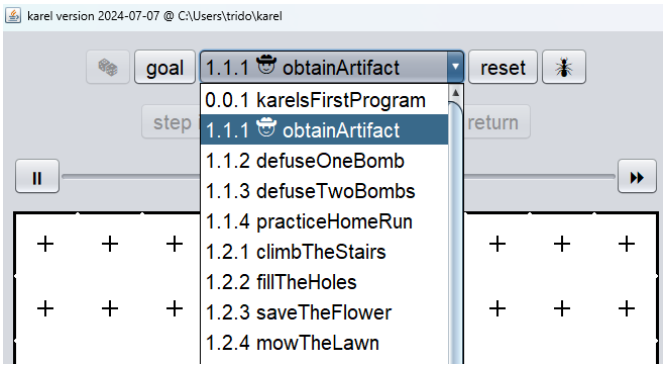
3.1 Karel, der Roboter – Das Handbuch

Im Folgenden lernst du eine einführende Programmierumgebung kennen, in dem ihr einen Roboter steuert, der einfache Probleme lösen soll. Dieser Roboter heißt **Karel**.

Karel soll die Grundlagen des Programmierens vermitteln, in dem ihr die grundlegenden Kontrollstrukturen in einer einfachen Umgebung lernen könnt, die frei von den Komplexitäten ist, die die meisten Programmiersprachen charakterisieren. Karel wurde in einführenden Informatikkursen auf der ganzen Welt eingesetzt und hat Millionen von Studenten unterrichtet. Mehrere Generationen von Stanford-Schülern haben gelernt, wie das Programmieren mit Karel funktioniert, und es ist immer noch die sanfte Einführung in die Programmierung, die in Stanford verwendet wird.

Zunächst werden dir alle Befehle von Karel vorgestellt, die du für deine Programme nutzen kannst. Anschließend werden dir zwei grundlegende Kontrollstrukturen (if-else Verzweigung und die Schleife) kurz erläutert. Die Erklärung sind bewusst kurz gehalten, da insbesondere ein intuitiver und spielerischer Zugang gewählt wird, damit du verschiedene Ansätze austesten kannst. Im späteren Verlauf des Unterrichts werden die Kontrollstrukturen erneut ausführlich für die Programmiersprache Java erläutert.

Die Programmierumgebung Karel enthält bereits verschiedene Aufgabenstellungen, die du in der oberen Leiste auswählen kannst:



Mit **goal** wird dir visualisiert, was Karel in der Aufgabenstellung machen muss. Mit **reset** wird Karel wieder auf die Ausgangsposition gesetzt.

Darunter befindet sich eine Leiste, in der du die Geschwindigkeit bei der Ausführung deines Programms verändern kannst.

Beachte, dass alle Aufgaben innerhalb der Methode, mit der richtigen Bezeichnung geschrieben werden.

Zum Beispiel das Programm 0.0.1 karelsFirstProgramm wird in die Methode **`karelsFirstProgram()`** geschrieben. Das Programm 1.1.1 wird in die Methode **`obtainArtifact()`** geschrieben.

```
1  void karelsFirstProgram() {
2      moveForward();
3      ...
4  }
5
6  void obtainArtifact() {
7
8      ...
9  }
```

Hinweis:

Verwende bei der Programmierung die Shortcuts F1-F11, damit du die ganzen Befehle nicht mit der Tastatur ausschreiben musst und dadurch Zeit sparst. Der Fokus liegt hier in der Entwicklung von Programmierideen und nicht im Erlernen einer Syntax!

Karels Befehle

Kurz-taste	**Befehl**	**Beschreibung**
F1	moveForward();	Karel bewegt sich ein Feld vorwärts in die Richtung, in die er gerade schaut. Er macht nichts, wenn eine Wand vor ihm ist.
F2	turnLeft();	Karel dreht sich um 90° nach links.
F3	turnAround();	Karel dreht sich um 180°.
F4	turnRight();	Karel dreht sich um 90° nach rechts.
F5	pickBeeper();	Karel hebt einen Beeper von dem Feld auf, auf dem er sich gerade befindet. Er macht nichts, wenn kein Beeper vorhanden ist.
F6	dropBeeper();	Karel legt einen Beeper auf das Feld, auf dem er sich gerade befindet. Er macht nichts, wenn bereits ein Beeper vorhanden ist.
F7	onBeeper();	Karel überprüft, ob sich ein Beeper auf dem Feld befindet, auf dem er sich gerade befindet.
F8	beeperAhead();	Karel überprüft, ob sich ein Beeper auf dem Feld direkt vor ihm befindet.
F9	leftIsClear();	Karel überprüft, ob sich eine Wand links von ihm befindet.
F10	frontIsClear();	Karel überprüft, ob sich eine Wand vor ihm befindet.
F11	rightIsClear();	Karel überprüft, ob sich eine Wand rechts von ihm befindet.

Feste Wiederholung repeat(x)

Anstatt die gleiche Anweisungsfolge mehrfach zu schreiben, kannst du **`repeat`** (Wiederholung) verwenden und die Anweisung nur einmal notieren. In Klammern wird hinter **`repeat`** die feste Anzahl der Wiederholungen geschrieben.

```
1  void dance()
2  {
3      moveForward();
4      turnLeft();
5      moveForward();
6      turnLeft();
7      moveForward();
8      turnLeft();
9      moveForward();
10     turnLeft();
11 }
```

Ist äquivalent zu:

```
1  void dance()
2  {
3      repeat(4)
4      {
5          moveForward();
6          turnLeft();
7      }
8  }
```

If/else

Manchmal soll Karel nur dann etwas tun, wenn eine bestimmte Bedingung erfüllt ist:

```
1  if (onBeeper())
2  {
3      pickBeeper();
4  }
```

Optional kannst du auch angeben, was zu tun ist, falls die Bedingung nicht erfüllt ist:

```
1  if (onBeeper())
2  {
3      pickBeeper();
4  }
5  else
6  {
7      dropBeeper();
8  }
```

FALLS Karel sich auf einen Beeper befindet, soll er diesen aufheben, **SONST** soll er einen Beeper ablegen.

If/else if

```
1  if (leftIsClear())
2  {
3      turnLeft();
4  }
5  else if (rightIsClear())
6  {
7      turnRight();
8  }
9  else
10 {
11     dropBeeper();
12 }
```

Bei einer **`else if`** – Anweisung, werden die Bedingungen nacheinander so lange geprüft, bis die erste erfüllt ist! Die Anweisungen davon werden dann ausgeführt und die restlichen Bedingungen werden nicht mehr betrachtet.

In dem Beispiel würde zuerst überprüft werden (Zeile 1), ob links von Karel eine Wand ist. Falls ja dreht er sich nach links und das Programm ist fertig. Sonst würde er überprüfen, ob rechts von ihm eine Wand ist (Zeile 5) und sich dann nach rechts drehen. Anschließend wäre das Programm fertig. Falls die ersten beiden Bedingungen falsch wären, würde er in Zeile 9 in den else-Zweig gehen und einen Beeper ablegen.

Logische Operatoren

Operator	Beschreibung
`!a`	wahr, wenn nicht a gilt.
`a && b`	wahr, wenn sowohl a als auch b wahr sind.
`a || b`	wahr, wenn entweder a oder b oder a und b wahr sind.
`a || !b && c`	`a || ((!b) && c) //Betrachte, wie der Computer vorrangig die Klammern setzt.`

Nicht !

Ein `if/else` mit einem leeren ersten Block:

```
1  if (onBeeper())
2  {
3  }
4  else
5  {
6      dropBeeper();
7  }
```

kann vereinfacht werden, indem die Bedingung mit einem vorangestellten `!` negiert wird.

```
1  if (!onBeeper())
2  {
3      dropBeeper();
4  }
```

Logisches Und &&

Ein `if`, dass nichts anderes als ein weiteres `if` enthält:

```
1  if (frontIsClear())
2  {
3      if (beeperAhead())
4      {
5          moveForward();
6          pickBeeper();
7      }
8  }
```

kann vereinfacht werden, indem man beide Bedingungen mit einem logischen Und **`&&`** kombiniert.

```
1  if (frontIsClear() && beeperAhead())
2  {
3      moveForward();
4      pickBeeper();
5  }
```

Logisches Oder ||

Ein `if/else if` mit identischen Blöcken:

```
1  if (!frontIsClear())
2  {
3      turnRight();
4  }
5  else if (beeperAhead())
6  {
7      turnRight();
8  }
```

kann vereinfacht werden, indem man beide Bedingungen mit einem logischen Oder **||** kombiniert.

```
1  if (!frontIsClear() || beeperAhead())
2  {
3      turnRight();
4  }
```

While-Schleife

Mit der **`repeat(x)`** –Kontrollstruktur, konnte man Anweisungsfolgen mit einer vorher festgelegten Anzahl wiederholen lassen.

```
1  void dance()
2  {
3      repeat(4)
4      {
5          moveForward();
6          turnLeft();
7      }
8  }
```

Es gibt jedoch auch Fälle, in denen vorher nicht klar ist, wie häufig eine Anweisung wiederholt werden sollte. Beispielsweise wenn man möchte, dass Karel bis zu einer Wand geradeauslaufen soll. Dies hängt nämlich von Karels aktueller Position ab. In diesem Fall verwendet man die **`while`**-Schleife.

Betrachte folgendes Beispiel:

```
1  void moveForwardSafely()
2  {
3      if (frontIsClear())
4      {
5          moveForward(); // Diese Anweisung wird 0 oder 1 mal ausgeführt.
6      }
7  }
```

`If` prüft die Bedingung and führt den Anweisungsblock maximal einmal aus.

`While` prüft hingegen die Bedingung jedes Mal, nachdem der Anweisungsblock ausgeführt wurde. Betrachte dazu den folgenden Programmcode.

```
1  void moveToWall()
2  {
3      while (frontIsClear())
4      {
5          moveForward(); // Die Anweisung wird bis zu 0 bis 9 mal ausgeführt.
6      }
7  }
```