RK4

"A new spin on an old bot"

Dies ist eine inoffizielle Weiterentwicklung von Robot Karol 3, bekannt aus dem Informatikunterricht.

Alles neu?

Im Vergleich zum Original bringt diese Version einige Neuerungen mit sich:

- **3D-Welt**, in der **mehrere** Roboter gleichzeitig wuseln können
- Eine (nahezu) voll ausgestattete Skriptingsprache, die es u.a. erlaubt...
  - eigene **Variablen**, **Funktionen** und **Klassen** zu definieren.
  - **Objekte** zu instanziieren.
  - "echte" neue **Methoden** (mit **Parametern**!) für die Roboter zu vereinbaren.
  - in Schleifen klassischen Kontrollfluss wie **zurück**, **abbrechen** und **weiter** zu verwenden.
  - mathematische Ausdrücke zu verwenden, z.B. ***meineZahl * 4***
  - usw.
- Mehrstufige Aufgaben in JSON/CSV-Format, in denen Zielzustände pro Feld definiert werden können.
- Live-Generierung von **Struktogrammen**, **Klassenkarten** und **Flussdiagramme**
- Introspektion in **Roboter-Objektkarten**
- Live-Fehlerkorrektur während **Lexing**, **Parsing** und **Runtime**

> [!WARNING] Wer nach einer klassischeren Robot-Karol-Version sucht, die aber auch das Skripting in Python, Java und Code-Blöcken erlaubt, der ist wahrscheinlich besser bei karol.arrrg.de aufgehoben.

RKBasic

Die neue Skriptingsprache (Dateiendung .rk) ist sehr nah an der ursprünglichen Karol-Syntax und damit Teil der BASIC-Familie. Zudem bietet aber sie aber sehr simple Objektorientierung (ohne Vererbung) und passt somit besser in den Lehrplan der 6. und 7. Klasse.

Vordefinierte Robotermethoden

Die Roboter-Objekte haben stets Namen wie k1, k2, usw. Um Sie zu bewegen, sie Ziegel legen oder aufzuheben zu lassen, Marken zu setzen oder ihre Sensoren abzufragen, *MUSS* die Punktschreibweise verwendet werden:

```
k1.schritt()            // Roboter k1 macht einen Schritt vorwärts.

k1.linksDrehen()        // Roboter k1 dreht sich um 90° nach links.
k1.rechtsDrehen()       // ... dreht sich um 90° nacht rechts.

k2.hinlegen()           // Roboter k2 legt einen Ziegel vor sich.
k2.aufheben()           // ... hebt den obersten Ziegel vor sich auf.

k1.hinlegen(rot)        // Manche Methoden nehmen optionale Parameter,
                        // wie hier die Ziegelfarbe
```

```
k1.markeSetzen(grün)    // Roboter k1 setzt eine grüne Marke unter sich.
k1.markeEntfernen()     // ... löscht die Marke unter sich.

k1.siehtWand()          // Gibt wahr zurück, wenn k1 vor einer Wand steht.
k1.siehtAbgrund()       // Gibt wahr zurück, wenn k1 vor dem Abgrund steht.
k1.siehtZiegel(gelb)    // Gibt wahr zurück, wenn der oberste Ziegel
                        // im Stapel vor k1 gelb ist.
                        // Ohne Parameter ist die Ziegelfarbe egal.
k1.istAufMarke(rot)     // Gibt wahr zurück, wenn die Marke unter k1
                        // rot ist. Auch hier ist der Parameter optional.

k1.x()          // Gibt die x-Koordinate von k1 zurück.
k2.y()          // Gibt die y-Koordinate zurück.
k3.richtung()   // Gibt die Richtung als Text zurück: "N", "S", "W" oder "O".
```

Wiederholungen

Im Gegensatz zu Robot Karol 3 werden *ALLE* Blöcke in RKBasic mit dem Schlüsselwort `ende` beendet. D.h. es gibt kein `endeWiederhole` oder `*Anweisung` mehr.

```
// k1 geht 7 Schritte vorwärts
wiederhole 7 mal
    k1.schritt()
ende

// k2 dreht sich solange, bis er nach Norden schaut
// Mehr zu = und anderen Vergleichsoperatoren später
wiederhole solange k2.richtung() = "N"
    k2.linksDrehen()
ende

// k3 läuft unendlich lang im Kreis
wiederhole immer
    k3.linksDrehen()
    k3.schritt()
ende
```

Bedingte Anweisungen

Anweisungen *MÜSSEN* in RKBasic in einer neuen Zeile stehen, Semikolons (😉 werden *NICHT* akzeptiert. Einrücken wird empfohlen, ist aber nicht nötig. Bedingte Anweisungen sehen dann z.B. so aus:

```
// Hier zeigen wir das Wort "Geschafft" in der Konsole an,
// wenn die aktuelle Teilaufgabe gelöst wurde.
wenn welt.fertig() dann
    zeig "Geschafft!"
ende
```

```
// Hier mit sonst.
wenn k1.siehtWand() dann
    zeig "Oh Nein, eine Wand"
    k1.linksDrehen()
sonst
    k1.schritt()
ende

// Auch 'sonst wenn' ist legal...
// ... wird aber als geschachtelte bedingte Anweisung interpretiert.
wenn k1.x() > 0 dann
    k1.schritt()
sonst wenn k1.y() > 0 dann
    k1.rechtsDrehen()
    k1.schritt()
sonst
    zeig "Nicht am Rand!"
ende
```

❓ Bedingungen

Bedingungen sind im allgemeinen Wahrheitswerte. Diese können buchstäblich `wahr` oder `falsch` sein, aber auch berechnet werden.

```
zeig wahr oder falsch       // In der Konsole: >> wahr
zeig wahr und falsch        // >> falsch
zeig nicht falsch           // >> wahr
zeig nicht k1.siehtWand()   // >> falsch, wenn k1 die Wand sieht
zeig 7 > 8                  // >> falsch
zeig 7 < 100                // >> wahr
zeig 17 = 17                // >> wahr
```

🔑 Besondere Schlüsselwörter

RKBasic besitzt einige besondere Schlüsselwörter, die es in Robot Karol 3 noch nicht gab.

- `zeig x, y, z, ...` gibt die Werte x, y, z, usw. mit Lehrzeichen getrennt in der Konsole aus. Vergleichbar mit `print` in Python 2.
- `abbrechen` entspricht `break` und beendet eine Wiederholung vorzeitig.
- `weiter` entspricht `continue` und überspringt den Rest der aktuellen Anweisungssequenz in einer Wiederholung.
- `zurück x` entspricht `return` und beendet das Program bzw. den Funktionsaufruf und gibt den Wert x zurück.

> [!WARNING] Achtung, den Rückgabewert kann man (aktuell) nicht auslassen! Wenn einem der Rückgabewert egal ist, schreibt man einfach `zurück nix`.

💬 Kommentare

RKBasic unterstützt drei Arten von Kommentare. C-Style-Kommentare für eine Zeile `// ...` oder mehrere Zeilen `/* ... */` werden bei der Ausführung ignoriert. Python-Style-Kommentare für eine Zeile `# ...` werden zwar auch ignoriert, erscheinen aber als Label im Struktogramm. Mehrere aufeinanderfolgende Zeilen werden dabei zu einem einzelnen Label zusammengefasst.

```
// Zeile wird ignoriert.

/*
Mehrere
zeilen
werden
ignoriert.
*/

# Diese Zeile wird im Label angezeigt.
# Und diese auch - im selben Label!
```

> [!WARNING] Achtung, #-Kommentare *MÜSSEN* in einer eigenen Zeile stehen.

🔢 Variablen

RKBasic kennt grundsätzlich vier verschiedene Datentypen: `Zahl`, `Text`, `Wahrheitswert` und `Objekt`. Eine Deklaration muss immer auch mit einer Wertzuweisung mit `ist` oder `als` passieren:

```
Zahl x ist 7 // Deklariert eine Zahl namens x und setzt ihren Wert auf 7
Text t ist "Hallo"
Wahrheitswert w ist wahr
```

Bei Objekten ist es ein bisschen anders:

```
Objekt v als Vektor // Erzeugt ein neues Objekt v der Klasse 'Vektor'.
```

👉 Funktionen

Funktionen ersetzen die Anweisungen aus früheren Versionen und können mit dem Schlüsslwort `Funktion` definiert werden. Wichtig: Bei Parametern müssen Datentypen angegeben werden.

```
Funktion hallo(Text name)
    zeig "Hallo", name
ende

hallo("Karol") // >> Gibt 'Hallo Karol' aus
```

🏛 Klassen

Es ist - wie gesagt - auch möglich, eigene Klassen mit Attributen und Methoden zu definieren. Um Attribute anzupassen muss man seine eigenen Setter schreiben oder die Attribute direkt setzen.

```
Klasse Vektor
    Zahl x ist 0
    Zahl y ist 0

    Methode setzeXY(Zahl sx, Zahl sy)
        x ist sx
        y ist sy
    ende

    Methode plus(Objekt v)
        x ist x + v.x
        y ist y + v.y
    ende

    Methode zeigmich()
        zeig x, y
    ende
ende

Objekt v1 als Vektor
v1.x ist 4
v1.y ist 6

Objekt v2 als Vektor
v2.setzeXY(3, -9)

v1.plus(v2)

v1.zeigmich() // >> 7 -3
```

Es gibt auch die Möglichkeit, Klassen mithilfe von Parametern zu erzeugen. Dieser "Konstruktor" ist allerdings auch auf das `als`-Schlüsselwort beschränkt:

```
Klasse Foo(Zahl n)
    Zahl bar ist n
    Zahl zweiBar ist 2 * n
ende

Objekt f als Foo(4)
zeig f.bar // >> 4
zeig f.zweiBar // >> 8
```

🤖 Externe Methoden

Relevanter für einen Unterrichtskontext sind dafür sogenannte *externe Methoden*. Mit diesen ist es möglich, (mithilfe des `für` Schlüsselworts) für bereits existierende Klassen, wie z.B. `Roboter` oder `Welt` neue Methoden zu definieren, die dann ganz normal über die Punktschreibweise ausgeführt werden können.

```
Methode umdrehen() für Roboter
    linksDrehen() // Da wir 'im' Roboter sind, kann man hier 'k1.' weglassen
    linksDrehen()
ende

Methode gehen(Zahl n) für Roboter
    wiederhole n mal
        wenn siehtWand() dann
            zurück falsch // wir haben bei einer Wand gestoppt
        ende
        schritt()
    ende
    zurück wahr // wir wurden nicht behindert
ende

Methode feldAufräumen() für Roboter
    wiederhole solange siehtZiegel()
        aufheben()
    ende
ende

k1.umdrehen() // Funktioniert ebenso wie...
k2.umdrehen() // ... das!
```

➗ Mathematische Operationen

Wie bereits vorher beschrieben, lassen sich Zahlen mit `=`, `>` und `<` miteinander vergleichen. Da RKBasic als Zuweisungsoperator `ist` hat, ist kein übliches Doppel-Ist-Gleich (`==`) nötig.

Daneben gibt es natürlich die mathematischen Grundoperationen:

```
zeig 5 + (-4)   // >> 1,    Addition & Subtraktion
zeig 8 * 12     // >> 96,   Multiplikation
zeig 22 : 2     // >> 11,   Division
zeig 33 / 3     // >> 11,   geht auch so
zeig 33 % 2     // >> 1,    Modulus
```

Für Text ist hingegen nur `+` definiert.

```
zeig "Halli" + "hallo" // >> Hallihallo
```

Aufgaben

Weltformat

Eine Welt wird durch einen Semikolon-unterteilten String definiert und kann deshalb z.B. gut in Excel erstellt werden.

Beispiel:

```
x;4;5;6
S;_;_;_
_;Y;_;_
_;_;_;_
rr;_;_;_
_;_;_:b;_
```

Diese Welt ist 4 Felder lang (Westen → Osten), 5 Felder breit (Norden → Süden) und 6 Ziegelsteine hoch.

An der Position (0|0), also nord-westlich, steht ein Roboter und blickt nach Süden (`S`), bei (1|1) liegt eine gelbe Marke (`Y`) und bei (0|3) liegen zwei rote Ziegel (`rr`). An der Stelle (3|4) ist ein Feldziel definiert: Hier ist das Feld zuerst leer (`_`), um die Aufgabe zu lösen soll heir aber ein blauer Ziegel gelegt werden (`:b`).

> [!IMPORTANT] Allgemein zeigt der Doppelpunkt (`:`) an, dass es hier eine Aufgabe zu lösen gibt.

Bedeutung der Buchstaben

In jedem Feld kann also eine Reihe von Buchstaben stehen, die vorgibt, wie die Welt Aufgebaut werden soll:

	Beschreibung	**Kommentar**
`x`	Beginnt neue Teilaufgabe	muss immer am Anfang neben den Weltdimensionen stehen
`N`	Platziert Roboter, Blickrichtung NORD	wird im Aufgabenmodus ignoriert
`S`	Platziert Roboter, Blickrichtung SÜD	wird im Aufgabenmodus ignoriert
`W`	Platziert Roboter, Blickrichtung WESTEN	wird im Aufgabenmodus ignoriert
`E`	Platziert Roboter, Blickrichtung OSTEN	wird im Aufgabenmodus ignoriert
	Ziegel	
`r`	Platziert ROTEN Ziegel	
`g`	Platziert GRÜNEN Ziegel	

	Beschreibung	Kommentar
b	Platziert BLAUEN Ziegel	
y	Platziert GELBEN Ziegel	
	Marken	
R	Platziert ROTE Marker	
G	Platziert GRÜNE Marke	
B	Platziert BLAUE Marke	
Y	Platziert GELBE Marke	
	Spezial	
_	Initialisiert ein LEERES Feld	
#	Setzt WAND	
:	Schaltet in den AUFGABENMODUS um	rrrr:_ heißt z.B., dass die 4 roten Blöcke entfernt werden müssen
	Zufallsziegel	
.	Platziert EINEN oder KEINEN roten Ziegel	
*	Platziert NULL oder MEHRERE rote Ziegel bis zur Welthöhe	
+	Platziert EINEN oder MEHRERE rote Ziegel bis zur Welthöhe	
	Zufallsmarke	
?	Platziert EINE oder KEINE gelbe Marke	
!	Platziert GEGENTEIL des aktuellen Markerzustandes	`Y!` würde dafür Sorgen, dass keine Marke auf dem Feld erscheint; `?:!` würde dafür sorgen, dass die Aufgabe ist, einen Marker zu setzen oder zu entfernen, je nachdem ob es einen gibt oder nicht.

Aufgabenformat

Eine Aufgabe kann im JSON-Format erstellt werden und beinhaltet neben dem mehrstufigen Welt-String auch noch informationen zum Titel, der Aufgabenbeschreibung und Code, der vorgeladen werden soll (preload).

Beispiel:

```
{
    "title": "Dreierreihe",
    "description": "Erstelle eine Methode für die Roboterklasse, die eine Reihe
aus drei Ziegeln legt.",
    "preload": "// Nichts vorgegeben",
    "world": "x;1;4;6;\nS:_\n_:r\n_:r\n_:r\n"
}
```

Debug

Struktogramme

Im Debug-Bereich rechts lässt sich der Ablauf des Programmes mithilfe von Struktogrammen nachvollziehen. Diese sind eine partielle Implementation von Nassi-Shneidermann-Diagrammen.

Diese werden live aus dem eingetippten Code generiert.

Klassenkarten

Daneben befinden sich live-generierte Klassenkarten, die alle Attribute und methoden selbstdefinierter Klassen zeigen.

Flussdiagramme

Es gibt auch die Möglichkeit, das Programm, Funktionen und Methoden als Flussdiagramme anzeigen zu lassen. Diese werden mithilfe von Mermaid generiert.

Objektkarten

In der Roboter-Ansicht kann durch einen Klick auf den Namen eines der Roboter sein Zustand in einem Objektdiagram, das auch während der Laufzeit live upgedated wird, angezeigt werden.