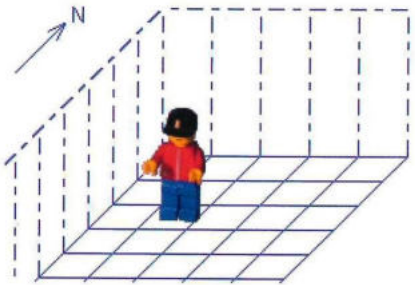
## Robot Karol

Robot Karol ist eine Programmiersprache, die entwickelt wurde, um Schülerinnen und Schülern einen möglichst motivierenden Einstieg in die Programmierung und Algorithmik zu gewähren.
Programmiert wird dabei ein Roboter namens Karol (ein Objekt der Klasse ROBOTER), der in seiner eigenen rechteckigen Welt, umrandet von einer Mauer, lebt. Bewegen kann er sich auf gleich großen quadratischen Feldern. Auf jedem Feld können entweder ein oder mehrere gestapelte Ziegel oder ein Quader liegen. Befindet sich kein Ziegel oder Quader auf einem Feld, kann Karol dieses betreten und dabei in eine der vier Himmelsrichtungen blicken. Die Klasse **ROBOTER** hat somit die Attribute *PositionX, PositionY* und *Blickrichtung*.

| ROBOTER |
| --- |
| PositionX<br>PositionY<br>Blickrichtung |
| Schritt()<br>LinksDrehen()<br>RechtsDrehen()<br>... |

Klassenkarte ROBOTER




**Punktnotation:**

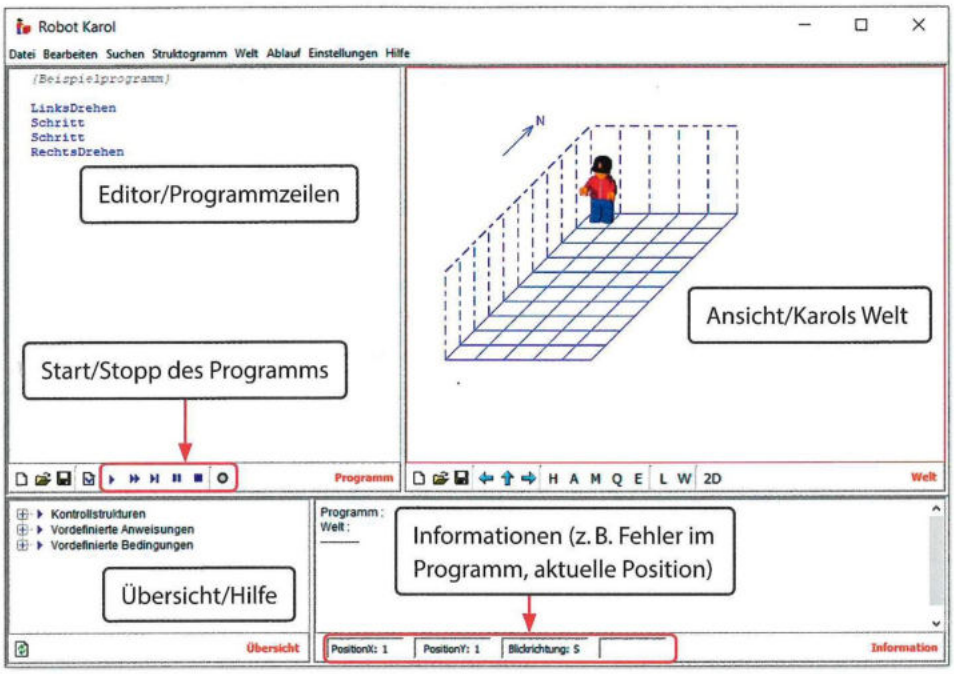
Karol.PositionX = 2

Karol.PositionY = 3

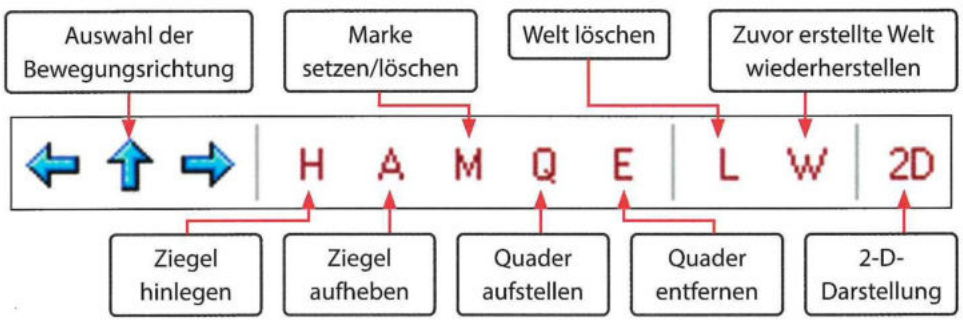
Karol.Blickrichtung = Süd

## Die Programmierumgebung Robot Karol

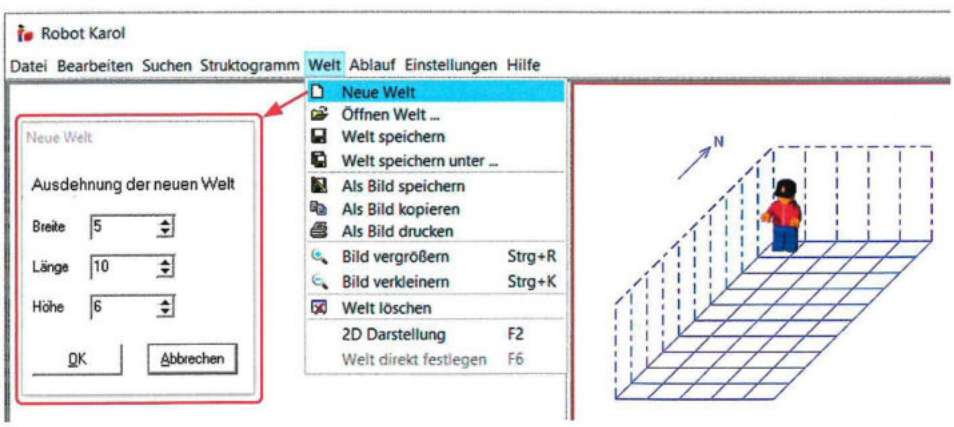
### Benutzeroberfläche von Robot Karol:




Karol kann mit Hilfe von Befehlen direkt von dir gesteuert werden und das ohne vorher ein Programm schreiben zu müssen. Sogar Marken, Ziegel und Quader kannst du per Hand auf der Welt platzieren:



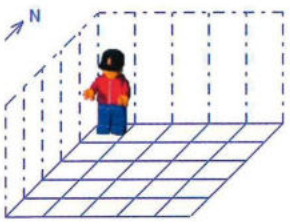

Will man die Größe der Welt verändern, kann man über den Reiter „Welt" und die Option „Neue Welt" die Ausdehnung der neuen Welt festsetzen. Die Höhe gibt vor, wie hoch Karol später maximal seine Ziegel stapeln kann.



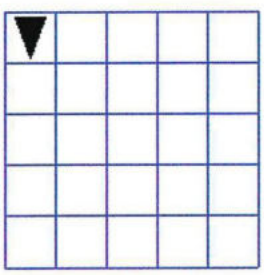

Befindet man sich in der 2-D-Darstellung, können die Objekte der Welt durch Mausklick direkt gesetzt bzw. gelöscht werden. Öffne dazu das Werkzeugfenster über das Menü „Welt" und klicke auf „Welt direkt festlegen".

Im Fenster Übersicht/Hilfe" findest du vordefinierte Anweisungen (z. B. Schritt, Linksdrehen) und Bedingungen (z. B. IstWand, IstZiegel).

Marke, Ziegel und Quader

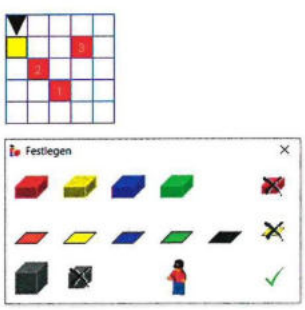


3-D-Ansicht



2-D-Ansicht

**3.** Öffne das Programm Robot Karol und erstelle eine neue Welt mit der Ausdehnung 10×10×10.



Ziegel und Marken in verschiedenen Farben, Quader und die Figur Karol. Auch mehrere Ziegel übereinander sind möglich.

**4.** Recherchiere im Internet, welche „Mini-Languages" es noch gibt.

## Roboter Karols Sprache

Robot Karol beherrscht die Sprache "Mini-Language". Dies ist eine Programmiersprache, die bewusst über einen kleinen, übersichtlichen Sprachumfang verfügt.

Aus objektorientierter Sicht kann der Roboter **Karol** als ein **Objekt** der **Klasse ROBOTER** betrachtet werden. Mit folgenden …

**… Attributen** (Eigenschaften): Blickrichtung, Ablaufgeschwindigkeit, Position, Sprunghöhe

… und **Methoden** (Fähigkeiten):

1. Mit einer **vordefinierten Anweisung** (z. B. Schritt) sendet man dem Objekt Karol den Befehl, mit der Methode „einen Schritt nach vorne gehen" zu reagieren.
2. Karol kennt aber auch Methoden, um bei einer Abfrage (z. B. „Stehst du vor einem Ziegel?") mit **Wahr** oder **Falsch** zu antworten. Der Aufruf einer Methode dieser Art heißt **Bedingung.**

Die folgende Tabelle zeigt einige vordefinierte Anweisungen:

| Vordefinierte Anweisungen | **Bedeutung (Aktion von Karol)** |
|---|---|
| Schritt | Macht einen Schritt vorwärts. |
| LinksDrehen | Drehung um 90° nach links. |
| RechtsDrehen | Drehung um 90° nach rechts. |
| Hinlegen | Legt vor sich einen Ziegel hin. |
| Aufheben | Hebt den Ziegel vor sich auf. |
| MarkeSetzen | Setzt auf dem Feld unter sich eine Marke. |
| MarkeLöschen | Löscht die Marke auf dem Feld unter sich. |
| Warten | Wartezeit eine Sekunde. |
| Ton | Spielt einen Ton ab. |
| Langsam | Bewegt sich langsam. |
| Schnell | Bewegt sich schnell. |
| Beenden | Programm wird gestoppt. |

Die folgende Tabelle zeigt einige vordefinierte Bedingungen:

| Vordefinierte Bedingungen | **Ausgabe WAHR wenn Karol …** |
|---|---|
| IstWand | vor einer Wand (oder auch ein Quader) steht. |
| NichtIstWand | keine Wand vor sich hat. |
| IstZiegel | vor einem Ziegel oder einem Ziegelturm steht. |
| NichtIstZiegel | kein Ziegel oder Ziegelturm vor sich. |
| IstMarke | auf einer Marke steht. |
| NichtIstMarke | keine Marke unter sich hat. |
| IstNorden | in genau diese jeweiligen Richtungen schaut. |
| IstOsten | |
| IstSüden | |
| IstWesten | |

Du kannst dem Objekt Karol neue Anweisungen und Bedingungen beibringen. Wie das funktioniert wird in einem späteren Kapitel erklärt.

## Abläufe mit der Karol-Sprache beschreiben

Wir wissen bereits (vgl. Kapitel 2.1, Seite 38 ff.):

- Abläufe werden durch eine oder mehreren Anweisungen beschrieben.
- In der Informatik nennt man solche Verarbeitungsvorschriften (Regeln, Anleitungen, Rezepte, Lösungsverfahren, …) **Algorithmen.**
- Der Begriff **Algorithmus** beschreibt eine endliche Folge aus eindeutigen und ausführbaren Anweisungen zur Lösung eines bestimmten Problems. Dabei kann der Algorithmus aus vielen Einzelschritten bestehen, die in der Summe zur Lösung des definierten Problems führen.
- Algorithmen bestehen aus Sequenzen (Folgen von Anweisungen), Schleifen und Fallunterscheidungen.

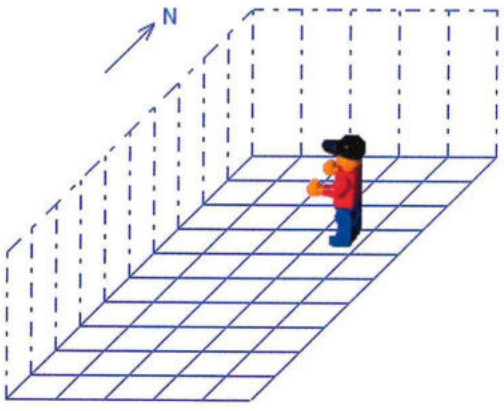

Ablauf des Bindens einer Krawatte als Algorithmus in Form von 6 Anweisungen dargestellt

All das werden wir nun anhand der Karol-Sprache vertiefen.

Zunächst werden wir linear programmieren, das heißt, alle Anweisungen stehen direkt untereinander und bilden eine **Sequenz.**

*Beispiel 1:* Erste Gehversuche.

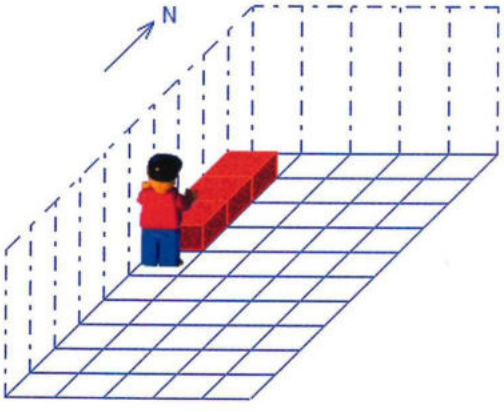
```
Schritt
Schritt
Schritt
LinksDrehen
Schritt
Schritt
Schritt
LinksDrehen
LinksDrehen
```




**Sequenz:** Hintereinander auszuführende Programmanweisungen.

*Beispiel 2:* Karol baut sich die erste Mauer.

```
Hinlegen
Schritt
Hinlegen
Schritt
```




**5.** Der Fleckenteufel hat zugeschlagen. Vervollständige den Programmcode aus dem *Beispiel 2* so, wie auf dem Bild mit Karol zu sehen ist.

**Verwendung von Parametern:**

Um Schreibarbeit zu sparen und die Übersichtlichkeit von Programmen zu gewährleisten gibt es einige vordefinierte Anweisungen und Bedingungen, die durch einen Parameterwert (Zusatzinformation) erweitert werden können.

Dazu gehören

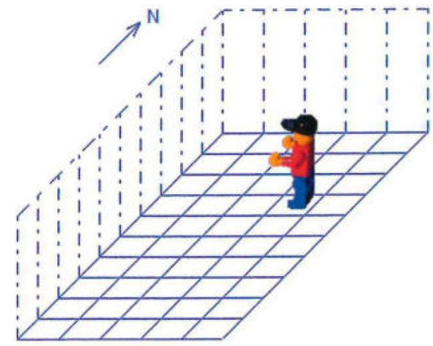
- Anweisungen: Schritt(x), Hinlegen(x), Aufheben(x), MarkeSetzen(x)
- und Bedingungen: IstZiegel(x), NichtIstZiegel(x) und HatZiegel(x).

Als Parameterwerte sind nur positive Ganzzahlwerte erlaubt, bei den Bedingungen noch zusätzlich die Zahl 0.

Es mit Robot Karol (ab Version 3.0) möglich, den Methoden *Hinlegen* und *MarkeSetzen* eine Farbe (rot, gelb, blau und grün) zu übergeben.

Mit den Parameterwerten kann man den Programmtext also deutlich verkürzen und übersichtlicher gestalten. Der Effekt wird sichtbar, wendet man die Parameterwerte bei *Beispiel 1* an:

```
Schritt(3)
LinksDrehen
Schritt(3)
LinksDrehen
LinksDrehen
```




**6.** Lege einen Ziegel in jede Ecke deiner Welt (5 × 5 × 6).

**7.** Lasse Karol deine Initialen mit Ziegeln schreiben.
Es entstehen dabei Buchstaben in Pixelgrafik!

Für Java-Programme gilt z. B. folgende Konvention:
„Code Conventions for the Java Programming Language"
(*https://www.oracle.com/technetwork/java/codeconventions-150003.pdf*)

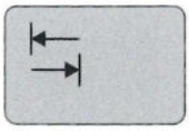
**Programme richtig kommentieren:**
Ein guter Programmcode enthält immer Kommentare, um ihn für andere verständlich zu machen und damit man selbst zu einem späteren Zeitpunkt noch weiß, warum man dieses oder jenes programmiert hat.
Es ist darauf zu achten, dass die Kommentare kurz und knapp gehalten werden. Zudem sollte man nicht jede offensichtliche Stelle im Code beschreiben, sondern sich auf das Wesentliche, d. h. auf die komplexen Zeilen, beschränken.
Kommentare werden bei Roboter Karol ab Version 3 mit der Kombination „{" und „}" von dem Programmcode abgegrenzt und dienen zur Erklärung von Programmcode-Abschnitten.

*Beispiel:*

{Einzeilige und mehrzeilige Kommentare werden mit einer geschweiften Klammer begonnen und auch wieder beendet.}

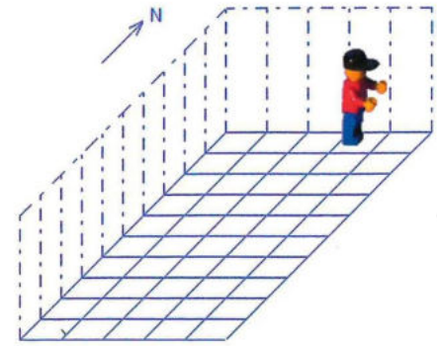
**Wohin mit dem Kommentar?**
Es gibt zwei wichtige Möglichkeiten den Kommentar zu platzieren:
- hinter der Programmzeile
- über dem Code

**Regeln der Code-Formatierung:**
Regeln bei der Programmierung helfen ebenfalls den Programmcode übersichtlich zu halten und die Lesbarkeit der Zeilen zu vereinfachen. Die verschiedenen Programmiersprachen haben jeweils allgemeingültige Vereinbarungen (Konventionen), an die sich die Programmierer halten sollten.
Für Robot Karol gibt es folgende Regeln:
- Um das Hauptprogramm eindeutig von weiteren Programmabschnitten abzuheben, wird dieses mit den Begriffen Programm und endeProgramm umschlossen.
- Es ist darauf zu achten, dass untergeordnete Code-Blöcke mit der Tabulator-Taste eingerückt werden.
- Verwende Leerzeilen zwischen Programmabschnitten.



Tabulator-Taste

```
{Hauptprogramm}
Programm
    LinksDrehen
    Schritt(3)
endeProgramm
```



## Kontrollstrukturen

**Bedingungen** sind **Methoden**, die Fragen mit **wahr** oder **falsch** beantworten.
Mit Bedingungen kannst du Anweisungen formulieren, die den Programmlauf bestimmen:

Die *Anweisungen* in einer **einseitig bedingten Anweisung** werden nur ausgeführt, wenn die Bedingung (bzw. ihre Verneinung) zutrifft. Ansonsten läuft das Programm nach „endewenn" weiter.

Die einseitige bedingte Anweisung hat folgende Syntax:

```
wenn [nicht] Bedingung dann
  Anweisung
  ...          } Sequenz
  Anweisung
endewenn
```

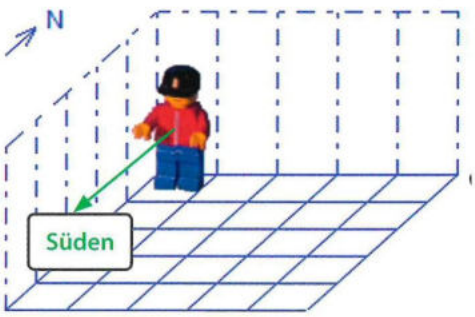
Struktogramm:

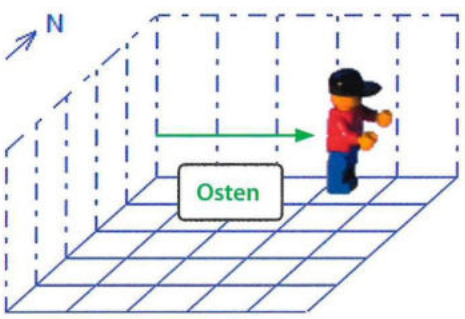
| Bedingung | |
|---|---|
| wahr | falsch |
| Anweisung | |

*Beispiel 3:* Robot Karol prüft, ob er Richtung Süden schaut. Falls ja, dreht er sich und geht nach Osten.

```
wenn IstSüden dann
  LinksDrehen
  Schritt(3)
endewenn
```

| IstSüden | |
|---|---|
| wahr | falsch |
| LinksDrehen<br>Schritt(3) | |




Von einer **zweiseitig bedingten Anweisung** spricht man, wenn es zusätzlich zu dem *wenn-Block* noch einen *sonst-Block* gibt.
Ist die vorgegebene *Bedingung* (bzw. ihre Verneinung) erfüllt, wird die *Anweisung1* (Sequenz) ausgeführt, sonst *Anweisung2* (Sequenz).

Die zweiseitig bedingte Anweisung hat folgende Syntax:

```
wenn [nicht] Bedingung dann
  Anweisung1
  ...
  Anweisung1
sonst
  Anweisung2
  ...
  Anweisung2
endewenn
```

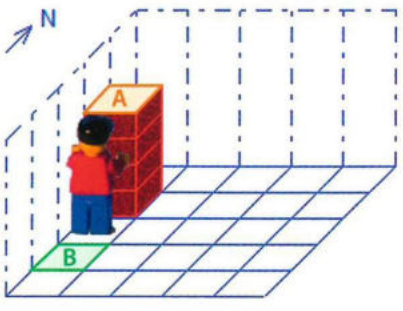
Struktogramm:

| Bedingung | |
|---|---|
| wahr | falsch |
| Anweisung 1 | Anweisung 2 |

Das Struktogramm (Nassi-Shneiderman-Diagramm) ist ein Diagrammtyp zur Darstellung des Programmablaufs. Es wurde 1972/73 von ISAAC NASSI und BEN SHNEIDERMAN entwickelt. Eine andere Möglichkeit der Darstellung ist der Programmablaufplan (PAP).

Die **Syntax** regelt den Aufbau und legt die zulässigen Sprachelemente einer Programmiersprache fest. Außerdem regelt sie, inwiefern diese Elemente in einem Programm verwendet werden dürfen.
Jedes Element hat eine Bedeutung – die **Semantik.** Syntaktisch falsche Elemente/Programme haben keine Semantik.
Beispielsweise bedeutet die Anweisung „LinksDrehen" → „Drehung um 90° nach links." (vgl. Seite 90)

**8.** Du hast einen Ferienjob in einem Getränkelager und sollst nun das Lager neu ordnen.
Schreibe ein Programm, welches den ganzen Stapel Getränkekisten (hier ein Stapel Ziegel) von A nach B bringt.



Mit einer **Endlos-Wiederholung** wird ein Block von Anweisungen dauerhaft wiederholt.

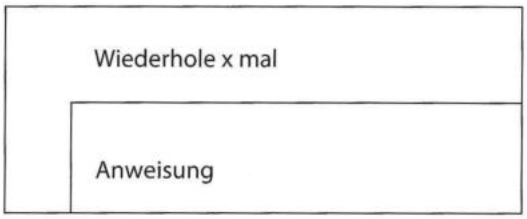
```
wiederhole immer
  Anweisung
  ...
  Anweisung
endewiederhole
```

## Wiederholungen (Schleifen)

**Wiederholung mit fester Anzahl (Zählschleife):**
Soll ein Befehl oder auch eine Sequenz von Befehlen mehrmals wiederholt werden und steht die Anzahl der Wiederholungen fest, verwendet man zur Vereinfachung die Wiederholung mit fester Anzahl.
Du schreibst also den Befehl bzw. die Sequenz nicht mehrmals in dein Programm, sondern gibst vorher an, wie oft wiederholt werden soll.

Die Wiederholung mit fester Anzahl hat folgende Syntax:

```
wiederhole x mal
  Anweisung
  ...
  Anweisung
endewiederhole
```

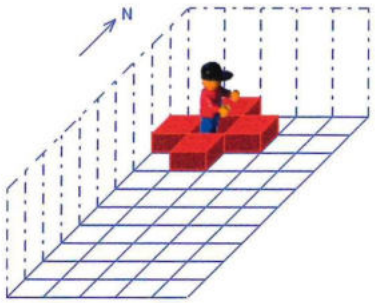



**9.** Öffne die Umgebung „Robot Karol" und erstelle das Programm *„Mitten im Stern"* und führe das Programm durch einen Klick auf die Start-Taste aus.

**10.** Zeichne ein Struktogramm zu diesem Programm.

*Beispiel 4:* Mitten im Stern

```
Schritt
LinksDrehen
Schritt
wiederhole 4 mal
  Hinlegen
  RechtsDrehen
endewiederhole
```



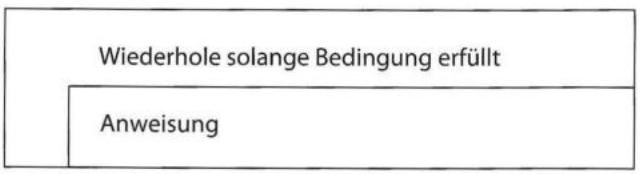

**Wiederholung mit Anfangsbedingung:**
Ein Befehl oder auch eine Sequenz von Befehlen soll wiederholt werden, solange eine gewisse Bedingung (bzw. ihre Verneinung) erfüllt ist. Die Überprüfung der Bedingung erfolgt am Anfang der Wiederholung. Dadurch ist es möglich, dass die Anweisung gar nicht ausgeführt wird, sollte die Bedingung nicht zutreffen.

> **Wiederholung mit Anfangsbedingung** werden immer dann verwendet, wenn die Aktion nur bei einer sinnvollen Bedingung durchgeführt werden soll.

*Beispiel:*
Zuerst prüft man, ob man noch etwas im Glas hat, und trinkt dann. Nicht umgekehrt. Das wird wiederholt, bis das Glas leer ist.

Die bedingte Wiederholung mit Anfangsbedingung hat folgende Syntax:

```
wiederhole solange [Nicht] Bedingung
  Anweisung
  ...
  Anweisung
endewiederhole
```



Wie unterscheiden sich *<[Nicht] Bedingung>* und *<Bedingung>* in der Syntax-Darstellung der Wiederholung mit Anfangsbedingung (▶Seite 92 unten)?

Die Bedingung **IstWand** liefert entweder *wahr* oder *falsch:*
- Wahr: Es ist eine Wand vor Karol.
- Falsch: Es ist keine Wand vor Karol.

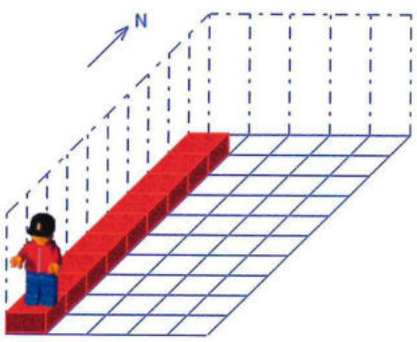
Die Bedingung **NichtIstWand (Negation)** liefert entweder *wahr* oder *falsch:*
- Wahr: Es ist keine Wand vor Karol.
- Falsch: Es ist eine Wand vor Karol.



Mit der **Negation** wird ein Wert invertiert, also umgekehrt. Das heißt, aus wahr wird falsch und aus falsch wird wahr.

*Beispiel 5:* Robot Karol kann nur dann gehen, wenn keine Wand vor ihm ist. Das heißt, die Sequenz läuft *nur dann* ab, wenn die Bedingung NichtIstWand **wahr** zurückgibt, also wenn keine Wand vor Karol zu finden ist. Sobald die Bedingung falsch ist, wird das Programm am Ende der Wiederholung fortgesetzt und Karol gibt einen Signalton als Warnung aus.

```
{Hauptprogramm}
Programm
  wiederhole solange NichtIstWand
    Hinlegen
    Schritt
  endewiederhole
  Ton
endeProgramm
```



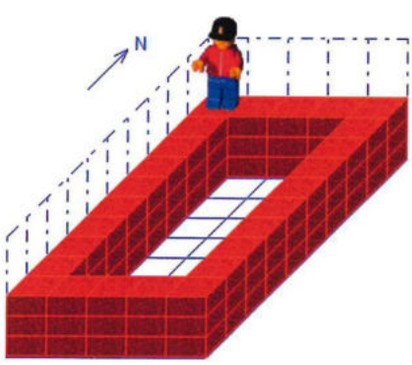

**Verschachtelte Schleifen:**
Es gibt keinerlei Einschränkungen bezüglich der Anweisungen, die im Körper einer Schleife vorkommen dürfen. Daher ist es auch möglich, dass eine Schleife eine oder mehrere Schleifen enthält. Die Konstruktion, bei der Schleifen durch übergeordnete Schleifen kontrolliert werden, heißt **verschachtelte Schleifen.** Sie gehören zum Repertoire eines Programmierers, da viele Problemstellungen sich nur mit ihrem Einsatz lösen lassen.

Der Anzahl an ineinander verschachtelten Wiederholungen sind prinzipiell keine Grenzen gesetzt. Allerdings führt eine Verschachtelungstiefe von mehr als drei Ebenen zu schwer verständlichem Code.

Hier ist ein Beispiel für eine Verschachtelung von Schleifen:

```
{Hauptprogramm}
Programm
  wiederhole x mal
    wiederhole solange [Nicht] Bedingung
      Anweisung
      ...
      Anweisung
    endewiederhole
    Anweisung
    ...
  endewiederhole
endeProgramm
```

**11.** Schreibe ein Programm mit verschachtelten Schleifen und lasse Karol dieses Schwimmbadbecken bauen:




**12.** Erstelle ein passendes allgemeines Struktogramm dazu.

Man kann sich die Verschachtelung gut vorstellen, wenn man sie mit einer Uhr vergleicht. Die äußere Schleife entspricht dabei dem Stundenzeiger, die Inneren dem Minutenzeiger: Während der Stundenzeiger um eine Einheit vorrückt, absolviert der Minutenzeiger einen kompletten Umlauf.
Genauso vollzieht die innere Schleife sämtliche Anweisungen während eines einzigen Durchlaufs der äußeren Schleife.

**Wiederholung mit Endbedingung:**
Bei dieser Schleife findet hier die Überprüfung der Bedingung am Ende statt und daher wird die Anweisung mindestens einmal ausgeführt!

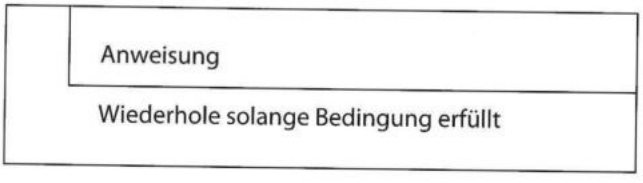
> **Wiederholung mit Endbedingung** werden immer dann verwendet, wenn das Ergebnis einer Aktion geprüft wird. Man führt zuerst aus und prüft dann.

*Beispiel:* Zuerst gibt man das Ziel einer Bahnreise ein und prüft danach, ob eine ICE-Verbindung angeboten wird. Wenn nicht, muss eine andere Verbindung oder Zugart ausgewählt werden.

Die bedingte Wiederholung mit Endbedingung hat folgende Syntax:

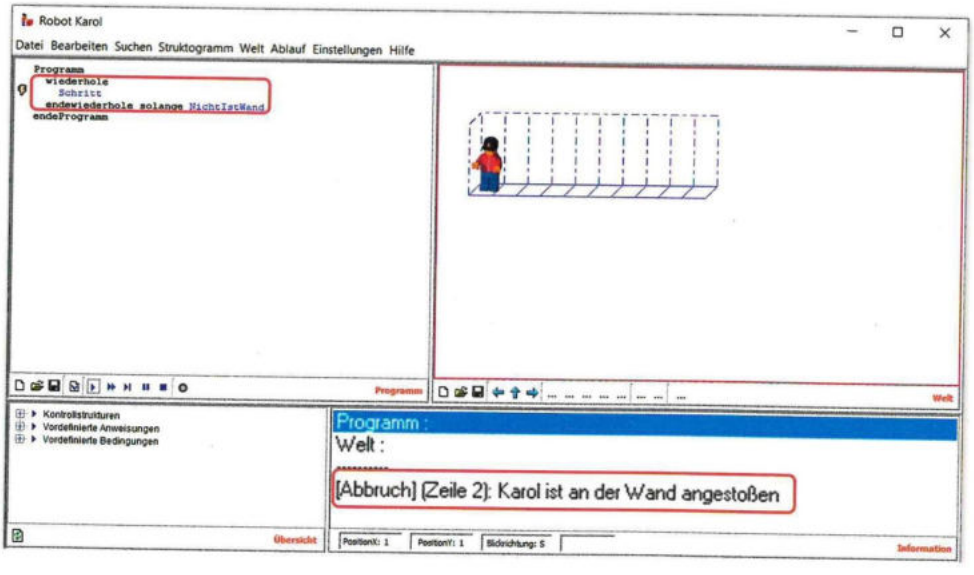
```
wiederhole
  Anweisung
  ...
  Anweisung
endewiederhole solange [nicht] Bedingung
```

**„bis"** ist statt „solange" auch erlaubt.



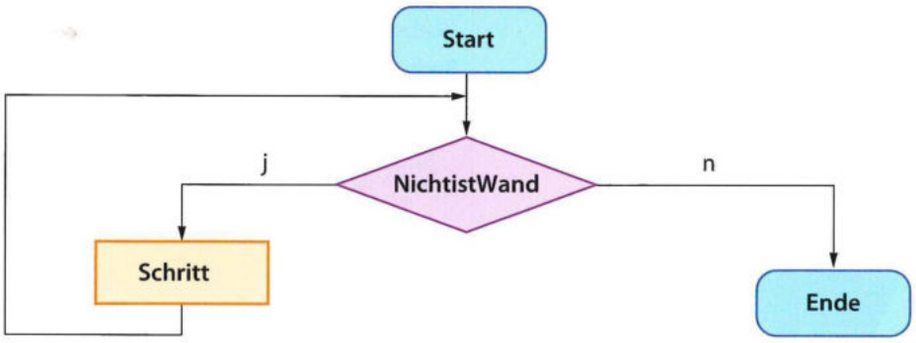

Wird vom Benutzer eine bestimmte Eingabe erwartet, eignet sich dafür die Schleife *mit Endbedingung.*
*Beispiel:* Nur wenn das richtige Passwort eingegeben wird, erlangt man Zutritt in einen Raum.

**Vergleich von Anfangsbedingung und Endbedingung:**
Die Wiederholung mit Endbedingung gibt hier eine Fehlermeldung aus:




Besser ist es in diesem Fall eine Wiederholung mit Anfangsbedingung zu verwenden. Diese prüft erst ob eine Wand vor Karol ist und stoppt das Programm, wenn dieser Fall zutrifft.

```
Programm
  wiederhole solange NichtIstWand
    Schritt
  endewiederhole
endeProgramm
```




Programmablaufplan (PAP)

Überlege dir genau, ob du eine Wiederholung mit Endbedingung oder Anfangsbedingung verwendest.

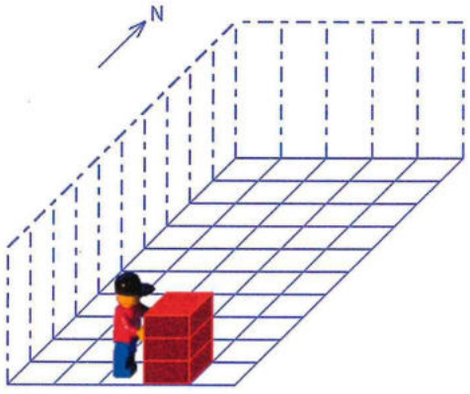
## Anweisungen und Bedingungen selbst schreiben

Es ist möglich, die neuen Anweisungen und Bedingungen mit einem frei gewählten Namen zu benennen. Dieser kann Buchstaben, Ziffern und den Unterstrich („_“) enthalten.

**Selbstdefinierte Anweisungen:**
Wie in anderen Programmiersprachen auch, können eigene Anweisungen direkt im Programm definiert werden.

```
{Selbstdefinierte Anweisung}
Anweisung LegeZiegel
  wiederhole 3 mal
    Hinlegen
  endewiederhole
endeAnweisung

{Hauptprogramm}
Programm
  wiederhole solange NichtIstWand
    Schritt
  endewiederhole
  LinksDrehen
  Schritt(2)
  LegeZiegel
endeProgramm
```




Die Bezeichner der Anweisungen können Buchstaben (auch Umlaute), Ziffern und „_“ enthalten. Neben dem Schlüsselwort **Anweisung** ist auch das Schlüsselwort **Methode** für die Kennzeichnung der Definition möglich.

**Selbstdefinierte Bedingungen:**
Der Roboter Karol versteht nicht nur vorgegebene Bedingungen, er kann auch mit selbstdefinierten Bedingungen umgehen. Laut Syntax muss WAHR oder FALSCH in der konstruierten Bedingung vorkommen, damit festgelegt ist, welchen Wert die Bedingung zurückgibt.

*Beispiel 6:* Robot Karol untersucht eine Mauer auf Lücken. Sobald links von ihm kein Ziegel liegt, bleibt er stehen (▶Seite 96):

Beachte: Eine neue Anweisung muss erst definiert werden, bevor sie verwendet werden kann.

Der Roboter Karol kann Befehle auf zwei Arten ausführen: **langsam** oder **schnell.**

Vor allem bei umfangreichen Programmen (z. B. Pyramide bauen) ist es nützlich den Programmablauf zu beschleunigen.
Dabei gilt die Beschleunigung *schnell* nur bis zum nächsten *langsam* und umgekehrt.
Außerdem sind sie nur in einem Block gültig.

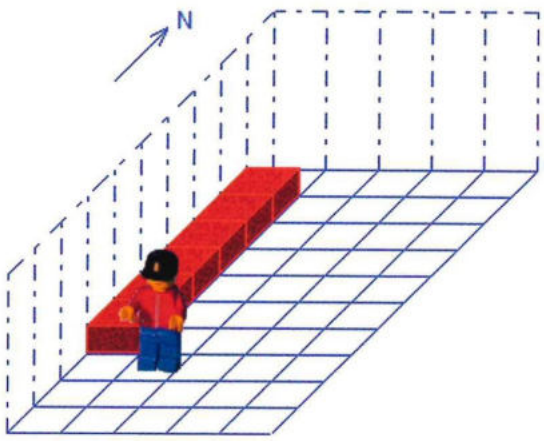
*Beispiel:*

```
Bedingung IstZiegelRechts
  schnell
  falsch
  RechtsDrehen
  wenn IstZiegel dann
    wahr
  endewenn
  LinksDrehen
endeBedingung

Programm
  ...
```

```
{Selbstdefinierte Bedingung}
Bedingung IstZiegelLinks
  falsch
  RechtsDrehen
  wenn IstZiegel dann
    wahr
  endewenn
  LinksDrehen
endeBedingung

Programm
  Schritt
  wiederhole solange IstZiegelLinks
    Schritt
  endewiederhole
endeProgramm
```




Die Bedingung wird zu Beginn immer auf falsch gesetzt. Dadurch bleibt Karol stehen, wenn kein Ziegel neben ihm ist. Die Wenn-dann-Abfrage setzt den Rückgabewert auf wahr, falls sich ein Ziegel rechts neben ihm befindet.

## Vorgehensweise bei Programmtest und -optimierung

Bei Programmierungen können immer wieder Schwierigkeiten und Fehlermeldungen beim Programmablauf auftreten. Um die Arbeit der Fehlersuche möglichst gering zu halten, solltest du den Programmcode bereits während deiner Arbeiten überprüfen und kritisch hinterfragen:

- Treten Anweisungen mehrmals hintereinander auf? Verwende Schleifen.
- Ist dein Roboter auf alle Eventualitäten vorbereitet? Das Programm darf nicht mit einer Fehlermeldung abbrechen.
- Erzielt der Code die gewünschten Ergebnisse?

Das Programm aus Beispiel 1 (▶Seite 89) wird verbessert:

| | | |
|---|---|---|
| Programm<br>  Schritt(3)<br>  LinksDrehen<br>  Schritt(3)<br>  LinksDrehen<br>  LinksDrehen<br>endeProgramm | Anweisung Vorwärts<br>  wiederhole 3 mal<br>  Schritt<br>  endewiederhole<br>endeAnweisung<br><br>Programm<br>  Vorwärts<br>  LinksDrehen<br>  Vorwärts<br>  LinksDrehen<br>  LinksDrehen<br>endeProgramm | Anweisung Vorwärts<br>  wenn NichtIstWand dann<br>  wiederhole 3 mal<br>  Schritt<br>  endewiederhole<br>  endewenn<br>endeAnweisung<br><br>Programm<br>  Vorwärts<br>  LinksDrehen<br>  Vorwärts<br>  LinksDrehen<br>  LinksDrehen<br>endeProgramm |
| | • Die Anweisungen „Schritt" werden in einen Anweisungsblock „Vorwärts" ausgelagert.<br>• Identische Anweisungen, die hintereinander ablaufen sollen, werden in eine Schleife gepackt. | • Sicherheitsüberprüfung, ob Karol an der Wand steht. |

In der Informatik wird der Vorgang des Fehlersuchens auch **Debuggen** genannt.
Ein **Debugger** ist ein Werkzeug, mit dem man in der Lage ist Fehler in Programmen zu finden und zu beheben.
Bei der Fehlerbeseitigung verfolgst du den Ablauf deines Programms Anweisung für Anweisung und überprüfst dabei jeweils das Ergebnis.

Bei Robot Karol kannst du deine Programmzeilen Schritt für Schritt auf Richtigkeit überprüfen: