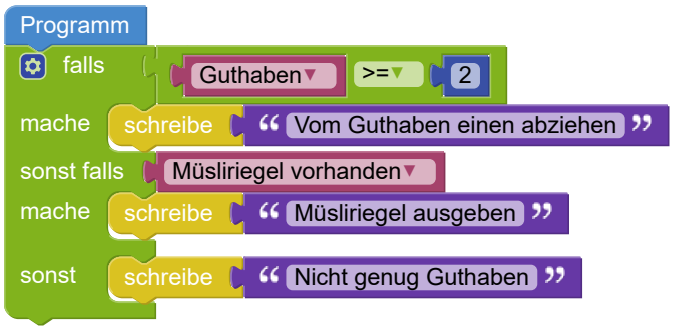
Aufgabe 6.7 – Snack-O-Mat

In der Schule steht ein Snack-Automat, an dem Juna für 2 Euro einen Müsliriegel kaufen kann.

(a) Überlege dir, was ein korrektes Verhalten für den Automaten wäre. Trage in der folgenden Tabelle für jede Situation die richtige Reaktion ein.

Guthaben	**Müsliriegel vorhanden?**	**Snack ausgeben?**
2,50 Euro	`wahr`	☐ `wahr` ☐ `falsch`
2,50 Euro	`falsch`	☐ `wahr` ☐ `falsch`
1 Euro	`wahr`	☐ `wahr` ☐ `falsch`
1 Euro	`falsch`	☐ `wahr` ☐ `falsch`

(b) Juna hat sich überlegt, dass der Automat vermutlich folgendermaßen arbeitet:



Überlege, wie Junas Programm in welcher Situation reagiert:

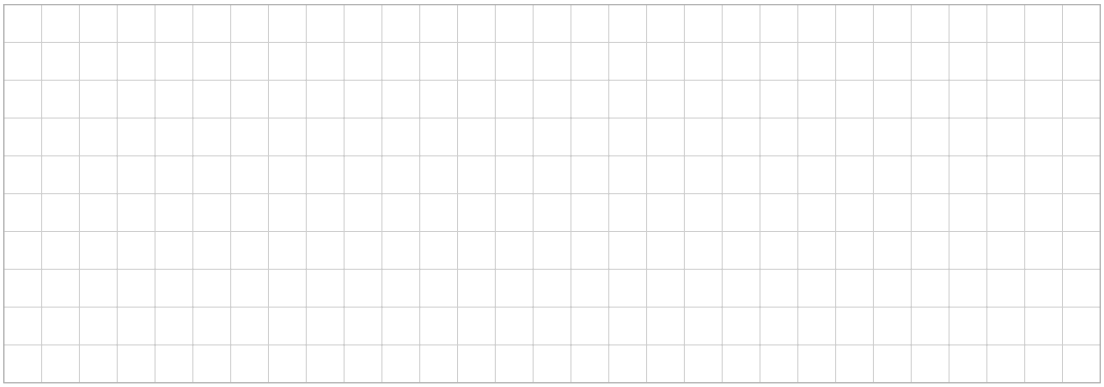
Guthaben	Müsliriegel vorhanden?	Snack ausgeben?
2,50 Euro	`wahr`	☐ `wahr` ☐ `falsch`
2,50 Euro	`falsch`	☐ `wahr` ☐ `falsch`
1 Euro	`wahr`	☐ `wahr` ☐ `falsch`
1 Euro	`falsch`	☐ `wahr` ☐ `falsch`

Erkläre, warum diese Lösung nicht richtig sein kann. Unter welcher Voraussetzung wird die Bedingung `"Müsliriegel vorhanden"` überhaupt erst geprüft?

(c) Formuliere nun die korrekte Regel, die du in der Tabelle von Aufgabe (a) festgelegt hast. Vervollständige den Satz:

Der Snack-O-Mat gibt genau dann einen Müsliriegel aus, falls ...

(d) Entwirf ein korrektes Programm für den Snack-O-Mat. Verwende dafür genau eine `Falls-Dann-Sonst`-Anweisung.



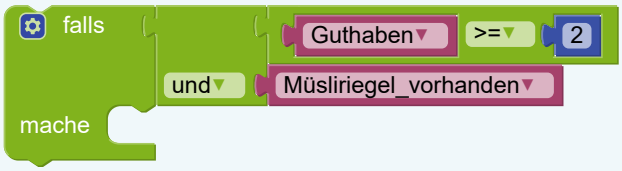
(e) Überprüfe dein Programm und vergleiche mit den Anforderungen aus (a):

Guthaben	Müsliriegel vorhanden?	Snack ausgeben?
2,50 Euro	`wahr`	☐ `wahr` ☐ `falsch`
2,50 Euro	`falsch`	☐ `wahr` ☐ `falsch`
1 Euro	`wahr`	☐ `wahr` ☐ `falsch`
1 Euro	`falsch`	☐ `wahr` ☐ `falsch`

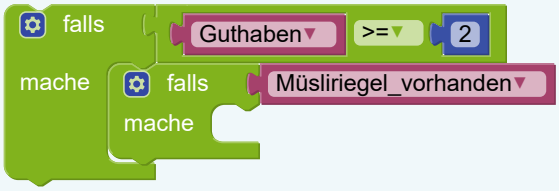
Hinweis

Eine gemeinsame oder mehrere aufeinanderfolgende Entscheidungen?

Beim Snack-O-Mat müssen zwei Bedingungen erfüllt sein, damit die entsprechende Handlung ausgeführt wird: Das Guthaben muss ausreichen **und** ein Müsliriegel muss vorhanden sein. Deshalb lassen sich beide Bedingungen übersichtlich mit `UND` zu einer gemeinsamen Bedingung verbinden.



Man könnte aber auch schrittweise vorgehen: Zuerst wird das Guthaben geprüft. Nur wenn es ausreicht, wird anschließend geprüft, ob ein Müsliriegel vorhanden ist. Dafür setzt man eine `FALLS`-Anweisung in den Dann-Bereich einer anderen `FALLS`-Anweisung. Das nennt man eine **verschachtelte bedingte Anweisung**.



Mit `UND` oder `ODER` verbindest du mehrere Bedingungen zu **einer Entscheidung**.

Verschachtelte `FALLS`-Anweisungen verwendest du, wenn nach einer Entscheidung nur in einem bestimmten Fall **weiter geprüft und entschieden** werden soll.

Beim Snack-O-Mat führen beide Möglichkeiten zum gleichen Ergebnis. Bei umfangreicheren Entscheidungen kann eine Verschachtelung jedoch unterschiedliche weitere Prüfungen und Handlungen enthalten.

7 Wiederholungen mit Bedingung: Programme, die wissen, wann Schluss ist

Du bist jetzt ein Profi darin, Anweisungen eine feste Anzahl von Malen zu wiederholen (Kapitel 4 und 5). Aber was ist, wenn du vorher gar nicht weißt, wie oft etwas wiederholt werden muss? Stell dir einen Roboter vor, der eine Murmel in einem Raum finden soll. Du kannst ihm nicht sagen: `Mache 10 Schritte`. Vielleicht liegt die Murmel schon nach 3 Schritten vor ihm, vielleicht braucht er aber auch 30.

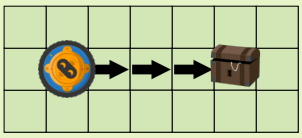
Für genau solche Fälle, in denen die Anzahl der Wiederholungen ungewiss ist, gibt es die **bedingte Wiederholung**. Sie verbindet die Macht der Wiederholung mit der Intelligenz der Bedingung, die du in Kapitel 6 kennengelernt hast. Eine bedingte Wiederholung fragt nicht „Wie oft“, sondern sie fragt vor jedem Durchlauf: „Soll ich überhaupt noch weitermachen?“. In unserer Programmierumgebung gibt es dafür zwei sehr nützliche Varianten.

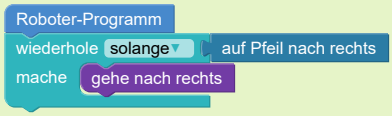
Die `Wiederhole solange`-Anweisung

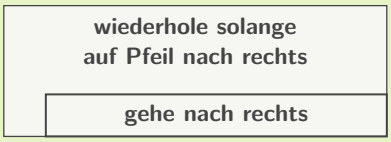
Diese Anweisung wiederholt eine Sequenz, **solange** eine bestimmte Bedingung **`wahr`** ist. Vor jedem Durchlauf wird geprüft: Ist die Bedingung noch `wahr`? Wenn ja, wird die Sequenz ausgeführt. Wenn nein, wird die Sequenz übersprungen und das Programm macht nach der Wiederholung weiter.

Beispiel

Der Roboter soll sich nach rechts bewegen, solange er sich auf dem Pfeil nach rechts befindet. Sobald er nicht mehr auf dem Pfeil nach rechts steht (die Bedingung wird `falsch`), hört der Roboter auf, sich zu bewegen.





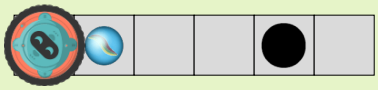


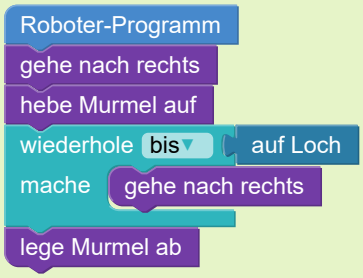
Die `Wiederhole bis`-Anweisung

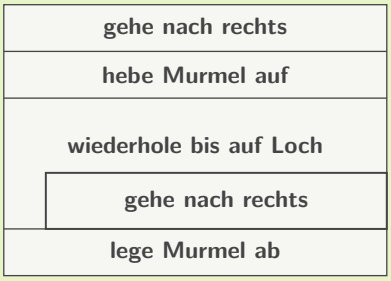
Diese Anweisung funktioniert genau umgekehrt zur `Wiederhole solange`-Anweisung. Sie wiederholt eine Sequenz, **bis** eine bestimmte Bedingung endlich **`wahr`** wird. Sie läuft also, solange die Bedingung noch `falsch` ist, und hört auf, sobald das gewünschte Ziel erreicht ist.

Beispiel

Der Roboter soll gehen, bis er sich auf einem Loch befindet. Er macht Schritt für Schritt (während die Bedingung noch `falsch` ist). Sobald er das Loch erreicht hat (die Bedingung wird `wahr`), stoppt er.







Der Roboter soll gehen, bis er sich auf einem Loch befindet. Er macht Schritt für Schritt (während die Bedingung noch `falsch` ist). Sobald er das Loch erreicht hat (die Bedingung wird `wahr`), stoppt er.

`Wiederhole solange`: Die Aktion läuft, während ein Zustand anhält.
`Wiederhole bis`: Die Aktion läuft, um einen Zielzustand zu erreichen.

Mit diesen intelligenten Wiederholungen werden deine Programme noch vielseitiger und können auf unterschiedlichste Situationen reagieren.

Merke

Eine **bedingte Wiederholung** führt eine Sequenz so lange aus, wie eine Bedingung erfüllt ist (oder bis sie erfüllt ist), wobei die Anzahl der Durchläufe nicht vorab feststeht sondern von der Bedingung abhängt.

Aufgabe 7.1 – Solange oder Bis?

Entscheide für jedes Szenario: Ist **`Wiederhole solange`** oder **`Wiederhole bis`** besser geeignet? Begründe deine Wahl kurz.

(a) Ein Roboter soll eine Pflanze gießen. (Bedingung: `erdeIstFeucht` / Anweisung: `gießeWasser`)

(b) Du liest ein Buch. (Bedingung: `duBistNichtMüde` / Anweisung: `leseEineSeite`)

(c) Ein Roboter poliert ein Auto. (Bedingung: `autoGlänzt` / Anweisung: `poliereEinStück`)

(d) Ein selbstfahrendes Auto hält Abstand. (Bedingung: `FahrzeugIstVorne` / Anweisung: `halteAbstand`)

Aufgabe 7.2 – Junos Begrüßungsautomat 2.0

Erinnerst du dich an den Begrüßungsautomaten aus Kapitel 4? Dort wusste Juno genau, dass vier Freunde zu Besuch kommen. Das Programm hat eine Sequenz viermal wiederholt.

Jetzt ist die Situation anders: Juno veranstaltet regelmäßig Spieleabende, aber er weiß nie genau, wie viele seiner Freunde Zeit haben. Die Namen stehen wieder untereinander im Eingabefeld, aber die Anzahl ändert sich jedes Mal. Sein altes Programm funktioniert also nicht mehr.

Juno braucht ein Programm, das so lange läuft, bis alle Namen gelesen wurden. Dafür gibt es einen neuen `Sensor`-Baustein: 

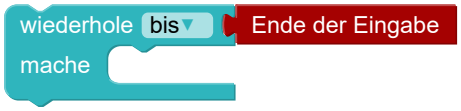
Dieser Sensor gibt `wahr` zurück, wenn das Ende des Eingabefeldes erreicht ist, und `falsch`, solange noch Text vorhanden ist.

(a) Juno überlegt, wie er den Sensor mit einer Wiederholungsanweisung kombinieren kann. Analysiere Junos erste beiden Ideen.

Idee 1:

Beschreibe den Programmablauf, wenn Juno diese Wiederholungsanweisung verwendet. Begründe, ob in diesem Fall überhaupt ein Name aus dem Eingabefeld gelesen wird.

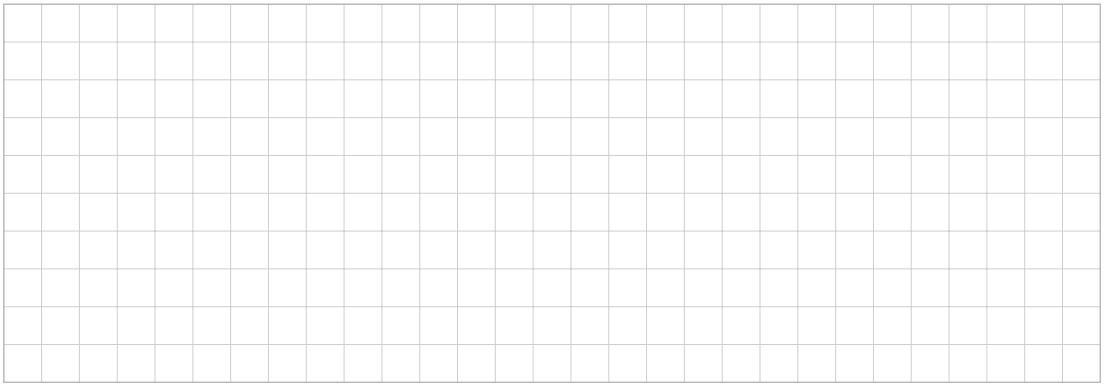
Idee 2:



Beurteile, ob diese Wiederholungsanweisung das Problem korrekt löst. Erläutere dazu genau, unter welcher Bedingung die Wiederholung angehalten wird.

Eine der Ideen funktioniert nicht wie gewünscht. Entwickle einen Lösungsvorschlag, indem du die fehlerhafte Idee mit einem Baustein aus Kapitel 6 (Exkurs) so umformulierst, dass sie korrekt funktioniert.

(b) Entwirf nun das neue, flexible Programm für den Begrüßungsautomaten. Verwende dafür eine der funktionierenden Wiederholungsanweisungen aus Aufgabe (a). Stelle deinen Entwurf in Form eines Struktogramms oder einer Textbeschreibung dar.

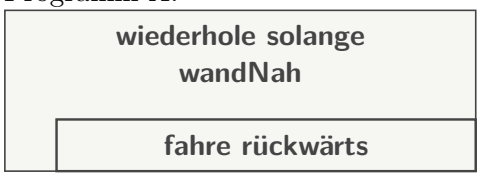


Aufgabe 7.3 – Der Park-Assistent

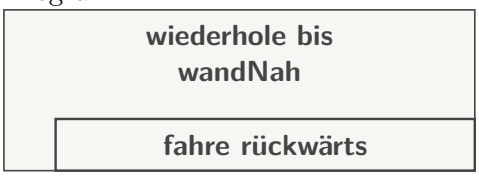
Ein selbstfahrendes Auto soll rückwärts in eine Parklücke setzen, bis es nah an der Wand ist. Es hat einen Sensor `WandNah`, der `wahr` meldet, sobald das Auto nur noch 20 cm von der Wand entfernt ist.

Analysiere die beiden folgenden Programme. Entscheide, welches Programm das Auto sicher einparkt und welches einen Unfall verursachen könnte. Begründe deine Entscheidung.

Programm A:

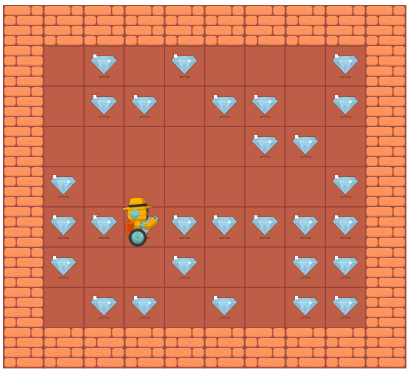


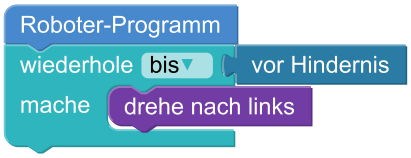
Programm B:



Aufgabe 7.4 – Der Code-Inspektor

Bedingte Wiederholungen können tückisch sein: Der Computer prüft die Bedingung stur vor jedem Durchlauf. Wenn die Anweisung innerhalb der Wiederholungsanweisung aber nichts daran ändert, dass die Bedingung `wahr` oder `falsch` wird, hört der Computer niemals auf. Man nennt das eine **Endlosschleife**.
Analysiere dieses Programm eines Roboters, der die Diamanten einsammeln soll:





Erläutere, warum die Aktion des Roboters innerhalb der Wiederholungsanweisung niemals dazu führen kann, dass die Bedingung `vor Hindernis wahr` wird. Warum wird der Roboter also niemals anhalten? **Tipp**: Überlege, ob die Aktion `drehe nach links` dazu führt, dass der Roboter näher an die Wand herankommt.

Korrigiere den Fehler. Welche Anweisung müsste statt `drehe nach links` in der Wiederholung stehen, damit der Roboter zur Wand gelangt und die Wiederholung erfolgreich beendet?

Info

Manchmal passiert es, dass man eine Bedingung nicht richtig formuliert hat und der Roboter oder das Programm in eine Endlosschleife gerät. Das merkst du daran, dass der Knopf zur Ausführung des Programms durch eine Grafik aus rotierenden Punkten ersetzt wird:

Keine Panik! Du kannst das Programm jederzeit abbrechen. Klicke dazu einfach auf den Zurücksetzen-Pfeil, den du links neben dem Startknopf findest.

8 Variablen: Das Gedächtnis deines Programms

Bisher waren deine Programme zwar schon ziemlich clever, aber auch ein wenig vergesslich. Sie konnten Anweisungen und Wiederholungen ausführen, sich aber keine Ergebnisse für später merken. Das ändern wir jetzt! Denn erst mit Variablen werden deine Algorithmen richtig flexibel und allgemeingültig.

Stell dir vor, du hast eine beschriftete Kiste, in die du eine Information legen kannst – z.B. eine Zahl oder ein Wort. Du kannst jederzeit nachsehen, was in der Kiste ist, den Inhalt verändern oder durch etwas Neues ersetzen. Genau so eine Kiste ist eine **Variable**. Sie ist ein Speicherplatz mit einem festgelegten Namen im Gedächtnis des Computers. Der Name Variable kommt dabei daher, dass sich ihr Inhalt während des Programmablaufs verändern kann – er ist also variabel.

Einer Variable einen Wert zuweisen

Um einer Variable einen Wert zu geben, benutzt du den Baustein 

. Der Name der Variable ist dabei die Beschriftung deiner Kiste. Der Wert, den du ihr gibst, kann dabei unterschiedlicher Art sein:

Eine Zahl: 

Ein Text: 

Den Wert einer Variable verändern

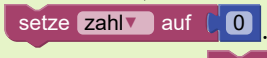
Wenn du den Wert einer Variable, die eine Zahl enthält, verändern möchtest, gibt es einen besonders praktischen Baustein: 

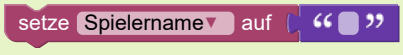
. Anstatt kompliziert zu rechnen, kannst du damit ganz einfach eine Zahl zum aktuellen Wert hinzufügen. Das ist perfekt, um z.B. einen Punktestand zu erhöhen oder mitzuzählen, wie oft eine Wiederholung schon durchgelaufen ist.

Der saubere Start: Initialisierung

Bevor du in einem Programm zum ersten Mal mit einer Variable rechnest oder Text zusammensetzt, musst du sicherstellen, dass sie einen bestimmten Startwert hat. Man kann nicht zu einer unbekannten Zahl etwas hinzufügen! Diesen ersten Schritt nennt man Initialisierung.

Beispiel

Eine Variable, mit der du zählen oder rechnen willst, setzt du am Anfang meist auf 0: 

. Eine Variable, in der du Text sammeln willst, setzt du am Anfang auf einen leeren Text: 

Merke

Eine **Variable** ist ein benannter Speicherplatz für einen Wert. Dieser Wert kann während des Programmablaufs ausgelesen, verändert, gespeichert und im Verlauf des Programms genutzt werden.

Aufgabe 8.1 – Was ist in der Kiste?

Stell dir vor, du hast eine Variable mit dem Namen `Zähler`. Die folgende Tabelle zeigt dir ein kurzes Programm.

(a) Analysiere das Programm Zeile für Zeile und vervollständige die Tabelle, indem du einträgst, welcher Wert nach jeder Anweisung in der „Kiste“ (Variable) liegt.

Schritt	Anweisung	Wert der Variable Zähler
1.	`Setze Zähler auf 0`	0
2.	`Erhöhe Zähler um 5`	
3.	`Erhöhe Zähler um 2`	
4.	`Setze Zähler auf 10`	
5.	`Erhöhe Zähler um -3`	
6.	`Erhöhe Zähler um 1`	

(b) Wie du in der Tabelle gesehen hast, verhalten sich die Anweisungen im Schritt 3 und Schritt 4 unterschiedlich.

Vergleiche die Wirkung der Bausteine `Setze Zähler auf...` und `Erhöhe Zähler um...` auf den Inhalt der Variable. Erläutere dabei insbesondere, was mit dem alten Wert der Variable passiert, wenn der Baustein `Setze Zähler auf...` ausgeführt wird.

Aufgabe 8.2 – Junos Begrüßungsautomat 3.0

Juno möchte seinen Begrüßungsautomaten weiterentwickeln. Statt vieler einzelner Begrüßungen soll er am Ende einen gemeinsamen Satz ausgeben.

Beispiel: Statt `"Hallo Juna! Hallo Peter!"` soll am Ende stehen: `"Hallo Juna Peter!"`. Dafür nutzt der Automat eine Variable mit dem Namen `Begrüßung`, in der `"Hallo "` und anschließend alle gelesenen Namen gesammelt werden.

(a) Juno hat sich schon mal überlegt, welche Schritte der Automat ausführen muss.

Bringe die Schritte in die richtige logische Reihenfolge, indem du die Zahlen 1 bis 5 einträgst.

☐ Wiederhole solange das Ende des Eingabefeldes noch nicht erreicht ist.

☐ Gib den fertigen Satz (Inhalt der Variable Begrüßung) aus.

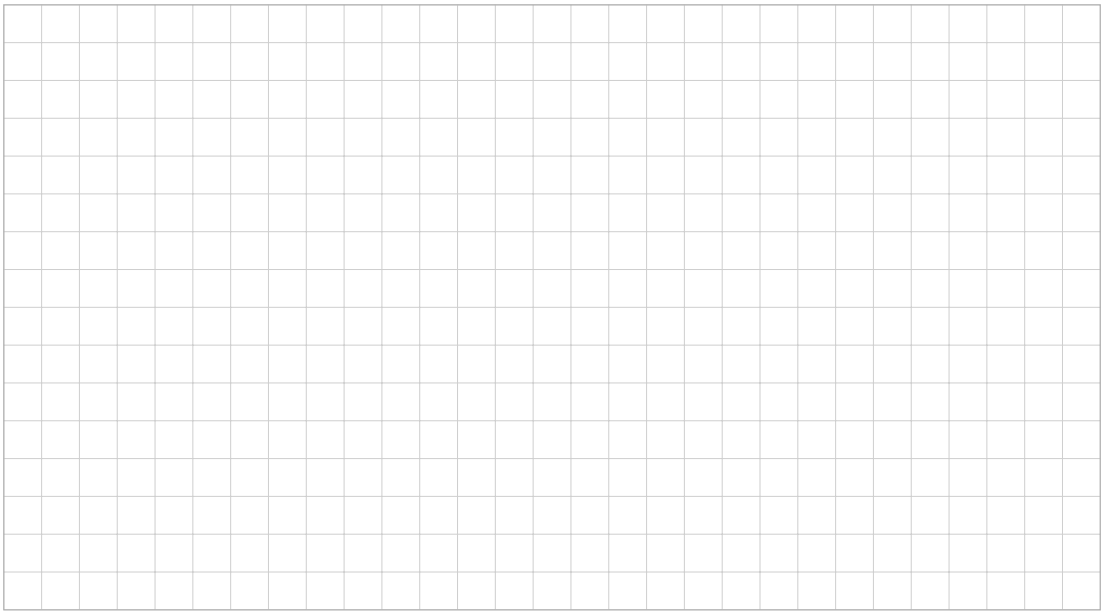
☐ Initialisiere die Variable `Begrüßung` mit dem Text `"Hallo "`.

☐ Lies den nächsten Namen aus dem Eingabefeld.

☐ Aktualisiere die Variable `Begrüßung`: Verbinde den bisherigen Inhalt mit dem neuen Namen und einem Leerzeichen `" "`.

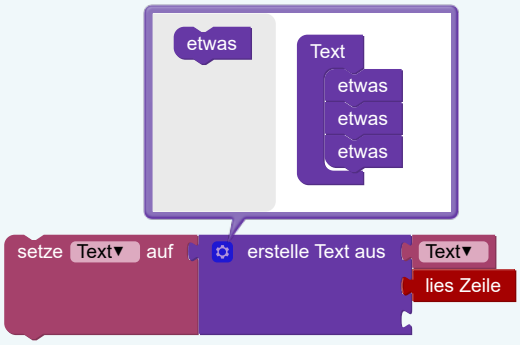
(b) Beurteile, an welcher Stelle im Programm der inlinebildimages/chapter2/print.pdf-Baustein stehen muss. Begründe deine Entscheidung kurz.

(c) Entwirf das vollständige Programm als Struktogramm, Programmcode oder Textbeschreibung. **Hinweis**: Der Automat muss den Namen zuerst lesen, bevor er ihn an `Begrüßung` anhängt.



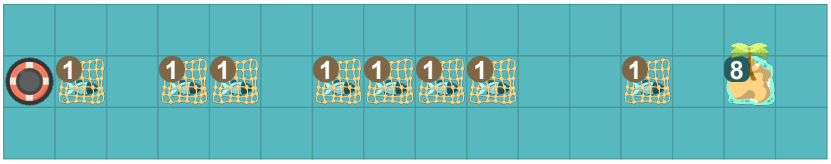
Hinweis

Wie du schon beim `Falls-Dann`-Baustein im Kapitel 6 gesehen hast, kannst du mit dem Zahnrad die Struktur eines Bausteins verändern.
Das funktioniert auch bei diesem Baustein (und überall, wo der Baustein ein Zahnrad hat)!

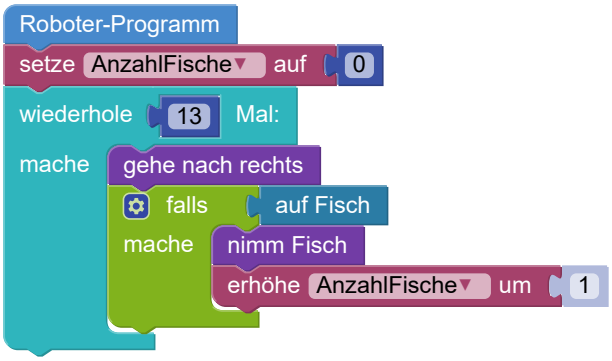


Aufgabe 8.3 – Die Fisch-Mission

Ein Roboter sammelt auf dem Wasser Fische aus Netzen ein. Damit die Fischer wissen, wie groß der Fang war, zählt der Roboter unterwegs mit einer Variable.



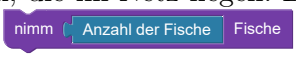
(a) Analysiere den ersten Programmteil. Erkläre den Ablauf in eigenen Worten. Überlege dabei, welche Rolle die Variable `AnzahlFische` spielt und unter welcher Bedingung sie verändert wird.



(b) Entwirf den zweiten Programmteil für die Ankunft auf der Insel: Der Roboter soll genau so viele Fische ablegen, wie er gezählt hat. Überlege, welche Wiederholungsart hier geeignet ist. Begründe kurz. (**Tipp**: Weiß der Roboter zu diesem Zeitpunkt schon genau, wie oft er die Aktion 

ausführen muss?)

(c) Erweiterung: In einem Netz können mehrere Fische liegen. Der Roboter hat dafür einen neuen `Sensor`-Baustein: 

. Dieser Baustein gibt dir direkt die Anzahl der Fische zurück, die im Netz liegen. Du kannst ihn auch in Verbindung mit diesem Baustein nutzen: 

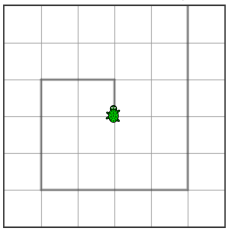
Beurteile die notwendigen Änderungen im ersten Teil des Programms aus (a). Erläutere, welchen Baustein du nun verwenden musst, um die Variable `AnzahlFische` korrekt zu aktualisieren, wenn sich mehrere Fische im Netz befinden.

Analysiere, ob der zweite Teil des Programms (Ablegen auf der Insel) an diese neue Situation angepasst werden muss. Begründe deine Entscheidung unter Berücksichtigung der Variable `AnzahlFische`.

(d) Entwirf das vollständige Programm für die neue Situation als Struktogramm, Programmcode oder Textbeschreibung.

Aufgabe 8.4 – Die wachsende Spirale

Die Schildkröte soll eine eckige Spirale (Schnecke) zeichnen.



(a) Analysiere die Zeichnung oben in der Grafik und vervollständige die folgenden Regeln, an die sich die Schildkröte halten muss.

Die Schildkröte zeichnet insgesamt _____ gerade Linien.

Nach jeder Linie dreht sie sich um _____.

Jede neue Linie muss um _____ als die vorherige.

(b) Folgende Bausteine stehen der Schildkröte zur Verfügung:

Wähle die passenden Bausteine aus und setze sie zu einem funktionierenden Programm zusammen. Schreibe oder zeichne das vollständige Programm in der richtigen Reihenfolge.

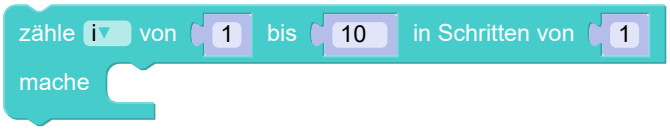
(c) Erkläre in einem Satz, warum für diese Aufgabe eine Variable unverzichtbar ist. Was wäre das Ergebnis, wenn du statt der Variable `anzahlSchritte` immer einen festen Wert (z.B. `Gehe 10 Schritte`) verwenden würdest?

_____

_____

9 Die mitzählende Wiederholung: Zählen und Wiederholen in einem Schritt

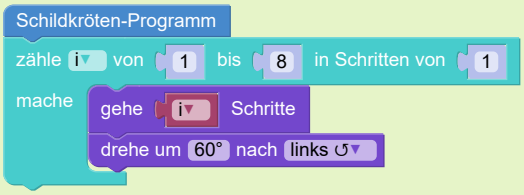
In Kapitel 8 hast du gelernt, wie du eine Variable nutzt, um innerhalb einer Wiederholung selbst mitzuzählen. Das funktioniert super, erfordert aber zwei Schritte: eine eigene Variable anlegen und sie in jedem Durchlauf manuell erhöhen. Für diesen häufigen Fall gibt es eine elegantere Abkürzung: die **mitzählende Wiederholung** (auch for-Schleife genannt).



`zähle i`	Das `i` ist eine besondere Zählvariable, die automatisch erstellt und verwaltet wird.
`von 1`	Das ist der Startwert. Hier legst du fest, bei welcher Zahl `i` im ersten Durchlauf beginnen soll.
`bis 10`	Das ist der Endwert. Die Wiederholung läuft so lange, bis die Zählvariable diesen Wert erreicht hat. Wichtig dabei ist: Der Durchlauf, in dem die Zählvariable genau den Endwert hat, wird ebenfalls noch ausgeführt.
`in Schritten von 1`	Das ist die Schrittweite. Hiermit bestimmst du, um wie viel die Zählvariable `i` nach jedem Durchlauf erhöht werden soll. Du kannst hier auch in 2er-Schritten zählen oder sogar rückwärts, indem du eine negative Zahl eingibst.

Beispiel

Die Wiederholung startet: `i` hat den Wert 1. Die Anweisungen im Inneren werden ausgeführt. **Nächster Durchlauf:** `i` hat jetzt automatisch den Wert 2. Die Anweisungen werden wieder ausgeführt. **...und so** weiter, bis zum letzten Durchlauf, in dem `i` den Wert 8 hat. Danach ist die Wiederholung beendet.



Der riesige Vorteil ist, dass du dich nicht mehr selbst um das Erstellen und Erhöhen einer Zählvariable kümmern musst. Der eigentliche Clou ist aber, dass du die Zählvariable `i` innerhalb der Wiederholungsanweisung für deine Anweisungen nutzen kannst. So kannst du Programme schreiben, deren Verhalten sich mit jedem Durchlauf anpasst – z.B., indem du eine Figur immer genau `i` Schritte weit bewegst.

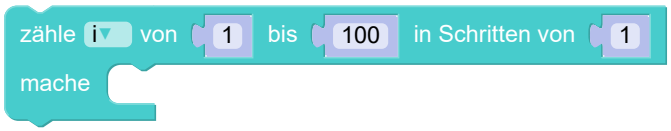
Merke

Die **mitzählende Wiederholung** (auch `for-Schleife` genannt) ist eine spezielle Form der Wiederholung, die eine Zählvariable automatisch von einem definierten Startwert bis zu einem Endwert in einer festgelegten Schrittweite durchlaufen lässt.

Aufgabe 9.1 – Der Fachbegriff-Checker

Markiere im Baustein die vier Teile der mitzählenden Wiederholung und verbinde sie mit den Begriffen:

Zählvariable Schrittweite Startwert Endwert



Aufgabe 9.2 – Zahlenreihen vorhersagen

Die mitzählende Wiederholung erzeugt bei jedem Durchlauf eine neue Zahl für die Zählvariable `i`. Gib für die folgenden Wiederholungsanweisungen an, welche Zahlen `i` nacheinander annehmen würde.

(a) `zähle i von 1 bis 5 in Schritten von 1`

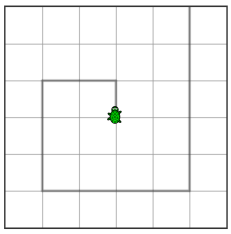
(b) `zähle i von 0 bis 8 in Schritten von 2`

(c) `zähle i von 10 bis 7 in Schritten von -1`

(d) `zähle i von 5 bis 20 in Schritten von 5`

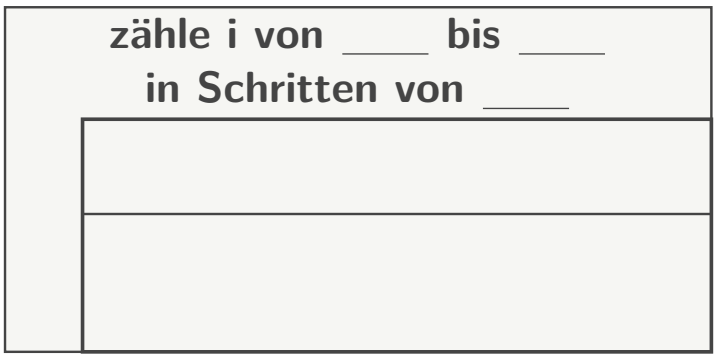
Aufgabe 9.3 – Die wachsende Spirale 2.0

Erinnerst du dich an die wachsende Spirale aus dem letzten Kapitel 8? Die bisherige Lösung funktioniert, ist aber etwas umständlich. Die alte Lösung sah so aus:



(a) Streiche in der alten Lösung alle Bausteine, die durch die mitzählende Wiederholung ersetzt werden.

(b) Vervollständige das neue Programm im Struktogramm: Nutze die Zählvariable `i` an der richtigen Stelle.

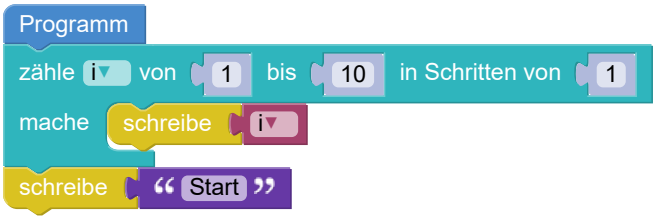


Aufgabe 9.4 – Der Code-Inspektor

In den folgenden zwei Programmen haben sich Fehler eingeschlichen. Das gewünschte Ergebnis wird nicht erreicht.

(a) Raketen-Countdown:

Es soll von 10 bis 1 heruntergezählt und am Ende `"Start "` ausgegeben werden.

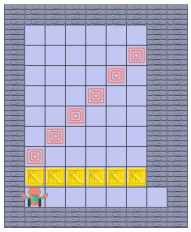


Analysiere: Was ist das Problem bei diesem Programm? Beschreibe den Fehler.

Korrigiere den fehlerhaften Baustein.

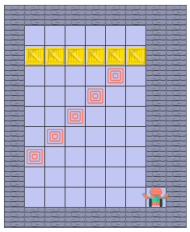
(b) Schieben der Kisten:

Der Roboter soll die 6 Kisten auf die markierten Positionen im Raum bringen.



Ausgangssituation

Programm



Nach der Ausführung

Analysiere: Was ist das Problem bei diesem Programm? Beschreibe den Fehler.

Korrigiere den fehlerhaften Baustein.

Aufgabe 9.5 – Der Muster-Meister

Die mitzählende Wiederholung ist auch perfekt geeignet, um mathematische Reihen und Muster zu erzeugen. Deine Aufgabe ist es, ein Programm zu entwerfen, das die 7er-Reihe des Einmaleins ausgibt.

Gewünschtes Ergebnis auf dem Bildschirm:

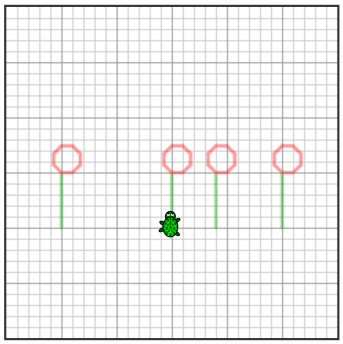
```
7 x 1 = 7
7 x 2 = 14
7 x 3 = 21
...
7 x 10 = 70
```

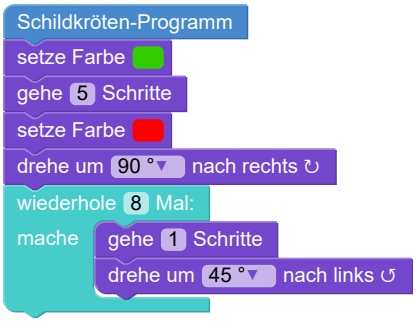
Entwirf ein vollständiges Struktogramm oder einen Programmcode, das bzw. der genau diese Ausgabe erzeugt.

- **Tipp 1**: Nutze eine mitzählende Wiederholung von 1 bis 10.
- **Tipp 2**: Baue die Ausgabe mit `i` und dem Produkt 7 x `i` zusammen.

10 Funktionen: Baue deine eigenen Bausteine

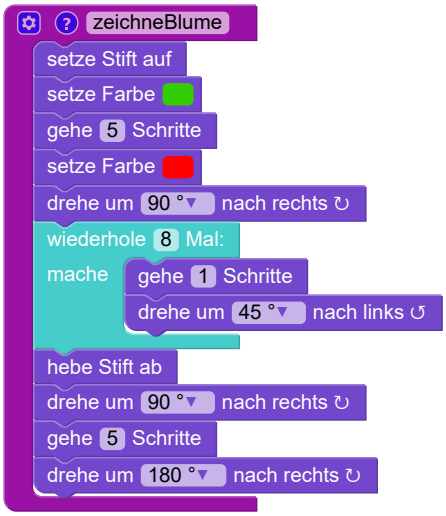
Stell dir vor, du willst vier gleiche Blumen zeichnen, die an unterschiedlichen Positionen auf der Zeichenfläche verteilt sind. Du müsstest die Sequenz zum Zeichnen einer Blume viermal kopieren und zwischen den Kopien die Turtle neu positionieren. Dein Programm würde dadurch lang, unübersichtlich und bei jeder kleinen Änderung müsstest du an vier Stellen arbeiten.





An dieser Stelle kommt ein zentrales Konzept der Informatik ins Spiel: die **Funktion** (manchmal auch **Methode** genannt).

Eine Funktion ist eine neue Fähigkeit, die du dem Computer beibringst und unter einem Namen speicherst. Du definierst einmal, wie eine Blume gezeichnet wird, und nennst diese Anleitung z.B. `zeichneBlume`. Dadurch erstellst du deinen eigenen, neuen Baustein. Statt vieler einzelner Schritte nimmst du dann einfach deinen `zeichneBlume`-Baustein.



Dies bringt mehrere Vorteile mit sich:

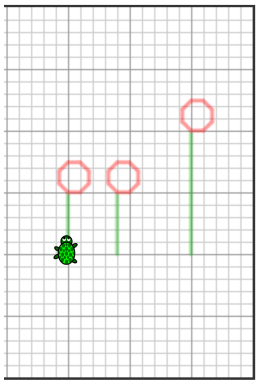
- **Wiederverwendbarkeit**: Einmal definiert, kann eine Funktion beliebig oft genutzt werden.
- **Übersichtlichkeit**: Dein Hauptprogramm beschreibt, was passiert (`zeichneBlume`), nicht wie.
- **Einfache Wartung**: Wenn du die Farbe der Blume ändern willst, musst du das nur an einer einzigen Stelle tun: in der Definition deiner Funktion. Alle mit dieser Funktion gezeichneten Blumen haben dann die neue Farbe.

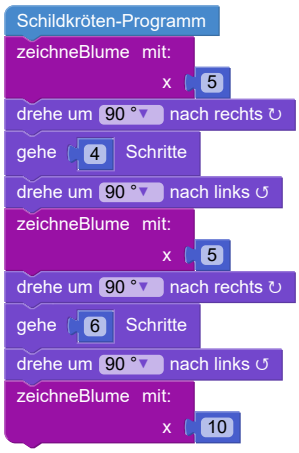
Funktionen mit Parametern – Für mehr Flexibilität

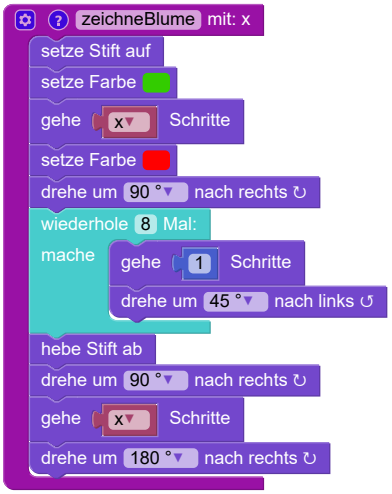
Was aber, wenn die Blumen nicht alle gleich aussehen sollen? Vielleicht sollen sie unterschiedlich lange Stiele haben. Auch dafür gibt es eine Lösung: Du kannst deiner Funktion beim Aufruf eine Information mitgeben. Diese Information nennt man **Parameter**. Du könntest deine Funktion so definieren, dass sie einen Wert für die Stiellänge `x` erwartet, z.B. `zeichneBlume(x)`.

Wenn du sie dann aufrufst, gibst du einfach die gewünschte Länge als Zahl mit:

- `zeichneBlume(10)` malt eine Blume mit einem langen Stiel.
- `zeichneBlume(5)` malt eine Blume mit einem kurzen Stiel.







Die Funktion nutzt dann den übergebenen Wert (den Parameter), um den Stiel in der passenden Länge zu zeichnen. So werden deine selbstgebauten Bausteine nicht nur Abkürzungen, sondern flexible Werkzeuge.

Mit Funktionen bist du jetzt nicht mehr nur wie bisher ein Anwender von Bausteinen – du bist ein Erschaffer von neuen, eigenen Bausteinen.

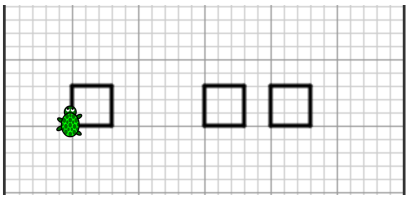
Merke

Eine **Funktion** (oder **Methode**) gibt einem Objekt (wie dem Roboter oder der Turtle) eine neue Fähigkeit. Sie ist ein wiederverwendbarer **Teil-Algorithmus**, der unter einem eigenen Namen eine Teilaufgabe löst und durch die Übergabe von **Parametern** flexibel gesteuert werden kann.

Aufgabe 10.1 – Die Quadrat-Reihe 2.0

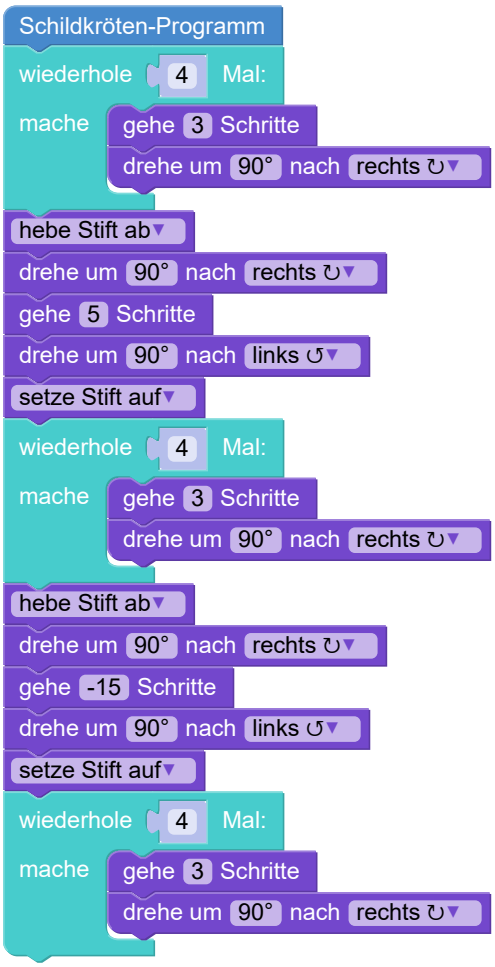
Im Kapitel 5 hast du gelernt, mit einer verschachtelten Wiederholung ganz einfach eine Reihe von vier Quadraten zu zeichnen. Das Programm war kurz und elegant, weil sich die Position der Quadrate nach einem einfachen Muster wiederholt hat.

Aber was passiert, wenn die Quadrate nicht mehr in einer sauberen Reihe stehen, sondern an ganz unterschiedlichen Positionen auf dem Bildschirm erscheinen sollen? Hier hilft uns eine einfache Wiederholung nicht mehr weiter, da die Position nach jedem Quadrat individuell geändert werden muss.

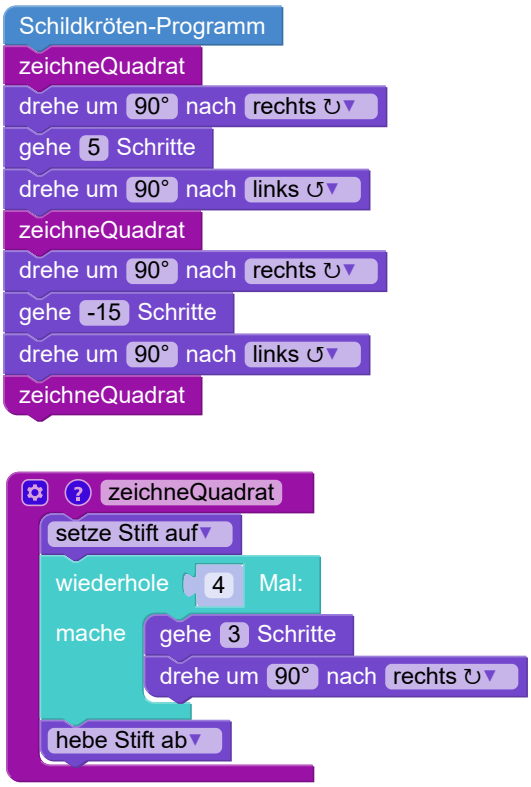


Analysiere die beiden folgenden Programme, die dasselbe Ergebnis erzeugen: drei Quadrate an unterschiedlichen Positionen.

Programm A:



Programm B:



Beantworte die folgenden Fragen.

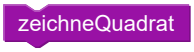
(a) In welchem Programm (A oder B) ist der Zweck des Programms auf den ersten Blick klarer? Begründe deine Wahl.

(b) Stell dir vor, du möchtest die Seitenlänge aller Quadrate auf 10 ändern. Zähle, wie viele Änderungen du in Programm A und wie viele du in Programm B machen müsstest.

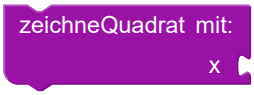
Programm A: _____ Änderungen Programm B: _____ Änderungen

(c) Erkläre, warum die Funktion im Programm B hier die bessere Methode ist als eine verschachtelte Wiederholung aus Kapitel 5.

Aufgabe 10.2 – Das flexible Quadrat

Eine Funktion 

ist nützlich, aber was, wenn man Quadrate unterschiedlicher Größe braucht?
Erweitere eine einfache Quadrat-Funktion, sodass sie einen Parameter für die Seitenlänge `x` akzeptiert.



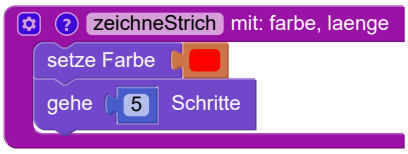
(a) Zeichne das Programm oder das Struktogramm für die Funktion `zeichneQuadrat(x)`: Die Funktion soll die übergebene Seitenlänge `x` verwenden, um die vier Seiten des Quadrats zu zeichnen.

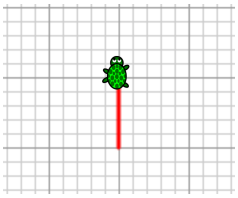
(b) Schreibe den Aufruf für ein kleines Quadrat mit der Seitenlänge 3.

(c) Schreibe den Aufruf für ein großes Quadrat mit der Seitenlänge 10.

Aufgabe 10.3 – Der Code-Inspektor

Dein Auftrag als Code-Inspektor ist es, Juno zu helfen, seine Funktion `zeichneStrich(farbe, laenge)` zu untersuchen. Juno wollte eine Funktion programmieren, die einen Strich in einer bestimmten Länge und Farbe zeichnet. Sein Programm funktioniert aber nicht wie erwartet.



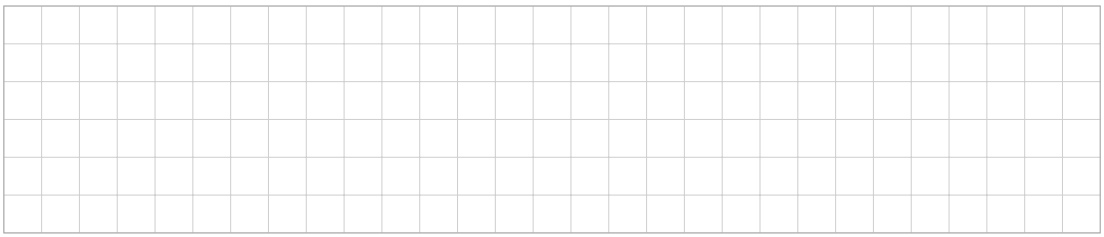


(a) Beschreibe das erwartete Ergebnis, das durch den Aufruf der Funktion im Schildkröten-Programm entstehen sollte.

(b) Beschreibe das tatsächliche Ergebnis, das im Bild zu sehen ist.

(c) Erkläre, warum die Funktion die übergebenen Parameter `farbe` und `laenge` ignoriert.

(d) Korrigiere die Funktion so, dass sie richtig arbeitet.



Aufgabe 10.4 – Die Zaun-Meisterschaft

Juna möchte an der großen Zaun-Meisterschaft teilnehmen. Ihre Aufgabe ist hierfür, ein Programm für die Schildkröte zu schreiben, das Zaunpläne für unterschiedliche Kunden zeichnet. Eine gute Programmiererin muss auf die Wünsche ihrer Kunden eingehen können!

Die Kunden und ihre Wünsche:

- Eine Hasenfamilie: „Wir brauchen einen breiten Zaun mit vielen Latten, damit unser großes Gehege geschützt ist. Aber wir sind klein, der Zaun muss also nicht hoch sein.“

- Eine Giraffe: „Mein Hals ist lang! Ich brauche einen sehr, sehr hohen Zaun, damit ich nicht darüber stolpere. Er muss aber nicht besonders breit sein.“

Juna überlegt: Ein starres Programm reicht nicht. Sie muss eine flexible Funktion für die Schildkröte entwickeln, um auf die Wünsche der Kunden eingehen zu können. Deine Aufgabe ist es, Juna bei der Entwicklung dieses Programms zu helfen.

(a) Analysiere die Anforderungen:

Ermittle die beiden Eigenschaften des Zauns, die sich für jeden Kunden ändern müssen, um ihre Wünsche zu erfüllen.

Eigenschaft 1: _____

Eigenschaft 2: _____

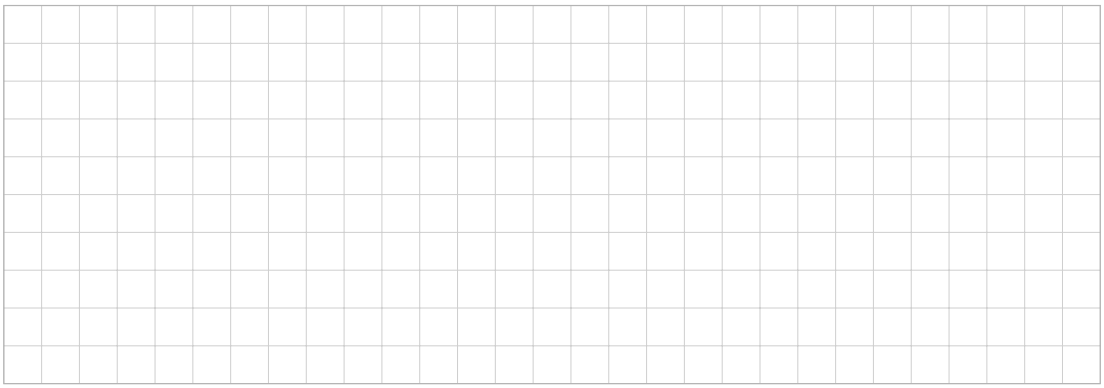
(b) Definiere den neuen Baustein:

Definiere den Beginn der Funktion, die Juna schreiben soll. Benenne die beiden Parameter, die du in Teil 1 ermittelt hast, und schreibe sie als Parameter in die Klammern.

```
zeichneZaun( _____, _____ )
```

(c) Entwirf nun das vollständige Struktogramm oder ein Programm für die Funktion `zeichneZaun(...)`. Benutze die von dir gewählten Parameter, um die Logik im Inneren zu steuern, die die Schildkröte ausführen wird.

- **Hinweis 1**: Du brauchst eine Wiederholungsanweisung, die so oft läuft, wie einer deiner Parameter es vorgibt.
- **Hinweis 2**: Innerhalb der Wiederholung musst du die andere Eigenschaft mit dem anderen Parameter steuern.
- **Hinweis 3**: Vergiss nicht, die Schildkröte nach jeder Latte ein Stück zur Seite zu bewegen.

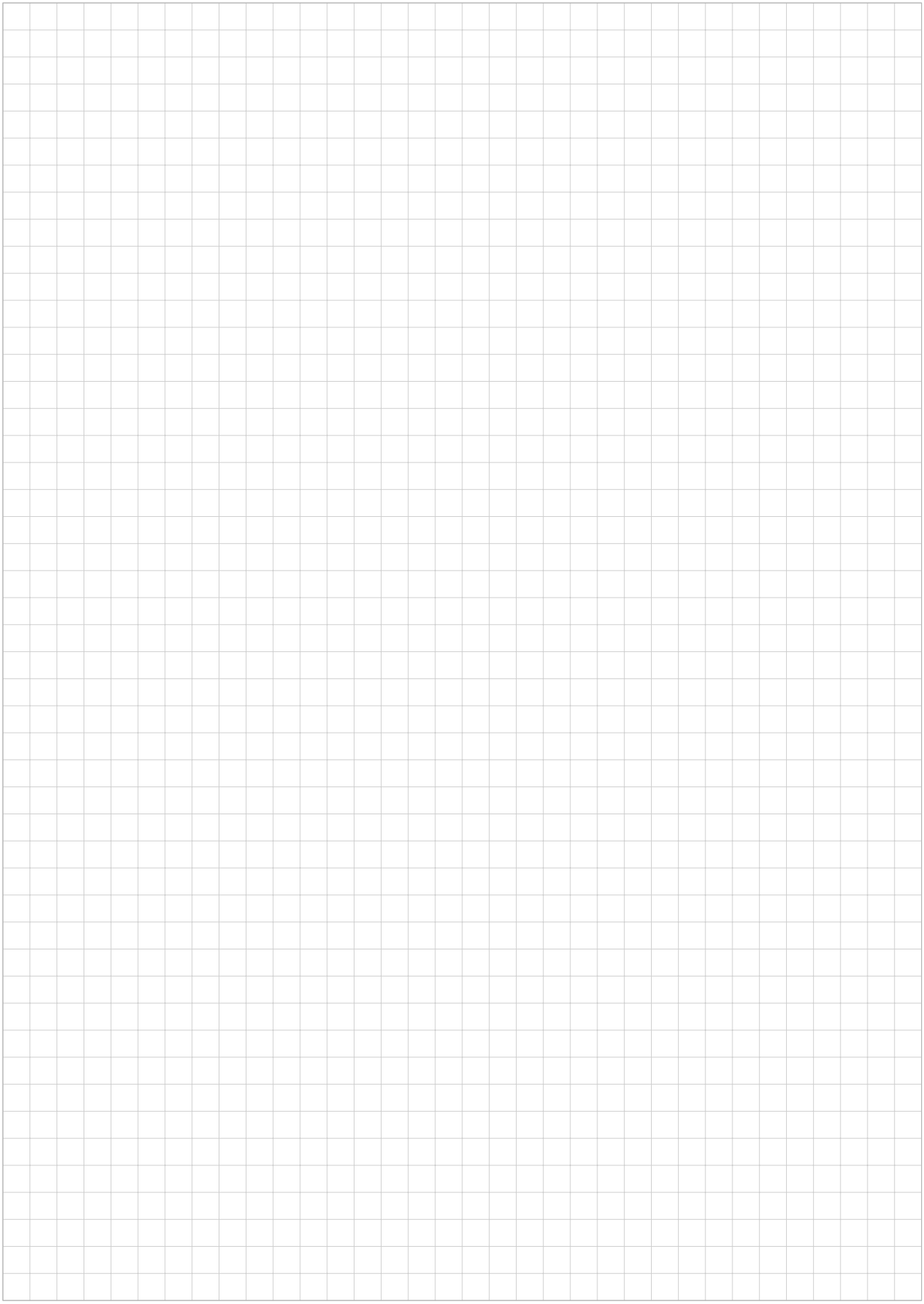


(d) Wende die Funktion an:

Schreibe die korrekten Aufrufe, die Juna in ihr Hauptprogramm schreiben muss, um die Zäune für die beiden Kunden von der Schildkröte zeichnen zu lassen.

Für die Hasenfamilie: _____

Für die Giraffe: _____



Unterstützung:

GESELLSCHAFT FÜR INFORMATIK

Fraunhofer IUK-Technologie

MAX-PLANCK-INSTITUT FÜR INFORMATIK

Gefördert vom:

Bundesministerium für Bildung, Familie, Senioren, Frauen und Jugend

Von der Kultusministerkonferenz empfohlene Wettbewerbe

Unter der Schirmherrschaft des Bundespräsidenten