Offizielles Handbuch

Programmieren lernen auf Deutsch

Für Kinder ab 10 Jahren | Lehrkräfte | Einsteiger

Inhaltsverzeichnis

CodeFuchs Handbuch — Kapitel 00: Einleitung

Was ist CodeFuchs?

Stell dir vor, du lernst eine neue Sprache — nicht Englisch oder Französisch, sondern die Sprache der Computer. **CodeFuchs** ist genau das: eine Programmiersprache, die auf Deutsch funktioniert. Du musst kein Englisch können, um damit anzufangen. Du schreibst Befehle wie `sag`, `wenn` oder `wiederhole` — ganz normale deutsche Wörter — und der Computer versteht sie und führt sie aus.

Das klingt einfach, und das ist es auch. Das ist kein Zufall. CodeFuchs wurde von Grund auf so gebaut, dass der Einstieg so leicht wie möglich ist.

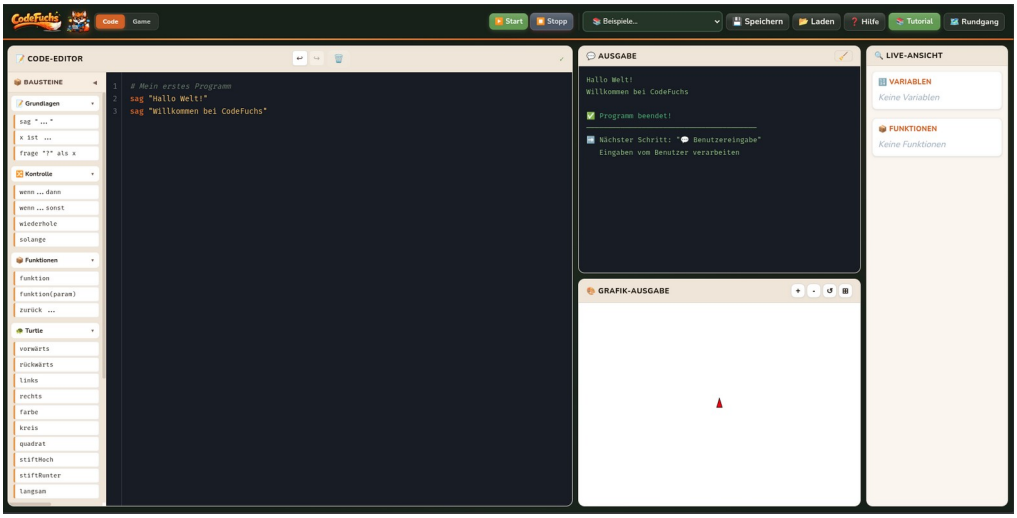
Was kannst du damit machen?

CodeFuchs hat zwei Bereiche:

CodeFuchs (die Programmier-Sandbox)

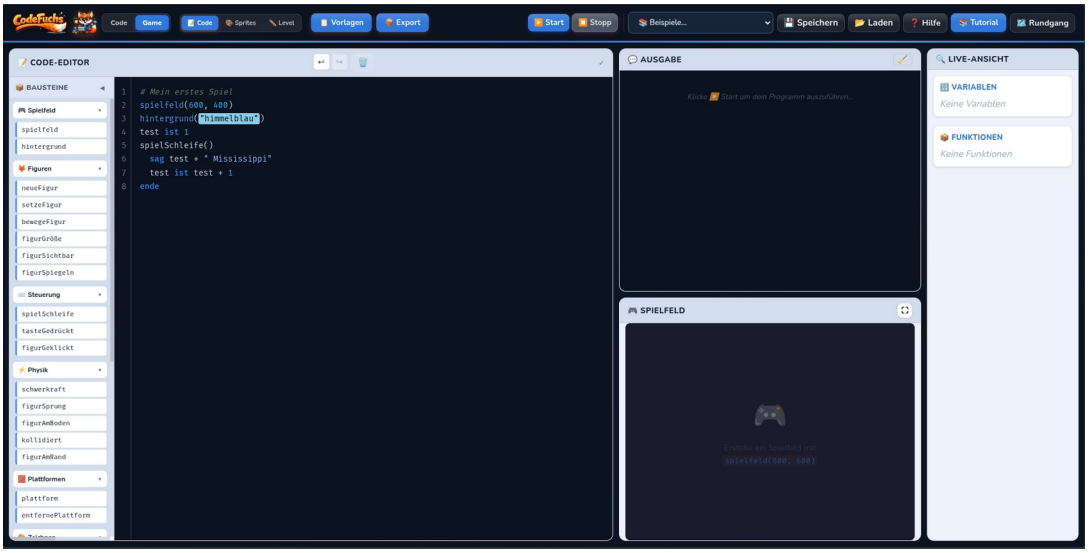
Hier lernst du die Grundlagen. Du kannst:

- Text ausgeben und Berechnungen durchführen
- Den Computer Fragen stellen und Antworten verarbeiten
- Muster und Bilder mit der **Schildkröte** zeichnen lassen
- Eigene Funktionen schreiben und wiederverwenden



GameFuchs (die Spiele-Werkstatt)

Wenn du die Grundlagen beherrschst, wartet die große Bühne: Du programmierst eigene **2D-Spiele**. Figuren, Bewegung, Kollisionen, Punkte — alles mit denselben deutschen Befehlen. Und am Ende kannst du dein Spiel als eine einzige HTML-Datei exportieren und mit Freunden teilen.



Für wen ist dieses Handbuch?

Dieses Handbuch richtet sich an **alle**, die CodeFuchs kennenlernen wollen:

- **Kinder ab 10 Jahren**: Die Kapitel sind in kleinen Schritten aufgebaut. Am Anfang reicht es, einfach mitzumachen.
- **Eltern und Lehrkräfte**: Jedes Kapitel erklärt nicht nur *was* zu tun ist, sondern auch *warum* — damit ihr die Kinder begleiten könnt.
- **Fortgeschrittene**: Die späteren Kapitel (05 und 06) sind vollständige Referenzen aller Befehle und Funktionen — übersichtlich und präzise.

Die erste Minute: So startest du CodeFuchs

Schritt 1 — Die IDE öffnen

Du brauchst nichts zu installieren. Öffne einfach deinen Browser und gehe zur CodeFuchs-Adresse. Die Oberfläche lädt sofort — sie läuft komplett im Browser, ohne Server, ohne Konto.

Schritt 2 — Dein erstes Programm

Im großen Textfeld links (dem **Editor**) gibst du folgendes ein:

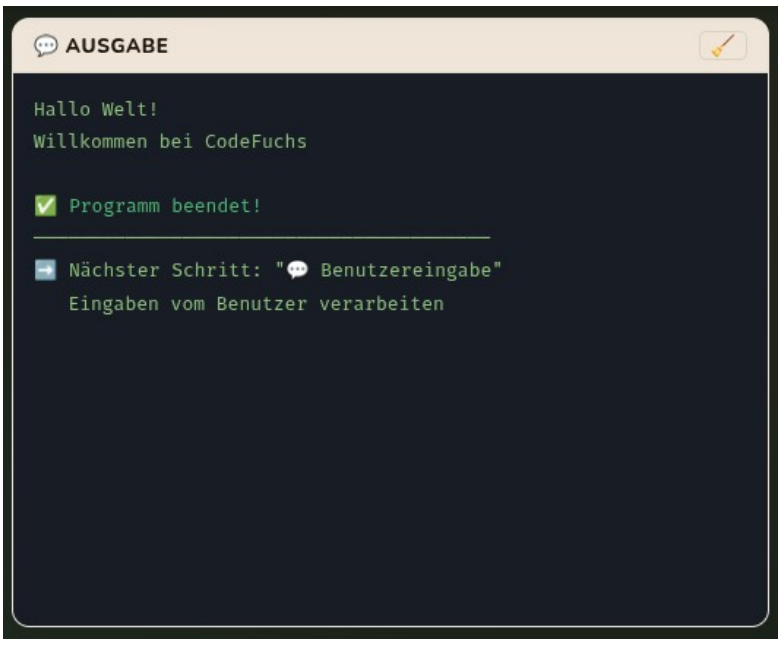
```
sag "Hallo Welt!"
```

Schritt 3 — Ausführen

Klick auf den grünen Button **"▶ Ausführen"** oben rechts. Im **Ausgabebereich** erscheint sofort:

```
Hallo Welt!
```

Das war dein erstes Programm. Du hast dem Computer gesagt, er soll etwas ausgeben — und er hat es getan.



Wie CodeFuchs denkt — ein kurzes Bild

Stell dir einen Koch vor. Du gibst ihm ein Rezept (dein Code). Er liest es Zeile für Zeile von oben nach unten und kocht genau das, was dort steht. Wenn er eine Zeile nicht versteht, hört er auf und erklärt dir freundlich, was schiefgelaufen ist.

CodeFuchs funktioniert genauso. Der **Interpreter** ist dieser Koch. Er liest deinen Code, übersetzt ihn Schritt für Schritt und führt ihn aus.

Wenn etwas nicht klappt, bekommst du keine kryptische Fehlermeldung in Englisch. Du bekommst einen Hinweis auf Deutsch, der erklärt, in welcher Zeile der Fehler steckt.

Was CodeFuchs absichtlich weglässt

CodeFuchs folgt einem einzigen Grundprinzip: **So einfach wie möglich.**

Das bedeutet:

- Kein Semikolon am Zeilenende nötig
- Keine kryptischen Sonderzeichen für Blöcke — Blöcke enden einfach mit `ende`
- Keine englischen Fachbegriffe als Befehle
- Keine Installation, keine Konfiguration, kein Build-Schritt

Was du siehst, ist alles, was du brauchst.

Bereit?

Dann geht es weiter mit **Kapitel 01 — Grundlagen CodeFuchs**. Dort lernst du Schritt für Schritt alle Bausteine der Sprache kennen — von der ersten Ausgabe bis zu eigenen Funktionen.

Viel Spaß beim Programmieren.

CodeFuchs Handbuch — Kapitel 01: Grundlagen CodeFuchs

Dieses Kapitel führt dich Schritt für Schritt durch alle Bausteine der CodeFuchs-Sprache. Jeder Abschnitt ist ein kleines Tutorial. Du lernst zuerst das Konzept, dann siehst du ein Beispiel, dann kannst du selbst üben. Fang oben an und arbeite dich nach unten vor. Jedes Kapitel baut auf dem vorherigen auf.

Lektion 1 — Ausgabe: `sag`

Das Konzept

Der einfachste Befehl in CodeFuchs ist `sag`. Er zeigt Text oder Zahlen im Ausgabebereich an. Stell dir vor, du gibst dem Computer einen Zettel mit einer Nachricht — er liest ihn laut vor.

Syntax

```
sag "Dein Text hier"
sag Zahl_oder_Variable
sag "Text" + variable
```

Beispiele

```
sag "Hallo Welt!"
sag "Ich bin 10 Jahre alt"
sag 42
sag "Das Ergebnis ist: " + 100
```

Ausgabe:

```
Hallo Welt!
Ich bin 10 Jahre alt
42
Das Ergebnis ist: 100
```

Wichtige Regeln

- Text muss immer in **Anführungszeichen** stehen: `"so"`
- Zahlen brauchen **keine** Anführungszeichen: `sag 42`
- Du kannst Text und Zahlen mit `+` verbinden: `"Wert: " + 5`
- Jede `sag`-Zeile erscheint als eigene Zeile in der Ausgabe

Kommentare

Zeilen, die mit `#` beginnen, werden ignoriert. Das ist praktisch für Erklärungen im Code:

```
# Das ist ein Kommentar – der Computer liest das nicht
sag "Hallo"  # Kommentare dürfen auch am Zeilenende stehen
```

Übung

Schreib ein Programm, das deinen Namen, dein Alter und deine Lieblingsfarbe ausgibt. Drei Zeilen, drei `sag`-Befehle.

Lektion 2 — Variablen: `ist`

Das Konzept

Eine Variable ist wie eine beschriftete Schachtel. Du legst einen Wert hinein und kannst ihn später wieder herausholen — und der Name auf der Schachtel bleibt gleich, auch wenn du den Inhalt änderst.

Syntax

```
schachtelName ist Wert
```

Beispiele

```
name ist "Anna"
alter ist 11
lieblingszahl ist 7
grossAlt ist alter * 10

sag name
sag alter
sag grossAlt
```

Ausgabe:

```
Anna
11
110
```

Rechenoperationen

Du kannst mit Variablen rechnen wie mit einem Taschenrechner:

Zeichen	Bedeutung	Beispiel
`+`	Addition	`3 + 4` → `7`
`-`	Subtraktion	`10 - 3` → `7`
`*`	Multiplikation	`6 * 7` → `42`
`/`	Division	`20 / 4` → `5`

```
a ist 10
b ist 3
summe ist a + b
produkt ist a * b
sag "Summe: " + summe
sag "Produkt: " + produkt
```

Ausgabe:

```
Summe: 13
Produkt: 30
```

Variablentypen

CodeFuchs kennt drei Arten von Werten:

Typ	Beispiel	Beschreibung
Zahl	`42` , `3.14`	Ganze Zahlen und Dezimalzahlen
Text	`"Hallo"`	Immer in Anführungszeichen
Wahrheitswert	`wahr` , `falsch`	Für Ja/Nein-Entscheidungen

```
zahl ist 42
text ist "Hallo"
istRichtig ist wahr
```

Übung

Erstelle drei Variablen: den Preis einer Schokolade, die Anzahl der Tafeln, die du kaufen willst, und den Gesamtpreis. Gib den Gesamtpreis aus.

Lektion 3 — Eingabe: `frage`

Das Konzept

Mit `frage` kannst du den Benutzer nach einer Antwort fragen. Das Programm hält an, zeigt eine Frage an, und wartet — genau wie wenn jemand eine Frage stellt und auf die Antwort wartet.

Syntax

```
frage "Deine Frage hier?" als variablenname
```

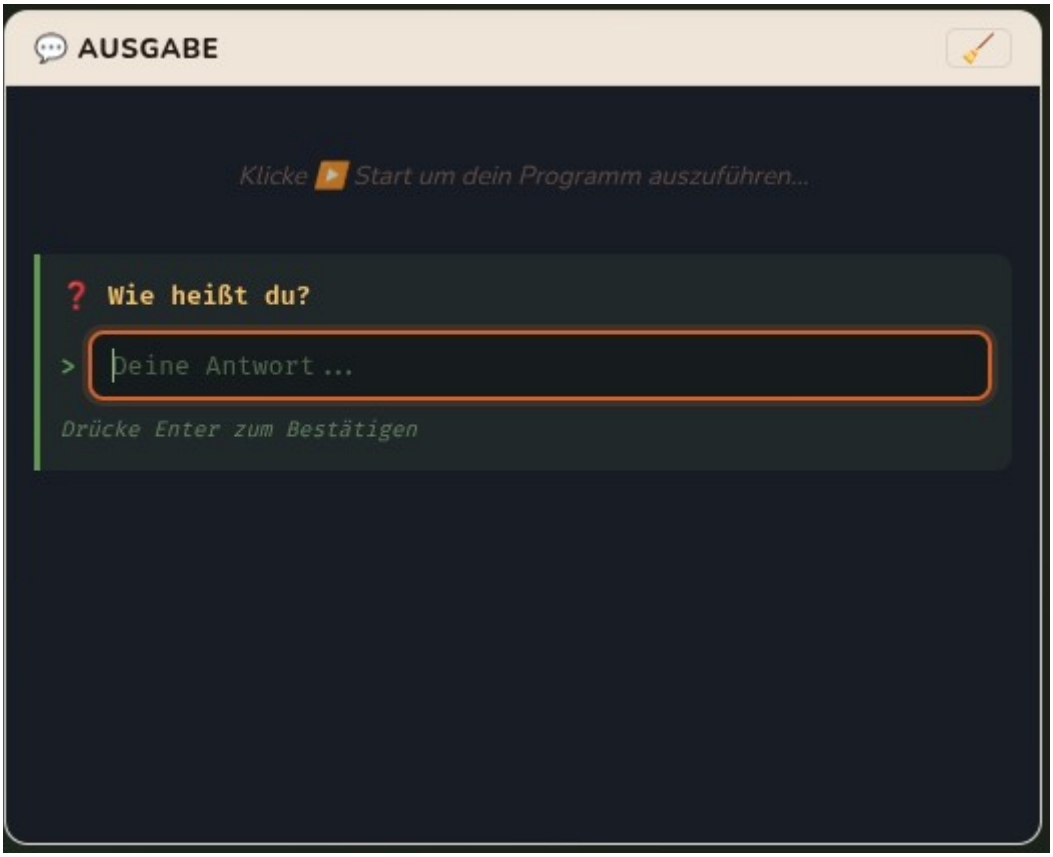
Das Schlüsselwort `als` sagt dem Computer: "Speichere die Antwort in dieser Variable."

Beispiele

```
frage "Wie heißt du?" als name
sag "Hallo, " + name + "!"

frage "Wie alt bist du?" als alter
sag "Du bist " + alter + " Jahre alt."
```

So sieht das aus:

```
→ Wie heißt du?
[Benutzer tippt: Lisa]
Hallo, Lisa!
→ Wie alt bist du?
[Benutzer tippt: 10]
Du bist 10 Jahre alt.
```



Wichtig

Die Antwort wird immer als **Text** gespeichert, auch wenn du eine Zahl tippst. Wenn du mit der Antwort rechnen willst, musst du das bedenken:

```
frage "Nenne eine Zahl:" als zahl
verdoppelt ist zahl * 2   # Achtung: funktioniert nur wenn Zahl eingegeben
wird
sag "Das Doppelte ist: " + verdoppelt
```

Übung

Schreib ein Programm, das nach Vorname und Nachname fragt und dann den vollständigen Namen ausgibt.

Lektion 4 — Bedingungen: `wenn / dann / sonst / ende`

Das Konzept

Manchmal soll der Computer eine Entscheidung treffen: "Wenn es regnet, nimm einen Regenschirm mit — sonst nicht." In CodeFuchs heißt das `wenn`.

Syntax

```
wenn Bedingung dann
  # Code, der ausgeführt wird wenn die Bedingung wahr ist
ende
```

Mit einem `sonst`-Zweig:

```
wenn Bedingung dann
  # Code wenn wahr
sonst
  # Code wenn falsch
ende
```

Vergleichsoperatoren

Operator	Bedeutung	Beispiel
`=`	Gleich	`alter = 10`
`!=`	Ungleich	`name != "Bob"`
`>`	Größer als	`punkte > 100`
`<`	Kleiner als	`temp < 0`
`>=`	Größer oder gleich	`alter >= 18`
`<=`	Kleiner oder gleich	`punkte <= 50`

Beispiele

```
frage "Wie alt bist du?" als alter

wenn alter >= 18 dann
  sag "Du bist volljährig!"
sonst
  sag "Du bist noch nicht volljährig."
ende
```

Ein Beispiel mit Zahlenvergleich:

```
zahl ist 7

wenn zahl > 10 dann
  sag "Die Zahl ist groß"
sonst
  wenn zahl > 5 dann
    sag "Die Zahl ist mittel"
  sonst
    sag "Die Zahl ist klein"
  ende
ende
```

Logische Verknüpfungen

Du kannst Bedingungen kombinieren:

```
# UND: Beide Bedingungen müssen wahr sein
wenn alter >= 10 und alter <= 18 dann
  sag "Du bist ein Teenager"
ende

# ODER: Mindestens eine Bedingung muss wahr sein
wenn farbe = "rot" oder farbe = "orange" dann
  sag "Warme Farbe!"
ende
```

Einrückung

Der Code *innerhalb* eines `wenn` -Blocks wird eingerückt (mit Leerzeichen oder Tab). Das macht den Code lesbarer. CodeFuchs macht das automatisch, wenn du Enter drückst.

Übung

Schreib ein Programm, das nach einer Zahl fragt und ausgibt, ob sie positiv, negativ oder null ist.

Lektion 5 — Zählschleife: `wiederhole / mal / ende`

Das Konzept

Eine Schleife führt Code mehrmals aus. Statt zehnmal das Gleiche zu schreiben, sagst du: "Wiederhole das zehnmal."

Stell dir vor, du musst 10 Liegestütze machen. Du zählst innerlich mit — das macht `wiederhole` für dich.

Syntax

```
wiederhole Anzahl mal
  # Code der wiederholt wird
ende
```

Beispiele

```
wiederhole 5 mal
  sag "Hallo!"
ende
```

Ausgabe:

```
Hallo!
Hallo!
Hallo!
Hallo!
Hallo!
```

Mit einer Variablen als Anzahl:

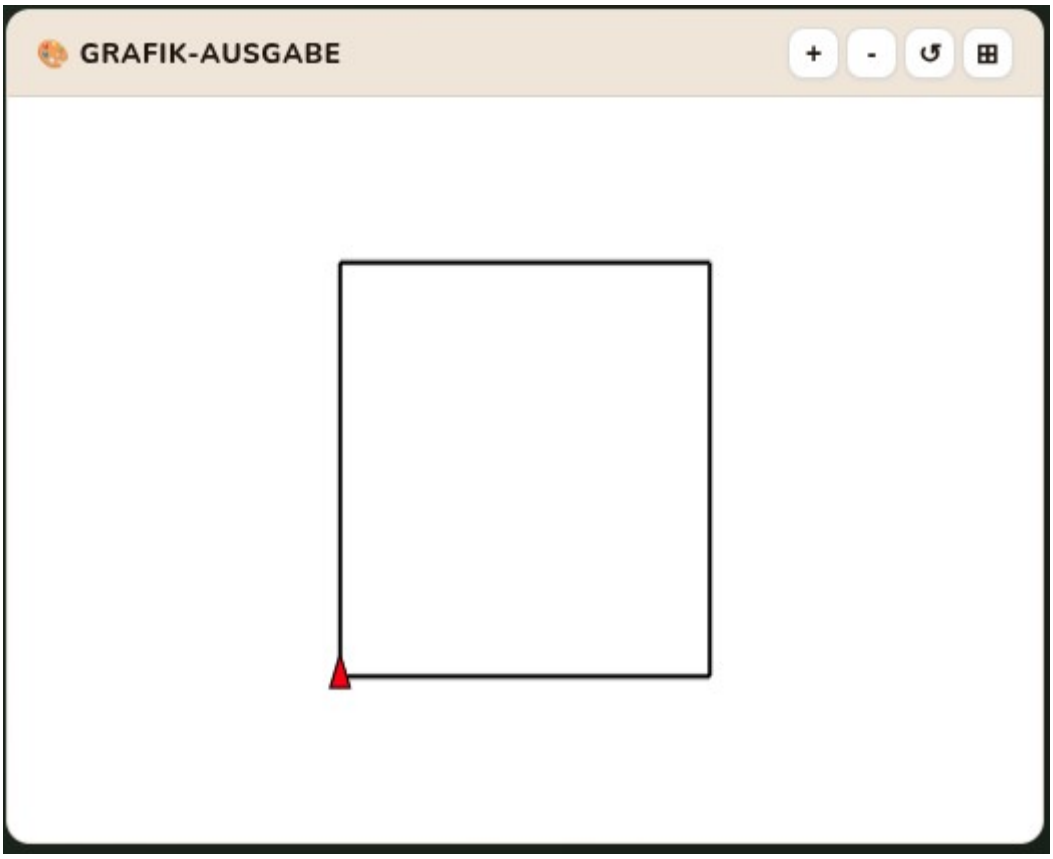
```
anzahl ist 3
wiederhole anzahl mal
  sag "Noch einmal!"
ende
```

Turtle-Grafik mit Schleifen

Schleifen sind perfekt für Zeichenaufgaben:

```
# Ein Quadrat zeichnen
wiederhole 4 mal
  vorwärts(100)
  rechts(90)
ende
```

```
# Ein Sechseck
wiederhole 6 mal
  vorwärts(80)
  rechts(60)
ende
```



Übung

Zeichne mit der Schildkröte einen gleichseitigen Dreieck. Tipp: Ein Dreieck hat 3 Seiten und an jeder Ecke wird um 120 Grad gedreht.

Lektion 6 — Bedingungsschleife: `solange / ende`

Das Konzept

`wiederhole` läuft eine feste Anzahl mal. `solange` läuft so lange, wie eine Bedingung wahr ist — wie ein Wecker, der klingelt, bis du ihn abstellst.

Syntax

```
solange Bedingung
  # Code der wiederholt wird
ende
```

Beispiele

```
zahl ist 1
solange zahl <= 5
  sag zahl
  zahl ist zahl + 1
ende
```

Ausgabe:

```
1
2
3
4
5
```

Ein Ratespiel als Beispiel:

```
geheimzahl ist 7
frage "Rate eine Zahl zwischen 1 und 10:" als tipp

solange tipp != geheimzahl
  sag "Falsch! Versuche es nochmal."
  frage "Nochmal raten:" als tipp
ende

sag "Richtig! Du hast gewonnen!"
```

Vorsicht: Endlosschleife

Wenn die Bedingung nie falsch wird, läuft die Schleife für immer. Das blockiert das Programm. Stelle sicher, dass sich der Wert in der Bedingung verändert:

```
# FALSCH — läuft ewig:
x ist 1
solange x > 0
  sag "Hilfe!"
  # x ändert sich nie!
ende

# RICHTIG:
x ist 10
solange x > 0
  sag x
  x ist x - 1
ende
```

Übung

Schreib ein Programm, das alle geraden Zahlen von 2 bis 20 ausgibt. Tipp: Eine Zahl ist gerade, wenn `zahl / 2 * 2 = zahl` .

Lektion 7 — Listen (Arrays)

Das Konzept

Eine Liste ist wie eine nummerierte Einkaufsliste. Du kannst mehrere Werte unter einem Namen speichern und einzeln darauf zugreifen.

Syntax

```
# Leere Liste erstellen
meineListe ist []

# Liste mit Werten erstellen
farben ist ["rot", "blau", "grün"]
zahlen ist [1, 2, 3, 4, 5]

# Auf ein Element zugreifen (Nummerierung startet bei 0!)
sag farben[0]   # → rot
sag farben[1]   # → blau
sag farben[2]   # → grün
```

Listen-Funktionen

Funktion	Bedeutung	Beispiel
`länge(liste)`	Anzahl der Elemente	`länge(farben)` → `3`
`anhängen(liste, wert)`	Element hinzufügen	`anhängen(farben, "gelb")`
`entfernen(liste, index)`	Element entfernen	`entfernen(farben, 0)`
`enthält(liste, wert)`	Prüfen ob enthalten	`enthält(farben, "rot")` → `wahr`
`finde(liste, wert)`	Position finden	`finde(farben, "blau")` → `1`

Beispiele

```
einkauf ist ["Milch", "Brot", "Butter"]
sag "Einkaufsliste:"
sag einkauf[0]
sag einkauf[1]
sag einkauf[2]

anhängen(einkauf, "Käse")
sag "Jetzt haben wir " + länge(einkauf) + " Dinge auf der Liste."
```

Liste mit Schleife durchgehen

```
noten ist [1, 2, 3, 2, 1]
i ist 0
solange i < länge(noten)
  sag "Note " + i + ": " + noten[i]
  i ist i + 1
ende
```

Übung

Erstelle eine Liste mit 5 Lieblingsspeisen und gib alle mit einer Schleife aus.

Lektion 8 — Funktionen: `funktion / ende`

Das Konzept

Eine Funktion ist ein wiederverwendbares Code-Paket. Du gibst ihr einen Namen, und überall dort, wo du diesen Namen aufrufst, wird der Code ausgeführt.

Stell dir vor, du hast ein Rezept. Du musst das Rezept nicht jedes Mal neu schreiben — du sagst einfach: "Mach das Rezept."

Syntax

```
funktion name()
  # Code
ende

# Aufrufen:
name()
```

Mit Parametern (Eingabewerten):

```
funktion name(parameter1, parameter2)
  # Code mit Parametern
ende

# Aufrufen:
name(wert1, wert2)
```

Beispiele

Eine einfache Funktion ohne Parameter:

```
funktion begrüße()
  sag "Hallo!"
  sag "Schön, dich zu sehen."
ende

begrüße()
begrüße()
```

Ausgabe:

```
Hallo!
Schön, dich zu sehen.
Hallo!
Schön, dich zu sehen.
```

Eine Funktion mit Parametern:

```
funktion begrüßeMit(name, alter)
  sag "Hallo, " + name + "!"
  sag "Du bist " + alter + " Jahre alt."
ende

begrüßeMit("Finn", 11)
begrüßeMit("Mia", 9)
```

Rückgabewerte: `zurück`

Eine Funktion kann auch ein Ergebnis zurückgeben. Das ist wie ein Taschenrechner — du gibst Zahlen ein und bekommst das Ergebnis zurück.

```
funktion addiere(a, b)
  ergebnis ist a + b
  zurück ergebnis
ende

summe ist addiere(10, 5)
sag "10 + 5 = " + summe

sag "20 + 30 = " + addiere(20, 30)
```

Ausgabe:

```
10 + 5 = 15
20 + 30 = 50
```

Praxisbeispiel: Einmaleins

```
funktion malRechnen(zahl)
  i ist 1
  solange i <= 10
    ergebnis ist zahl * i
    sag zahl + " × " + i + " = " + ergebnis
    i ist i + 1
  ende
ende

malRechnen(7)
```

Übung

Schreib eine Funktion `istGerade(zahl)`, die `wahr` zurückgibt, wenn die Zahl gerade ist, und `falsch` wenn sie ungerade ist.

Lektion 9 — Turtle-Grafik

Das Konzept

Die Turtle (Schildkröte) ist ein kleiner Zeichenroboter auf dem Canvas. Du sagst ihr, wie weit sie laufen soll und wie weit sie sich drehen soll — und sie hinterlässt dabei eine Linie.

Stell dir vor, du hältst einen Stift und läufst rückwärts über ein großes Blatt Papier. Du zeichnest dabei genau den Weg, den du gehst.

Befehle

Befehl	Bedeutung	Beispiel
`vorwärts(pixel)`	Vorwärts laufen	`vorwärts(100)`
`rückwärts(pixel)`	Rückwärts laufen	`rückwärts(50)`
`rechts(grad)`	Nach rechts drehen	`rechts(90)`
`links(grad)`	Nach links drehen	`links(45)`
`stiftHoch()`	Stift anheben (nicht zeichnen)	`stiftHoch()`
`stiftRunter()`	Stift absetzen (zeichnen)	`stiftRunter()`
`farbe("farbe")`	Farbe wechseln	`farbe("rot")`
`kreis(radius)`	Kreis zeichnen	`kreis(50)`
`quadrat(größe)`	Quadrat zeichnen	`quadrat(80)`
`langsam(1)`	Animation aktivieren	`langsam(1)`

Winkel verstehen

Die Turtle startet und schaut nach rechts (0 Grad). Drehungen gehen im Uhrzeigersinn:

- `rechts(90)` → schaut nach unten
- `rechts(180)` → schaut nach links
- `rechts(270)` oder `links(90)` → schaut nach oben

Beispiele

Ein Quadrat:

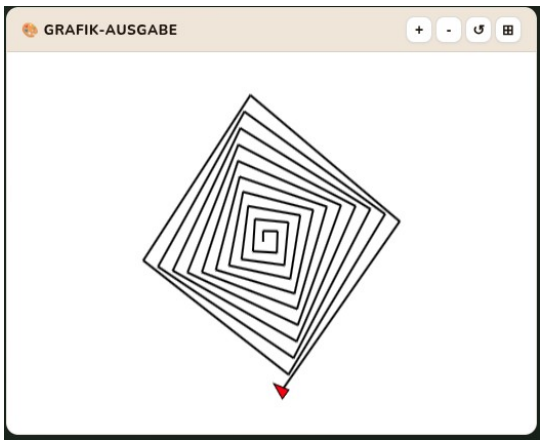
```
wiederhole 4 mal
  vorwärts(100)
  rechts(90)
ende
```

Ein buntes Muster:

```
farbe("rot")
vorwärts(80)
rechts(90)
farbe("blau")
vorwärts(80)
rechts(90)
farbe("grün")
vorwärts(80)
rechts(90)
farbe("gelb")
vorwärts(80)
rechts(90)
```

Ein Spiralmuster:

```
i ist 10
solange i <= 200
  vorwärts(i)
  rechts(91)
  i ist i + 5
ende
```



Mit Funktionen kombinieren:

```
funktion stern(größe)
  i ist 0
  solange i < 5
    vorwärts(größe)
    rechts(144)
    i ist i + 1
  ende
ende

farbe("gold")
stern(100)

stiftHoch()
vorwärts(150)
stiftRunter()

farbe("orange")
stern(60)
```

Zoom und Pan

Der Canvas unterstützt Zoom und Verschieben:

- **Mausrad drehen:** Rein- oder rauszoomen
- **Klicken und ziehen:** Canvas verschieben
- **"Alles anzeigen":** Automatisch auf Zeichnung zoomen

Übung

Zeichne eine Reihe von 5 Kreisen nebeneinander. Jeder Kreis soll eine andere Farbe haben. Tipp: Benutze `stiftHoch()` zum Bewegen und `stiftRunter()` zum Zeichnen.

Lektion 10 — Eingebaute Mathematik-Funktionen

Verfügbare Funktionen

Funktion	Bedeutung	Beispiel	Ergebnis
`zufallsZahl(max)`	Zufallszahl von 0 bis max-1	`zufallsZahl(6)`	z.B. `3`
`wurzel(zahl)`	Quadratwurzel	`wurzel(25)`	`5`
`abs(zahl)`	Absolutwert (immer positiv)	`abs(-7)`	`7`

Beispiele

```
# Würfeln simulieren
wurf ist zufallsZahl(6) + 1
sag "Du hast gewürfelt: " + wurf

# Hypotenuse berechnen
a ist 3
b ist 4
c ist wurzel(a*a + b*b)
sag "Hypotenuse: " + c
```

Zusammenfassung

Du kennst jetzt alle Bausteine der CodeFuchs-Sprache:

Baustein	Befehl	Wofür
Ausgabe	`sag`	Text und Werte anzeigen
Variable	`ist`	Werte speichern
Eingabe	`frage ... als`	Benutzereingabe
Bedingung	`wenn / dann / sonst / ende`	Entscheidungen
Zählschleife	`wiederhole / mal / ende`	N-mal wiederholen
Bedingungsschleife	`solange / ende`	Solange wiederholen wie...
Liste	`[]` + Listenfunktionen	Mehrere Werte speichern
Funktion	`funktion / ende` + `zurück`	Code wiederverwenden
Turtle	`vorwärts`, `rechts`, ...	Zeichnen

Als nächstes: **Kapitel 02** erklärt dir alle Knöpfe und Bereiche der Oberfläche. **Kapitel 03** zeigt dir, wie du mit GameFuchs eigene Spiele baust.

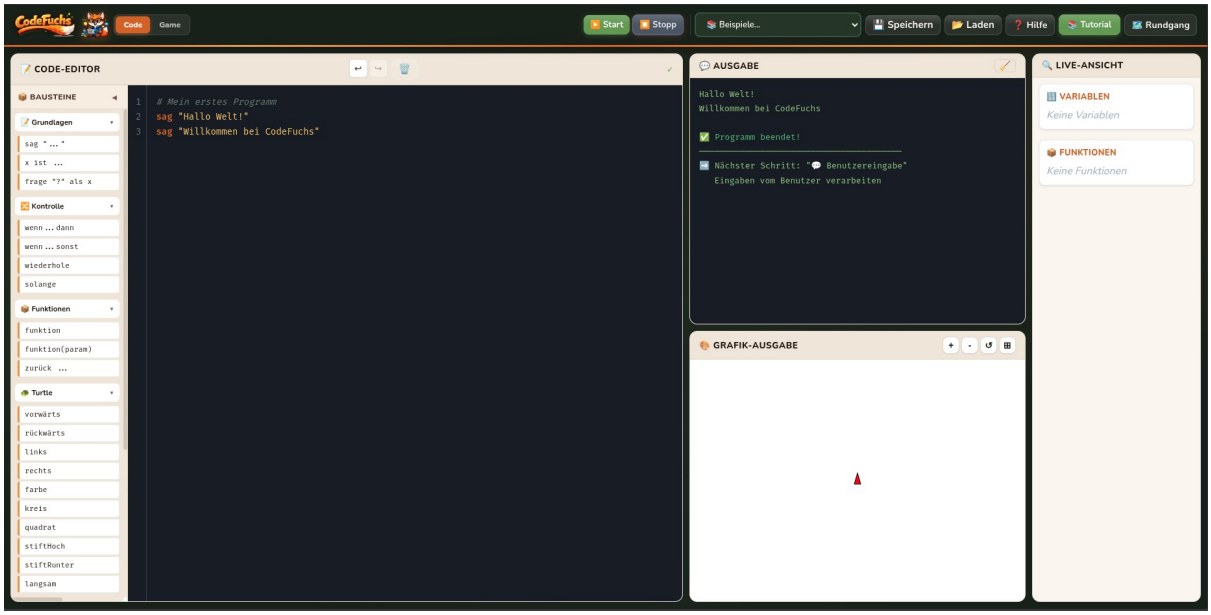
Tipp: Das Hilfe-System der IDE (Button) enthält alle Befehle auf einen Blick — ideal zum Nachschlagen während du programmierst.

CodeFuchs Handbuch — Kapitel 02: UI-Guide CodeFuchs

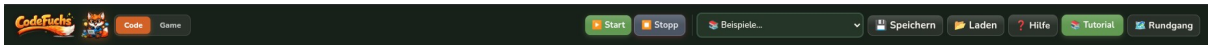
Dieses Kapitel erklärt jeden Bereich und jeden Knopf der CodeFuchs-Oberfläche. Du musst das nicht auswendig lernen — nutze es als Nachschlagewerk, wenn du einen Knopf nicht kennst.

Überblick: Die vier Bereiche

Wenn du CodeFuchs öffnest, siehst du vier Hauptbereiche:

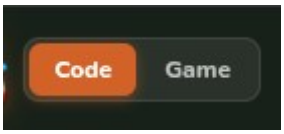


Der Header (Oben)



Logo und Modus-Umschalter

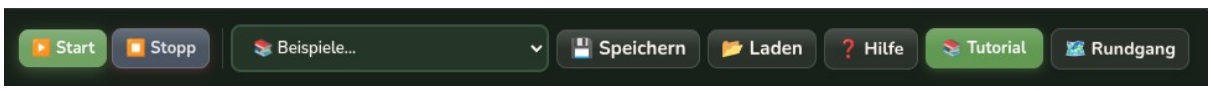
Links oben siehst du das CodeFuchs-Logo. Daneben gibt es zwei Buttons:



- **Code** — aktiviert den normalen CodeFuchs-Modus (Turtle-Grafik, Textausgabe)
- **Game** — wechselt zu GameFuchs (Spielmodus mit Sprites und Spielfeld)

Steuerungs-Buttons

Button	Tastenkürzel	Funktion
▶ Start	—	Führt den Code im Editor aus
Stopp	—	Bricht ein laufendes Programm ab
Beispiele...	—	Öffnet ein Dropdown mit Beispielprogrammen
Speichern	Strg+S	Speichert das aktuelle Projekt im Browser
Laden	—	Lädt ein gespeichertes Projekt
Hilfe	—	Öffnet das Hilfe-Fenster mit allen Befehlen
Tutorial		Öffnet die Tutorial Sidebar
Rundgang		Zeigt den UI Rundgang erneut an



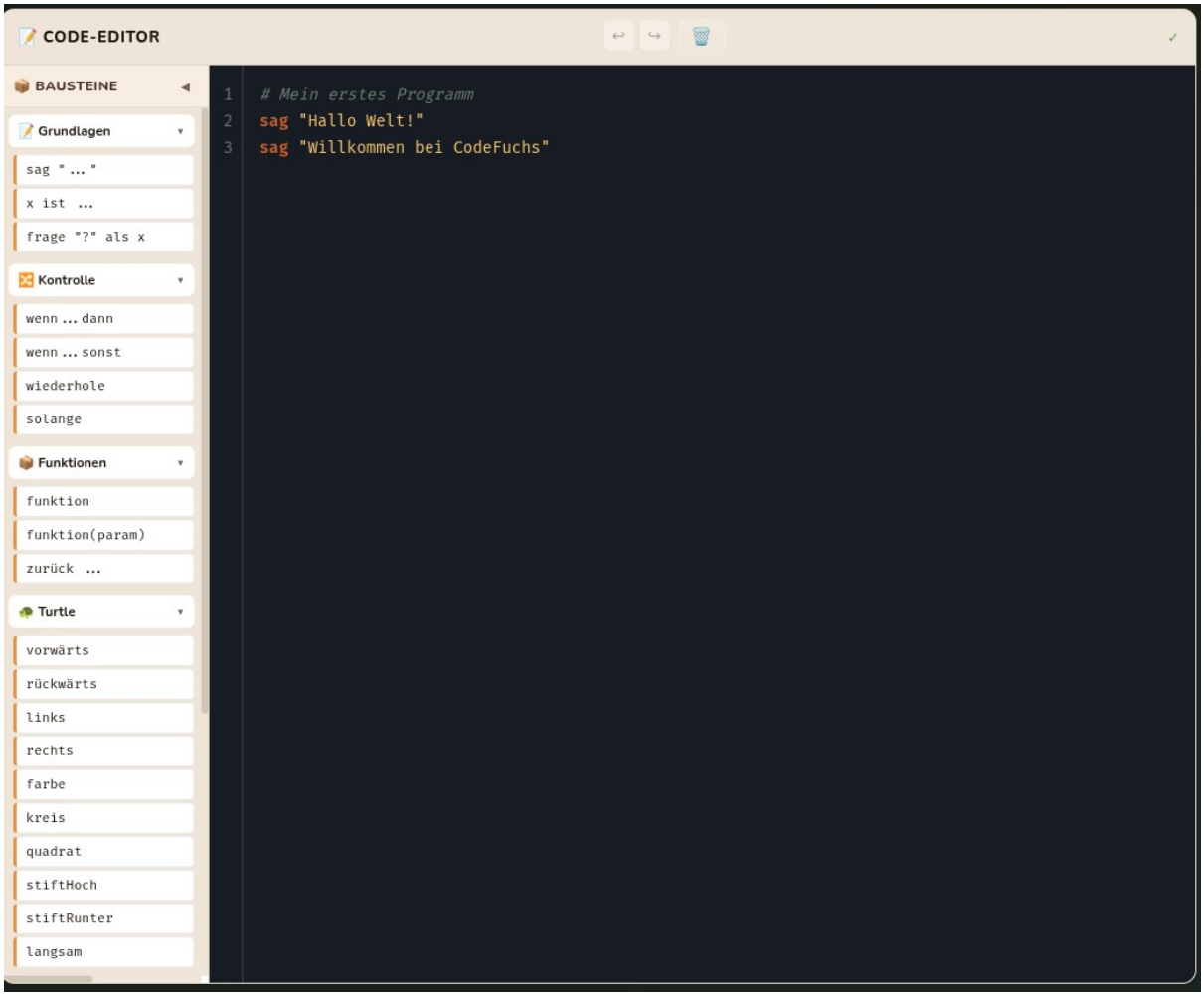
So führst du Code aus

- Schreibe deinen Code in den Editor
- Klicke ▶ **Start**
- Schau rechts in den Ausgabebereich und auf den Canvas

Wenn ein Fehler passiert, erscheint eine Fehlermeldung mit einer Erklärung auf Deutsch.

Der Code-Editor (Links)

Der Editor ist das Textfeld, in dem du deinen Code schreibst. Er sieht aus wie ein normales Schreibprogramm — aber er versteht CodeFuchs.



Syntax-Highlighting

Der Editor färbt deinen Code automatisch ein:

Farbe	Bedeutung
Orange/Gelb	Schlüsselwörter (`sag` , `wenn` , `funktion` , ...)
Grün	Texte (alles in Anführungszeichen)
Hellblau	Zahlen
Grau/Kursiv	Kommentare (Zeilen mit `#`)
Weiß/Hell	Variablennamen und Funktionsaufrufe

Zeilennummern und Fehleranzeige

Links im Editor siehst du die Zeilennummern. Wenn ein Fehler in einer Zeile ist, wird diese Zeile rot markiert — so findest du Fehler sofort.

Automatische Einrückung

Wenn du nach `wenn ... dann` , `wiederhole ... mal` , `solange` , `funktion` oder `spielSchleife` Enter drückst, rückt der Editor die nächste Zeile automatisch ein. Das macht den Code lesbarer.

Editor-Toolbar

Über dem Editor gibt es kleine Buttons:

Button	Tastenkürzel	Funktion
↩ Undo	Strg+Z	Letzte Änderung rückgängig machen
↪ Redo	Strg+Y	Rückgängig gemachte Änderung wiederholen
** Leeren**	—	Gesamten Code löschen (mit Bestätigung)

Tastaturkürzel im Editor

Taste	Funktion
`Tab`	Einrückung um 2 Leerzeichen
`Enter`	Neue Zeile mit automatischer Einrückung
`Strg+Z` / `Cmd+Z`	Undo
`Strg+Y` / `Cmd+Y`	Redo
`Strg+S` / `Cmd+S`	Speichern

Die Bausteine-Palette

Auf der linken Seite des Editors findest du die **Bausteine-Palette** — ein aufklappbares Menü mit fertigen Code-Schnipseln.

So verwendest du die Bausteine

Methode 1 — Klicken:

Klicke auf einen Baustein. Der Code-Schnipsel wird an der Cursor-Position im Editor eingefügt.

Methode 2 — Drag & Drop:

Ziehe einen Baustein per Maus direkt an die gewünschte Stelle im Editor.

Baustein-Kategorien (Code-Modus)

Grundlagen

- `sag "..."` — Text ausgeben
- `x ist ...` — Variable erstellen
- `frage "?" als x` — Benutzereingabe holen

Kontrolle

- `wenn ... dann ... ende` — Einfache Bedingung
- `wenn ... dann ... sonst ... ende` — Bedingung mit Alternative
- `wiederhole X mal ... ende` — Zählschleife
- `solange ... ende` — Bedingungsschleife

Funktionen

- `funktion name() ... ende` — Funktion ohne Parameter
- `funktion name(param) ... ende` — Funktion mit Parameter
- `zurück ...` — Wert zurückgeben

Turtle-Grafik

- `vorwärts()` , `rückwärts()` , `links()` , `rechts()`
- `farbe("")` , `stiftHoch()` , `stiftRunter()`
- `kreis()` , `quadrat()`
- `langsam()` — Animation einschalten

Mathe

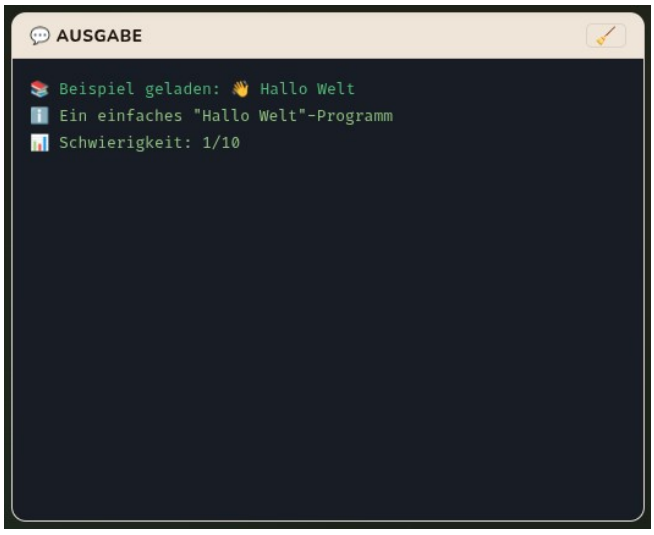
- `zufallsZahl()` , `wurzel()` , `abs()`

Listen

- `... ist []` — Leere Liste
- `... ist [..., ...]` — Liste mit Werten
- `länge()` , `anhängen()` , `entfernen()`

Der Ausgabebereich (Rechts oben)

Hier erscheinen alle Ausgaben deines Programms — alles, was du mit `sag` ausgibst.

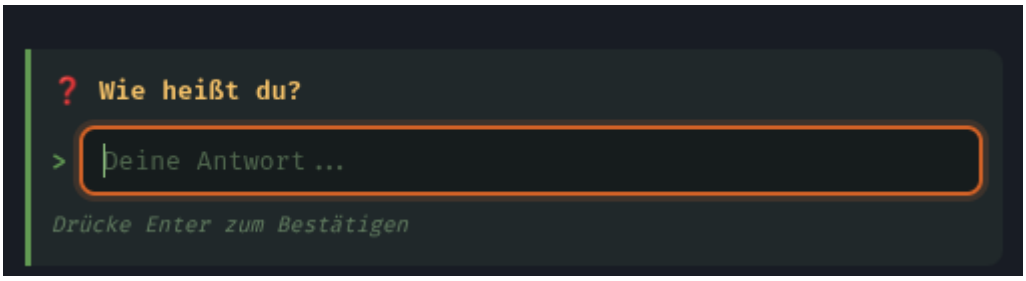


Was du hier siehst

- **Normale Ausgabe** — weiße Textzeilen (von `sag`)
- **Eingabe-Aufforderung** — wenn `frage` aufgerufen wird, erscheint das Eingabefeld
- **Fehlermeldungen** — rote oder orangefarbene Zeilen mit Erklärung

Das Eingabefeld

Wenn dein Programm `frage` verwendet, erscheint unten im Ausgabebereich ein Textfeld:



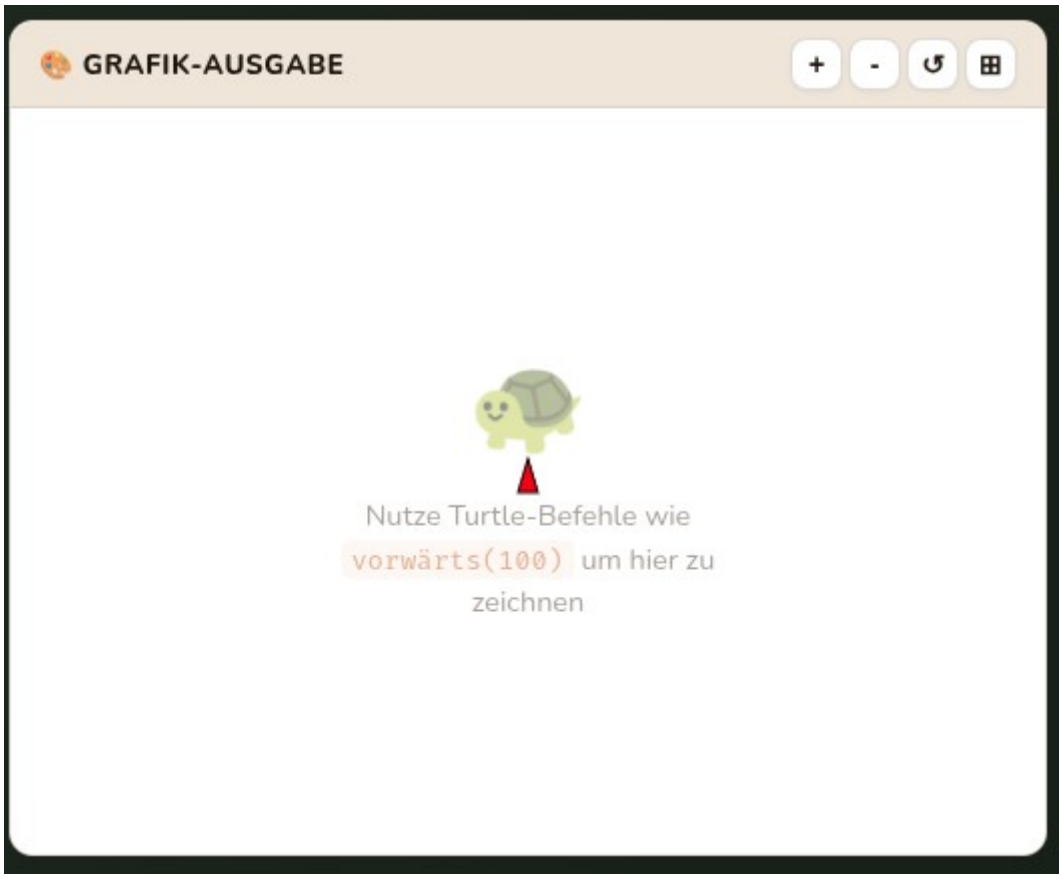
Tippe deine Antwort und drücke **OK** oder **Enter**.

Ausgabe leeren

Der **-Button** oben rechts im Ausgabebereich löscht alle bisherigen Ausgaben.

Der Grafik-Canvas (Rechts unten)

Hier erscheinen alle Zeichnungen der Turtle-Grafik. Der Canvas ist ein großes weißes Blatt mit einem Raster.



Die Schildkröte

Die kleine grüne Figur auf dem Canvas ist die Schildkröte. Sie zeigt an:

- **Position:** Wo sich die Turtle gerade befindet
- **Richtung:** Wohin die Turtle schaut (und sich beim nächsten `vorwärts` bewegt)

Unter dem Canvas siehst du die aktuellen Koordinaten:

```
X: 200    Y: 150    Richtung: 0°
```

Zoom und Navigation

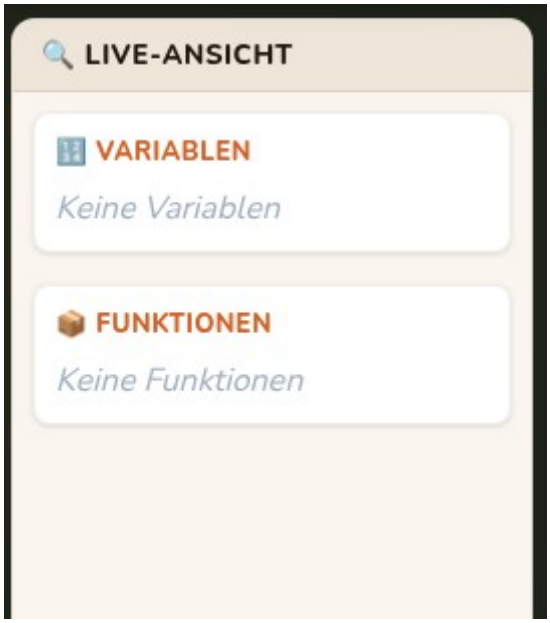
Aktion	Funktion
Mausrad drehen	Zoom rein / raus
Linke Maustaste + ziehen	Canvas verschieben
** + Button**	Reinzoomen
** − Button**	Rauszoomen
↺ Button	Ansicht zurücksetzen
⊞ Button	Auf die gesamte Zeichnung zoomen

Canvas zurücksetzen

Wenn du **Stopp** klickst oder das Programm neu startest, wird der Canvas geleert und die Schildkröte kehrt in die Mitte zurück.

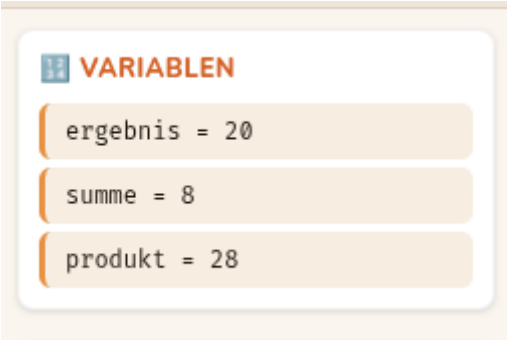
Die Sidebar (Rechts, Variablen und Funktionen)

Die Sidebar zeigt dir in Echtzeit, was in deinem Programm passiert.



Variablen-Bereich

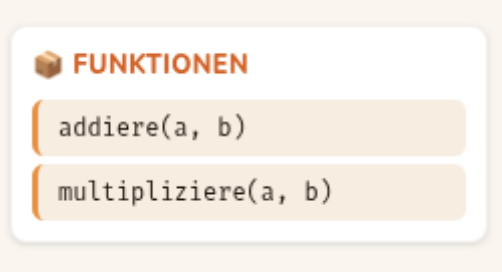
Alle Variablen, die dein Programm erstellt, erscheinen hier mit ihrem aktuellen Wert:



Das ist sehr nützlich zum Debuggen — du siehst sofort, was in jeder Variable gespeichert ist.

Funktionen-Bereich

Alle definierten Funktionen erscheinen hier mit ihren Parametern:



Das Hilfe-System

Das **-Symbol** oben öffnet das Hilfe-Fenster. Hier findest du alle Befehle erklärt — mit Beispielen und Code-Snippets zum Kopieren.

Navigation im Hilfe-Fenster

Das Hilfe-Fenster hat Tabs für verschiedene Themen:

```
   Start |    Grundlagen |    Variablen |    Listen |
   Bedingungen |    Schleifen |    Funktionen |    Turtle-Grafik |
Referenz
```

Code-Beispiele kopieren

Jeder Code-Block im Hilfe-Fenster hat einen **-Button**. Ein Klick kopiert den Code in die Zwischenablage — dann einfach im Editor einfügen.

Das Beispiel-Dropdown

Der **Beispiele...-Button** öffnet eine Liste mit fertigen Programmen, geordnet nach Schwierigkeit:

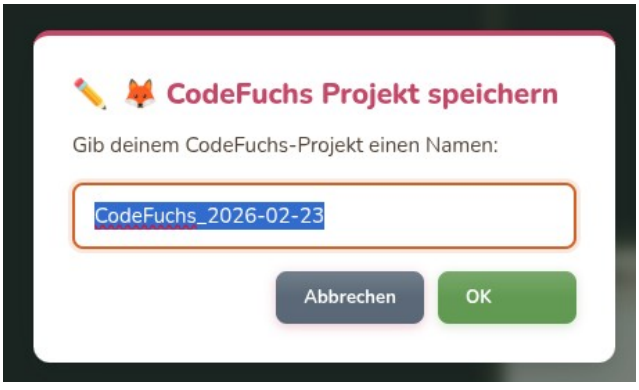
Kategorie	Beispiele
Grundlagen	Hallo Welt, Rechnen, Benutzereingabe
Kontrolle	Bedingungen, Zählschleife, Bedingungsschleife
Funktionen	Listen, Einfache Funktion, Funktion mit Rückgabe
Grafik	Quadrat, Stern, Spirale, Fraktal-Baum
Projekte	Zahlen-Ratespiel, Einmaleins-Trainer

Ein Klick auf ein Beispiel lädt den Code direkt in den Editor. Der bisherige Code wird dabei ersetzt (falls du einen wichtigen Code hast, erst speichern!).

Speichern und Laden

Projekt speichern

Klicke **Speichern** oder drücke **Strg+S**. Ein Dialog fragt nach einem Namen für dein Projekt:



Der Code wird im Browser gespeichert — er ist auch nach dem Schließen noch da.

Projekt laden

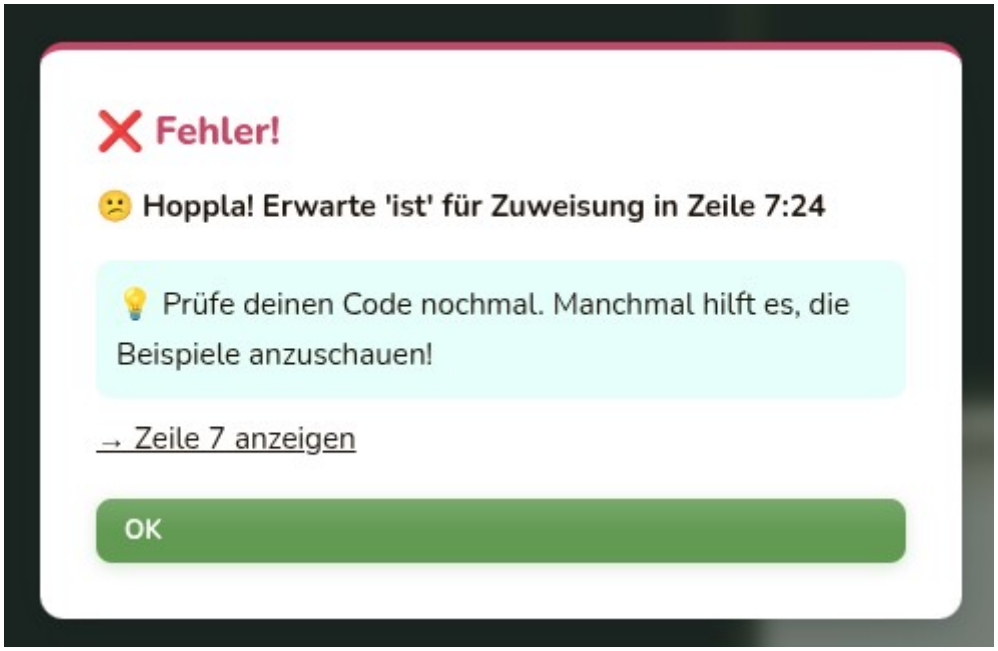
Klicke **Laden**. Eine Liste aller gespeicherten Projekte erscheint:

Klicke auf einen Eintrag, um ihn zu laden.

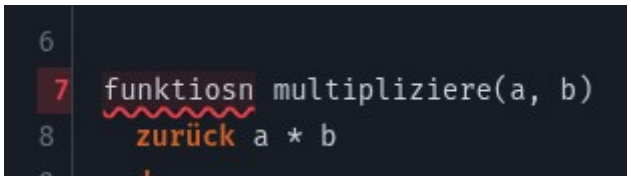
Wichtig: Der aktuelle Code im Editor wird dabei ersetzt. Vorher speichern, wenn nötig.

Fehlermeldungen verstehen

Wenn dein Code einen Fehler enthält, erscheint ein rotes Fenster:



Gleichzeitig wird die fehlerhafte Zeile im Editor rot markiert.



Häufige Fehler:

Fehlermeldung	Bedeutung	Lösung
Unbekannter Befehl	Tippfehler im Befehlsnamen	Schreibweise prüfen
`ende` fehlt	Ein Block wurde nicht geschlossen	`ende` am Ende des Blocks ergänzen
Variable nicht gefunden	Variable benutzt, bevor sie definiert wurde	Variable zuerst mit `ist` anlegen
Erwarte `dann`	`wenn` ohne `dann` geschrieben	Nach der Bedingung `dann` einfügen

Kurz-Übersicht: Alle Tastaturkürzel

Shortcut	Funktion
`Strg+S` / `Cmd+S`	Speichern
`Strg+Z` / `Cmd+Z`	Undo
`Strg+Y` / `Cmd+Y`	Redo
`Tab`	Einrückung
`Enter`	Neue Zeile mit Auto-Indent

*Weiter zu **Kapitel 03 — Grundlagen GameFuchs**, um zu lernen, wie du eigene Spiele programmierst.*

CodeFuchs Handbuch — Kapitel 03: Grundlagen GameFuchs

GameFuchs ist die Spiele-Werkstatt innerhalb von CodeFuchs. Du programmierst 2D-Spiele mit denselben deutschen Befehlen — und kannst sie am Ende als eine einzige Datei exportieren und mit Freunden teilen.

Voraussetzung: Du hast Kapitel 01 durchgearbeitet. Du kennst `wenn`, `solange`, `funktion` und `ist`.

Was ist der Unterschied zu CodeFuchs?

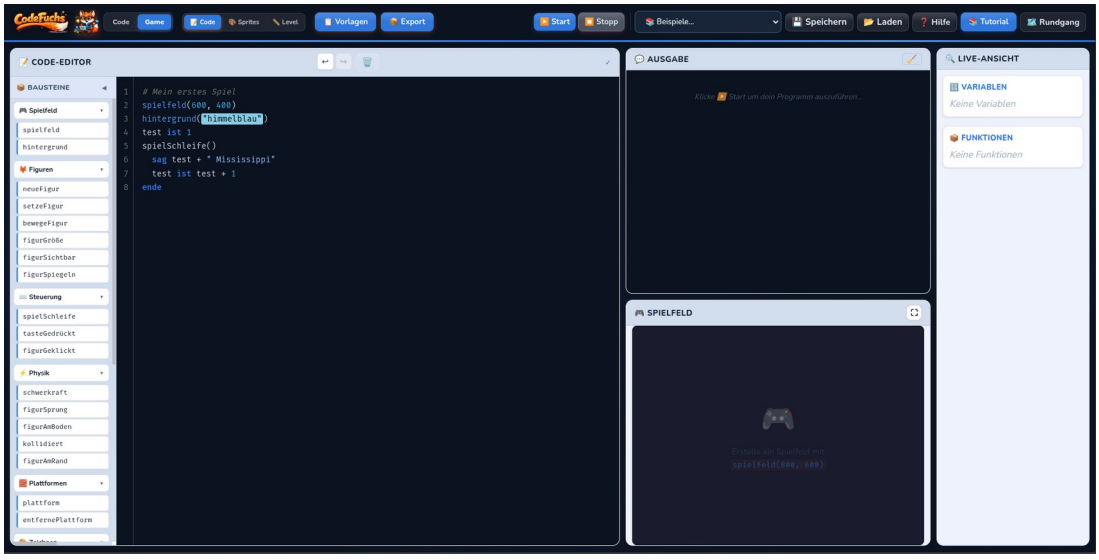
In CodeFuchs läuft dein Programm einmal von oben nach unten und hört dann auf. In GameFuchs läuft dein Programm in einer **Schleife** — 60 Mal pro Sekunde. Das ist notwendig für ein Spiel: Figuren bewegen sich, Tasten werden gedrückt, Kollisionen werden geprüft — das alles passiert jede Sekunde dutzende Male.

Das zentrale neue Konzept heißt **Spielschleife**.

Schritt 1: Modus wechseln

Klicke oben links im Header auf **[Game]**.

Die Oberfläche ändert sich: Der Turtle-Canvas verschwindet, stattdessen erscheint der **Game-Canvas**. In der Sidebar gibt es jetzt einen **Sprite-Manager** statt der Variablen-Ansicht.



Lektion 1 — Das Spielfeld einrichten

Das Spielfeld

Jedes Spiel beginnt mit dem Spielfeld. Du bestimmst seine Größe und Farbe:

```
spielfeld(400, 300)
hintergrund("himmelblau")
```

- `spielfeld(breite, höhe)` — Größe in Pixeln
- `hintergrund("farbe")` — Hintergrundfarbe (alle deutschen Farbwörter und Hex-Codes funktionieren)

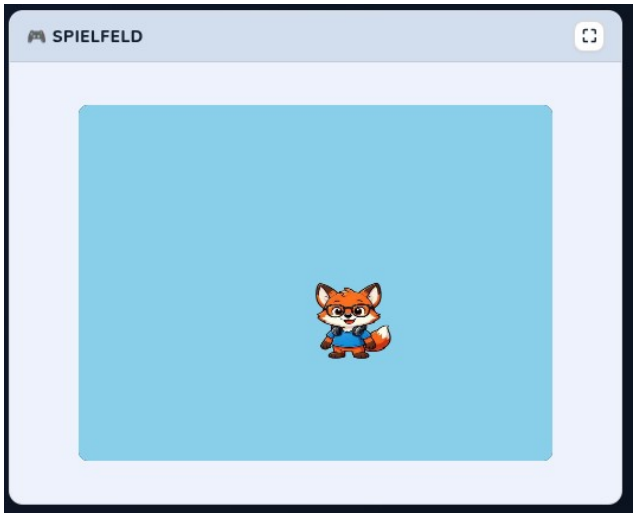
Erste Figur erstellen

Eine Figur (auch Sprite genannt) ist ein Bild, das sich auf dem Spielfeld bewegen kann. Du erstellst sie mit `neueFigur`:

```
spielfeld(400, 300)
hintergrund("himmelblau")

held ist neueFigur("fuchs")
setzeFigur(held, 200, 150)
```

- `neueFigur("fuchs")` — erstellt eine neue Figur mit dem Bild "fuchs", gibt eine ID zurück
- `setzeFigur(held, 200, 150)` — platziert die Figur an Position X=200, Y=150

Führe diesen Code aus. Der Fuchs erscheint auf dem Canvas.



Koordinatensystem

Das Koordinatensystem startet oben links:

- **X** wächst nach rechts (X=0 ist der linke Rand)
- **Y** wächst nach unten (Y=0 ist der obere Rand)

```
(0, 0)──────────────→ X
  │
  │      (200, 150) ← Fuchs ist hier
  │
  ↓ Y
```

Lektion 2 — Die Spielschleife

Das Konzept

Die Spielschleife ist der Motor jedes Spiels. Alles, was sich im Spiel bewegt oder verändert, passiert innerhalb der Spielschleife. Stell dir einen Filmprojektor vor: Er zeigt 60 Bilder pro Sekunde. Zwischen jedem Bild kann das Programm Figuren bewegen, Eingaben verarbeiten und Punkte zählen. Der Spieler sieht das als fließende Bewegung.

Syntax

```
spielSchleife()
  # Dieser Code wird 60x pro Sekunde ausgeführt
ende
```

Grundstruktur eines Spiels

Jedes GameFuchs-Spiel folgt diesem Muster:

```
# 1. SETUP – läuft einmal
spielfeld(400, 300)
hintergrund("dunkelblau")
held ist neueFigur("fuchs")
setzeFigur(held, 50, 150)

# 2. SPIELSCHLEIFE – läuft 60x pro Sekunde
spielSchleife()

  # Eingabe verarbeiten
  wenn tasteGedrückt("rechts") dann
    bewegeFigur(held, 3, 0)
  ende

  # Anzeige aktualisieren
  zeigeText("Bewege den Fuchs!", 10, 20)

ende
```

Lektion 3 — Tastatur-Steuerung

Der `tasteGedrückt`-Befehl

`tasteGedrückt("tastenname")` gibt `wahr` zurück, solange die Taste gedrückt ist.

Wichtig: Dieser Befehl funktioniert nur *innerhalb* der Spielschleife.

Tastennamen

CodeFuchs-Name	Taste
`"links"`	Pfeil links ←
`"rechts"`	Pfeil rechts →
`"hoch"`	Pfeil hoch ↑
`"runter"`	Pfeil runter ↓
`"leertaste"`	Leertaste
`"eingabe"`	Enter
`"escape"`	Escape
`"a"` bis `"z"`	Buchstaben A–Z
`"1"` bis `"9"`	Zahlen 1–9

Vollständige 4-Richtungs-Steuerung

```
spielfeld(400, 300)
hintergrund("grün")
held ist neueFigur("fuchs")
setzeFigur(held, 200, 150)

spielSchleife()

  wenn tasteGedrückt("links") dann
    bewegeFigur(held, -4, 0)
    figurSpiegeln(held, wahr)
  ende

  wenn tasteGedrückt("rechts") dann
    bewegeFigur(held, 4, 0)
    figurSpiegeln(held, falsch)
  ende
```

```
  wenn tasteGedrückt("hoch") dann
    bewegeFigur(held, 0, -4)
  ende

  wenn tasteGedrückt("runter") dann
    bewegeFigur(held, 0, 4)
  ende

  zeigeText("Bewege den Fuchs!", 10, 20)

ende
```

- `bewegeFigur(held, dx, dy)` — bewegt die Figur relativ zur aktuellen Position
  - `dx=4` → 4 Pixel nach rechts
  - `dx=-4` → 4 Pixel nach links
  - `dy=-4` → 4 Pixel nach oben (Y wird kleiner!)
- `figurSpiegeln(held, wahr)` — spiegelt die Figur horizontal (nützlich für Richtungswechsel)

Lektion 4 — Kollisionserkennung

Zwei Figuren kollidieren

```
kollidiert(figur1, figur2)
```

Gibt `wahr` zurück, wenn sich die Rechtecke der beiden Figuren überschneiden.

Beispiel: Münze sammeln

```
spielfeld(400, 300)
hintergrund("dunkelblau")

held ist neueFigur("fuchs")
setzeFigur(held, 50, 150)

münze ist neueFigur("robert")
setzeFigur(münze, 300, 150)

punkte ist 0

spielSchleife()

  # Steuerung
  wenn tasteGedrückt("rechts") dann
    bewegeFigur(held, 4, 0)
```

```
  ende
  wenn tasteGedrückt("links") dann
    bewegeFigur(held, -4, 0)
  ende

  # Kollision prüfen
  wenn kollidiert(held, münze) dann
    punkte ist punkte + 1
    setzeFigur(münze, zufallsZahl(360), zufallsZahl(260))
  ende

  # Punktestand anzeigen
  textFarbe("weiß")
  zeigeText("Punkte: " + punkte, 10, 25)

ende
```

Randkollision

`figurAmRand(figur)` gibt `wahr` zurück, wenn die Figur den Spielfeld-Rand berührt:

```
wenn figurAmRand(held) dann
  sag "Rand erreicht!"
  spielEnde("Du bist rausgelaufen!")
ende
```

Lektion 5 — Text und Anzeige

Text im Spiel anzeigen

```
zeigeText(text, x, y)
zeigeText(text, x, y, größe, farbe)
```

- `x`, `y` — Position des Textes (oben links des Textes)
- `größe` — Schriftgröße in Pixeln (optional, Standard: 20)
- `farbe` — Textfarbe (optional)

```
zeigeText("Punkte: " + punkte, 10, 30)
zeigeText("SPIELENDE", 150, 150, 40, "rot")
```

Schriftgröße und Farbe setzen

Du kannst auch Standard-Einstellungen setzen, die dann für alle folgenden `zeigeText` -Aufrufe gelten:

```
textGröße(24)
textFarbe("weiß")
zeigeText("Level 1", 10, 30)
zeigeText("Punkte: " + punkte, 10, 60)
```

Formen zeichnen

```
zeichneRechteck(x, y, breite, höhe, farbe)
zeichneKreis(x, y, radius, farbe)
```

Wichtig: Zeichnungen werden am Anfang jedes Frames gelöscht. Du musst sie in der Spielschleife jedes Mal neu zeichnen.

```
spielSchleife()
  zeichneRechteck(10, 10, 200, 20, "dunkelgrün")   # Energie-Balken
  zeichneRechteck(10, 10, leben * 20, 20, "grün")  # Gefüllter Anteil
  zeigeText("Leben: " + leben, 220, 25)
ende
```

Lektion 6 — Figur-Eigenschaften

Eigenschaften lesen und setzen

Du kannst direkt auf die Position einer Figur zugreifen:

```
sag held.x    # X-Position des Fuchses
sag held.y    # Y-Position
sag held.w    # Breite
sag held.h    # Höhe
```

Das ist nützlich, um zu prüfen wo eine Figur gerade ist:

```
# Figur soll nicht aus dem Spielfeld laufen
wenn held.x < 0 dann
  setzeFigur(held, 0, held.y)
ende
wenn held.x > 370 dann
  setzeFigur(held, 370, held.y)
ende
```

Figur-Größe ändern

```
figurGröße(held, 64, 64)   # Breite 64px, Höhe 64px
```

Figur-Sichtbarkeit

```
figurSichtbar(münze, falsch)  # Münze verstecken
figurSichtbar(münze, wahr)    # Münze zeigen
```

Figur-Bild wechseln

```
figurBild(held, "fuchs_winkt")   # Anderen Sprite-Frame laden
```

Lektion 7 — Schwerkraft und Plattformer

Schwerkraft aktivieren

`schwerkraft(figur, stärke)` aktiviert die Schwerkraft für eine Figur. Die Figur fällt automatisch nach unten.

```
spielfeld(400, 300)
hintergrund("himmelblau")
held ist neueFigur("fuchs")
setzeFigur(held, 50, 50)
schwerkraft(held, 0.5)
```

Plattformen erstellen

```
plattform(x, y, breite, höhe, farbe)
```

Plattformen sind unsichtbare (oder gefärbte) Rechtecke, auf denen Figuren mit Schwerkraft landen:

```
plattform(0, 280, 400, 20, "braun")    # Boden
plattform(100, 200, 120, 15, "grün")   # Schwebende Plattform
plattform(250, 150, 80, 15, "grün")    # Höhere Plattform
```

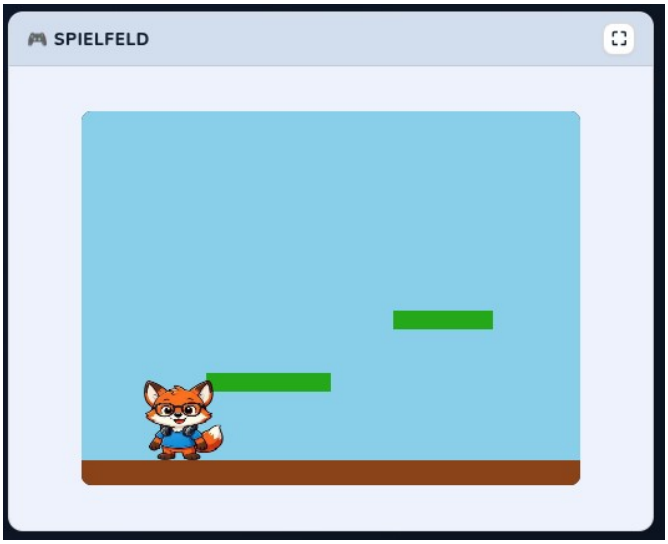
Springen

```
figurSprung(figur, kraft)
```

Lässt die Figur nach oben springen. Funktioniert nur wenn die Figur am Boden ist.

Vollständiges Plattformer-Beispiel

```
spielfeld(400, 300)
hintergrund("himmelblau")

held ist neueFigur("fuchs")
setzeFigur(held, 50, 200)
schwerkraft(held, 0.5)

plattform(0, 280, 400, 20, "braun")
plattform(100, 210, 100, 15, "grün")
plattform(250, 160, 80, 15, "grün")

spielSchleife()

  wenn tasteGedrückt("links") dann
    bewegeFigur(held, -3, 0)
    figurSpiegeln(held, wahr)
  ende

  wenn tasteGedrückt("rechts") dann
    bewegeFigur(held, 3, 0)
    figurSpiegeln(held, falsch)
  ende

  wenn tasteGedrückt("leertaste") und figurAmBoden(held) dann
    figurSprung(held, 10)
  ende

ende
```



Lektion 8 — Sounds

Sounds abspielen

```
spieleSound("name")
```

Verfügbare Sounds:

Name	Beschreibung
`"sprung"`	Hüpfgeräusch
`"punkt"`	Punkte sammeln
`"treffer"`	Getroffen werden
`"fehler"`	Fehler / Verloren
`"gewonnen"`	Sieg-Fanfare
`"verloren"`	Niederlage
`"klick"`	Klick-Geräusch
`"münze"`	Münze einsammeln

```
wenn kollidiert(held, münze) dann
  punkte ist punkte + 1
  spieleSound("münze")
  setzeFigur(münze, zufallsZahl(380), zufallsZahl(280))
ende
```

Lektion 9 — Partikel-Effekte

Partikel sind kleine animierte Punkte, die Effekte wie Explosionen oder Konfetti erzeugen.

Vordefinierte Effekte

```
partikel(x, y, typ)
```

Typ	Aussehen
`"explosion"`	Orange/gelbe Explosion
`"funken"`	Goldene Funken
`"rauch"`	Aufsteigender Rauch
`"konfetti"`	Buntes Konfetti
`"feuer"`	Aufsteigende Flammen
`"sterne"`	Glitzernde Sterne
`"wasser"`	Wasser-Spritzer

```
wenn kollidiert(held, gegner) dann
  partikel(gegner.x, gegner.y, "explosion")
  spieleSound("treffer")
  leben ist leben - 1
ende
```

Eigene Partikel

```
partikelEffekt(x, y, anzahl, tempo, dauer, größe, farbe)
```

- `anzahl` — Wie viele Partikel
- `tempo` — Geschwindigkeit (1–10)
- `dauer` — Wie lange sie sichtbar sind (in Frames)
- `größe` — Partikelgröße in Pixeln
- `farbe` — Farbe

```
partikelEffekt(200, 150, 30, 5, 60, 4, "gold")
```

Lektion 10 — Spielsteuerung

Spiel beenden

```
spielEnde("Nachricht")
```

Stoppt die Spielschleife und zeigt einen Endscreen mit der Nachricht:

```
wenn punkte >= 10 dann
  spielEnde("Du hast gewonnen! " + punkte + " Punkte!")
ende
```

Spiel pausieren und fortsetzen

```
spielPause()    # Schleife pausieren
spielWeiter()   # Fortsetzen
```

Spiel neu starten

```
spielNeustart()   # Gesamten Code von vorn ausführen
```

Spielzeit abfragen

```
zeit ist spielZeit()   # Sekunden seit Spielstart
zeigeText("Zeit: " + zeit + "s", 10, 30)
```

Lektion 11 — Klick-Interaktion

Figur-Klick erkennen

`figurGeklickt(figur)` gibt `wahr` zurück, wenn die Maus oder ein Finger auf die Figur getippt hat:

```
spielSchleife()

  wenn figurGeklickt(knopf) dann
    punkte ist punkte + 1
    spieleSound("klick")
  ende

  zeigeText("Klicks: " + punkte, 10, 30)

ende
```

Das ist die Basis für Klicker-Spiele.

Lektion 12 — Das Spiel exportieren

Wenn dein Spiel fertig ist, kannst du es als eine einzige HTML-Datei exportieren. Diese Datei läuft in jedem Browser — ohne Installation, ohne Internet, ohne CodeFuchs.

So exportierst du

- Klicke oben auf **Export**
- Gib deinem Spiel einen Namen
- Klicke **Herunterladen**
- Die Datei `mein-spiel.html` wird heruntergeladen
- Öffne sie im Browser — dein Spiel läuft!

Du kannst diese Datei:

- Per E-Mail verschicken
- Auf einen USB-Stick kopieren
- Auf eine Website hochladen

Zusammenfassung: Aufbau eines GameFuchs-Programms

```
# PHASE 1: Setup (läuft einmal)
spielfeld(breite, höhe)
hintergrund("farbe")
figur ist neueFigur("assetName")
setzeFigur(figur, x, y)
# ... weitere Figuren, Plattformen, Variablen

# PHASE 2: Spielschleife (läuft 60x/Sekunde)
spielSchleife()

  # Eingabe
  wenn tasteGedrückt("links") dann ... ende
  wenn figurGeklickt(figur) dann ... ende

  # Spiellogik
  wenn kollidiert(a, b) dann ... ende
  wenn figur.x > 400 dann ... ende

  # Physik (automatisch durch schwerkraft())
  # Plattform-Kollision (automatisch)

  # Anzeige
  zeigeText("Punkte: " + punkte, 10, 30)
  zeichneRechteck(...)

  # Spielende prüfen
  wenn punkte >= 10 dann
    spielEnde("Gewonnen!")
  ende

ende
```

Lernpfad: Beispiele nach Schwierigkeit

Die eingebauten GameFuchs-Beispiele helfen dir Schritt für Schritt:

Schwierigkeit	Beispiel	Lernziel
1	Erste Schritte	Spielfeld und Figur
2	Spielschleife	60 FPS Loop verstehen
3	Tastatur	Steuerung
4	Kollision	Kollisionserkennung
5	Eigenschaften	figur.x, figur.y
6	Punkte sammeln	Vollständiges Mini-Spiel
7	Plattformer	Schwerkraft, Sprung
8	Pong	KI-Gegner, Spiellogik
9	Snake	Schlangen-System
9	Flappy Fuchs	Komplexe Kollision

Lade die Beispiele aus dem **Beispiele...-Dropdown**, schau dir den Code an und verändere ihn — das ist der schnellste Weg zu lernen.

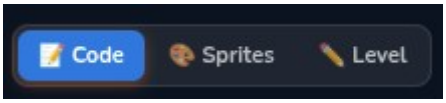
*Weiter zu **Kapitel 04 — UI-Guide GameFuchs** für eine Erklärung aller spielspezifischen Oberflächen-Elemente.*

CodeFuchs Handbuch — Kapitel 04: UI-Guide GameFuchs

Dieser Guide erklärt alle Bereiche und Werkzeuge, die im GameFuchs-Modus zusätzlich zur Verfügung stehen. Aktiviere den GameFuchs-Modus mit dem **[Game]**-Button oben links.

Überblick: Das GameFuchs-Layout

Im Game-Modus gibt es drei Haupt-Ansichten, zwischen denen du oben wechselst:



- **Code** — Der Code-Editor (wie in CodeFuchs, mit Game-spezifischen Bausteinen)
- **Sprites** — Der Pixel-Art-Editor zum Zeichnen eigener Figuren
- **Level** — Der Spielfeld-Editor zum visuellen Platzieren von Objekten

Ansicht 1: Code-Editor (Game-Modus)

Der Code-Editor funktioniert wie in CodeFuchs — aber die Bausteine-Palette zeigt jetzt Game-spezifische Befehle.

Bausteine-Palette im Game-Modus

Die Palette hat neue Kategorien:

Spielfeld

```
spielfeld(400, 300)
hintergrund("farbe")
hintergrundBild("name")
```

Figuren

```
figur ist neueFigur("name")
setzeFigur(figur, x, y)
bewegeFigur(figur, dx, dy)
figurGröße(figur, w, h)
figurSichtbar(figur, wahr)
figurSpiegeln(figur, wahr)
figurBild(figur, "neuesBild")
```

⌨ Steuerung

```
spielSchleife()
  ...
ende

wenn tasteGedrückt("links") dann
  ...
ende

wenn figurGeklickt(figur) dann
  ...
ende
```

⚡ Physik

```
schwerkraft(figur, 0.5)
figurSprung(figur, 10)
figurAmBoden(figur)
kollidiert(figur1, figur2)
figurAmRand(figur)
rechteckKollision(figur, x, y, w, h)
```

Plattformen

```
plattform(x, y, w, h, "farbe")
entfernePlattform(id)
```

Zeichnen

```
zeichneRechteck(x, y, w, h, "farbe")
zeichneKreis(x, y, radius, "farbe")
zeigeText("...", x, y)
zeigeText("...", x, y, größe, "farbe")
textGröße(20)
textFarbe("weiß")
```

Spielsteuerung

```
spielEnde("Nachricht")
spielPause()
spielWeiter()
spielNeustart()
spielZeit()
```

Sound

```
spieleSound("sprung")
spieleSound("punkt")
spieleSound("treffer")
spieleSound("münze")
spieleSound("gewonnen")
spieleSound("verloren")
```

Partikel

```
partikel(x, y, "explosion")
partikel(x, y, "funken")
partikel(x, y, "konfetti")
partikel(x, y, "feuer")
partikel(x, y, "sterne")
partikelEffekt(x, y, anzahl, tempo, dauer, größe, "farbe")
partikelLöschen()
```

Der Game-Canvas

Rechts siehst du den Game-Canvas — das ist das Spielfeld, auf dem dein Spiel läuft.

Canvas-Steuerelemente

Über dem Canvas gibt es eine Toolbar:

Button	Funktion
+	Reinzoomen
−	Rauszoomen
↺	Zoom zurücksetzen
⛶ Vollbild	Canvas auf Vollbild vergrößern

Navigation auf dem Canvas

Aktion	Funktion
Mausrad	Zoom
Linke Maustaste + Ziehen	Canvas verschieben
Touch (1 Finger)	Verschieben

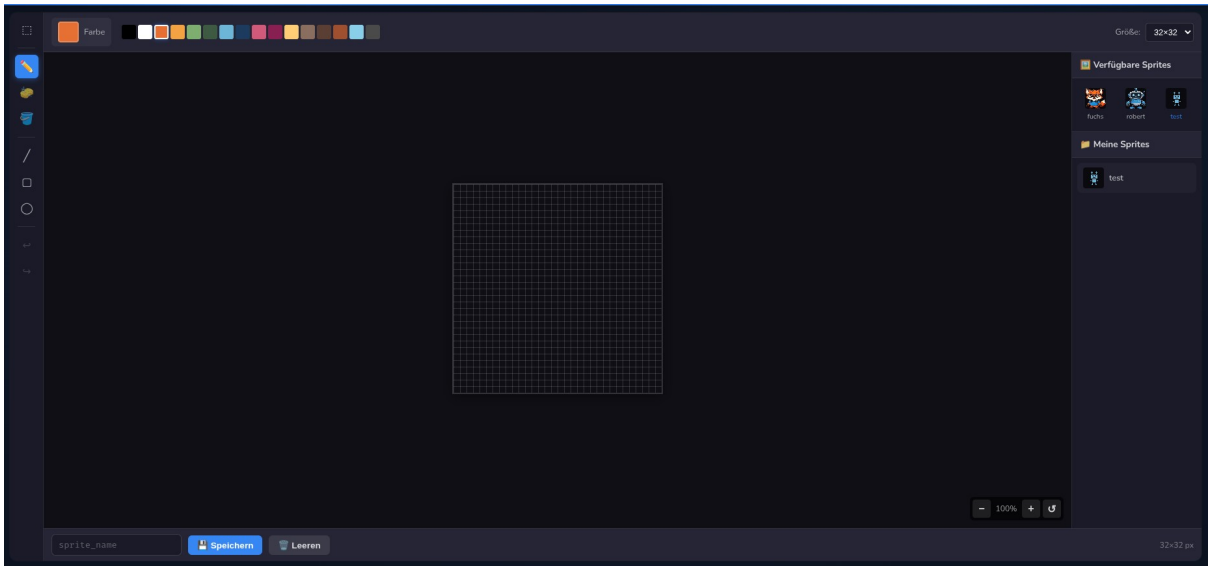
Vollbild-Modus

Der ⛶ **Vollbild**-Button vergrößert den Canvas auf den gesamten Bildschirm. Das ist praktisch zum Testen, wie das Spiel auf verschiedenen Bildschirmgrößen aussieht. Mit **Escape** beendest du den Vollbild-Modus.

Ansicht 2: Sprite-Editor (Sprites)

Der Sprite-Editor ist ein Pixel-Art-Zeichenprogramm. Hier kannst du eigene Spielfiguren zeichnen.

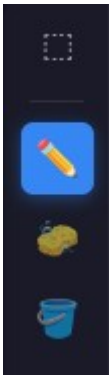
Aufbau des Sprite-Editors



Werkzeuge

Icon	Taste	Werkzeug	Funktion
⬚	M	Markieren	Bereich auswählen, verschieben, löschen
✏	P	Stift	Einzelne Pixel zeichnen
	E	Radierer	Pixel löschen (transparent machen)
	F	Fülleimer	Zusammenhängend

Icon	Taste	Werkzeug	Funktion
			e Fläche füllen
╱	L	Linie	Gerade Linie zwischen zwei Punkten
▢	R	Rechteck	Ausgefülltes oder hohles Rechteck
◯	C	Kreis	Ausgefüllter oder hohler Kreis
↩	Strg+Z	Undo	Letzte Aktion rückgängig
↪	Strg+Y	Redo	Aktion wiederherstellen



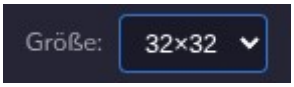
Farb-Palette

Oben siehst du 16 vordefinierte Farben:

Klicke auf eine Farbe, um sie auszuwählen. Die aktuelle Farbe wird links als Vorschau angezeigt.

Sprite-Größen

Du kannst die Auflösung des Sprites wählen:



Größe	Pixel	Verwendung
16×16	256 Pixel	Kleine Icons, Münzen, kleine Details
32×32	1024 Pixel	Standard-Spielfiguren
64×64	4096 Pixel	Große Figuren, Bosse

Navigation im Sprite-Editor

Aktion	Funktion
Mausrad	Zoom
Leertaste + Ziehen	Canvas verschieben
Klick	Zeichnen / Werkzeug anwenden
Klick + Ziehen	Linie / Rechteck / Kreis zeichnen

Sprite speichern

- Gib unten im Namensfeld einen Namen ein (max. 20 Zeichen, keine Sonderzeichen)
- Klicke **Speichern**

Der Sprite ist jetzt unter diesem Namen verfügbar und kann im Code verwendet werden:

```
figur ist neueFigur("meinSpriteName")
```

Reservierte Namen: `fuchs` und `robert` sind die eingebauten Sprites und können nicht überschrieben werden.

Sprite laden und bearbeiten

In der rechten Sidebar siehst du alle verfügbaren Sprites:

- **Vorhanden** — die eingebauten Sprites (`fuchs` , `robert`)
- **Meine Sprites** — deine selbst erstellten

Klicke auf einen Sprite-Namen, um ihn zur Bearbeitung zu laden.

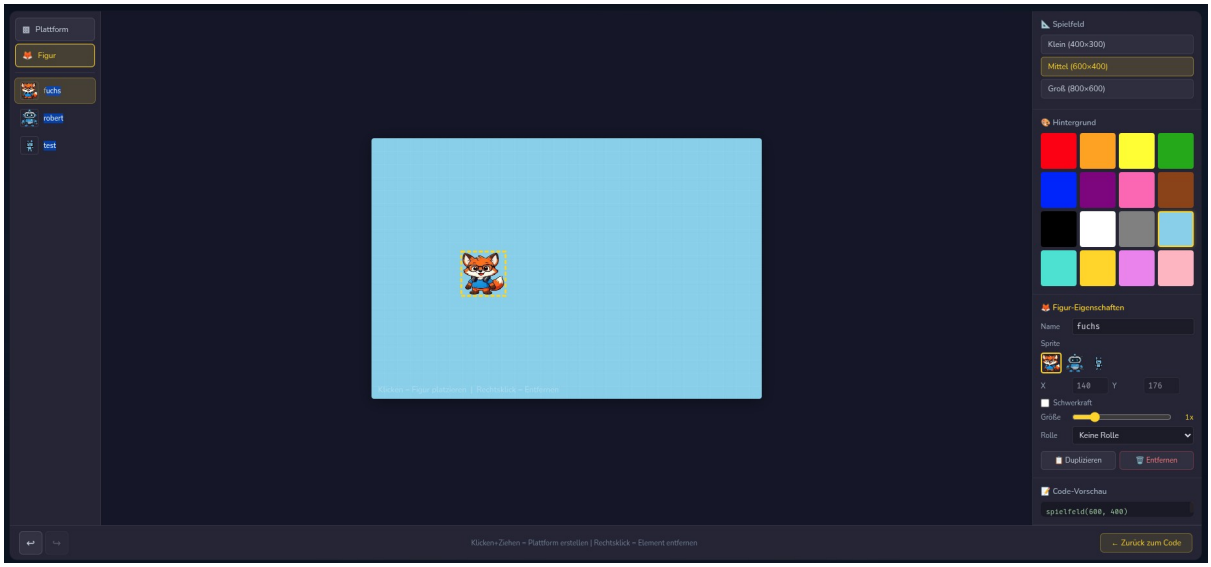
Sprite löschen

Neben jedem eigenen Sprite in der Liste gibt es einen **Löschen**-Button. Eingebaute Sprites können nicht gelöscht werden.

Ansicht 3: Spielfeld-Editor (✏ Level)

Der Spielfeld-Editor erlaubt es dir, Plattformen und Figuren visuell zu platzieren, statt ihre Koordinaten im Code einzutippen.

Aufbau des Spielfeld-Editors



Spielfeld konfigurieren (Linke Sidebar)

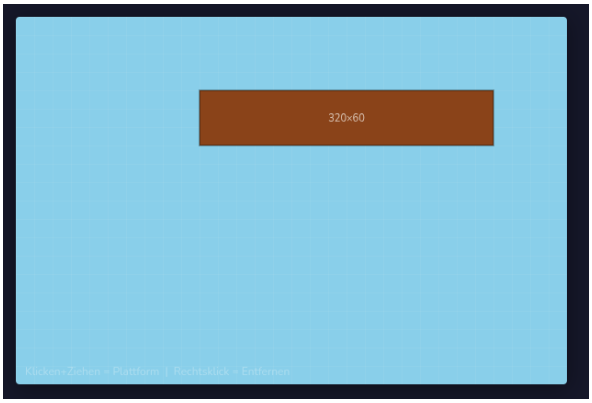
- **Breite / Höhe** — Größe des Spielfelds in Pixeln
- **Farbe** — Hintergrundfarbe des Spielfelds (Farbwähler oder Texteingabe)
-

Änderungen werden sofort im Canvas sichtbar.

Plattformen platzieren

Klicke in den Canvas und ziehe:

- **Klick + Ziehen** → Neue Plattform erstellen (braunes Rechteck)
- Die Größe ergibt sich aus dem gezogenen Bereich



Figuren platzieren

Klicke in der linken Sidebar auf einen Sprite-Namen und dann auf die gewünschte Position im Canvas. Die Figur wird dort platziert.

Du kannst platzierte Figuren durch Klicken und Ziehen verschieben.

Objekte löschen

Rechtsklick auf ein Objekt (Plattform oder Figur) löscht es.

Undo / Redo

Alle Aktionen im Spielfeld-Editor sind rückgängig zu machen:

- ↩ **Undo** (Strg+Z) — Letzter Schritt rückgängig
- ↪ **Redo** (Strg+Y) — Wiederherstellen

Code-Vorschau (Rechte Sidebar)

Die rechte Sidebar zeigt den generierten Setup-Code. Dieser Code ist schreibgeschützt — er wird automatisch aus deinen Platzierungen generiert.

```
# Setup-Code (automatisch generiert)
spielfeld(600, 400)
hintergrund("himmelblau")
held ist neueFigur("fuchs")
setzeFigur(held, 80, 300)
plattform(0, 380, 600, 20, "braun")
plattform(150, 270, 150, 15, "grün")
```

Klicke **← Zurück** oder wechsle auf den **Code**-Tab, um den generierten Code in deinen Editor zu übernehmen.

Der Export-Button

Im GameFuchs-Modus erscheint oben rechts ein **Export**-Button.

Export-Ablauf

- Klicke **Export**
- Gib einen Spieltitel ein
- Die Datei wird in deinem Ordner bereitgestellt.

Was die exportierte Datei enthält

Die erzeugte `.html` -Datei enthält:

- Deinen gesamten Spiel-Code (als JavaScript)
- Alle verwendeten Sprite-Bilder (eingebettet)
- Die komplette Spiel-Engine (Mini-Version, ~3KB)
- Kein CodeFuchs-Editor, keine Sidebar, kein Tutorial — nur das Spiel

Die Datei funktioniert komplett **offline** und ohne Installation.

Vorlagen-Dropdown

Der **Vorlagen**-Button (nur im Game-Modus) öffnet eine Liste mit fertigen Spielgerüsten:

Vorlage	Beschreibung
Jump'n'Run	Spieler mit Schwerkraft, Plattformen, Münzen sammeln
Top-Down	Vogelperspektive, 4-Richtungs-Steuerung
Klicker	Figur anklicken für Punkte
Snake	Schlangen-Spiel-Grundgerüst

Eine Vorlage lädt einen fertig funktionierenden Code-Rahmen in den Editor. Du musst nur noch deine Sprites und Spiellogik einfügen.

Tastaturkürzel im GameFuchs-Modus

Shortcut	Bereich	Funktion
`Strg+S` / `Cmd+S`	Global	Speichern
`Strg+Z` / `Cmd+Z`	Alle Ansichten	Undo
`Strg+Y` / `Cmd+Y`	Alle Ansichten	Redo
`M`	Sprite-Editor	Markierungs-Tool
`P`	Sprite-Editor	Stift-Tool
`E`	Sprite-Editor	Radierer-Tool
`F`	Sprite-Editor	Fülleimer-Tool
`L`	Sprite-Editor	Linien-Tool
`R`	Sprite-Editor	Rechteck-Tool
`C`	Sprite-Editor	Kreis-Tool
`Leertaste + Ziehen`	Sprite-Editor	Canvas verschieben
`Mausrad`	Sprite-Editor	Zoom
`Escape`	Game-Canvas	Vollbild beenden

Sprite-Manager Panel (während das Spiel läuft)

Wenn ein Spiel läuft, erscheint in der Sidebar eine Live-Ansicht aller erstellten Figuren mit ihren aktuellen Werten:

```
 Figuren (Live)
held
  x: 142    y: 280
  w: 64     h: 64
  sichtbar: ja
münze
  x: 312    y: 150
  w: 32     h: 32
  sichtbar: ja
```

Das ist nützlich zum Debuggen — du siehst in Echtzeit, wo jede Figur gerade ist.

Speichern im Game-Modus

Projekte im Game-Modus werden getrennt von CodeFuchs-Projekten gespeichert.

Speicherort: Browser-Speicher unter `GameFuchs_projects`

Das heißt: Ein Code-Modus-Projekt und ein Game-Modus-Projekt können denselben Namen haben, ohne sich zu überschreiben.

*Weiter zu **Kapitel 05 — Syntax-Referenz** für die vollständige Dokumentation aller CodeFuchs-Befehle.*

CodeFuchs Handbuch — Kapitel 05: Syntax-Referenz

Dieses Kapitel ist das vollständige Nachschlagewerk aller CodeFuchs-Befehle. Hier findest du Syntax, Parameter, Rückgabewerte und Beispiele zu jedem Befehl.

Nutze dieses Kapitel nicht zum Lernen — dafür ist Kapitel 01 besser. Hier schaust du nach, wenn du im Code einen Befehl vergessen hast.

Grundstruktur

Ein CodeFuchs-Programm besteht aus Anweisungen, eine pro Zeile. Leerzeilen werden ignoriert. Die Reihenfolge ist sequenziell von oben nach unten.

```
# Das ist ein Kommentar
sag "Hallo Welt!"     # Kommentar darf auch am Zeilenende stehen
```

Kommentare: Alles nach `#` in einer Zeile wird ignoriert.

Block-Ende: Alle Blöcke (`wenn` , `wiederhole` , `solange` , `funktion` , `spielSchleife`) enden mit dem Schlüsselwort `ende` .

Einrückung: Empfohlen, aber nicht erzwungen. Der Editor rückt automatisch ein.

Ausgabe — `sag`

Gibt einen Wert in den Ausgabebereich aus. Jeder `sag` -Aufruf erzeugt eine neue Zeile.

Syntax

```
sag ausdruck
sag "Text"
sag zahl
sag variable
sag "Text" + variable + "mehr Text"
```

Parameter

Parameter	Typ	Beschreibung
`ausdruck`	beliebig	Was ausgegeben wird. Kann Text, Zahl, Variable oder ein Ausdruck mit `+` sein.

Beispiele

```
sag "Hallo Welt!"
sag 42
sag 3.14
sag wahr
sag name
sag "Du heißt: " + name
sag "Ergebnis: " + (a + b)
sag "Alter: " + alter + " Jahre"
```

Rückgabe

Keiner. `sag` hat keinen Rückgabewert.

Eingabe — `frage`

Hält das Programm an und zeigt eine Eingabe-Aufforderung. Die Antwort des Benutzers wird in einer Variable gespeichert.

Syntax

```
frage "Fragetext" als variablenname
```

Parameter

Parameter	Typ	Beschreibung
`"Fragetext"`	Text	Die Frage, die dem Benutzer angezeigt wird
`variablenname`	Bezeichner	Name der Variable, in der die Antwort gespeichert wird

Hinweis zum Typ

Die Antwort wird immer als **Text** gespeichert, auch wenn der Benutzer eine Zahl eingibt. Arithmetik funktioniert in der Regel trotzdem, da CodeFuchs automatisch zwischen Text und Zahl konvertiert, wenn der Kontext es erlaubt.

Beispiele

```
frage "Wie heißt du?" als name
sag "Hallo, " + name + "!"

frage "Wie alt bist du?" als alter
wenn alter >= 18 dann
  sag "Du bist volljährig."
ende

frage "Nenne eine Zahl:" als eingabe
ergebnis ist eingabe * 2
sag "Das Doppelte ist: " + ergebnis
```

Variablen — `ist`

Weist einer Variablen einen Wert zu. Variablen müssen nicht vorher deklariert werden — die erste Zuweisung erstellt sie.

Syntax

```
variablenname ist ausdruck
```

Parameter

Parameter	Typ	Beschreibung
`variablenname`	Bezeichner	Der Name der Variable. Darf Buchstaben, Ziffern und `_` enthalten, muss mit Buchstabe beginnen. Groß-/Kleinschreibung wird unterschieden.
`ausdruck`	beliebig	Der Wert, der gespeichert

Parameter	Typ	Beschreibung
		wird.

Beispiele

```
name ist "Anna"
alter ist 11
pi ist 3.14159
istAktiv ist wahr
liste ist [1, 2, 3]

# Variablen können überschrieben werden
zahl ist 5
zahl ist zahl + 1    # zahl ist jetzt 6

# Variablen können in Ausdrücken verwendet werden
a ist 10
b ist 20
summe ist a + b
```

Gültigkeitsbereich

- **Globale Variablen:** Erstellt außerhalb von Funktionen. Überall im Programm sichtbar.
- **Lokale Variablen:** Erstellt innerhalb einer Funktion. Nur innerhalb dieser Funktion sichtbar.

```
x ist 10      # global

funktion test()
  y ist 5     # lokal – nur in test() sichtbar
  sag x       # globale Variable ist sichtbar
  sag y
ende

test()
sag x         # funktioniert
# sag y       # würde einen Fehler geben – y existiert hier nicht
```

Datentypen

CodeFuchs ist dynamisch typisiert — eine Variable kann jeden Typ enthalten, und der Typ kann sich ändern.

Zahlen

Ganze Zahlen und Dezimalzahlen:

```
ganzeZahl ist 42
dezimal ist 3.14
negativ ist -7
```

Text (Strings)

Zeichenketten, immer in doppelten Anführungszeichen:

```
text ist "Hallo Welt"
leer ist ""
mitZahl ist "Ich bin 10 Jahre alt"
```

Wahrheitswerte (Boolean)

```
istRichtig ist wahr
istFalsch ist falsch
```

Arrays (Listen)

Geordnete Sammlungen von Werten:

```
zahlen ist [1, 2, 3, 4, 5]
texte ist ["rot", "grün", "blau"]
gemischt ist [1, "hallo", wahr]
leer ist []
```

Operatoren

Arithmetische Operatoren

Operator	Bedeutung	Beispiel	Ergebnis
`+`	Addition	`5 + 3`	`8`
`-`	Subtraktion	`10 - 4`	`6`
`*`	Multiplikation	`6 * 7`	`42`
`/`	Division	`15 / 3`	`5`
`+` (Text)	Verkettung	`"Ha" + "llo"`	`"Hallo"`

Vergleichsoperatoren

Liefern immer `wahr` oder `falsch` :

Operator	Bedeutung	Beispiel	Ergebnis
`=`	Gleich	`5 = 5`	`wahr`
`!=`	Ungleich	`5 != 3`	`wahr`
`>`	Größer als	`10 > 5`	`wahr`
`<`	Kleiner als	`3 < 7`	`wahr`
`>=`	Größer oder gleich	`5 >= 5`	`wahr`
`<=`	Kleiner oder gleich	`4 <= 10`	`wahr`

Logische Operatoren

Operator	Bedeutung	Beispiel
`und`	Beide müssen wahr sein	`alter > 10 und alter < 18`
`oder`	Mindestens eine muss wahr sein	`farbe = "rot" oder farbe = "blau"`

Operator-Priorität

- Klammern `( )` — höchste Priorität
- `*` und `/`
- `+` und `-`
- Vergleiche (`=` , `!=` , `<` , `>` , `<=` , `>=`)
- `und`
- `oder` — niedrigste Priorität

```
ergebnis ist (3 + 4) * 2     # = 14
ergebnis ist 3 + 4 * 2       # = 11 (Punkt vor Strich)
```

Bedingungen — `wenn / dann / sonst / ende`

Syntax (einfach)

```
wenn bedingung dann
  # Anweisungen
ende
```

Syntax (mit Alternative)

```
wenn bedingung dann
  # Anweisungen wenn wahr
sonst
  # Anweisungen wenn falsch
ende
```

Syntax (verschachtelt)

```
wenn bedingung1 dann
  # ...
sonst
  wenn bedingung2 dann
    # ...
  sonst
    # ...
  ende
ende
```

Parameter

Parameter	Typ	Beschreibung
`bedingung`	Wahrheitswert	Ausdruck, der `wahr` oder `falsch` ergibt. Vergleichsoperatoren, logische Operatoren, oder direkte Wahrheitswerte.

Beispiele

```
# Einfach
wenn alter >= 18 dann
  sag "Volljährig"
ende

# Mit sonst
wenn zahl > 0 dann
  sag "Positiv"
sonst
  sag "Null oder negativ"
ende

# Verschachtelt
wenn zahl > 0 dann
  sag "Positiv"
sonst
  wenn zahl < 0 dann
    sag "Negativ"
  sonst
    sag "Null"
  ende
ende

# Mit logischen Operatoren
wenn alter >= 13 und alter <= 19 dann
  sag "Du bist ein Teenager"
ende
```

Zählschleife — `wiederhole / mal / ende`

Führt einen Block eine bestimmte Anzahl von Malen aus.

Syntax

```
wiederhole anzahl mal
  # Anweisungen
ende
```

Parameter

Parameter	Typ	Beschreibung
`anzahl`	Zahl oder Variable	Wie viele Male der Block ausgeführt wird. Muss eine positive ganze Zahl sein.

Beispiele

```
# Direkte Zahl
wiederhole 5 mal
  sag "Hallo"
ende

# Variable als Anzahl
n ist 10
wiederhole n mal
  sag "Zeile"
ende

# Verschachtelt (Multiplikationstabelle)
wiederhole 3 mal
  wiederhole 3 mal
    sag "*"
  ende
ende

# Turtle-Quadrat
wiederhole 4 mal
  vorwärts(100)
  rechts(90)
ende
```

Bedingungsschleife — `solange / ende`

Führt einen Block wiederholt aus, solange eine Bedingung wahr ist.

Syntax

```
solange bedingung
  # Anweisungen
ende
```

Parameter

Parameter	Typ	Beschreibung
`bedingung`	Wahrheitswert	Wie bei `wenn`. Wird vor jedem Durchlauf geprüft.

Wichtig

Die Bedingung wird **vor** jedem Durchlauf geprüft. Wenn sie beim ersten Mal bereits falsch ist, wird der Block kein einziges Mal ausgeführt.

Stelle sicher, dass sich die Bedingung innerhalb des Blocks irgendwann ändert — sonst entsteht eine Endlosschleife.

Beispiele

```
# Zähler
i ist 0
solange i < 10
  sag i
  i ist i + 1
ende

# Benutzer-Eingabe
antwort ist ""
solange antwort != "stop"
  frage "Schreibe etwas (stop zum Beenden):" als antwort
  sag "Du schriebst: " + antwort
ende

# Zufallsspiel
zahl ist zufallsZahl(6) + 1
versuche ist 0
solange zahl != 6
  zahl ist zufallsZahl(6) + 1
  versuche ist versuche + 1
ende
sag "Nach " + versuche + " Versuchen endlich eine 6!"
```

Funktionen — `funktion / zurück / ende`

Funktion definieren

```
funktion name()
  # Anweisungen
ende

funktion name(param1, param2)
  # Anweisungen
ende
```

Funktion aufrufen

```
name()
name(argument1, argument2)
ergebnis ist name(argument1)
```

Rückgabewert

```
funktion name()
  zurück wert
ende
```

Parameter

Element	Beschreibung
`name`	Bezeichner der Funktion
`param1, param2, ...`	Parameternamen (optional). Kommagetrennt. Innerhalb der Funktion als lokale Variablen verfügbar.
`zurück wert`	Beendet die Funktion und gibt `wert` zurück. Optional — ohne `zurück` gibt die Funktion keinen Wert zurück.

Beispiele

```
# Ohne Parameter, ohne Rückgabe
funktion trennlinie()
  sag "--------------------"
ende

trennlinie()
sag "Hallo"
trennlinie()

# Mit Parametern
funktion begrüße(name, alter)
  sag "Hallo, " + name + "!"
  sag "Du bist " + alter + " Jahre alt."
ende

begrüße("Lisa", 11)
begrüße("Tom", 9)

# Mit Rückgabewert
funktion addiere(a, b)
  zurück a + b
ende

summe ist addiere(10, 5)
sag "10 + 5 = " + summe

# Rekursive Funktion (Fakultät)
funktion fakultät(n)
  wenn n <= 1 dann
    zurück 1
  sonst
    zurück n * fakultät(n - 1)
  ende
ende

sag "5! = " + fakultät(5)   # → 120
```

Arrays / Listen

Erstellen

```
leere ist []
zahlen ist [1, 2, 3, 4, 5]
namen ist ["Anna", "Ben", "Clara"]
gemischt ist [1, "hallo", wahr, [1, 2]]
```

Zugriff

Arrays sind 0-indiziert (das erste Element hat Index 0):

```
farben ist ["rot", "blau", "grün"]
sag farben[0]    # → rot
sag farben[1]    # → blau
sag farben[2]    # → grün
```

Element setzen

```
farben[1] ist "gelb"    # "blau" wird durch "gelb" ersetzt
```

Listen-Funktionen

`länge(array)`

Gibt die Anzahl der Elemente zurück.

```
länge([1, 2, 3])          # → 3
länge("Hallo")            # → 5 (auch Strings haben eine Länge)
länge(meineListe)         # → aktuelle Länge
```

`anhängen(array, wert)`

Fügt ein Element am Ende hinzu.

```
liste ist [1, 2, 3]
anhängen(liste, 4)    # liste ist jetzt [1, 2, 3, 4]
```

`entfernen(array, index)`

Entfernt das Element am angegebenen Index und gibt es zurück. Ohne Index wird das letzte Element entfernt.

```
liste ist ["a", "b", "c"]
entfernen(liste, 0)    # entfernt "a", liste ist jetzt ["b", "c"]
entfernen(liste)       # entfernt "c" (letztes Element)
```

`enthält(array, wert)`

Gibt `wahr` zurück, wenn der Wert in der Liste ist.

```
farben ist ["rot", "blau", "grün"]
enthält(farben, "blau")    # → wahr
enthält(farben, "gelb")    # → falsch
```

`finde(array, wert)`

Gibt den Index des ersten Vorkommens zurück, oder `-1` wenn nicht gefunden.

```
farben ist ["rot", "blau", "grün"]
finde(farben, "blau")     # → 1
finde(farben, "gelb")     # → -1
```

Arrays mit Schleifen

```
namen ist ["Anna", "Ben", "Clara", "David"]
i ist 0
solange i < länge(namen)
  sag (i + 1) + ". " + namen[i]
  i ist i + 1
ende
```

Ausgabe:

```
1. Anna
2. Ben
3. Clara
4. David
```

Turtle-Grafik

Turtle-Befehle funktionieren nur im **Code-Modus** (nicht in GameFuchs). Die Turtle startet in der Mitte des Canvas und schaut nach rechts (0°).

Bewegung

`vorwärts(pixel)`

Bewegt die Turtle vorwärts und zeichnet eine Linie (wenn Stift unten).

```
vorwärts(100)
vorwärts(50)
```

`rückwärts(pixel)`

Bewegt die Turtle rückwärts.

```
rückwärts(50)
```

Drehung

`rechts(grad)`

Dreht die Turtle um `grad` Grad im Uhrzeigersinn.

```
rechts(90)     # 90° im Uhrzeigersinn
rechts(45)     # 45° im Uhrzeigersinn
rechts(360)    # vollständige Drehung
```

`links(grad)`

Dreht die Turtle um `grad` Grad gegen den Uhrzeigersinn.

```
links(90)
```

Stift

`stiftHoch()`

Hebt den Stift an. Die Turtle bewegt sich ohne zu zeichnen.

```
stiftHoch()
vorwärts(100)    # bewegt sich, zeichnet aber nicht
stiftRunter()
```

`stiftRunter()`

Setzt den Stift ab. Alle folgenden Bewegungen erzeugen Linien.

```
stiftRunter()
vorwärts(100)    # zeichnet eine Linie
```

Farbe und Stil

`farbe("farbe")`

Setzt die Stiftfarbe. Akzeptiert deutsche Farbwörter und Hex-Codes.

```
farbe("rot")
farbe("dunkelblau")
farbe("#FF5500")
farbe("#FFA")      # Kurzschreibweise
```

Formen

`kreis(radius)`

Zeichnet einen Kreis um die aktuelle Position der Turtle.

```
kreis(50)     # Kreis mit Radius 50px
```

`quadrat(größe)`

Zeichnet ein Quadrat um die aktuelle Position.

```
quadrat(100)    # 100×100px Quadrat
```

Animation

`langsam(geschwindigkeit)`

Schaltet den Animations-Modus ein. Die Turtle bewegt sich sichtbar langsam.

Wert	Geschwindigkeit
`1`	Sehr langsam
`5`	Mittel
`10`	Schnell

```
langsam(3)
wiederhole 4 mal
  vorwärts(100)
  rechts(90)
ende
```

Ohne `langsam()` wird die gesamte Zeichnung sofort angezeigt.

Mathematik-Funktionen

`zufallsZahl(max)`

Gibt eine zufällige ganze Zahl von `0` bis `max - 1` zurück.

```
zufallsZahl(6)       # → 0, 1, 2, 3, 4 oder 5
zufallsZahl(6) + 1   # → 1, 2, 3, 4, 5 oder 6 (Würfel)
zufallsZahl(100)     # → 0 bis 99
```

`wurzel(zahl)`

Gibt die Quadratwurzel zurück.

```
wurzel(9)     # → 3
wurzel(2)     # → 1.4142...
wurzel(25)    # → 5
```

`abs(zahl)`

Gibt den Absolutwert zurück (immer positiv).

```
abs(-5)    # → 5
abs(5)     # → 5
abs(-3.7)  # → 3.7
```

Farbwörter

Alle folgenden Wörter können überall dort verwendet werden, wo eine Farbe erwartet wird.

Farbwort	Hex-Code
`rot`	`#FF0000`
`dunkelrot`	`#8B0000`
`hellrot`	`#FF6B6B`
`grün`	`#00AA00`
`hellgrün` / `hellgruen`	`#90EE90`
`dunkelgrün` / `dunkelgruen`	`#006400`
`blau`	`#0000FF`
`hellblau`	`#ADD8E6`
`dunkelblau`	`#00008B`
`himmelblau`	`#87CEEB`
`gelb`	`#FFFF00`
`orange`	`#FFA500`
`lila`	`#800080`
`pink`	`#FF69B4`
`rosa`	`#FFB6C1`
`fuchsia`	`#FF00FF`
`violett`	`#EE82EE`
`magenta`	`#FF00FF`
`türkis` / `tuerkis`	`#40E0D0`
`cyan`	`#00FFFF`
`marine`	`#000080`
`braun`	`#8B4513`
`gold`	`#FFD700`
`beige`	`#F5DEB3`
`khaki`	`#F0E68C`
`schwarz`	`#000000`
`weiß` / `weiss`	`#FFFFFF`
`grau`	`#808080`
`hellgrau`	`#D3D3D3`
`dunkelgrau`	`#404040`
`silber`	`#C0C0C0`
`transparent`	durchsichtig

Alternativen Farbangaben

Neben deutschen Farbwörtern akzeptiert CodeFuchs auch:

```
farbe("#FF0000")          # Hex-Code (6 Zeichen)
farbe("#F00")             # Hex-Code Kurzform (3 Zeichen)
farbe("#FF000088")        # Hex mit Alpha (8 Zeichen, halb-transparent)
```

Schlüsselwörter — vollständige Liste

Die folgenden Wörter haben in CodeFuchs eine besondere Bedeutung und dürfen nicht als Variablen- oder Funktionsnamen verwendet werden:

Keyword	Verwendung
`sag`	Ausgabe
`frage`	Eingabe
`als`	Teil von `frage ... als`
`ist`	Zuweisung
`wenn`	Bedingung
`dann`	Teil von `wenn ... dann`
`sonst`	Else-Zweig
`ende`	Block-Abschluss
`wiederhole`	Zählschleife
`mal`	Teil von `wiederhole ... mal`
`solange`	Bedingungsschleife
`funktion`	Funktionsdefinition
`zurück`	Rückgabewert
`und`	Logisches UND
`oder`	Logisches ODER
`wahr`	Wahrheitswert true
`falsch`	Wahrheitswert false
`spielSchleife`	Game-Loop (GameFuchs)

CodeFuchs Handbuch — Kapitel 06: Referenz GameEngine (GameFuchs)

Dieses Kapitel dokumentiert alle Funktionen, die im GameFuchs-Modus verfügbar sind. Es ist das Nachschlagewerk für die Spielentwicklung. Alle CodeFuchs-Grundbefehle (`sag` , `wenn` , `wiederhole` , `funktion` etc.) funktionieren auch in GameFuchs — hier sind nur die zusätzlichen Spielfunktionen dokumentiert.

Wichtig: Alle GameFuchs-Funktionen funktionieren nur im **Game-Modus**. Im Code-Modus gibt es eine Fehlermeldung.

Spielfeld & Hintergrund

`spielfeld(breite, höhe)`

Setzt die Größe des Game-Canvas.

Parameter:

- `breite` — Breite in Pixeln
- `höhe` — Höhe in Pixeln

Hinweis: Muss vor allen anderen Spielbefehlen aufgerufen werden. Standardgröße ist 400×300.

```
spielfeld(800, 600)    # HD-Spielfeld
spielfeld(400, 300)    # Kompaktes Spielfeld
spielfeld(320, 240)    # Klassische Retro-Größe
```

`hintergrund("farbe")`

Setzt die Hintergrundfarbe des Spielfelds. Alle deutschen Farbwörter und Hex-Codes sind gültig.

```
hintergrund("himmelblau")
hintergrund("schwarz")
hintergrund("#1a1a2e")
```

`hintergrundBild("name")`

Setzt ein geladenes Sprite-Asset als Hintergrund.

Parameter:

- `"name"` — Name des Assets (muss als Sprite geladen oder im Sprite-Editor gespeichert sein)

```
hintergrundBild("himmel")
hintergrundBild("stadtlandschaft")
```

Figuren (Sprites)

Eine Figur ist ein rechteckiges Bild, das sich auf dem Spielfeld bewegen kann. Jede Figur hat eine eindeutige **ID**, die beim Erstellen zurückgegeben wird.

`neueFigur("assetName")`

Erstellt eine neue Figur und gibt ihre ID zurück.

Parameter:

- `"assetName"` — Name des Sprite-Assets. Eingebaut: `"fuchs"`, `"robert"`. Eigene Sprites nach Namen im Sprite-Editor.

Rückgabe: Figur-ID (ein interner Wert, in einer Variable speichern)

```
held ist neueFigur("fuchs")
gegner ist neueFigur("robert")
münze ist neueFigur("fuchs")     # Dasselbe Bild, neue Figur
```

`setzeFigur(id, x, y)`

Platziert eine Figur an einer absoluten Position.

Parameter:

- `id` — Figur-ID (aus `neueFigur()`)
- `x` — Horizontale Position (0 = linker Rand)
- `y` — Vertikale Position (0 = oberer Rand)

```
setzeFigur(held, 100, 200)
setzeFigur(held, 0, 0)          # Obere linke Ecke
setzeFigur(held, 400, 300)      # Mitte bei 800×600-Spielfeld
```

`bewegeFigur(id, dx, dy)`

Bewegt eine Figur relativ zu ihrer aktuellen Position.

Parameter:

- `id` — Figur-ID
- `dx` — Verschiebung horizontal (positiv = rechts, negativ = links)
- `dy` — Verschiebung vertikal (positiv = unten, negativ = oben)

```
bewegeFigur(held, 5, 0)      # 5px nach rechts
bewegeFigur(held, -5, 0)     # 5px nach links
bewegeFigur(held, 0, -5)     # 5px nach oben
bewegeFigur(held, 3, 3)      # diagonal nach unten-rechts
```

`figurGröße(id, breite, höhe)`

Ändert die Größe einer Figur.

Parameter:

- `id` — Figur-ID
- `breite` — Neue Breite in Pixeln
- `höhe` — Neue Höhe in Pixeln

```
figurGröße(held, 64, 64)      # Standard-Größe
figurGröße(boss, 128, 128)    # Doppelt so groß
figurGröße(münze, 24, 24)     # Klein
```

`figurSichtbar(id, sichtbar)`

Zeigt oder versteckt eine Figur.

Parameter:

- `id` — Figur-ID
- `sichtbar` — `wahr` = sichtbar, `falsch` = unsichtbar

```
figurSichtbar(münze, falsch)    # Münze verstecken (gesammelt)
figurSichtbar(münze, wahr)      # Münze wieder zeigen
```

`figurBild(id, "assetName")`

Wechselt das Bild einer Figur zu einem anderen Asset.

```
figurBild(held, "fuchs_winkt")    # Bild wechseln
figurBild(held, "fuchs")          # Zurück zum Original
```

`figurSpiegeln(id, gespiegelt)`

Spiegelt eine Figur horizontal.

Parameter:

- `id` — Figur-ID
- `gespiegelt` — `wahr` = gespiegelt (schaut nach links), `falsch` = normal

Nützlich, um eine Figur in die richtige Richtung blicken zu lassen, ohne zwei verschiedene Sprites zu benötigen.

```
wenn tasteGedrückt("links") dann
  bewegeFigur(held, -4, 0)
  figurSpiegeln(held, wahr)     # Schaut nach links
ende
wenn tasteGedrückt("rechts") dann
  bewegeFigur(held, 4, 0)
  figurSpiegeln(held, falsch)   # Schaut nach rechts
ende
```

Figur-Eigenschaften lesen

Auf die aktuellen Werte einer Figur kann mit Punkt-Notation zugegriffen werden:

Eigenschaft	Bedeutung	Typ
`figur.x`	X-Position	Zahl
`figur.y`	Y-Position	Zahl
`figur.w`	Breite	Zahl
`figur.h`	Höhe	Zahl

```
sag "Held ist bei X=" + held.x + ", Y=" + held.y

# Figur am rechten Rand stoppen
wenn held.x > spielfeldBreite - held.w dann
  setzeFigur(held, spielfeldBreite - held.w, held.y)
ende
```

Tastatur-Eingabe

`tasteGedrückt("taste")`

Gibt `wahr` zurück, solange die angegebene Taste gedrückt gehalten wird.

Nur innerhalb der Spielschleife verwenden.

Rückgabe: `wahr` oder `falsch`

Gültige Tastenbezeichner:

Bezeichner	Taste
`"links"`	← Pfeil links
`"rechts"`	→ Pfeil rechts
`"hoch"`	↑ Pfeil oben
`"runter"`	↓ Pfeil unten
`"leertaste"`	Leertaste
`"eingabe"`	Enter
`"escape"`	Escape
`"a"` bis `"z"`	Buchstabe A–Z (Kleinschreibung)
`"1"` bis `"9"`	Ziffer 1–9

```
spielSchleife()

  wenn tasteGedrückt("links") dann
    bewegeFigur(held, -4, 0)
  ende

  wenn tasteGedrückt("rechts") dann
    bewegeFigur(held, 4, 0)
  ende

  wenn tasteGedrückt("leertaste") und figurAmBoden(held) dann
    figurSprung(held, 12)
    spieleSound("sprung")
  ende

  wenn tasteGedrückt("escape") dann
    spielPause()
  ende

ende
```

Maus / Touch-Eingabe

`figurGeklickt(id)`

Gibt `wahr` zurück, wenn auf die Figur geklickt oder getippt wurde (in diesem Frame).

Rückgabe: `wahr` oder `falsch`

```
spielSchleife()

  wenn figurGeklickt(knopf) dann
    punkte ist punkte + 1
    spieleSound("klick")
  ende

  zeigeText("Punkte: " + punkte, 10, 30)

ende
```

Kollision

`kollidiert(id1, id2)`

Gibt `wahr` zurück, wenn sich die Bounding-Boxes der zwei Figuren überlappen (AABB-Kollision).

Parameter:

- `id1`, `id2` — Figur-Ids

Rückgabe: `wahr` oder `falsch`

```
wenn kollidiert(held, münze) dann
  punkte ist punkte + 1
  figurSichtbar(münze, falsch)
  spieleSound("münze")
ende

wenn kollidiert(held, gegner) dann
  leben ist leben - 1
  partikel(held.x, held.y, "explosion")
  spieleSound("treffer")
  setzeFigur(held, 50, 50)    # Zurück zum Start
ende
```

`figurAmRand(id)`

Gibt `wahr` zurück, wenn die Figur den Spielfeld-Rand berührt oder überschreitet.

Rückgabe: `wahr` oder `falsch`

```
wenn figurAmRand(ball) dann
  spielEnde("Du bist rausgelaufen!")
ende

# Oder: Figur am Rand abprallen lassen
wenn figurAmRand(ball) dann
  richtungX ist richtungX * -1
ende
```

`rechteckKollision(id, x, y, breite, höhe)`

Prüft ob eine Figur mit einem bestimmten Rechteck (keine andere Figur) kollidiert.

Parameter:

- `id` — Figur-ID
- `x`, `y` — Position des Rechtecks
- `breite`, `höhe` — Größe des Rechtecks

Rückgabe: `wahr` oder `falsch`

```
# Prüfen ob der Held die "Ziellinie" (unsichtbares Rechteck) berührt
wenn rechteckKollision(held, 750, 0, 50, 600) dann
  spielEnde("Ziel erreicht!")
ende
```

Physik

`schwerkraft(id, stärke)`

Aktiviert Schwerkraft für eine Figur. Die Figur fällt automatisch nach unten und landet auf Plattformen.

Parameter:

- `id` — Figur-ID
- `stärke` — Schwerkraftbeschleunigung (0.1 = sehr leicht, 1.0 = stark)

Hinweis: Muss im Setup-Bereich aufgerufen werden (vor der Spielschleife).

```
schwerkraft(held, 0.5)    # Normale Schwerkraft
schwerkraft(ball, 0.3)    # Leichte Schwerkraft (z.B. für Monde)
```

`figurSprung(id, kraft)`

Lässt eine Figur nach oben springen. Funktioniert nur, wenn die Figur gerade am Boden ist (`figurAmBoden(id)` ist `wahr`).

Parameter:

- `id` — Figur-ID
- `kraft` — Sprungkraft (5–15 typische Werte)

```
wenn tasteGedrückt("leertaste") und figurAmBoden(held) dann
  figurSprung(held, 12)
  spieleSound("sprung")
ende
```

`figurAmBoden(id)`

Gibt `wahr` zurück, wenn die Figur auf einer Plattform oder dem Boden steht.

Rückgabe: `wahr` oder `falsch`

```
wenn figurAmBoden(held) dann
  zeigeText("Am Boden", 10, 30)
sonst
  zeigeText("In der Luft", 10, 30)
ende
```

Plattformen

`plattform(x, y, breite, höhe, "farbe")`

Erstellt eine Plattform. Figuren mit Schwerkraft landen auf Plattformen.

Parameter:

- `x` , `y` — Position der oberen linken Ecke
- `breite` , `höhe` — Größe in Pixeln
- `"farbe"` — Farbe der Plattform (optional, Standard: `"braun"`)

Rückgabe: Plattform-ID (zum späteren Entfernen)

```
# Boden
plattform(0, 380, 800, 20, "braun")

# Schwebende Plattformen
plattform(100, 280, 120, 15, "grün")
plattform(300, 210, 100, 15, "grün")
plattform(500, 150, 80, 15, "grün")

# Unsichtbare Kollisionszone (transparent)
bodenID ist plattform(0, 399, 800, 1, "transparent")
```

`entfernePlattform(id)`

Entfernt eine Plattform aus dem Spielfeld.

Parameter:

- `id` — Plattform-ID (Rückgabewert von `plattform()`)

```
plattformID ist plattform(200, 200, 100, 15, "braun")

# Später entfernen:
entfernePlattform(plattformID)
```

Anzeige: Text und Formen

Wichtig: Alle Zeichnungen werden am Ende jedes Frames gelöscht. Alles, was sichtbar bleiben soll, muss in der Spielschleife jedes Frame neu gezeichnet werden.

`zeigeText(text, x, y, größe, "farbe")`

Zeichnet Text auf den Canvas.

Parameter:

- `text` — Anzuzeigender Text. Variablen mit `+` einbinden.
- `x` , `y` — Position der oberen linken Ecke des Textes
- `größe` — Schriftgröße in Pixeln (optional, Standard: 20)
- `"farbe"` — Textfarbe (optional, Standard: aktuelle Textfarbe)

```
zeigeText("Punkte: " + punkte, 10, 30)
zeigeText("GAME OVER", 200, 200, 48, "rot")
zeigeText("Level " + level, 10, 30, 24, "weiß")
```

`textGröße(größe)`

Setzt die Standard-Schriftgröße für alle folgenden `zeigeText()` -Aufrufe.

```
textGröße(24)
zeigeText("Groß", 10, 30)
textGröße(14)
zeigeText("Klein", 10, 60)
```

`textFarbe("farbe")`

Setzt die Standard-Textfarbe für alle folgenden `zeigeText()` -Aufrufe.

```
textFarbe("weiß")
zeigeText("Heller Text", 10, 30)
textFarbe("rot")
zeigeText("Roter Text", 10, 60)
```

`zeichneRechteck(x, y, breite, höhe, "farbe")`

Zeichnet ein ausgefülltes Rechteck.

Parameter:

- `x` , `y` — Position der oberen linken Ecke
- `breite` , `höhe` — Größe in Pixeln
- `"farbe"` — Füllfarbe

```
# Lebensbalken
zeichneRechteck(10, 10, 200, 20, "dunkelgrau")      # Hintergrund
zeichneRechteck(10, 10, leben * 2, 20, "grün")       # Gefüllter Anteil

# Dekorative Elemente
zeichneRechteck(0, 0, 800, 40, "dunkelblau")         # HUD-Hintergrund
```

`zeichneKreis(x, y, radius, "farbe")`

Zeichnet einen ausgefüllten Kreis.

Parameter:

- `x` , `y` — Mittelpunkt des Kreises
- `radius` — Radius in Pixeln
- `"farbe"` — Füllfarbe

```
zeichneKreis(400, 300, 50, "gelb")      # Sonne
zeichneKreis(held.x, held.y, 5, "rot") # Markierungspunkt
```

Sound

`spieleSound("name")`

Spielt einen kurzen Soundeffekt ab.

Verfügbare Sounds:

Name	Typischer Einsatz
`"sprung"`	Figur springt
`"punkt"`	Punkt oder Erfolg
`"münze"`	Münze oder Item einsammeln
`"treffer"`	Schaden erleiden
`"klick"`	Button oder Figur antippen
`"fehler"`	Falsche Aktion
`"gewonnen"`	Spiel gewonnen
`"verloren"`	Spiel verloren

```
wenn kollidiert(held, münze) dann
  punkte ist punkte + 1
  spieleSound("münze")
ende

wenn leben = 0 dann
  spieleSound("verloren")
  spielEnde("Game Over!")
ende
```

Partikel-Effekte

`partikel(x, y, "typ")`

Erzeugt einen vordefinierten Partikel-Effekt an der angegebenen Position.

Parameter:

- `x` , `y` — Startposition der Partikel
- `"typ"` — Name des Effekts

Vordefinierte Typen:

Typ	Beschreibung	Typischer Einsatz
`"explosion"`	Orange/gelbe Explosion	Figur wird getroffen oder zerstört
`"funken"`	Goldene Funken	Treffer, Kontakt
`"rauch"`	Grauer aufsteigender Rauch	Motor, Zerstörung
`"konfetti"`	Buntes Konfetti-Regen	Gewonnen, Erfolg
`"feuer"`	Aufsteigende Flammen	Feuer, Gefahr
`"sterne"`	Glitzernde Sterne	Magie, Bonus
`"wasser"`	Wasser-Spritzer	Wasserberührung

```
wenn kollidiert(held, gegner) dann
  partikel(gegner.x, gegner.y, "explosion")
  spieleSound("treffer")
ende

wenn punkte >= 10 dann
  partikel(400, 300, "konfetti")
  spieleSound("gewonnen")
  spielEnde("Glückwunsch!")
ende
```

`partikelEffekt(x, y, anzahl, tempo, dauer, größe, "farbe")`

Erzeugt einen vollständig anpassbaren Partikel-Effekt.

Parameter:

- `x` , `y` — Startposition
- `anzahl` — Anzahl der Partikel (1–100)
- `tempo` — Geschwindigkeit der Partikel (1–10)
- `dauer` — Lebensdauer in Frames (30 = ~0.5 Sekunden bei 60 FPS)
- `größe` — Partikelgröße in Pixeln
- `"farbe"` — Farbe der Partikel

```
# Goldener Glitzer
partikelEffekt(200, 100, 50, 3, 90, 3, "gold")

# Großer roter Ausbruch
partikelEffekt(400, 300, 80, 8, 45, 6, "rot")

# Feines weißes Glimmen
partikelEffekt(held.x, held.y, 20, 2, 120, 2, "weiß")
```

`partikelLöschen()`

Entfernt alle aktiven Partikel sofort vom Canvas.

```
partikelLöschen()    # Aufräumen beim Neustart
```

Spielsteuerung

`spielEnde("nachricht")`

Beendet die Spielschleife und zeigt einen Endscreen mit der Nachricht.

```
spielEnde("Du hast gewonnen! Punkte: " + punkte)
spielEnde("Game Over!")
spielEnde("Neuer Highscore: " + punkte)
```

`spielPause()`

Hält die Spielschleife an. `zeigeText()` und Zeichenbefehle werden nicht mehr ausgeführt. Die Eingabeverarbeitung stoppt ebenfalls.

```
wenn tasteGedrückt("escape") dann
  spielPause()
ende
```

`spielWeiter()`

Setzt eine pausierte Spielschleife fort.

```
# (In einem anderen Kontext, z.B. nach einem Timer)
spielWeiter()
```

`spielNeustart()`

Startet das gesamte Programm von vorn (Setup und Spielschleife werden neu ausgeführt).

```
wenn tasteGedrückt("eingabe") dann
  spielNeustart()
ende
```

`spielZeit()`

Gibt die vergangene Zeit seit dem Start des Spiels in Sekunden zurück.

Rückgabe: Zahl (Sekunden, Dezimalzahl)

```
zeit ist spielZeit()
zeigeText("Zeit: " + zeit + "s", 10, 30)

# Zeitlimit prüfen
wenn spielZeit() > 60 dann
  spielEnde("Zeit abgelaufen!")
ende
```

Schlangen-System

Das Schlangen-System enthält vorgefertigte Logik für Snake-artige Spiele. Eine "Schlange" ist eine Kette von Segmenten, die sich gleichmäßig durch das Spielfeld bewegt.

`neueSchlange(segmentGröße, "farbe")`

Erstellt eine neue Schlange.

Parameter:

- `segmentGröße` — Größe jedes Segments in Pixeln (z.B. 20)
- `"farbe"` — Farbe der Schlange (optional, Standard: `"grün"`)

Rückgabe: Schlangen-ID

```
schlange ist neueSchlange(20, "grün")
```

`schlangeRichtung(id, dx, dy)`

Setzt die Bewegungsrichtung der Schlange.

Parameter:

- `id` — Schlangen-ID
- `dx` — Horizontale Richtung (-1, 0 oder 1)
- `dy` — Vertikale Richtung (-1, 0 oder 1)

```
wenn tasteGedrückt("rechts") dann
  schlangeRichtung(schlange, 1, 0)
ende
wenn tasteGedrückt("links") dann
  schlangeRichtung(schlange, -1, 0)
ende
wenn tasteGedrückt("hoch") dann
  schlangeRichtung(schlange, 0, -1)
ende
wenn tasteGedrückt("runter") dann
  schlangeRichtung(schlange, 0, 1)
ende
```

`schlangeTempo(id, framesProSchritt)`

Setzt die Bewegungsgeschwindigkeit.

Parameter:

- `id` — Schlangen-ID
- `framesProSchritt` — Wie viele Frames zwischen jedem Schritt vergehen (niedriger = schneller)

```
schlangeTempo(schlange, 10)    # Gemäßigt
schlangeTempo(schlange, 5)     # Schnell
schlangeTempo(schlange, 20)    # Langsam (gut zum Einsteigen)
```

`schlangeWachsen(id)`

Lässt die Schlange um ein Segment wachsen (beim nächsten Schritt).

```
wenn kollidiert(schlangeKopf, futter) dann
  schlangeWachsen(schlange)
  spieleSound("münze")
  setzeFigur(futter, zufallsZahl(380), zufallsZahl(380))
ende
```

`schlangeKopfX(id)` / `schlangeKopfY(id)`

Gibt die X- bzw. Y-Position des Schlangenkopfes zurück.

Rückgabe: Zahl (Pixelposition)

```
kopfX ist schlangeKopfX(schlange)
kopfY ist schlangeKopfY(schlange)

# Rand-Kollision prüfen
wenn kopfX < 0 oder kopfX > 400 oder kopfY < 0 oder kopfY > 400 dann
  spielEnde("Wand getroffen!")
ende
```

`schlangeSelbstkollision(id)`

Gibt `wahr` zurück, wenn der Schlangenkopf ein eigenes Körpersegment berührt.

Rückgabe: `wahr` oder `falsch`

```
wenn schlangeSelbstkollision(schlange) dann
  spieleSound("verloren")
  spielEnde("Du hast dich selbst gebissen!")
ende
```

`schlangeKollision(id, x, y, breite, höhe)`

Prüft, ob der Schlangenkopf ein Rechteck berührt.

Parameter:

- `id` — Schlangen-ID
- `x`, `y` — Position des Rechtecks
- `breite`, `höhe` — Größe des Rechtecks

Rückgabe: `wahr` oder `falsch`

```
# Futter einsammeln (als Rechteck)
wenn schlangeKollision(schlange, futter.x, futter.y, 20, 20) dann
  schlangeWachsen(schlange)
  punkte ist punkte + 1
  setzeFigur(futter, zufallsZahl(380), zufallsZahl(380))
ende
```

Die Spielschleife

`spielSchleife() ... ende`

Das zentrale Konstrukt jedes GameFuchs-Spiels. Der Block zwischen `spielSchleife()` und `ende` wird 60 Mal pro Sekunde ausgeführt.

Wichtig:

- `spielSchleife()` ist kein normaler Funktionsaufruf
- Es darf nur **einmal** pro Programm aufgerufen werden
- Alles vor `spielSchleife()` ist der Setup-Bereich (läuft einmal)
- Innerhalb der Spielschleife können `tasteGedrückt()` und alle Zeichenbefehle verwendet werden

Empfohlene Reihenfolge innerhalb der Spielschleife

```
spielSchleife()

  # 1. Eingabe verarbeiten
  wenn tasteGedrückt("links") dann ... ende

  # 2. Spiellogik prüfen
  wenn kollidiert(held, münze) dann ... ende
  wenn figurAmRand(held) dann ... ende

  # 3. Anzeige aktualisieren (am Ende)
  textFarbe("weiß")
  zeigeText("Punkte: " + punkte, 10, 30)
  zeichneRechteck(...)

  # 4. Spielende prüfen
  wenn punkte >= ziel dann
    spielEnde("Gewonnen!")
  ende

ende
```

Vollständiges Beispiel: Punkte-Sammler

Dieses Beispiel kombiniert alle wichtigen Konzepte:

```
# SETUP
spielfeld(400, 300)
hintergrund("dunkelblau")

held ist neueFigur("fuchs")
setzeFigur(held, 200, 150)
figurGröße(held, 40, 40)

münze ist neueFigur("robert")
figurGröße(münze, 30, 30)
setzeFigur(münze, zufallsZahl(370), zufallsZahl(270))

punkte ist 0
ziel ist 5

# SPIELSCHLEIFE
spielSchleife()

  # Eingabe
  wenn tasteGedrückt("links") dann
    bewegeFigur(held, -4, 0)
    figurSpiegeln(held, wahr)
  ende
  wenn tasteGedrückt("rechts") dann
    bewegeFigur(held, 4, 0)
    figurSpiegeln(held, falsch)
  ende
  wenn tasteGedrückt("hoch") dann
    bewegeFigur(held, 0, -4)
  ende
  wenn tasteGedrückt("runter") dann
    bewegeFigur(held, 0, 4)
  ende

  # Spiellogik
  wenn kollidiert(held, münze) dann
    punkte ist punkte + 1
    spieleSound("münze")
    partikel(münze.x, münze.y, "sterne")
    setzeFigur(münze, zufallsZahl(370), zufallsZahl(270))
  ende

  # Anzeige
  textFarbe("weiß")
  textGröße(20)
  zeigeText("Punkte: " + punkte + "/" + ziel, 10, 30)

  # Spielende
  wenn punkte >= ziel dann
    partikel(200, 150, "konfetti")
    spieleSound("gewonnen")
    spielEnde("Gewonnen! Du hast " + punkte + " Punkte gesammelt!")
  ende

ende
```

Übersicht: Alle GameFuchs-Funktionen

Funktion	Kategorie	Rückgabe
`spielfeld(b, h)`	Setup	—
`hintergrund("farbe")`	Setup	—
`hintergrundBild("name")`	Setup	—
`neueFigur("name")`	Figuren	Figur-ID
`setzeFigur(id, x, y)`	Figuren	—
`bewegeFigur(id, dx, dy)`	Figuren	—
`figurGröße(id, w, h)`	Figuren	—
`figurSichtbar(id, bool)`	Figuren	—
`figurBild(id, "name")`	Figuren	—
`figurSpiegeln(id, bool)`	Figuren	—
`tasteGedrückt("taste")`	Eingabe	`wahr` / `falsch`
`figurGeklickt(id)`	Eingabe	`wahr` / `falsch`
`kollidiert(id1, id2)`	Kollision	`wahr` / `falsch`
`figurAmRand(id)`	Kollision	`wahr` / `falsch`
`rechteckKollision(id, x, y, w, h)`	Kollision	`wahr` / `falsch`
`schwerkraft(id, stärke)`	Physik	—
`figurSprung(id, kraft)`	Physik	—
`figurAmBoden(id)`	Physik	`wahr` / `falsch`
`plattform(x, y, w, h, "farbe")`	Plattformen	Plattform-ID
`entfernePlattform(id)`	Plattformen	—
`zeigeText(text, x, y)`	Anzeige	—
`zeigeText(text, x, y, gr, "farbe")`	Anzeige	—
`textGröße(gr)`	Anzeige	—
`textFarbe("farbe")`	Anzeige	—
`zeichneRechteck(x, y, w, h, "farbe")`	Anzeige	—
`zeichneKreis(x, y, r, "farbe")`	Anzeige	—
`spieleSound("name")`	Sound	—
`partikel(x, y, "typ")`	Partikel	—

Funktion	Kategorie	Rückgabe
`partikelEffekt(x, y, n, t, d, gr, "farbe")`	Partikel	—
`partikelLöschen()`	Partikel	—
`spielEnde("text")`	Steuerung	—
`spielPause()`	Steuerung	—
`spielWeiter()`	Steuerung	—
`spielNeustart()`	Steuerung	—
`spielZeit()`	Steuerung	Sekunden
`neueSchlange(größe, "farbe")`	Schlange	Schlangen-ID
`schlangeRichtung(id, dx, dy)`	Schlange	—
`schlangeTempo(id, frames)`	Schlange	—
`schlangeWachsen(id)`	Schlange	—
`schlangeKopfX(id)`	Schlange	Zahl
`schlangeKopfY(id)`	Schlange	Zahl
`schlangeSelbstkollision(id)`	Schlange	`wahr` / `falsch`
`schlangeKollision(id, x, y, w, h)`	Schlange	`wahr` / `falsch`