EIN TUTORIAL ZUR PROGRAMMIERSPRACHE KISTE

KistePix

Programmieren lernen auf Deutsch —
bau deinen eigenen Pixel-Editor

von **Sascha Rust**

kiste-lang.org · erstellt mit kiste 0.9.38 · Stand 20.07.2026

Inhalt

Willkommen

Dieses Buch bringt dir das Programmieren bei — von Grund auf, auf Deutsch, und mit einem echten Ziel vor Augen: Am Ende hast du dein eigenes **Zeichenprogramm** gebaut, mit dem man Pixel-Bilder malt. Es heißt **KistePix**.

Du brauchst dafür **kein Vorwissen**. Keine Mathematik über die Grundschule hinaus, keine Erfahrung mit anderen Programmiersprachen, kein Englisch. Nur Neugier und die Bereitschaft, Dinge selbst auszuprobieren.

Was ist Kiste?

Kiste ist eine Programmiersprache — die Sprache, in der du dem Computer sagst, was er tun soll. Das Besondere an ihr: Sie ist **vollständig auf Deutsch**. Ihre Befehlswörter heißen `wenn`, `solange`, `funktion`, `gib` oder `nimm` — du liest ein Programm also fast wie einen deutschen Satz.

Die allermeisten Programmiersprachen sind auf Englisch. Wer Programmieren lernt, muss deshalb oft zwei Dinge auf einmal stemmen: die fremde Sprache *und* das Denken in Abläufen. Kiste nimmt dir die erste Hürde ab. So kannst du dich ganz darauf konzentrieren, *wie* ein Programm denkt — die Wörter selbst stehen dir nicht im Weg. Was du hier lernst, gilt anschließend für jede andere Sprache genauso; die Ideen sind überall dieselben.

Wie Kiste entstand

Ein kurzes, persönliches Wort dazu, woher Kiste kommt. Ich habe mich **schon immer fürs Programmieren interessiert** und war lange auf der Suche nach der „perfekten" Programmiersprache. Die gibt es natürlich nicht — aber die Suche ließ mich nicht los.

Als ich anfing, Kiste zu entwickeln, hätte ich nie gedacht, dass ich es wirklich zu Ende bringe. Es war zuerst ein **Spaßprojekt**, mehr für mich selbst. Doch je länger ich mich damit beschäftigte und je mehr Nächte ich über einer kniffligen Frage grübelte, desto größer wurde die Freude, **etwas Eigenes zu erschaffen**. Als die Sprache langsam Gestalt annahm und ich ihr Potenzial erkannte, beschäftigte ich mich nur noch intensiver mit ihr — sie ließ mich nicht mehr los.

Der Traum dahinter: eine **deutsche** Programmiersprache mit der **Einfachheit von Python** und der **Geschwindigkeit einer großen Sprache** — eine, die dein Programm in ein echtes, schnelles Maschinenprogramm *übersetzt* (man sagt: **kompiliert**), statt es langsam Zeile für Zeile abzuarbeiten (zu *interpretieren*). Diese Verbindung aus „leicht zu lesen" und trotzdem „schnell" hat mich fasziniert — und ist bis heute der Kern von Kiste.

Und woher kommt der Name?

Weniger feierlich, als man denken könnte: Ich besaß damals die Internet-Adresse „kiste", eigentlich für ein anderes Projekt, aus dem am Ende nichts wurde. Also taufte ich meine Sprache kurzerhand **Kiste**. Wie sich herausstellte, passt der Name wie angegossen — denn eine Kiste ist genau das, was du beim Programmieren ständig benutzt: ein Behälter, in den du Werte hineinlegst und später wieder herausholst (man nennt sie **Variablen**). Und „Kiste" ist im Deutschen ja auch ein liebevolles Wort für den Computer selbst — „eine schnelle Kiste". Manchmal findet der beste Name einen ganz von allein.

Und was ist KistePix?

KistePix ist das Programm, das du in diesem Buch Schritt für Schritt baust: ein kleiner, aber richtiger **Pixel-Editor**. Du malst damit auf einem Raster aus großen, bunten Pixeln, wählst Farben, ziehst Rechtecke, füllst Flächen auf einen Klick und exportierst am Ende eine echte Bilddatei. Solche Pixel-Grafiken sind das Herz vieler Spiele — und mit KistePix verstehst du, wie so ein Werkzeug von innen funktioniert, weil du es selbst gebaut hast.

Wir halten den Editor bewusst **überschaubar**: Zeichenfläche, Stift, Radierer, Rechteck, Farben, Füllen, PNG-Export — und dann ist er fertig. Das ist kein Notbehelf, sondern Absicht: ein kleines, *vollständiges* Programm, das man ganz versteht, bringt dir mehr bei als ein riesiges, das man nur halb durchschaut. Und weil du danach das ganze Gerüst kennst, kannst du es selbst weiterbauen — das letzte Kapitel zeigt dir genau, wohin die Reise von dort aus geht (Ebenen, Animation, Kreise, Auswahl …).

KistePix ist von Anfang bis Ende ein **echtes Kiste-Programm**. Jede Zeile, die im Buch steht, wurde vor dem Druck wirklich ausgeführt — nichts ist erfunden oder bis zur Unwahrheit vereinfacht. Wo wir sagen „das hält auch bei großen Bildern", haben wir es unter Last *geprüft*, nicht bloß behauptet.

Wie du dieses Buch benutzt

Das Buch hat zwei große Teile und zwei Anhänge zum Nachschlagen:

- **Teil I — Programmieren mit Kiste.** Hier lernst du Kiste von null an: Ausgeben, Variablen, Rechnen, Text, Entscheidungen, Schleifen, Funktionen, Listen und die ersten Schritte mit Fenster und Maus. Jedes Thema kommt in kleinen Häppchen, mit einem winzigen Beispiel, das du selbst startest.
- **Teil II — KistePix bauen.** Jetzt setzen wir alles ein und bauen den Editor Stufe für Stufe — vom leeren Fenster bis zum fertigen Werkzeug mit Export.
- **Anhang A — Glossar.** Alle Fachbegriffe kurz erklärt, alphabetisch, zum schnellen Nachschlagen.
- **Anhang B — Sprachreferenz.** Die Schlüsselwörter, Funktionen und Module von Kiste kompakt beisammen — dein Nachschlagewerk für später.

Am Ende jedes Kapitels gibt es eine kleine **Prüfung** („Teste dich") und eine oder zwei **Aufgaben** mit **Lösung**. Nutze sie — sie festigen das Gelernte weit besser als bloßes Weiterlesen.

Der wichtigste Rat: tippe selbst

Widerstehe der Versuchung, Code nur zu überfliegen. **Tippe jedes Beispiel selbst ab und starte es.** Beim Tippen merkst du, wie eine Zeile aufgebaut ist; beim Starten siehst du sofort, ob du sie verstanden hast. Fehler sind dabei kein Malheur, sondern der Normalfall — man lernt Programmieren, indem man etwas kaputt macht und wieder repariert. Den **fertigen Code zu jedem Kapitel und jeder Bau-Stufe** gibt es außerdem zum Herunterladen unter **kiste-lang.org/de/kistepix-buch** — als Sicherheitsnetz und zum Vergleichen.

Bevor es losgeht, richten wir gemeinsam deine Werkstatt ein: **Kapitel 0** zeigt dir, wie du Kiste auf deinem Rechner installierst und dein allererstes Programm startest. Los geht's!

Programmieren mit Kiste

In diesem Teil lernst du Programmieren von Grund auf — mit der Sprache Kiste, ganz auf Deutsch. Wir fangen bei null an: kein Vorwissen nötig. Schritt für Schritt, an vielen kleinen Beispielen, die du selbst ausprobierst.

Jedes neue Wort, jedes Zeichen wird erklärt, bevor wir es verwenden. Du musst nie raten, was eine Zeile bedeutet. Und du tippst nichts ab, ohne zu verstehen, *warum* es da steht — genau das ist der Unterschied zwischen „Code kopieren" und „programmieren können".

Wie dieser Teil aufgebaut ist

- Wir nehmen **ein** Thema nach dem anderen — in kleinen Häppchen.
- Zu jedem Thema gibt es ein **winziges Beispiel**, das du wirklich startest.
- Wir gehen es **Wort für Wort** durch und zeigen dir die echte *Ausgabe*.
- Eine kleine **Übung** mit Lösung festigt das Gelernte, bevor es weitergeht.

Schon Erfahrung?

Wenn du eine andere Programmiersprache bereits beherrschst, kannst du Teil I überfliegen und dir vor allem ansehen, wo Kiste anders tickt. Für alle anderen gilt: Lass dir Zeit, tippe jedes Beispiel selbst ab und starte es. Verstehen kommt vom Tun — nicht vom Lesen allein.

Für unsere Schweizer Freunde (und alle ohne ß-Taste)

In diesem Buch schreiben wir das **ß** — das „Eszett" oder „scharfe S" —, wie es in Deutschland und Österreich üblich ist, etwa in `größe`. In der Schweiz gibt es das ß nicht; dort schreibt man **ss**. Das ist überhaupt kein Problem: **Kiste versteht beide Schreibweisen.** Eingebaute Funktionen darfst du mit ß *oder* mit ss tippen — `text.großgeschrieben` und `text.grossgeschrieben` bewirken genau dasselbe. Bei deinen *eigenen* Namen wählst du einfach eine Variante und bleibst dabei. So programmiert jede und jeder bequem mit — ganz ohne ß auf der Tastatur.

Kapitel 0 — Kiste installieren und der erste Start

Was du hier lernst

- Wie du **Kiste** kostenlos auf deinen Rechner holst.
- Wie du die **Kiste-IDE** installierst und zum ersten Mal öffnest.
- Warum Windows anfangs warnt — und wie du gelassen daran vorbeikommst.
- Wie du dein **allererstes Programm** tippst, speicherst und startest.

Bevor wir programmieren können, brauchen wir eine **Werkstatt**: ein Programm, in dem du deinen Kiste-Code schreibst und mit einem Tastendruck laufen lässt. Diese Werkstatt heißt **Kiste-IDE**. „IDE" ist die englische Abkürzung für *integrierte Entwicklungsumgebung* — ein etwas großes Wort für eine einfache

Idee: Alles, was du zum Programmieren brauchst, steckt in *einem* Fenster. Ein Texteditor für deinen Code, ein Knopf zum Starten und ein Bereich, in dem die Ausgabe erscheint. Mehr ist es nicht, und mehr brauchst du auch nicht.

Was du brauchst

Einen **Windows-PC** (Windows 10 oder 11) und eine Internetverbindung für den Download. Das war's — Kiste ist **kostenlos**, und du musst dich nirgends anmelden oder ein Konto anlegen. *(Linux? Ein kurzer Hinweis dazu steht am Kapitelende.)*

Schritt 1 — Kiste herunterladen

Öffne deinen Browser und gehe zu:

kiste-lang.org

Das ist die offizielle Seite von Kiste. Klicke dort oben im Menü auf **Download**. Du landest auf einer Seite mit zwei Angeboten — eines für **Windows**, eines für Linux. Klicke beim Windows-Kasten auf den Download-Knopf. Es lädt eine Datei mit einem Namen wie `Kiste-Setup-….exe` herunter (die Zahl darin ist die Version — sie ändert sich mit der Zeit, das ist völlig in Ordnung).

Wo landet die Datei?

In aller Regel im Ordner **Downloads**. Falls dein Browser unten eine Leiste zeigt, kannst du auch direkt dort auf die fertige Datei klicken.

Schritt 2 — Installieren (und die Windows-Warnung)

Mach einen Doppelklick auf die heruntergeladene `Kiste-Setup-….exe`. Jetzt passiert mit großer Wahrscheinlichkeit etwas, das dich im ersten Moment erschreckt: Windows zeigt ein blaues Fenster mit der Überschrift *„Der Computer wurde durch Windows geschützt"* und meldet, der Herausgeber sei unbekannt.

Keine Sorge — das ist normal

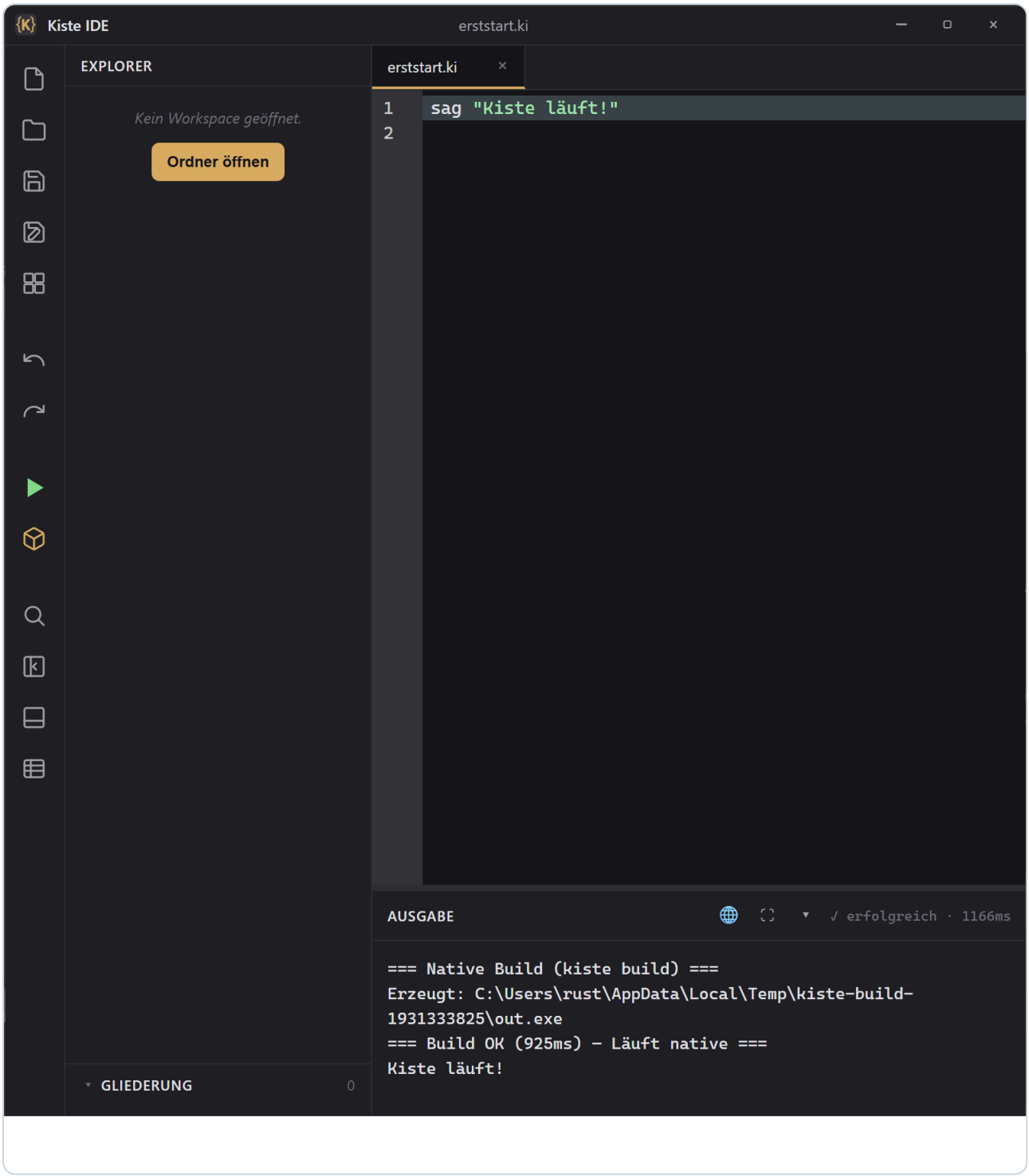
Diese Warnung heißt **nicht**, dass mit der Datei etwas nicht stimmt. Windows zeigt sie bei *jedem* Programm, das noch kein teures „Zertifikat" gekauft hat. Für ein junges, kostenloses Projekt wie Kiste lohnt sich so ein Zertifikat noch nicht — deshalb die Meldung. Auf **kiste-lang.org** kannst du unter dem Download jederzeit die Prüfsummen und einen Virenscan-Bericht (VirusTotal) einsehen, wenn du dich vergewissern möchtest.

So geht's weiter: Klicke im Warnfenster auf **„Weitere Informationen"**. Darunter erscheint dann ein Knopf **„Trotzdem ausführen"**. Klick ihn an — und der Installer startet ganz normal. Folge seinen Schritten (immer weiter auf *Weiter* bzw. *Installieren*). Nach ein paar Sekunden ist Kiste installiert, und die **Kiste-IDE** lässt sich wie jedes andere Programm über das Startmenü öffnen.

Schritt 3 — Die Kiste-IDE zum ersten Mal öffnen

Starte die **Kiste-IDE** aus dem Startmenü. Beim ersten Blick siehst du im Wesentlichen drei Bereiche:

- In der **Mitte** die große, noch leere Fläche — der **Editor**. Hier tippst du deinen Code.
- Ganz **oben** eine Leiste mit Knöpfen. Der wichtigste für uns heißt **„Ausführen (F5)"** — damit startest du dein Programm.
- **Unten** der **Ausgabe**-Bereich. Solange nichts läuft, ist er leer; sobald dein Programm etwas ausgibt, erscheint es genau hier.



Die Kiste-IDE nach dem ersten Lauf: links die Seitenleiste mit dem grünen **Ausführen**-Knopf (▶), in der Mitte der **Editor** mit deinem Programm, unten der **Ausgabe**-Bereich. Ganz unten steht das Ergebnis: `Kiste läuft!`

Falls der Ausgabe-Bereich bei dir nicht zu sehen ist: Mit **Strg + J** blendest du ihn ein und aus.

Schritt 4 — Dein allererstes Programm

Jetzt schreiben wir das kürzeste sinnvolle Programm der Welt: eine einzige Zeile, die Kiste etwas sagen lässt. Klick in den Editor (die große Fläche) und tippe genau das hier ab:

erststart.ki

```
sag "Kiste läuft!"
```

Kein Grund, jetzt schon jedes Zeichen zu verstehen — das machen wir gleich in Kapitel 1 gründlich. Für den Moment reicht: `sag` heißt „gib etwas aus", und zwischen den Anführungszeichen steht, *was* ausgegeben werden soll.

Und jetzt der große Moment: Drück **F5** (oder klick oben auf **Ausführen**). Kiste führt aus, was gerade im Editor steht. Unten im Ausgabe-Bereich erscheint:

```
AUSGABE
Kiste läuft!
```

Eine Zeile mehr als erwartet?

In der IDE steht über deiner Ausgabe noch eine kurze technische Notiz — etwas wie *„Native Build … Build OK … Läuft native*". Das ist völlig normal und kein Fehler: Kiste übersetzt dein Programm in Sekundenbruchteilen in ein echtes Windows-Programm und startet es dann. Die Zeile, auf die es ankommt, ist die **letzte** — und dort steht dein `Kiste läuft!`

Zum *schnellen Ausprobieren* darfst du also einfach tippen und F5 drücken — du musst nicht extra vorher speichern. Sobald du deine Arbeit aber **behalten** willst, musst du sie speichern, sonst ist sie beim Schließen der IDE weg. Drück dazu **Strg + S**; die IDE fragt nach einem Namen — nenn die Datei `erststart.ki`. Wichtig ist die Endung `.ki`: Daran erkennt Kiste, dass es sich um ein Kiste-Programm handelt (so wie `.docx` für ein Word-Dokument steht). Merke dir den Ordner, in den du speicherst; dort legen wir später auch die Dateien für KistePix ab.

Faustregel fürs Speichern

Ausprobieren geht jederzeit ohne Speichern — praktisch, wenn du nur kurz etwas testen willst. Aber gewöhn dir an, vor jedem Start kurz **Strg + S** zu drücken, sobald du an etwas Richtigem arbeitest. Gespeichertes geht nicht verloren.

Glückwunsch — das war dein erstes laufendes Programm! Der Computer hat getan, was du ihm aufgeschrieben hast. Genau darum geht es beim Programmieren, und alles Weitere in diesem Buch ist nur noch eine Steigerung dieses einen Erlebnisses.

Probier's aus

Ändere den Text zwischen den Anführungszeichen — schreib zum Beispiel deinen Namen hinein — und drück noch einmal **F5**. Die Ausgabe ändert sich mit. So arbeiten wir das ganze Buch über: tippen, F5 drücken, schauen, was passiert.

Speichern und wieder öffnen

Zwei Handgriffe wirst du ständig brauchen — präg sie dir gleich ein:

- **Speichern: Strg + S**. Tu das oft — am besten vor jedem Start. Gespeichert wird immer die Datei, an der du gerade arbeitest.
- **Öffnen: Strg + O** holt eine bereits gespeicherte `.ki`-Datei zurück in den Editor.

Wenn etwas klemmt

- **Nach F5 passiert nichts / der Ausgabe-Bereich ist leer.** Blende ihn mit **Strg + J** ein. Und schau nach, ob im Editor wirklich eine `sag`-Zeile steht — nur die erzeugt eine Ausgabe.
- **Es erscheint eine rote Fehlermeldung.** Das ist kein Beinbruch, sondern Kistes Art, dir zu helfen. Meist fehlt ein Anführungszeichen oder es hat sich ein Tippfehler eingeschlichen. Vergleiche deine Zeile Zeichen für Zeichen mit dem Beispiel oben — besonders die beiden `"` müssen paarweise da sein.
- **Windows lässt den Installer partout nicht zu.** Wiederhole Schritt 2 und achte darauf, wirklich zuerst auf *„Weitere Informationen"* zu klicken — erst dann taucht *„Trotzdem ausführen"* auf.

Und wenn ich Linux benutze?

Kiste gibt es auch für Linux — auf der Download-Seite als Paket zum Herunterladen (`kiste-linux-….tar.gz`). Der bequeme Ein-Klick-Installer wie unter Windows ist dort noch in Arbeit; bis dahin entpackst du das Paket von Hand und startest die IDE aus dem entpackten Ordner. Alles Weitere in diesem Buch — der Code, die Beispiele, KistePix selbst — funktioniert unter Linux **genau gleich**. Nur dieser eine Installations-Schritt sieht anders aus.

Geschafft

Deine Werkstatt steht: Kiste ist installiert, die IDE läuft, und du hast dein erstes Programm gestartet. Ab jetzt geht es ums Programmieren selbst. In **Kapitel 1** zerlegen wir diese eine Zeile Wort für Wort und schauen, was wirklich dahintersteckt.

Teste dich

1. Wie heißt die Datei-Endung, an der Kiste ein Programm erkennt?
2. Mit welcher Taste startest du dein Programm?
3. Windows warnt beim Installieren. Was bedeutet diese Warnung — und wie kommst du daran vorbei?

Lösung

(1) `.ki`. (2) **F5** (oder der Knopf „Ausführen"). (3) Sie bedeutet nur, dass Kiste (noch) kein kostenpflichtiges Zertifikat hat — nicht, dass etwas mit der Datei nicht stimmt. Du klickst auf *„Weitere Informationen"* und dann auf *„Trotzdem ausführen"*.

Kapitel 1 — Dein erstes Programm

Was du hier lernst

- Was ein Programm überhaupt ist.
- Wie du Kiste etwas **ausgeben** lässt — mit `sag`.
- Dass Anweisungen **von oben nach unten** ablaufen.
- Was ein **Kommentar** ist und wozu er gut ist.

In Kapitel 0 hast du deine Werkstatt eingerichtet und schon eine Zeile laufen lassen. Jetzt sehen wir uns in Ruhe an, *was* dabei eigentlich passiert ist — denn diese eine Zeile enthält schon die wichtigste Grundidee des Programmierens.

Was ist ein Programm?

Ein Computer tut nichts von allein. Er wartet auf **Anweisungen** — und führt sie genau so aus, wie du sie aufschreibst, eine nach der anderen. Ein **Programm** ist nichts anderes als eine solche Liste von Anweisungen. Programmieren heißt: dem Computer in seiner Sprache aufschreiben, was er tun soll. Unsere Sprache heißt **Kiste**, und sie ist auf Deutsch — du schreibst also fast wie in normalen Worten.

Fangen wir mit der kleinsten nützlichen Anweisung an: etwas *ausgeben*, also auf den Bildschirm schreiben.

Das kleinste Programm

Hier ist ein vollständiges Kiste-Programm. Es ist eine einzige Zeile lang:

hallo.ki

```
sag "Hallo, KistePix!"
```

Tippe es in die Kiste-IDE (die du in **Kapitel 0** eingerichtet hast) und drücke **F5**. Unten im Ausgabe-Bereich erscheint:

```
AUSGABE
Hallo, KistePix!
```

Zerlegt — Wort für Wort

Diese eine Zeile besteht aus zwei Teilen. Sehen wir sie uns ganz genau an:

`sag "Hallo, KistePix!"`

- `sag` ist eine **Anweisung**. Sie bedeutet: „Gib das Folgende aus." Du sprichst Kiste damit direkt an — wie einen Befehl: *sag …!*
- `"Hallo, KistePix!"` ist der **Text**, der ausgegeben werden soll. Die **Anführungszeichen** `" "` sind die „Verpackung": Sie markieren, wo der Text anfängt und wo er aufhört. Ausgegeben wird nur, was *zwischen* ihnen steht — die Anführungszeichen selbst erscheinen nicht.

Du könntest genauso `sag "Guten Morgen"` schreiben, oder `sag "12345"` — Kiste gibt immer genau das aus, was zwischen den Anführungszeichen steht. Einen solchen Text in Anführungszeichen nennt man übrigens eine **Zeichenkette** (weil er eine Kette einzelner Zeichen ist). Uns reicht fürs Erste das Bild von der „Verpackung".

Mehrere Anweisungen — eine nach der anderen

Ein Programm darf natürlich aus mehreren Zeilen bestehen. Kiste arbeitet sie **von oben nach unten** ab — in genau der Reihenfolge, in der sie dastehen:

mehrere.ki

```
sag "Guten Morgen!"
sag "Heute bauen wir einen Pixel-Editor."
sag "Fangen wir an."
```

Das ergibt drei Zeilen Ausgabe, in derselben Reihenfolge wie im Programm:

```
AUSGABE
Guten Morgen!
Heute bauen wir einen Pixel-Editor.
Fangen wir an.
```

Vertauschst du zwei `sag`-Zeilen im Programm, vertauscht sich auch die Ausgabe. Diese **Reihenfolge von oben nach unten** ist eine der wichtigsten Grundregeln überhaupt — wir werden uns noch oft darauf verlassen.

Kommentare — Notizen für Menschen

Manchmal willst du dir (oder anderen) im Programm etwas notieren, das Kiste *nicht* ausführen soll. Dafür gibt es **Kommentare**. Alles, was hinter zwei Schrägstrichen `//` steht, ignoriert Kiste vollständig — bis zum Zeilenende:

kommentar.ki

```
// Dieses Programm begrüßt dich.
// Zeilen mit zwei Schrägstrichen ignoriert Kiste vollständig.
sag "Ich werde ausgegeben."
// sag "Ich nicht – ich bin auskommentiert."
```

Obwohl im Programm vier Zeilen stehen, erscheint nur eine einzige in der Ausgabe — die anderen sind Kommentare:

```
AUSGABE
Ich werde ausgegeben.
```

Kommentare ändern also nie, *was* ein Programm tut. Sie erklären, *warum* es etwas tut — und sind ein praktischer Trick, um eine Zeile vorübergehend „stillzulegen", ohne sie zu löschen (wie bei der letzten `sag`-Zeile oben). Wir setzen sie im ganzen Buch ein, um Code zu erklären.

Häufige Stolperstelle

Vergisst du ein Anführungszeichen — etwa `sag "Hallo` ohne das schließende `"` — weiß Kiste nicht, wo der Text aufhört, und meldet einen Fehler. Anführungszeichen kommen immer **paarweise**: eines am Anfang, eines am Ende des Textes.

Teste dich

1. Was gibt `sag "42"` aus?
2. In welcher Reihenfolge führt Kiste mehrere Zeilen aus?
3. Was passiert mit einer Zeile, die mit `//` beginnt?

Aufgabe ★

Schreibe ein Programm, das sich in drei Zeilen vorstellt: deinen Namen, dein Alter und dein Lieblingstier — je eine `sag`-Zeile. Setze über das Programm einen Kommentar, der sagt, worum es geht. Speichere, starte es mit F5 und prüfe, ob die Ausgabe stimmt.

Lösung

Teste dich: (1) Genau `42` — Kiste gibt aus, was zwischen den Anführungszeichen steht. (2) Von oben nach unten, eine Zeile nach der anderen. (3) Kiste ignoriert sie vollständig; sie erscheint nicht in der Ausgabe.

Aufgabe: So könnte es aussehen (mit deinen eigenen Angaben):

`vorstellung.ki`

```
// Ich stelle mich vor.
sag "Ich heiße Robin."
sag "Ich bin 10 Jahre alt."
sag "Mein Lieblingstier ist der Fuchs."
```

Kapitel 2 — Variablen: Werte merken

Was du hier lernst

- Was eine **Variable** ist und wozu du sie brauchst.
- Wie du mit `nimm` einen Wert unter einem Namen **merkst**.
- Wie du Werte mit **geschweiften Klammern** `{ }` in Text einsetzt.
- Wie du einen Wert später **änderst** — und wann du ihn mit `fest` **festnagelst**, damit er sich *nicht* ändern kann.

Wozu Variablen?

Bisher haben wir Texte direkt ausgegeben. Aber Programme müssen sich Dinge **merken**: einen Namen, einen Punktestand, eine Farbe. Dafür gibt es **Variablen**. Erinnerst du dich an die Idee hinter dem Namen „Kiste"? Eine Variable ist genau so eine Kiste: ein **Behälter mit einem Namen**, in den du einen Wert legst und ihn später am Namen wieder herausholst.

Das ist der ganze Trick — und trotzdem eine der mächtigsten Ideen überhaupt. Schauen wir sie uns an.

Deine erste Variable

Wir legen einen Namen in eine Kiste namens `name`:

name.ki

```
nimm name = "Mira"
sag "Hallo, {name}!"
```

Das gibt aus:

```
AUSGABE
Hallo, Mira!
```

Zerlegt — Wort für Wort

`nimm name = "Mira"`

- `nimm` sagt Kiste: „Leg eine neue Variable an." Man liest es wörtlich — *nimm* eine Kiste.
- `name` ist der **Name** der Variablen — das Etikett auf der Kiste. Den denkst du dir selbst aus; er sollte sagen, was drinsteckt.
- Das **Gleichheitszeichen** `=` bedeutet hier nicht „ist gleich", sondern **„leg das da hinein"**. Man spricht es als *„bekommt"*: *name bekommt „Mira"*.
- `"Mira"` ist der **Wert**, der in die Kiste kommt — hier ein Text, also in Anführungszeichen (wie in Kapitel 1).

Ab jetzt steht der Name `name` überall für den Wert `Mira`.

Werte in Text einsetzen: die geschweiften Klammern

In der zweiten Zeile ist etwas Neues passiert. Sieh genau hin:

`sag "Hallo, {name}!"`

Innerhalb des Textes stehen **geschweifte Klammern** um einen Variablennamen: `{name}`. Diese Klammern sind ein **Platzhalter**: Kiste ersetzt `{name}` beim Ausgeben durch den *Inhalt* der Variablen. Aus `"Hallo, {name}!"` wird also `Hallo, Mira!`. Alles andere im Text bleibt, wie es ist — nur was in `{ }` steht, wird durch den Wert ersetzt.

Dieses Einsetzen nennt man **Interpolation** (ein Fachwort — du musst es dir nicht merken). Wichtig ist die Idee: **fester Text und Variablenwerte in einem Satz zusammenbauen.** Das brauchen wir ununterbrochen, und wir machen es im ganzen Buch *immer* so — es ist die klare, saubere Schreibweise.

> **Nur innerhalb von Anführungszeichen**
>
> Die geschweiften Klammern wirken **nur in einem Text**, also zwischen zwei `"`. Schreibst du `{name}` außerhalb, ist es kein Platzhalter. Und die Öffnungs-Klammer `{` gehört immer mit ihrer Schließungs-Klammer `}` zusammen — genau wie die Anführungszeichen paarweise kommen.

Zahlen in Variablen — und mehrere in einem Satz

In eine Variable passt nicht nur Text, sondern auch eine **Zahl**. Zahlen schreibst du *ohne* Anführungszeichen — sie sind keine Buchstabenkette, sondern ein Rechenwert (dazu mehr in Kapitel 3). Und du darfst beliebig viele Platzhalter in einen Satz setzen:

`mehrere.ki`

```
nimm name = "Mira"
nimm alter = 12
sag "{name} ist {alter} Jahre alt."
```

Kiste setzt beide Werte an ihren Platz:

```
AUSGABE
Mira ist 12 Jahre alt.
```

Beachte den Unterschied: `name` bekommt `"Mira"` *mit* Anführungszeichen (Text), `alter` bekommt `12` *ohne* (Zahl). Im `sag`-Satz dagegen stehen beide gleich als Platzhalter `{name}` und `{alter}` — Kiste weiß selbst, wie es jeden Wert einsetzt.

Einen Wert ändern

Eine mit `nimm` angelegte Variable ist **veränderlich** — du kannst ihr später einen neuen Wert geben. Dafür brauchst du `nimm` *nicht noch einmal*; die Kiste gibt es ja schon. Du schreibst einfach den Namen, `=` und den neuen Wert:

`aendern.ki`

```
nimm punkte = 0
sag "Punkte am Anfang: {punkte}"
punkte = punkte + 10
sag "Nach dem ersten Treffer: {punkte}"
```

Die Ausgabe zeigt beide Stände:

```
AUSGABE
Punkte am Anfang: 0
Nach dem ersten Treffer: 10
```

Was bedeutet `punkte = punkte + 10`?

Das sieht als Mathe-Gleichung unsinnig aus — aber `=` heißt ja „leg hinein", nicht „ist gleich". Kiste rechnet **zuerst die rechte Seite** aus: `punkte + 10`, also `0 + 10 = 10`. Dann legt es das Ergebnis zurück in `punkte`. Kurz gesagt: *„Nimm den alten Punktestand, zähl 10 dazu, und merk dir das als neuen Punktestand."* Genau so zählt später KistePix mit.

Werte, die sich nicht ändern dürfen: `fest`

Manchmal willst du das Gegenteil: einen Wert, der sich **garantiert nie** ändert — eine Höchstpunktzahl, die Zahl der Farben in der Palette, die Größe der Leinwand. So etwas nennt man eine **Konstante**. Du legst sie mit `fest` statt `nimm` an:

fest.ki

```
fest maximum = 100
nimm punkte = 90
sag "Du hast {punkte} von {maximum} Punkten."
```

```
AUSGABE
Du hast 90 von 100 Punkten.
```

Der Unterschied zeigt sich, wenn du eine Konstante doch zu ändern versuchst. Schreibst du irgendwo danach `maximum = 200`, lässt Kiste das **nicht** zu und meldet:

```
AUSGABE
Konstante "maximum" kann nicht überschrieben werden
```

Das ist kein Ärgernis, sondern ein **Schutz**: Kiste bewahrt dich davor, aus Versehen einen Wert zu verändern, der fest bleiben soll. Faustregel: **Was sich ändern soll, kommt in `nimm`; was fest bleiben soll, in `fest`.** Im Zweifel nimmst du `nimm` — das ist nie falsch.

Gute Namen wählen

Ein Variablenname sagt am besten, *was* drinsteckt: `punkte`, `name`, `maximum` — nicht `x` oder `zeug`. Erlaubt sind Buchstaben, Ziffern und der Unterstrich `_`; ein Name darf aber nicht mit einer Ziffer *beginnen* und enthält keine Leerzeichen. Statt eines Leerzeichens nimmt man den Unterstrich: `höchste_punktzahl`. Gute Namen machen ein Programm lesbar wie einen Satz — auch für dich selbst in einer Woche.

Teste dich

1. Womit legst du eine *veränderliche* Variable an — und womit eine, die fest bleibt?
2. Was gibt Kiste bei `sag "Ich mag {tier}"` aus, wenn vorher `nimm tier = "Katzen"` steht?
3. Kiste führt `nimm zahl = 5` und dann `zahl = zahl + 3` aus. Was steht danach in `zahl`?

Aufgabe ★

Leg eine Variable für deinen Namen und eine für dein Alter an. Gib einen Satz aus, der **beide** mit Platzhaltern enthält. Feiere dann „Geburtstag": erhöhe das Alter um 1 und gib den Satz noch einmal aus.

Lösung

Teste dich: (1) Veränderlich mit `nimm`, fest mit `fest`. (2) `Ich mag Katzen`. (3) `8` — erst wird rechts `5 + 3` gerechnet, dann zurückgelegt.

Aufgabe: So könnte es aussehen:

`geburtstag.ki`

```
nimm name = "Robin"
nimm alter = 10
sag "{name} ist {alter} Jahre alt."
// Ein Jahr später – Geburtstag!
alter = alter + 1
sag "Jetzt ist {name} {alter} Jahre alt."
```

```
AUSGABE
Robin ist 10 Jahre alt.
Jetzt ist Robin 11 Jahre alt.
```

Kapitel 3 — Mit Zahlen rechnen

Was du hier lernst

- Die zwei Sorten Zahlen: **Ganzzahlen** und **Kommazahlen**.
- Die Grundrechenarten `+` `-` `*` `/` — und dass du direkt im Platzhalter rechnen darfst.
- **Ganzzahlig teilen** mit `div` und den **Rest** mit `%` — die zwei heimlichen Stars für Raster wie in KistePix.
- **Punkt vor Strich** und wie du das mit Klammern steuerst.

Zwei Sorten Zahlen

Kiste kennt zwei Arten von Zahlen, und der Unterschied ist einfach:

- Eine **Ganzzahl** ist eine Zahl *ohne* Nachkommastelle: `0`, `7`, `256`, auch `-3`. Fachwort in Kiste: **Ganz**.
- Eine **Kommazahl** hat eine Nachkommastelle: `1.5`, `0.2`, `3.14`. Fachwort: **Komma**.

Wichtig: Das Komma schreibt man in Kiste als **Punkt** — `1.5`, nicht `1,5`. (Das ist in fast allen Programmiersprachen so.) Und Zahlen kommen *ohne* Anführungszeichen — sonst wären es Texte, mit denen man nicht rechnen kann.

Die Grundrechenarten

Rechnen sieht fast aus wie im Matheheft. Nur das Malzeichen ist ein **Stern** `*` (auf der Tastatur über der Ziffer `+`):

rechnen.ki

```
nimm bonbons = 3
nimm kekse = 4
sag "Zusammen: {bonbons + kekse} Naschereien"
sag "Doppelt so viele Kekse: {kekse * 2}"
sag "Ein Bonbon weniger: {bonbons - 1}"
```

AUSGABE

```
Zusammen: 7 Naschereien
Doppelt so viele Kekse: 8
Ein Bonbon weniger: 2
```

Rechnen mitten im Text

Schau dir `{bonbons + kekse}` an: Zwischen den geschweiften Klammern darf nicht nur ein Variablenname stehen, sondern eine **ganze Rechnung**. Kiste rechnet sie zuerst aus und setzt dann das Ergebnis ein. So baust du Text und frisch berechnete Werte in einem Satz zusammen — ganz ohne Zwischenschritt.

Zeichen	Bedeutung	Beispiel	Ergebnis
`+`	plus	`3 + 4`	7
`-`	minus	`3 - 1`	2
`*`	mal	`4 * 2`	8
`/`	geteilt	`17 / 5`	3.4

Kommazahlen

Mit Kommazahlen rechnet Kiste genauso:

komma.ki

```
nimm preis_stift = 1.50
nimm preis_blatt = 0.20
sag "Stift und Blatt kosten zusammen {preis_stift + preis_blatt}."
```

AUSGABE

```
Stift und Blatt kosten zusammen 1.7.
```

Warum `1.7` und nicht `1.70`?

`1.50 + 0.20` ergibt `1.7` — Kiste lässt eine **Null am Ende weg**, weil sie am Wert nichts ändert (`1.70` und `1.7` sind dieselbe Zahl). Das ist richtig gerechnet, sieht nur ungewohnt aus. Wie man Zahlen schön formatiert (etwa immer mit zwei Stellen), lernst du später — für jetzt zählt das Rechnen selbst.

Teilen — gleich auf drei Arten

Beim Teilen wird es spannend, denn es gibt drei nützliche Varianten. Sieh dir alle drei am selben Beispiel an — 17 geteilt durch 5:

teilen.ki

```
sag "17 / 5   = {17 / 5}"
sag "17 div 5 = {17 div 5}"
sag "17 % 5   = {17 % 5}"
```

```
AUSGABE
17 / 5   = 3.4
17 div 5 = 3
17 % 5   = 2
```

Die drei Teil-Arten

- `/` — das **normale Teilen**. Ergebnis ist eine Kommazahl: `17 / 5 = 3.4`.
- `div` — das **ganzzahlige Teilen**. Es fragt: „Wie oft passt die 5 *ganz* in die 17?" Antwort: `3` Mal (der Rest wird weggelassen).
- `%` — der **Rest** („modulo"). Es fragt: „Was bleibt übrig?" Nach 3 mal 5 bleiben von 17 noch `2` übrig.

`div` und `%` wirken unscheinbar, sind aber genau die Werkzeuge, mit denen man in einem **Raster** rechnet — und ein Raster ist das Herz von KistePix. Ein Vorgeschmack:

raster.ki

```
fest breite = 16
fest höhe = 16
sag "Ein Raster von {breite} mal {höhe} hat {breite * höhe} Pixel."
```

```
AUSGABE
Ein Raster von 16 mal 16 hat 256 Pixel.
```

Blick voraus: so findet KistePix jedes Pixel

Später legt KistePix alle Pixel der Leinwand in *einer* langen Reihe ab (das lernst du in Teil II). Aus einer Position im Raster — Spalte `x`, Zeile `y` — rechnet es die Stelle in dieser Reihe mit `y * breite + x` aus. Und umgekehrt holt es aus einer Stelle `nr` die Spalte mit `nr % breite` und die Zeile mit `nr div breite` zurück. Genau die zwei Zeichen von eben. Du siehst: Wir lernen hier nichts „auf Vorrat" — jedes Werkzeug taucht später wieder auf.

Punkt vor Strich — und Klammern

Kiste hält sich an die Rechenregel aus der Schule: **Punkt vor Strich**. Mal und Geteilt werden also vor Plus und Minus gerechnet. Mit **Klammern** bestimmst du die Reihenfolge selbst:

klammern.ki

```
sag "Ohne Klammern: {2 + 3 * 4}"
sag "Mit Klammern:   {(2 + 3) * 4}"
```

```
AUSGABE
Ohne Klammern: 14
Mit Klammern:   20
```

Ohne Klammern rechnet Kiste erst `3 * 4 = 12` und dann `2 + 12 = 14`. Mit Klammern kommt zuerst `(2 + 3) = 5` an die Reihe, dann `5 * 4 = 20`. Im Zweifel setzt du lieber eine Klammer zu viel — das macht die Absicht klar und kostet nichts.

Häufige Stolperstellen

- **Komma statt Punkt.** `1,5` ist für Kiste keine Zahl — schreib `1.5`.
- **Malzeichen vergessen.** `2(3)` versteht Kiste nicht; schreib `2 * 3`.
- **Zahl in Anführungszeichen.** `"3" + "4"` ist Text-Aneinanderhängen, nicht Rechnen — dazu gleich in Kapitel 4. Zum Rechnen: Zahlen ohne Anführungszeichen.

Teste dich

1. Was ergibt `2 + 3 * 4` — und warum?
2. Was ist der Unterschied zwischen `17 / 5`, `17 div 5` und `17 % 5`?
3. Wie schreibst du die Zahl „drei Komma fünf" in Kiste?

Aufgabe ★

Ein Vorrat hat eine bestimmte Zahl an `tage`n. Gib aus, wie viele **volle Wochen** das sind und wie viele **Tage übrig** bleiben. Tipp: volle Wochen mit `div 7`, die übrigen Tage mit `% 7`. Probier es mit `100` Tagen.

Lösung

Teste dich: (1) `14` — Punkt vor Strich: erst `3 * 4 = 12`, dann `2 + 12`. (2) `17 / 5 = 3.4` (normal, Kommazahl), `17 div 5 = 3` (wie oft passt 5 ganz hinein), `17 % 5 = 2` (was übrig bleibt). (3) `3.5` — mit Punkt.

Aufgabe:

wochen.ki

```
nimm tage = 100
sag "{tage} Tage sind {tage div 7} volle Wochen und {tage % 7} Tage."
```

```
AUSGABE
100 Tage sind 14 volle Wochen und 2 Tage.
```

Kapitel 4 — Mit Text arbeiten

Was du hier lernst

- Texte **zusammenfügen** — mit Platzhaltern und mit `+`.
- Einen **Werkzeugkasten** holen: `nutze text`.
- Text **umformen**: groß, klein, erster Buchstabe groß, rückwärts.
- Zeichen **zählen** mit `länge` und Text **durchsuchen** und **ersetzen**.

Text begegnet dir überall: Namen, Titel, Dateinamen, Meldungen. Kiste kann mit Text richtig umgehen — schauen wir uns an, wie.

Texte zusammenfügen

Die erste Art kennst du schon: die **Platzhalter** aus Kapitel 2. Damit baust du Text und Werte bequem in einem Satz zusammen:

zusammen.ki

```
nimm vorname = "Mira"
nimm nachname = "Bach"
sag "Hallo, {vorname} {nachname}!"
```

```
AUSGABE
Hallo, Mira Bach!
```

Es gibt noch eine zweite Art: Mit dem **Pluszeichen** `+` klebst du zwei Texte direkt aneinander. Bei Zahlen bedeutet `+` „zusammenzählen" — bei Texten bedeutet es **„aneinanderhängen"**:

verketten.ki

```
nimm anfang = "Hallo, "
nimm name = "Kiste"
nimm ganzer = anfang + name + "!"
sag ganzer
```

```
AUSGABE
Hallo, Kiste!
```

Welche Art wann?

Beide führen zum Ziel. **Faustregel:** Mischst du festen Text mit Werten, sind **Platzhalter** übersichtlicher (`"Hallo, {name}!"`). Klebst du nur ein paar fertige Texte zusammen, tut es auch `+`. Achte beim `+` auf die Leerzeichen — sie entstehen nicht von selbst: Darum steht in `"Hallo, "` extra ein Leerzeichen am Ende.

Ein Anführungszeichen mitten im Text

Was, wenn im Text selbst ein Anführungszeichen vorkommen soll — etwa `Sie rief: "Fertig!"`? Schreibst du das `"` einfach so hinein, denkt Kiste beim zweiten Anführungszeichen, der Text sei zu Ende, und meldet einen Fehler. Die Lösung ist ein kleines Sonderzeichen: der **Rückstrich** `\` (auf der Tastatur meist mit *AltGr + ß*). Setzt du ihn direkt vor ein Anführungszeichen — `\"` —, sagst du Kiste: „Dieses `"` gehört zum Text, es beendet ihn nicht."

`escape.ki`

```
sag "Sie rief: \"Fertig!\" und lächelte."
```

```
AUSGABE
Sie rief: "Fertig!" und lächelte.
```

Ausgegeben wird nur der Text — die Rückstriche selbst erscheinen nicht; sie sind bloß das Zeichen „das nächste `"` ist echt". Das brauchst du selten, aber wenn, ist gut zu wissen, wie es geht.

Einen Werkzeugkasten holen: `nutze text`

Fürs Zusammenfügen brauchst du nichts weiter. Für kniffligere Kunststücke — Text groß schreiben, umdrehen, durchsuchen — hat Kiste einen eigenen **Werkzeugkasten** namens `text`. So einen Werkzeugkasten nennt man **Modul**. Bevor du seine Werkzeuge verwenden kannst, holst du ihn einmal oben ins Programm — mit `nutze`:

```
nutze text
```

Ab da stehen dir die `text`-Werkzeuge zur Verfügung. Du rufst sie mit dem Modulnamen, einem Punkt und dem Werkzeugnamen auf — zum Beispiel `text.großgeschrieben(…)`. Der Punkt heißt so viel wie: „aus dem Kasten `text` das Werkzeug `großgeschrieben`".

`modul.ki`

```
nutze text
nimm gruss = "hallo welt"
sag "Groß geschrieben: {text.großgeschrieben(gruss)}"
sag "Erster Buchstabe groß: {text.kapitalisiert(gruss)}"
sag "Rückwärts: {text.umgekehrt(gruss)}"
```

```
AUSGABE
Groß geschrieben: HALLO WELT
Erster Buchstabe groß: Hallo welt
Rückwärts: tlew ollah
```

Zerlegt

- `text.großgeschrieben(gruss)` gibt den Text in **Großbuchstaben** zurück.
- `text.kapitalisiert(gruss)` macht nur den **ersten Buchstaben** groß — praktisch für Namen und Satzanfänge.
- `text.umgekehrt(gruss)` dreht den Text **rückwärts**.

Das Wort in den runden Klammern — hier `gruss` — ist der Text, auf den das Werkzeug angewendet wird. Wichtig: Diese Werkzeuge *verändern* deine Variable nicht, sie **geben ein neues Ergebnis zurück**. `gruss` selbst bleibt `"hallo welt"`.

Zeichen zählen mit `länge`

`länge` sagt dir, aus wie vielen Zeichen ein Text besteht. Es ist so grundlegend, dass du dafür *keinen* Werkzeugkasten brauchst — `länge` ist immer da:

laenge.ki

```
nimm wort = "Größe"
nimm cafe = "café"
sag "'{wort}' hat {länge(wort)} Zeichen."
sag "'{cafe}' hat {länge(cafe)} Zeichen."
```

```
AUSGABE
'Größe' hat 5 Zeichen.
'café' hat 4 Zeichen.
```

Ein Zeichen ist ein Zeichen

`"Größe"` hat **5** Zeichen, `"café"` hat **4** — obwohl `ö`, `ß` und `é` „besondere" Buchstaben sind. Kiste zählt sie ganz selbstverständlich als je *ein* Zeichen, so wie ein Mensch es täte. Darum musst du dir keine Gedanken machen; es zählt einfach, was du siehst. (Dasselbe `länge` zählt später auch, wie viele Dinge in einer Liste stecken — dazu in Kapitel 8.)

Durchsuchen und ersetzen

Zwei weitere `text`-Werkzeuge brauchst du oft: nachsehen, *ob* ein Wort vorkommt, und ein Wort *austauschen*:

enthaelt.ki

```
nutze text
nimm satz = "ich male gern"
nimm gefunden = text.enthält(satz, "male")
nimm neuer_satz = text.ersetzt(satz, "male", "zeichne")
sag "Kommt 'male' vor? {gefunden}"
sag "Ausgetauscht: {neuer_satz}"
```

```
AUSGABE
Kommt 'male' vor? wahr
Ausgetauscht: ich zeichne gern
```

`text.enthält(satz, "male")` antwortet mit `wahr` oder `falsch` — einem **Wahrheitswert**. Solche Ja/Nein-Antworten sind der Stoff, aus dem Entscheidungen gemacht werden; genau darum geht es im nächsten Kapitel. `text.ersetzt(satz, "male", "zeichne")` gibt den Satz zurück, in dem jedes `male` durch `zeichne` ersetzt ist.

Warum erst in eine Variable?

Wir legen das Ergebnis oben zuerst in eine Variable (`nimm gefunden = …`) und geben dann `{gefunden}` aus. Nötig ist das nicht — du dürftest den Aufruf auch direkt in den Platzhalter schreiben. Aber mit einem sprechenden Zwischennamen liest sich die `sag`-Zeile ruhiger, und du siehst auf einen Blick, was passiert. Ein kleiner Stilgriff, den wir öfter nutzen.

Wozu das in KistePix gut ist

Text steckt auch in KistePix an vielen Stellen: der Titel im Fensterrahmen, die Namen deiner gespeicherten Bilder, kleine Meldungen an dich. Immer, wenn dort etwas aus festem Text und einem Wert zusammengesetzt wird — „Bild `{name}` gespeichert" —, ist es genau das, was du hier gelernt hast.

Teste dich

1. Was ergibt `"Pixel" + "kunst"`?
2. Was macht `text.kapitalisiert("mira")` — und was `text.großgeschrieben("mira")`?
3. Wie viele Zeichen hat `"Straße"` laut `länge`?

Aufgabe ★

Leg zwei Variablen an: einen kleingeschriebenen Vor- und Nachnamen (z. B. `"robin"` und `"fuchs"`). Baue daraus einen vollen Namen, bei dem **beide Teile mit großem Anfangsbuchstaben** beginnen, und gib ihn aus. Gib außerdem aus, wie viele Zeichen der volle Name hat. Tipp: `text.kapitalisiert` und ein Leerzeichen zum Verbinden.

Lösung

Teste dich: (1) `Pixelkunst` — direkt aneinandergehängt, ohne Leerzeichen. (2) `kapitalisiert` → `Mira` (nur erster Buchstabe groß); `großgeschrieben` → `MIRA` (alles groß). (3) `6` — auch das `ß` zählt als ein Zeichen.

Aufgabe:

`voller_name.ki`

```
nutze text
nimm vorname = "robin"
nimm nachname = "fuchs"
nimm voll = text.kapitalisiert(vorname) + " " + text.kapitalisiert(nachname)
sag "Voller Name: {voll}"
sag "In Großbuchstaben: {text.großgeschrieben(voll)}"
sag "Der Name hat {länge(voll)} Zeichen (mit Leerzeichen)."
```

```
AUSGABE
Voller Name: Robin Fuchs
In Großbuchstaben: ROBIN FUCHS
Der Name hat 11 Zeichen (mit Leerzeichen).
```

Kapitel 5 — Entscheidungen treffen

Was du hier lernst

- Was ein **Wahrheitswert** ist (`wahr` / `falsch`).
- Wie dein Programm mit `wenn` … `sonst` **entscheidet**.
- Die **Vergleiche** `==` `!=` `<` `>` `<=` `>=` — und die klassische Falle `=` gegen `==` .
- Mehrere Fälle mit `sonstwenn` und Bedingungen kombinieren mit `und` , `oder` , `nicht` .

Bisher liefen unsere Programme stur von oben nach unten. Jetzt bringen wir sie zum **Mitdenken**: Sie sollen je nach Lage *Verschiedenes* tun. Das ist der Moment, in dem aus einer Befehlsliste ein echtes Programm wird.

Wahr oder falsch

In Kapitel 4 hat `text.enthält(…)` mit `wahr` oder `falsch` geantwortet. Das sind die zwei **Wahrheitswerte** — die einzigen Antworten auf eine Ja/Nein-Frage. Ein Vergleich wie „ist 7 kleiner als 10?" liefert genau so einen Wert. Und mit diesen Werten trifft ein Programm seine Entscheidungen.

Deine erste Entscheidung: `wenn`

Mit `wenn` knüpfst du eine Anweisung an eine **Bedingung**:

`wenn.ki`

```
nimm pinselgröße = 3
wenn pinselgröße > 1 {
    sag "Großer Pinsel – male dicke Striche."
} sonst {
    sag "Feiner Pinsel – male einzelne Pixel."
}
```

```
AUSGABE
Großer Pinsel – male dicke Striche.
```

Wort für Wort

- `wenn` leitet die Entscheidung ein.
- `pinselgröße > 1` ist die **Bedingung** — eine Ja/Nein-Frage, die `wahr` oder `falsch` ergibt.
- Die **geschweiften Klammern** `{ }` umschließen einen **Block**: alle Zeilen darin gehören zusammen und laufen *nur*, wenn die Bedingung `wahr` ist.
- `sonst { … }` ist der Gegenfall: Er läuft, wenn die Bedingung `falsch` ist. Den `sonst`-Teil darfst du auch weglassen — dann passiert bei `falsch` eben nichts.

Alles im Block ist eingerückt (ein Stück nach rechts geschoben). Das muss nicht sein, aber es macht auf einen Blick sichtbar, was zusammengehört — halte dich daran, dein späteres Ich dankt es dir.

Vergleiche

Bedingungen entstehen fast immer aus einem **Vergleich**. Sechs gibt es:

vergleiche.ki

```
nimm a = 7
nimm b = 10
sag "7 == 10 ? {a == b}"
sag "7 != 10 ? {a != b}"
sag "7 <  10 ? {a < b}"
sag "7 >= 7  ? {a >= 7}"
```

```
AUSGABE
7 == 10 ? falsch
7 != 10 ? wahr
7 <  10 ? wahr
7 >= 7  ? wahr
```

Zeichen	Frage	Beispiel	Ergebnis
`==`	gleich?	`7 == 10`	falsch
`!=`	ungleich?	`7 != 10`	wahr
`<`	kleiner?	`7 < 10`	wahr
`>`	größer?	`7 > 10`	falsch
`<=`	kleiner oder gleich?	`7 <= 7`	wahr
`>=`	größer oder gleich?	`7 >= 7`	wahr

Die wichtigste Falle: `=` gegen `==`

Zwei Zeichen, die leicht zu verwechseln sind — aber ganz Verschiedenes bedeuten:

- Ein **einzelnes** `=` heißt *„leg hinein"* (aus Kapitel 2): `alter = 12` steckt 12 in die Variable.
- Ein **doppeltes** `==` heißt *„ist gleich?"* — es *fragt* und antwortet mit `wahr` oder `falsch`: `alter == 12`.

Merksatz: In einer Bedingung (nach `wenn`) willst du *fragen*, also nimm `==`. Ein einzelnes `=` gehört dorthin, wo du etwas *merkst*.

Mehrere Fälle: `sonstwenn`

Oft gibt es mehr als zwei Möglichkeiten. Dann reihst du mit `sonstwenn` weitere Fälle aneinander:

sonstwenn.ki

```
nimm punkte = 75
wenn punkte >= 90 {
    sag "Sehr gut!"
} sonstwenn punkte >= 60 {
    sag "Bestanden."
} sonst {
    sag "Leider nicht bestanden."
}
```

```
AUSGABE
Bestanden.
```

Der erste Treffer gewinnt

Kiste prüft die Fälle **von oben nach unten** und nimmt den **ersten**, dessen Bedingung `wahr` ist — die restlichen schaut es dann gar nicht mehr an. Bei `punkte = 75` ist `>= 90` falsch, `>= 60` wahr → „Bestanden.", und Schluss. Deshalb ordnet man die Fälle sinnvoll (hier vom höchsten zum niedrigsten). `sonst` am Ende fängt alles ab, was auf keinen Fall gepasst hat.

Bedingungen kombinieren: `und`, `oder`, `nicht`

Manchmal hängt eine Entscheidung an *mehreren* Dingen zugleich. Dafür verknüpfst du Bedingungen:

undoder.ki

```
nimm temperatur = 24
nimm sonne = wahr
wenn temperatur > 20 und sonne {
    sag "Warm und sonnig – ab nach draußen!"
}

nimm alter = 15
nimm in_begleitung = wahr
wenn alter >= 16 oder in_begleitung {
    sag "Du darfst den Film sehen."
}
```

```
AUSGABE
Warm und sonnig – ab nach draußen!
Du darfst den Film sehen.
```

- `und` ist nur `wahr`, wenn **beide** Seiten wahr sind (warm *und* sonnig).
- `oder` ist schon `wahr`, wenn **mindestens eine** Seite wahr ist (alt genug *oder* in Begleitung).

Und `nicht` **dreht** einen Wahrheitswert um — aus `falsch` wird `wahr`:

nicht.ki

```
nimm gespeichert = falsch
wenn nicht gespeichert {
    sag "Achtung: Dein Bild ist noch nicht gespeichert!"
}
```

```
AUSGABE
Achtung: Dein Bild ist noch nicht gespeichert!
```

Weil `gespeichert` gerade `falsch` ist, ist `nicht gespeichert` `wahr` — und die Warnung erscheint. So prüft man elegant auf „noch nicht getan".

Genau so denkt KistePix

Entscheidungen sind das Rückgrat von KistePix. Ein Beispiel, das dir in Teil II wieder begegnet: Bevor der Editor ein Pixel setzt, muss er prüfen, ob die angeklickte Stelle **überhaupt auf der Leinwand liegt** — sonst würde er ins Leere malen. Genau ein `wenn` mit vier `und`-verknüpften Vergleichen:

grenze.ki

```
fest breite = 16
fest höhe = 16
nimm x = 5
nimm y = 20
wenn x >= 0 und x < breite und y >= 0 und y < höhe {
    sag "Pixel ({x}, {y}) liegt auf der Leinwand."
} sonst {
    sag "Pixel ({x}, {y}) liegt außerhalb – wird ignoriert."
}
```

```
AUSGABE
Pixel (5, 20) liegt außerhalb – wird ignoriert.
```

Die Zeile `y = 20` liegt außerhalb einer 16 Zellen hohen Leinwand — also fällt die Bedingung auf `falsch`, und das Pixel wird ignoriert. Dieselbe Grenzprüfung nutzt später der Füll-Eimer, damit er nicht über den Rand hinausläuft. Wieder gilt: nichts auf Vorrat — du siehst schon jetzt, wozu es gut ist.

Teste dich

1. Welches Zeichen fragt „ist gleich?" — `=` oder `==`?
2. Bei `und`: Wie viele Seiten müssen wahr sein, damit das Ganze wahr ist? Und bei `oder`?
3. Was gibt `nicht (3 > 5)` — `wahr` oder `falsch`?

Aufgabe ★

Schreibe einen kleinen **Wetter-Berater**: Leg eine Variable `grad` für die Temperatur an. Gib mit `wenn` / `sonstwenn` / `sonst` einen Rat aus — etwa ab 25 Grad „T-Shirt-Wetter", ab 10 Grad „Nimm eine Jacke mit", sonst „Dick einpacken". Probier verschiedene Werte für `grad` aus.

Lösung

Teste dich: (1) `==` fragt; `=` legt hinein. (2) Bei `und` müssen **beide** wahr sein, bei `oder` reicht **eine**. (3) `wahr` — `3 > 5` ist `falsch`, und `nicht falsch` ist `wahr`.

Aufgabe:

wetter.ki

```
nimm grad = 8
wenn grad >= 25 {
    sag "T-Shirt-Wetter."
} sonstwenn grad >= 10 {
    sag "Nimm eine Jacke mit."
} sonst {
    sag "Dick einpacken – es ist kalt!"
}
```

```
AUSGABE
Dick einpacken – es ist kalt!
```

Kapitel 6 — Etwas wiederholen

Was du hier lernst

- Etwas eine feste Anzahl Mal tun mit der `für`-Schleife.
- In **Schritten** zählen — auch rückwärts — mit `schritt`.
- Wiederholen, *solange* eine Bedingung gilt, mit `solange`.
- **Verschachtelte Schleifen** — und wie man damit ein ganzes Raster durchläuft (das Herzstück von KistePix).

Computer sind unermüdlich: Dieselbe Sache tausendmal zu tun, macht ihnen nichts aus — dir beim Tippen schon. Statt hundert `sag`-Zeilen zu schreiben, sagst du Kiste **einmal**, was zu tun ist, und **wie oft**. Das nennt man eine **Schleife**.

Die `für`-Schleife: eine feste Anzahl Mal

Wenn du im Voraus weißt, wie oft etwas passieren soll, nimmst du `für`:

fuer.ki

```
für zahl = 1 bis 5 {
    sag "Zeile {zahl}"
}
sag "Fertig!"
```

```
AUSGABE
Zeile 1
Zeile 2
Zeile 3
Zeile 4
Zeile 5
Fertig!
```

Wort für Wort

- `für` startet die Schleife.
- `zahl` ist die **Zählvariable**. Kiste legt sie selbst an und füllt sie bei jedem Durchlauf mit dem nächsten Wert.
- `1 bis 5` ist der **Bereich**: `zahl` nimmt nacheinander die Werte 1, 2, 3, 4, 5 an — beide Enden zählen mit.
- Der **Block** `{ … }` läuft *einmal für jeden Wert*. Fünf Werte, fünf Durchläufe.

Beachte: `"Fertig!"` steht *nach* dem Block und erscheint darum nur ein einziges Mal — es gehört nicht zur Schleife. Genau dafür ist die Einrückung da: Sie zeigt, was drinnen und was draußen ist.

In Schritten zählen: `schritt`

Normalerweise zählt `für` in Einerschritten aufwärts. Mit `schritt` bestimmst du die Schrittweite selbst — auch rückwärts, mit einer negativen Zahl:

schritt.ki

```
für i = 10 bis 0 schritt -2 {
    sag "Countdown: {i}"
}
sag "Los geht's!"
```

```
AUSGABE
Countdown: 10
Countdown: 8
Countdown: 6
Countdown: 4
Countdown: 2
Countdown: 0
Los geht's!
```

`schritt -2` zählt von 10 in Zweierschritten *abwärts* bis 0. So baust du zum Beispiel einen Countdown. Mit `schritt 2` würdest du vorwärts in Zweierschritten zählen.

Die `solange`-Schleife: wiederholen, bis etwas eintritt

Manchmal weißt du *nicht* im Voraus, wie viele Durchläufe nötig sind — nur, *wann* Schluss sein soll. Dann nimmst du `solange`. Es wiederholt den Block, **solange** eine Bedingung `wahr` ist:

solange.ki

```
nimm guthaben = 1
nimm verdopplungen = 0
solange guthaben < 100 {
    guthaben = guthaben * 2
    verdopplungen = verdopplungen + 1
}
sag "Nach {verdopplungen} Verdopplungen: {guthaben}"
```

```
AUSGABE
Nach 7 Verdopplungen: 128
```

Hier verdoppeln wir das Guthaben, solange es unter 100 liegt, und zählen die Verdopplungen. Kiste prüft die Bedingung *vor* jedem Durchlauf: Sobald `guthaben` 100 erreicht oder überschreitet (bei 128), hört die Schleife auf.

Vorsicht: die Endlosschleife

Eine `solange`-Schleife braucht im Block **etwas, das die Bedingung irgendwann `falsch` macht** — hier das `guthaben = guthaben * 2`. Fehlt das, läuft die Schleife **ewig**, und das Programm bleibt hängen. Frag dich bei jeder `solange`-Schleife: „Was hier drin sorgt dafür, dass sie einmal endet?" (Passiert es doch: Das laufende Programm lässt sich in der IDE abbrechen.)

Verschachtelte Schleifen: ein ganzes Raster

Jetzt kommt der Moment, auf den alles zuläuft. Setzt du eine Schleife *in* eine andere, kannst du ein **Raster** durchlaufen — jede Zeile, und in jeder Zeile jede Spalte. Erinnerst du dich an `y * breite + x` aus Kapitel 3? Genau das erzeugen wir hier für jede Zelle:

raster.ki

```
fest breite = 3
fest höhe = 2
für y = 0 bis höhe - 1 {
    für x = 0 bis breite - 1 {
        sag "Spalte {x}, Zeile {y}  ->  Nummer {y * breite + x}"
    }
}
```

```
AUSGABE
Spalte 0, Zeile 0  ->  Nummer 0
Spalte 1, Zeile 0  ->  Nummer 1
Spalte 2, Zeile 0  ->  Nummer 2
Spalte 0, Zeile 1  ->  Nummer 3
Spalte 1, Zeile 1  ->  Nummer 4
Spalte 2, Zeile 1  ->  Nummer 5
```

Wie das läuft

Die **äußere** Schleife geht über die Zeilen `y`. Für *jede* Zeile läuft die **innere** Schleife komplett über alle Spalten `x` durch. So besucht Kiste die Zellen ordentlich der Reihe nach: erst die ganze obere Zeile (Nummern 0, 1, 2), dann die untere (3, 4, 5). `bis höhe - 1` bzw. `bis breite - 1`, weil wir bei 0 anfangen zu zählen — zwei Zeilen sind also `0` und `1`.

Das ist die halbe Miete für KistePix

Genau dieses Doppel — `für y … { für x … { … } }` — durchläuft in KistePix die ganze Leinwand: um sie zu leeren, um jedes Pixel zu zeichnen, um das fertige Bild zu speichern. Wenn du dieses Muster verstanden hast, hast du das wichtigste Werkzeug von Teil II schon in der Hand.

Teste dich

1. Wie oft läuft der Block bei `für i = 1 bis 4 { … }`?
2. Wann nimmst du `solange` statt `für`?
3. Was ist die Gefahr bei `solange` — und wie vermeidest du sie?

Aufgabe ★

Gib mit einer `für`-Schleife die **Siebenerreihe** aus: von `1 mal 7` bis `10 mal 7`, jeweils mit dem Ergebnis. Tipp: In der Schleife rechnest du `i * 7`.

Lösung

Teste dich: (1) Viermal — für 1, 2, 3, 4. (2) Wenn du *nicht* vorher weißt, wie viele Durchläufe nötig sind, sondern nur die Abbruch-Bedingung kennst. (3) Die Endlosschleife; vermeide sie, indem im Block etwas die Bedingung irgendwann `falsch` werden lässt.

Aufgabe:

siebenerreihe.ki

```
für i = 1 bis 10 {
    sag "{i} mal 7 = {i * 7}"
}
```

AUSGABE

```
1 mal 7 = 7
2 mal 7 = 14
3 mal 7 = 21
4 mal 7 = 28
5 mal 7 = 35
6 mal 7 = 42
7 mal 7 = 49
8 mal 7 = 56
9 mal 7 = 63
10 mal 7 = 70
```

Kapitel 7 — Eigene Funktionen schreiben

Was du hier lernst

- Wie du mit `funktion` einen Arbeitsschritt unter einem **Namen bündelst**.
- Wie du einer Funktion Werte übergibst — die **Parameter**.
- Wie eine Funktion mit `gib` ein **Ergebnis zurückgibt**.
- Warum Funktionen dir in KistePix Tipparbeit und Fehler ersparen.

Bis jetzt liefen unsere Programme als ein langer Strang von Anweisungen. Sobald ein Programm größer wird, willst du es aber in **überschaubare Häppchen** zerlegen — jedes mit einem Namen, der sagt, was es tut. So ein benanntes Häppchen ist eine **Funktion**. Du schreibst sie *einmal* und rufst sie *so oft du willst* auf.

Deine erste Funktion

funktion.ki

```
funktion begrüße(name) {
    sag "Hallo, {name}! Schön, dass du da bist."
}

begrüße("Sascha")
begrüße("Mara")
```

```
AUSGABE
Hallo, Sascha! Schön, dass du da bist.
Hallo, Mara! Schön, dass du da bist.
```

Wort für Wort

- `funktion` kündigt an: „Jetzt kommt ein benannter Arbeitsschritt."
- `begrüße` ist der **Name**, den du dir aussuchst — am besten ein Verb, denn eine Funktion *tut* etwas.
- `(name)` ist der **Parameter**: ein Platz für einen Wert, den die Funktion beim Aufruf bekommt. Innerhalb der Funktion ist `name` eine ganz normale Variable.
- Der **Block** `{ … }` enthält, was die Funktion tut.

Wichtig: Das bloße *Hinschreiben* der Funktion tut noch **nichts** — sie wartet nur. Erst der **Aufruf** `begrüße("Sascha")` lässt sie laufen. Der Wert in den Klammern — hier `"Sascha"` — landet im Parameter `name`. Zwei Aufrufe, zwei Begrüßungen: Du hast den Text nur *einmal* geschrieben und trotzdem zweimal verwendet.

Etwas zurückgeben: `gib`

Die Funktion oben *erledigt* etwas (sie gibt aus). Oft soll eine Funktion aber etwas **ausrechnen und das Ergebnis zurückliefern**, damit du weiterdamit arbeiten kannst. Dafür gibt es `gib`:

gib.ki

```
funktion fläche(breite, höhe) {
    gib breite * höhe
}

nimm a = fläche(4, 3)
sag "Ein 4-mal-3-Rechteck hat die Fläche {a}."
sag "Ein 5-mal-2-Rechteck: {fläche(5, 2)}"
```

```
AUSGABE
Ein 4-mal-3-Rechteck hat die Fläche 12.
Ein 5-mal-2-Rechteck: 10
```

Wort für Wort

- `fläche` hat **zwei** Parameter, durch Komma getrennt: `breite` und `höhe`.
- `gib breite * höhe` rechnet das Produkt aus und **reicht es zurück** an die Stelle, wo die Funktion aufgerufen wurde. Nach `gib` ist die Funktion sofort zu Ende.
- Der Aufruf `fläche(4, 3)` ist damit im Grunde *der Wert 12*. Du kannst ihn in eine Variable legen (`nimm a = fläche(4, 3)`) oder direkt in einen Platzhalter schreiben (`{fläche(5, 2)}`).

Zwei Sorten Funktionen

Manche Funktionen **tun** etwas (sie geben aus, malen, speichern) — wie `begrüße`. Andere **rechnen etwas aus und geben es zurück** — wie `fläche`. Faustregel: Willst du das Ergebnis *weiterverwenden*, nimm `gib`. Soll die Funktion nur etwas erledigen, brauchst du kein `gib`.

Funktionen und Schleifen — ein starkes Team

Eine Funktion in einer Schleife aufzurufen, ist enorm nützlich: Du beschreibst *einmal*, wie eine Sache geht, und wendest sie auf viele Werte an:

schleife.ki

```
funktion quadrat(n) {
    gib n * n
}

für i = 1 bis 5 {
    sag "{i} im Quadrat ist {quadrat(i)}"
}
```

```
AUSGABE
1 im Quadrat ist 1
2 im Quadrat ist 4
3 im Quadrat ist 9
4 im Quadrat ist 16
5 im Quadrat ist 25
```

Standardwerte für Parameter

Ein Parameter darf einen **Standardwert** bekommen — er gilt, wenn beim Aufruf kein Wert angegeben wird:

standard.ki

```
funktion begrüße(name = "Welt") {
    sag "Hallo, {name}!"
}

begrüße()
begrüße("Tim")
```

```
AUSGABE
Hallo, Welt!
Hallo, Tim!
```

`begrüße()` ohne Wert nimmt den Standard `"Welt"`; `begrüße("Tim")` überschreibt ihn. Praktisch für Werte, die *meistens* gleich sind.

Genau so baut KistePix seine Werkzeuge

Erinnerst du dich an `y * breite + x` aus Kapitel 3 und 6 — die Rechnung, die aus Spalte und Zeile die Nummer einer Zelle macht? Statt sie im ganzen Programm immer wieder hinzuschreiben (und sich

dabei zu vertippen), gießt man sie **ein einziges Mal** in eine Funktion:

nummer.ki

```
funktion nummer(x, y, breite) {
    gib y * breite + x
}

fest breite = 16
sag "Pixel (5, 3) hat die Nummer {nummer(5, 3, breite)}."
sag "Pixel (0, 0) hat die Nummer {nummer(0, 0, breite)}."
sag "Pixel (0, 1) hat die Nummer {nummer(0, 1, breite)}."
```

AUSGABE

```
Pixel (5, 3) hat die Nummer 53.
Pixel (0, 0) hat die Nummer 0.
Pixel (0, 1) hat die Nummer 16.
```

Warum das so wertvoll ist

Jetzt steht die Raster-Rechnung an *einer* Stelle. Überall sonst schreibst du nur noch das sprechende `nummer(x, y, breite)` — das liest sich klarer *und* ist sicherer: Müsste die Formel je geändert werden, änderst du sie an genau einer Stelle statt an zwanzig. Genau nach diesem Muster baut KistePix seine kleinen Helfer, etwa „setze ein Pixel" oder „hol die Farbe einer Zelle". Du hast das Werkzeug jetzt, um sie zu verstehen.

Teste dich

1. Was ist der Unterschied zwischen dem *Hinschreiben* und dem *Aufrufen* einer Funktion?
2. Wozu dient `gib`?
3. Was ist ein **Parameter**?

Aufgabe ★

Schreibe eine Funktion `würfel(n)`, die die dritte Potenz zurückgibt (`n * n * n`). Gib damit in einer Schleife die Würfelzahlen von `1` bis `5` aus.

Lösung

Teste dich: (1) Das Hinschreiben legt die Funktion nur an — sie wartet; erst der Aufruf lässt sie laufen. (2) `gib` liefert ein Ergebnis aus der Funktion zurück, mit dem man weiterarbeiten kann. (3) Ein Platz für einen Wert, den die Funktion beim Aufruf bekommt — innen eine normale Variable.

Aufgabe:

wuerfel.ki

```
funktion würfel(n) {
    gib n * n * n
}

für i = 1 bis 5 {
    sag "{i} hoch 3 = {würfel(i)}"
}
```

AUSGABE

```
1 hoch 3 = 1
2 hoch 3 = 8
3 hoch 3 = 27
4 hoch 3 = 64
5 hoch 3 = 125
```

Kapitel 8 — Listen: viele Werte auf einmal

Was du hier lernst

- Wie du viele Werte in **einer** Variablen sammelst — einer **Liste**.
- Wie du einzelne Elemente über ihren **Index** holst (der bei `0` beginnt).
- Wie du eine Liste **durchläufst** und **änderst**.
- Wie eine **vorbereitete Liste** zur **Leinwand** von KistePix wird — der Moment, in dem alle Grundlagen zusammenkommen.

Bisher hielt eine Variable genau *einen* Wert. Aber oft gehören viele Werte zusammen: die Farben einer Palette, die Namen einer Rangliste, die Pixel eines Bildes. Sie alle in einzelne Variablen zu stecken, wäre mühsam. Dafür gibt es die **Liste**: *ein* Behälter für beliebig viele Werte, ordentlich der Reihe nach.

Eine Liste anlegen und darauf zugreifen

liste.ki

```
nimm palette = ["Rot", "Grün", "Blau"]
sag "Die Palette: {palette}"
sag "Erste Farbe:  {palette[0]}"
sag "Dritte Farbe: {palette[2]}"
sag "Anzahl Farben: {länge(palette)}"
```

```
AUSGABE
Die Palette: ["Rot", "Grün", "Blau"]
Erste Farbe:  Rot
Dritte Farbe: Blau
Anzahl Farben: 3
```

Wort für Wort

- Die **eckigen Klammern** `[ … ]` machen eine Liste; die Werte darin trennt ein Komma.
- `palette[0]` holt ein einzelnes Element über seinen **Index** — die Position in der Liste.
- `länge(palette)` sagt, wie viele Elemente drin sind (dasselbe `länge` wie bei Text in Kapitel 4).

Das Zählen beginnt bei 0

Der Index des **ersten** Elements ist `0`, nicht 1. Also: `palette[0]` ist „Rot", `palette[1]` ist „Grün", `palette[2]` ist „Blau". Das ist beim Programmieren fast überall so — und anfangs die häufigste Verwechslung. Merke: Bei einer Liste mit `länge` Elementen läuft der Index von `0` bis `länge - 1`. Das letzte Element ist also `palette[länge(palette) - 1]`.

Eine Liste durchlaufen

Willst du mit *jedem* Element etwas tun, läufst du die Liste mit `wiederhole … in …` durch — Element für Element:

durchlaufen.ki

```
nimm palette = ["Rot", "Grün", "Blau"]
wiederhole farbe in palette {
    sag "Farbe: {farbe}"
}
```

```
AUSGABE
Farbe: Rot
Farbe: Grün
Farbe: Blau
```

Das ist die bequemste Art, wenn du nur die *Werte* brauchst. Manchmal brauchst du aber auch die **Nummer** jedes Elements — dann läufst du mit einer `für`-Schleife über die Indizes:

nummeriert.ki

```
nimm palette = ["Rot", "Grün", "Blau"]
für i = 0 bis länge(palette) - 1 {
    sag "{i}: {palette[i]}"
}
```

```
AUSGABE
0: Rot
1: Grün
2: Blau
```

Hier siehst du, warum es `bis länge(palette) - 1` heißt: Die Liste hat 3 Elemente, die Indizes gehen von `0` bis `2`. Genau dieselbe „minus eins"-Grenze wie beim Raster in Kapitel 6 — kein Zufall.

Eine Liste ändern

Ein einzelnes Element überschreibst du direkt über seinen Index — mit `=`, wie bei einer Variablen. Und mit dem `liste`-Werkzeugkasten kannst du eine Liste auch **wachsen** und **schrumpfen** lassen:

aendern.ki

```
nutze liste

nimm palette = ["Rot", "Grün"]
sag "Vorher: {palette}"

liste.hänge_an(palette, "Blau")
sag "Nach Anhängen: {palette}"

palette[0] = "Dunkelrot"
sag "Nach Ändern von Nr. 0: {palette}"

liste.entferne(palette, 1)
sag "Nach Entfernen von Nr. 1: {palette}"
```

AUSGABE

```
Vorher: ["Rot", "Grün"]
Nach Anhängen: ["Rot", "Grün", "Blau"]
Nach Ändern von Nr. 0: ["Dunkelrot", "Grün", "Blau"]
Nach Entfernen von Nr. 1: ["Dunkelrot", "Blau"]
```

Wort für Wort

- `palette[0] = "Dunkelrot"` überschreibt das Element an Position 0.
- `liste.hänge_an(palette, "Blau")` hängt hinten ein neues Element an (dafür einmal oben `nutze liste`).
- `liste.entferne(palette, 1)` entfernt das Element an Position 1.

Die vorbereitete Liste — und damit die Leinwand

Jetzt kommt alles zusammen. Statt eine Liste Stück für Stück wachsen zu lassen, kannst du sie in **voller Größe vorbereiten** — mit `liste.gefüllt(anzahl, startwert)`. Das ergibt eine Liste aus `anzahl` gleichen Werten, die du danach über den Index gezielt änderst.

Und genau das ist die **Leinwand von KistePix**: Alle Pixel liegen in *einer* langen Liste, Zeile für Zeile. Welche Stelle ein Pixel hat, rechnest du — du ahnst es — mit `y * breite + x`:

leinwand.ki

```
nutze liste

fest breite = 4
fest höhe = 3
nimm leinwand = liste.gefüllt(breite * höhe, 0)
```

```
// Zwei Zellen einfärben (Farbnummer statt 0), über y * breite + x adressiert:
leinwand[0 * breite + 0] = 1
leinwand[1 * breite + 2] = 2

sag "Die Leinwand als eine Reihe: {leinwand}"
sag "Sie hat {länge(leinwand)} Zellen."
```

```
AUSGABE
Die Leinwand als eine Reihe: [1, 0, 0, 0, 0, 0, 2, 0, 0, 0, 0, 0]
Sie hat 12 Zellen.
```

Der rote Faden von Teil I

Schau dir die Ausgabe an: eine Reihe aus 12 Zahlen für ein 4×3-Raster. Die `1` steht ganz vorne (Zelle 0,0), die `2` an Position 6 — das ist Zeile 1, Spalte 2, denn `1 * 4 + 2 = 6`. Alles, was du in Teil I gelernt hast, steckt in diesen zwei Zeilen: eine **Liste** (Kap 8) hält die Pixel, ein **Index** aus `y * breite + x` (Kap 3) findet jedes, eine **Konstante** (Kap 2) merkt sich die Breite, eine **Schleife** (Kap 6) läuft darüber, eine **Funktion** (Kap 7) bündelt die Rechnung. In Teil II bauen wir aus genau diesem Raster einen echten Pixel-Editor. Du hast das Fundament jetzt komplett.

Teste dich

1. Welchen Index hat das **erste** Element einer Liste?
2. Wie kommst du an das **letzte** Element einer Liste namens `xs`?
3. Was macht `liste.gefüllt(5, 0)`?

Aufgabe ★

Leg eine Liste mit drei Lieblingsgerichten an. Gib sie als **nummerierte Liste** aus (1., 2., 3. — nicht 0., 1., 2.!). Häng dann ein viertes Gericht an und gib aus, wie viele es jetzt sind. Tipp: Index `i` läuft ab 0, aber du darfst `{i + 1}` ausgeben.

Lösung

Teste dich: (1) `0`. (2) `xs[länge(xs) - 1]`. (3) Es erzeugt eine Liste aus 5 Nullen (fünf gleiche Startwerte), die du danach über den Index füllst.

Aufgabe:

gerichte.ki

```
nutze liste

nimm essen = ["Pizza", "Pasta", "Sushi"]
für i = 0 bis länge(essen) - 1 {
    sag "{i + 1}. {essen[i]}"
}

liste.hänge_an(essen, "Glace")
sag "Jetzt sind es {länge(essen)} Gerichte."
```

```
AUSGABE
1. Pizza
2. Pasta
3. Sushi
Jetzt sind es 4 Gerichte.
```

Kapitel 9 — Fenster, Leinwand und Maus

Was du hier lernst

- Wie du ein echtes **Fenster** öffnest — raus aus der Konsole.
- Wie du auf einer **Leinwand** malst: Rechtecke, Farben, Koordinaten.
- Wie dein Programm auf **Mausklicks** reagiert.
- Wie daraus in wenigen Zeilen ein winziger Pixel-Editor wird — die Brücke zu Teil II.

Bisher haben unsere Programme in die **Konsole** geschrieben — den Ausgabe-Bereich der IDE. Jetzt öffnen wir ein richtiges **Fenster** mit Grafik und Maus. Dafür hat Kiste den Werkzeugkasten `gui` (von englisch *graphical user interface*, „grafische Bedienoberfläche").

GUI-Programme startest du mit F5 — und sie zeigen keine Konsolenausgabe

Zwei Dinge sind bei `gui`-Programmen anders. Erstens: Sie öffnen ein **eigenes Fenster** statt in die Konsole zu schreiben — in diesem Kapitel siehst du darum echte *Bildschirmfotos* statt Ausgabe-Kästen. Zweitens: `gui` läuft nur über den **F5**-Weg (Kiste baut dein Programm dazu blitzschnell in ein echtes Windows-Programm). Das merkst du im Alltag nicht — du drückst wie immer F5. Zum **Schließen** klickst du das Fenster einfach zu.

Das kleinste Fenster

fenster.ki

```
// Das kleinste GUI-Programm: ein leeres Fenster.
nutze gui

nimm fenster = gui.fenster_öffne("Mein erstes Fenster", 320, 220)
gui.starte()
```

Ein leeres Fenster, 320×220 Punkte groß, mit deinem Titel in der Leiste.

Wort für Wort

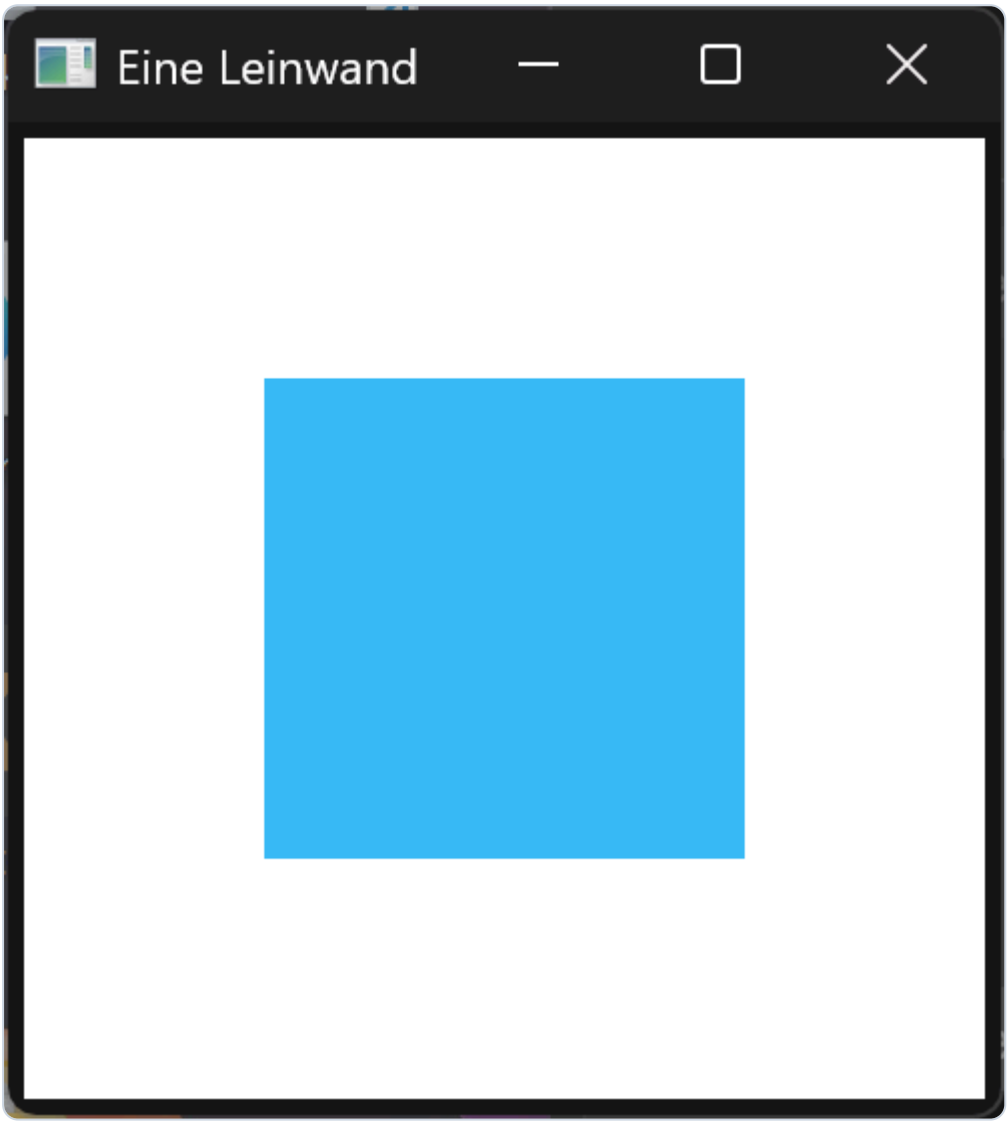
- `nutze gui` holt den GUI-Werkzeugkasten (wie `nutze text` in Kapitel 4).
- `gui.fenster_öffne("Mein erstes Fenster", 320, 220)` erstellt das Fenster: Titel, Breite, Höhe (in Bildpunkten). Zurück kommt ein **Handle** — ein Griff, über den du das Fenster später ansprichst.
- `gui.starte()` startet die **Ereignisschleife**: Ab hier bleibt das Fenster offen und wartet auf dich (Klicks, Tasten). Ohne diese Zeile wäre das Fenster sofort wieder weg. Sie steht darum immer als *letzte* Zeile.

Eine Leinwand zum Malen

Ein leeres Fenster ist langweilig. Legen wir eine **Leinwand** hinein — eine Fläche, auf die wir zeichnen — und malen ein Rechteck darauf:

leinwand.ki

```
// Eine Leinwand mit weißem Grund und einem blauen Quadrat.
nutze gui

nimm bild = gui.leinwand(240, 240)
gui.leeren(bild)
gui.rechteck(bild, 0, 0, 240, 240, "#ffffff")   // weißer Hintergrund
gui.rechteck(bild, 60, 60, 120, 120, "#38bdf8")  // blaues Quadrat in der Mitte

nimm fenster = gui.fenster_öffne("Eine Leinwand", 240, 240)
gui.zeige(fenster, bild)
gui.starte()
```



Weißer Hintergrund, ein blaues Quadrat in der Mitte — beides mit `gui.rechteck` gemalt.

Wort für Wort

- `gui.leinwand(240, 240)` erstellt eine Zeichenfläche und gibt ihr Handle zurück (hier `bild`).
- `gui.rechteck(bild, x, y, breite, höhe, farbe)` malt ein gefülltes Rechteck. Das erste volle Rechteck deckt alles weiß ein (der Hintergrund), das zweite setzt das blaue Quadrat darüber.
- `gui.zeige(fenster, bild)` hängt die Leinwand ins Fenster.

Das Koordinaten-System und die Farben

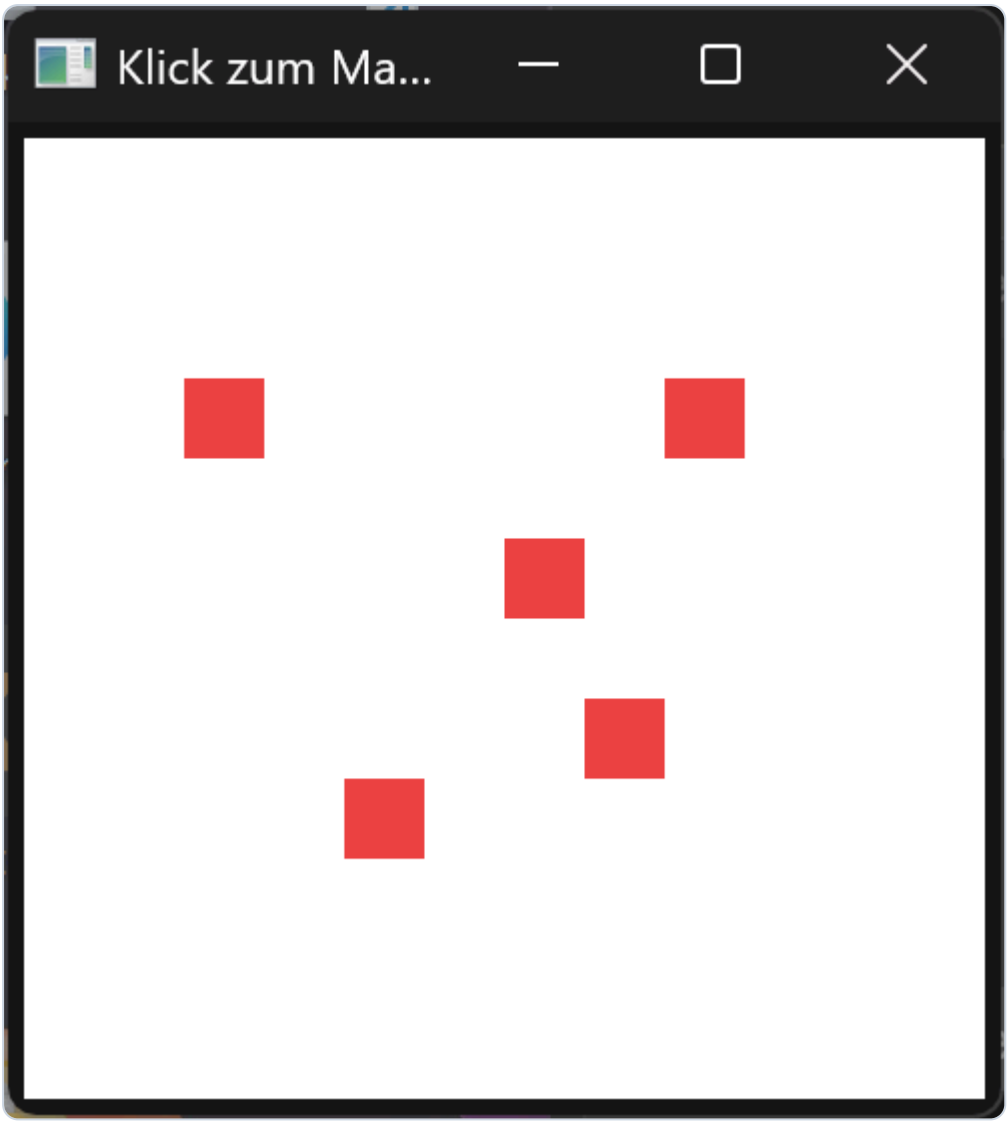
Der Punkt **(0, 0) ist oben links**. `x` wächst nach **rechts**, `y` nach **unten** (nicht nach oben wie im Matheheft!). Ein Rechteck gibst du mit seiner *linken oberen Ecke* `(x, y)` und seiner `breite` / `höhe` an. Farben schreibst du als **Hex-Code** in Anführungszeichen: `"#ffffff"` ist Weiß, `"#000000"` Schwarz, `"#ef4444"` ein kräftiges Rot. Die sechs Zeichen stehen für die Rot-, Grün- und Blau-Anteile — die genaue Bedeutung ist für jetzt nebensächlich; du darfst einfach fertige Codes verwenden.

Auf die Maus reagieren

Jetzt wird es lebendig. Wir wollen, dass ein Klick auf die Leinwand dort ein Kästchen malt. Dafür sagen wir Kiste: „*Wenn* jemand klickt, *dann* tu Folgendes." So ein Stück Code, das auf ein Ereignis wartet, nennt man einen **Ereignis-Behandler**:

malen.ki

```
// Klick zum Malen: Jeder Mausklick setzt ein rotes Kästchen ins Raster.
nutze gui

nimm ZELLE = 20   // Kantenlänge einer Rasterzelle in Bildpunkten

nimm bild = gui.leinwand(240, 240)
gui.leeren(bild)
gui.rechteck(bild, 0, 0, 240, 240, "#ffffff")

// Beim Klick: Mausposition holen, auf das Raster runden, Kästchen malen.
gui.bei_maus_klick(bild, funktion(b) {
    nimm x = gui.maus_x(b)
    nimm y = gui.maus_y(b)
    nimm sx = (x div ZELLE) * ZELLE
    nimm sy = (y div ZELLE) * ZELLE
    gui.rechteck(bild, sx, sy, ZELLE, ZELLE, "#ef4444")
})

nimm fenster = gui.fenster_öffne("Klick zum Malen", 240, 240)
gui.zeige(fenster, bild)
gui.starte()
```



Nach fünf Klicks: Jeder Klick hat ein rotes Kästchen ins Raster gesetzt — dein erster, winziger Pixel-Editor.

Wort für Wort

- `gui.bei_maus_klick(bild, funktion(b) { … })` verdrahtet die Leinwand mit einem Behandler: Der Block in `funktion(b) { … }` läuft **jedes Mal, wenn** auf die Leinwand geklickt wird. So eine namenlose Funktion, die man direkt übergibt, ist ganz normal — sie tut, was du in Kapitel 7 gelernt hast, nur ohne eigenen Namen.
- `gui.maus_x(b)` und `gui.maus_y(b)` liefern die **Klickposition** auf der Leinwand.
- `(x div ZELLE) * ZELLE` rundet die Position auf die nächste **Rasterzelle** ab — erinnerst du dich an `div` aus Kapitel 3? Genau dafür ist es da. So landet das Kästchen sauber im Gitter, egal wohin genau du klickst.
- Dann malt `gui.rechteck(…)` das rote Kästchen an die gerundete Stelle.

Merk dir dieses Programm — es ist KistePix im Kleinen

Lies die `malen.ki` noch einmal von oben: Fenster, Leinwand, und ein Klick, der eine Rasterposition ausrechnet und dort malt. **Genau das** ist der Kern von KistePix — nur mit mehr Farben, echten Pixeln in einer Liste (Kapitel 8) und weiteren Werkzeugen. Alles, was in Teil II folgt, wächst aus diesen paar Zeilen. Die kleine Vorschau-Funktion für Tastenkürzel gibt es übrigens auch: `gui.taste(fenster, "Strg+S", funktion(f) { … })` lässt dich später etwa mit *Strg* + *S* speichern — aber das bauen wir erst, wenn es so weit ist.

Teil I geschafft!

Das war der letzte Grundlagen-Baustein. Du kennst jetzt Ausgabe, Variablen, Zahlen, Text, Entscheidungen, Schleifen, Funktionen, Listen — und du kannst ein Fenster öffnen, darauf malen und auf die Maus reagieren. Mehr braucht es nicht, um KistePix zu bauen. **In Teil II geht es los.**

Teste dich

1. Welche Zeile hält das Fenster offen und wartet auf Klicks?
2. Wo liegt der Punkt `(0, 0)` auf der Leinwand, und in welche Richtungen wachsen `x` und `y`?
3. Womit fragst du im Klick-Behandler die Position des Klicks ab?

Aufgabe ★

Nimm `malen.ki` und verändere es: Male die Kästchen in einer anderen Farbe (z. B. `"#38bdf8"` für Blau), und mach die Zellen größer, indem du `ZELLE` auf `40` setzt. Starte, klick ein paar Mal und sieh, wie sich Größe und Farbe des Rasters ändern.

Lösung

Teste dich: (1) `gui.starte()` — die Ereignisschleife. (2) `(0, 0)` ist oben links; `x` wächst nach rechts, `y` nach unten. (3) Mit `gui.maus_x(b)` und `gui.maus_y(b)` im Behandler.

Aufgabe: Du änderst nur zwei Stellen — die Zeile `nimm ZELLE = 20` zu `nimm ZELLE = 40` und im `gui.rechteck(…)` die Farbe `"#ef4444"` zu `"#38bdf8"`. Der Rest bleibt gleich. Genau so, Stück für Stück, werden wir in Teil II aus dieser Vorlage einen vollen Editor machen.

II KistePix bauen

Jetzt kommt der spannende Teil: Wir bauen aus allem, was du in Teil I gelernt hast, einen echten kleinen Pixel-Editor — Stufe für Stufe, von der leeren Leinwand bis zum fertigen Bild, das du als Datei speicherst.

Jede Stufe fügt eine Fähigkeit hinzu und baut auf der vorigen auf. Am Ende jeder Stufe hast du ein **lauffähiges Programm**, das du starten und ausprobieren kannst — kein „erst-am-Schluss-läuft-es".

So gehen wir vor

- Zu jeder Stufe zeigen wir das **vollständige Programm** und gehen die neuen Teile Schritt für Schritt durch — anlegen, einhängen, verdrahten.
- Jedes Listing wurde vor dem Druck wirklich **gebaut und ausgeführt**; die Bildschirmfotos sind echt.
- Weil ein Editor ein **Fenster** ist, startest du die Programme mit **F5** (wie in Kapitel 9) — sie öffnen ein Fenster, statt in die Konsole zu schreiben.

Der Bauplan

- **S1 — Die Leinwand:** das leere Pixel-Raster als Gitter.
- **S2 — Der Stift:** mit der Maus einzelne Pixel malen.
- **S3 — Farben:** eine Palette und die aktive Farbe.
- **S4 — Ziehen & Radierer:** mit gedrückter Maus malen und wieder löschen.
- **S5 — Das Rechteck:** gefüllte Rechtecke aufziehen.
- **S6 — Füllen:** eine ganze Fläche auf einen Klick — richtig und schnell.
- **S7 — Exportieren:** dein Bild als echte PNG-Datei speichern.
- **S8 — Fertig:** und wohin die Reise von hier aus weitergeht.

Los geht's mit der Grundlage von allem: der Leinwand.

Stufe 1 — Die Leinwand: ein flaches Pixel-Raster

Was du hier baust

Das Fundament von KistePix: ein **leeres Pixel-Raster**, gezeichnet als sauberes Gitter aus Kästchen. Am Ende dieser Stufe öffnet sich ein Fenster mit deiner Leinwand — noch leer, aber bereit.

Eine Pixel-Grafik ist ein Raster aus lauter kleinen Quadraten. Genau das bauen wir zuerst: die Fläche, auf der später gemalt wird. Aus Teil I hast du dafür schon alles — wir setzen es jetzt zusammen.

Die Idee: alle Pixel in einer Liste

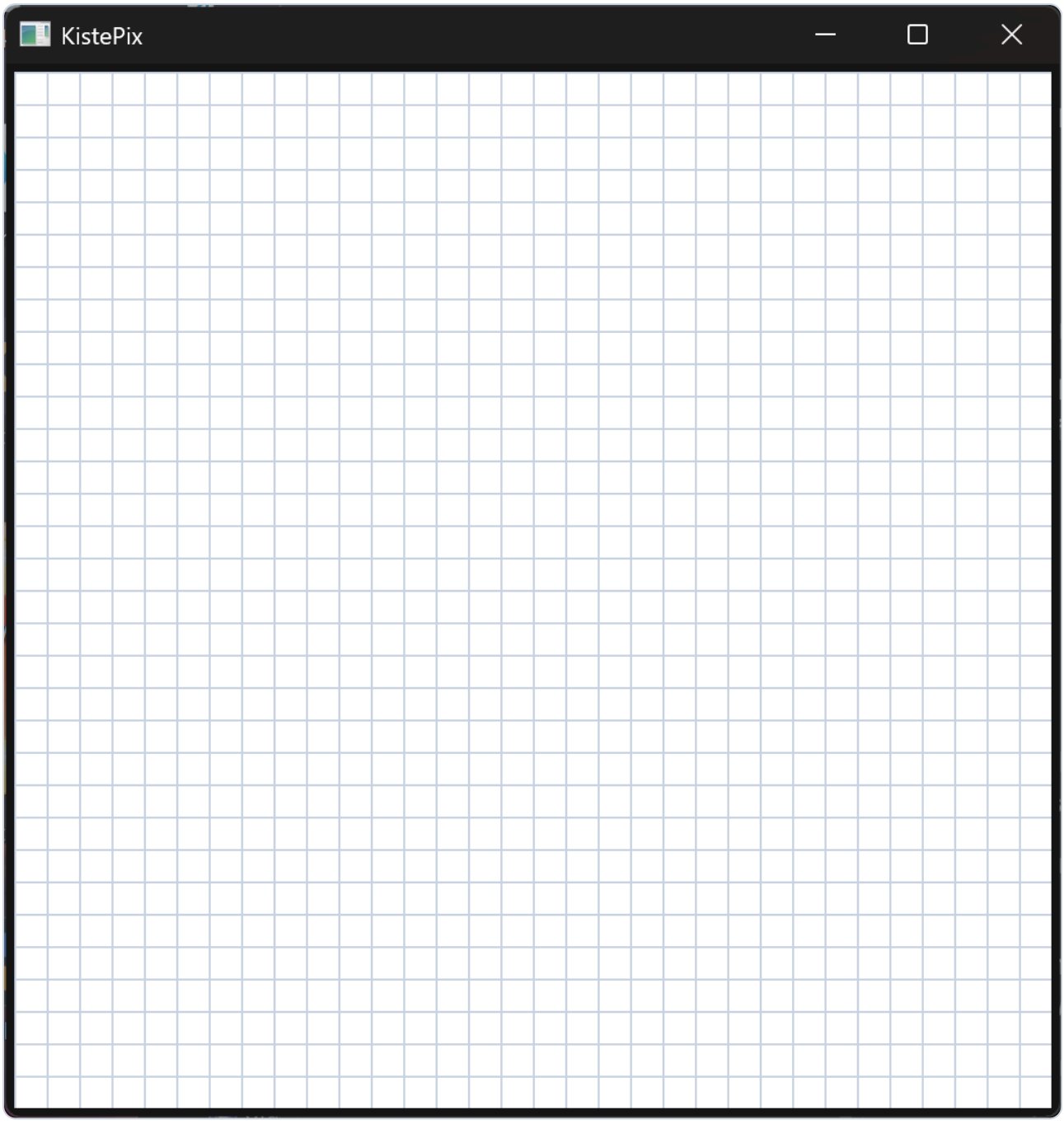
Erinnerst du dich an den „roten Faden" aus Kapitel 8? Die Leinwand ist **eine flache Liste**: alle Zellen hintereinander, Zeile für Zeile. Eine Zelle bei Spalte `x`, Zeile `y` liegt an der Stelle `y * LEIN_B + x`. Der Wert `0` heißt „leer". So einfach ist das Datenmodell — und es bleibt bis zum Schluss so.

Das ganze Programm

Hier ist die komplette Stufe 1. Sieht nach viel aus, ist aber lauter Bekanntes — wir gehen es gleich Stück für Stück durch. Tipp es ab oder lies mit:

kistepix.ki (Stufe 1)

```
// KistePix – Stufe 1: Die Leinwand.
// Ein flaches Pixel-Raster (eine Liste) wird als Gitter aus Kästchen gezeichnet.
nutze gui
nutze liste

fest LEIN_B = 32          // Breite der Leinwand in Zellen
fest LEIN_H = 32          // Höhe der Leinwand in Zellen
fest ZELLE = 16           // Kantenlänge einer Zelle in Bildpunkten
fest RAND = "#cbd5e1"     // Farbe der Gitterlinien
fest LEER = "#ffffff"     // Farbe einer leeren Zelle

// Das Herz von KistePix: alle Pixel in EINER langen Liste, Zeile für Zeile.
// 0 bedeutet „leer". Die Stelle einer Zelle ist y * LEIN_B + x (aus Teil I).
nimm pixel = liste.gefüllt(LEIN_B * LEIN_H, 0)

// Die Zeichenfläche, so groß wie das ganze Raster in Bildpunkten.
nimm bild_gui = gui.leinwand(LEIN_B * ZELLE, LEIN_H * ZELLE)

// Welche Farbe hat eine Zelle? (In S3 kommen echte Farben dazu; jetzt nur leer.)
funktion zellfarbe(wert) {
    gib LEER
}

// Eine einzelne Zelle zeichnen – später beim Malen genau eine (schnell).
funktion zeichne_zelle(x, y) {
    nimm f = zellfarbe(pixel[y * LEIN_B + x])
    // 1 Punkt Abstand lässt die graue Gitterlinie durchscheinen.
    gui.rechteck(bild_gui, x * ZELLE + 1, y * ZELLE + 1, ZELLE - 1, ZELLE - 1, f)
}

// Die ganze Leinwand zeichnen – beim Start und nach großen Änderungen.
funktion zeichne() {
    gui.rechteck(bild_gui, 0, 0, LEIN_B * ZELLE, LEIN_H * ZELLE, RAND)  // Gitter-Grund
    für y = 0 bis LEIN_H - 1 {
        für x = 0 bis LEIN_B - 1 {
            zeichne_zelle(x, y)
        }
    }
}

zeichne()

nimm fenster = gui.fenster_öffne("KistePix", LEIN_B * ZELLE, LEIN_H * ZELLE)
gui.zeige(fenster, bild_gui)
gui.starte()
```



Stufe 1: die leere Leinwand — ein 32×32-Raster mit feinen Gitterlinien.

Schritt für Schritt

Die Festlegungen ganz oben

- `LEIN_B` und `LEIN_H` sind Breite und Höhe der Leinwand *in Zellen* — hier 32×32. Als `fest`, denn diese Größe ändert sich im Programm nicht.
- `ZELLE` ist die Größe einer Zelle *in Bildpunkten* (16). Eine kleine Rasterzelle wird also als 16×16-Quadrat am Bildschirm gezeigt — sonst wären einzelne Pixel kaum zu treffen.
- `RAND` und `LEER` sind zwei Farben: das Grau der Gitterlinien und das Weiß einer leeren Zelle.

Das Raster und die Zeichenfläche

- `nimm pixel = liste.gefüllt(LEIN_B * LEIN_H, 0)` — die Leinwand als flache Liste, alle Zellen anfangs `0` (leer). Das ist die „vorbereitete Liste" aus Kapitel 8, jetzt für 1024 Zellen.
- `nimm bild_gui = gui.leinwand(LEIN_B * ZELLE, LEIN_H * ZELLE)` — die Zeichenfläche in Bildpunkten, also 512×512 groß (32×16).

Zeichnen — in zwei Funktionen

- `zellfarbe(wert)` sagt, welche Farbe ein Zell-Wert bekommt. Noch gibt es nur „leer" → Weiß; ab Stufe 3 kommen hier die echten Farben dazu. Wir gießen die Regel gleich in eine Funktion (Kapitel 7), damit später nur *eine* Stelle wächst.
- `zeichne_zelle(x, y)` malt **eine einzelne Zelle**. Der Versatz um 1 Punkt (`+ 1` und `ZELLE - 1`) lässt ringsum das graue Gitter durchscheinen.
- `zeichne()` malt zuerst den grauen Grund und dann mit einer **verschachtelten Schleife** (Kapitel 6!) jede Zelle darüber — genau das `für y { für x { … } }`, das du schon kennst.

Fenster auf

Zum Schluss dasselbe Muster wie in Kapitel 9: einmal `zeichne()` aufrufen, dann das Fenster öffnen, die Leinwand hineinhängen (`gui.zeige`) und mit `gui.starte()` die Ereignisschleife starten.

Warum genau so gezeichnet wird — und wie es groß wird

Wir zeichnen jede Zelle als eigenes Rechteck. Das ist einfach und — wichtig — **schnell genug**, weil wir gleich in Stufe 2 nur noch die *eine* Zelle neu zeichnen, auf die du klickst (nicht jedes Mal alle 1024). Die volle `zeichne()`-Runde läuft nur beim Start und nach großen Änderungen. Für eine handliche Leinwand wie unsere ist das flott. **Ehrlich gesagt:** Bei einer *riesigen* Leinwand (viele hundert Zellen je Kante) würden die vielen Einzel-Rechtecke träge — dann malt man die Leinwand als *ein* Bild in den Speicher und zeigt es skaliert an. Das ist die Technik der großen Vorlage „KistePix Deluxe" und ein prima nächstes Projekt (siehe Stufe 8). Für unser Buch bleiben wir bei der klaren, schnellen Raster-Variante.

Teste dich

1. Wo in der Liste `pixel` liegt die Zelle bei Spalte 3, Zeile 2 (bei `LEIN_B = 32`)?
2. Warum ist `LEIN_B` als `fest` angelegt und nicht als `nimm`?
3. Welche Funktion läuft beim Start und zeichnet die ganze Leinwand?

Aufgabe ★

Ändere die Leinwand: Mach sie mit `LEIN_B` und `LEIN_H` zu einem 16×16-Raster und setze `ZELLE` auf `28`. Starte und sieh, wie die Zellen größer und gröber werden. Probier auch eine andere Gitterfarbe für `RAND` (z. B. `"#94a3b8"`).

Lösung

Teste dich: (1) An Stelle `2 * 32 + 3 = 67`. (2) Weil sich die Leinwandgröße während des Programms nicht ändert — `fest` schützt sie vor versehentlichem Überschreiben (Kapitel 2). (3) `zeichne()`.

Aufgabe: Du änderst nur die drei `fest`-Zeilen oben — `LEIN_B = 16`, `LEIN_H = 16`, `ZELLE = 28` — und nach Wunsch die `RAND`-Farbe. Der restliche Code bleibt unverändert; er rechnet ja alles aus diesen Werten aus. Genau dafür haben wir sie oben als benannte Festlegungen angelegt.

Stufe 2 — Ein Pixel malen: der Stift

Was du hier baust

Den **Stift**: Ein Klick auf die Leinwand färbt die getroffene Zelle. Damit malst du deine ersten Pixel — und legst den Grundstein für alle Werkzeuge, die noch kommen.

Die Leinwand steht. Jetzt soll sie auf die Maus *hören*. Das Prinzip kennst du schon aus Kapitel 9: ein Klick-Behandler auf der Leinwand. Neu ist nur, dass wir die Klickstelle in eine **Rasterzelle** umrechnen und den Wert im `pixel`-Raster merken.

Von der Klickstelle zur Zelle

Die Maus liefert Bild*punkte* (etwa „204 Punkte von links"). Wir brauchen aber die *Zelle*. Weil jede Zelle `ZELLE` Punkte breit ist, ist die Umrechnung genau das ganzzahlige Teilen aus Kapitel 3:

```
nimm x = gui.maus_x(b) div ZELLE
nimm y = gui.maus_y(b) div ZELLE
```

Ein Klick bei 204 Punkten und `ZELLE = 16` ergibt `204 div 16 = 12` — also Spalte 12. Egal, wohin genau innerhalb der Zelle du triffst: `div` rundet auf die Zellennummer ab.

Die `male`-Funktion

Das eigentliche Malen bündeln wir in einer eigenen Funktion (Kapitel 7). Sie merkt sich den Wert im Raster und zeichnet **nur diese eine Zelle** neu:

kistepix.ki (Ausschnitt)

```
funktion male(x, y) {
    wenn x >= 0 und x < LEIN_B und y >= 0 und y < LEIN_H {
        pixel[y * LEIN_B + x] = 1
        zeichne_zelle(x, y)
    }
}
```

Wort für Wort

- Die `wenn`-Zeile ist die **Grenzprüfung** aus Kapitel 5 — Wort für Wort dieselbe: nur malen, wenn `(x, y)` wirklich auf der Leinwand liegt. So kann ein Klick auf den Rand nichts kaputt machen.
- `pixel[y * LEIN_B + x] = 1` merkt sich „diese Zelle ist gemalt" — die Stelle im Raster über `y * LEIN_B + x` (Kapitel 3), der Wert `1` statt `0`.
- `zeichne_zelle(x, y)` malt **nur diese eine Zelle** neu. Nicht die ganze Leinwand — das ist der Trick, der es schnell hält.

Den Behandler verdrahten

Jetzt hängen wir das Ganze an die Maus. Der Behandler rechnet die Zelle aus und ruft `male`:

kistepix.ki (Ausschnitt)

```
gui.bei_maus_klick(bild_gui, funktion(b) {
    nimm x = gui.maus_x(b) div ZELLE
    nimm y = gui.maus_y(b) div ZELLE
    male(x, y)
})
```

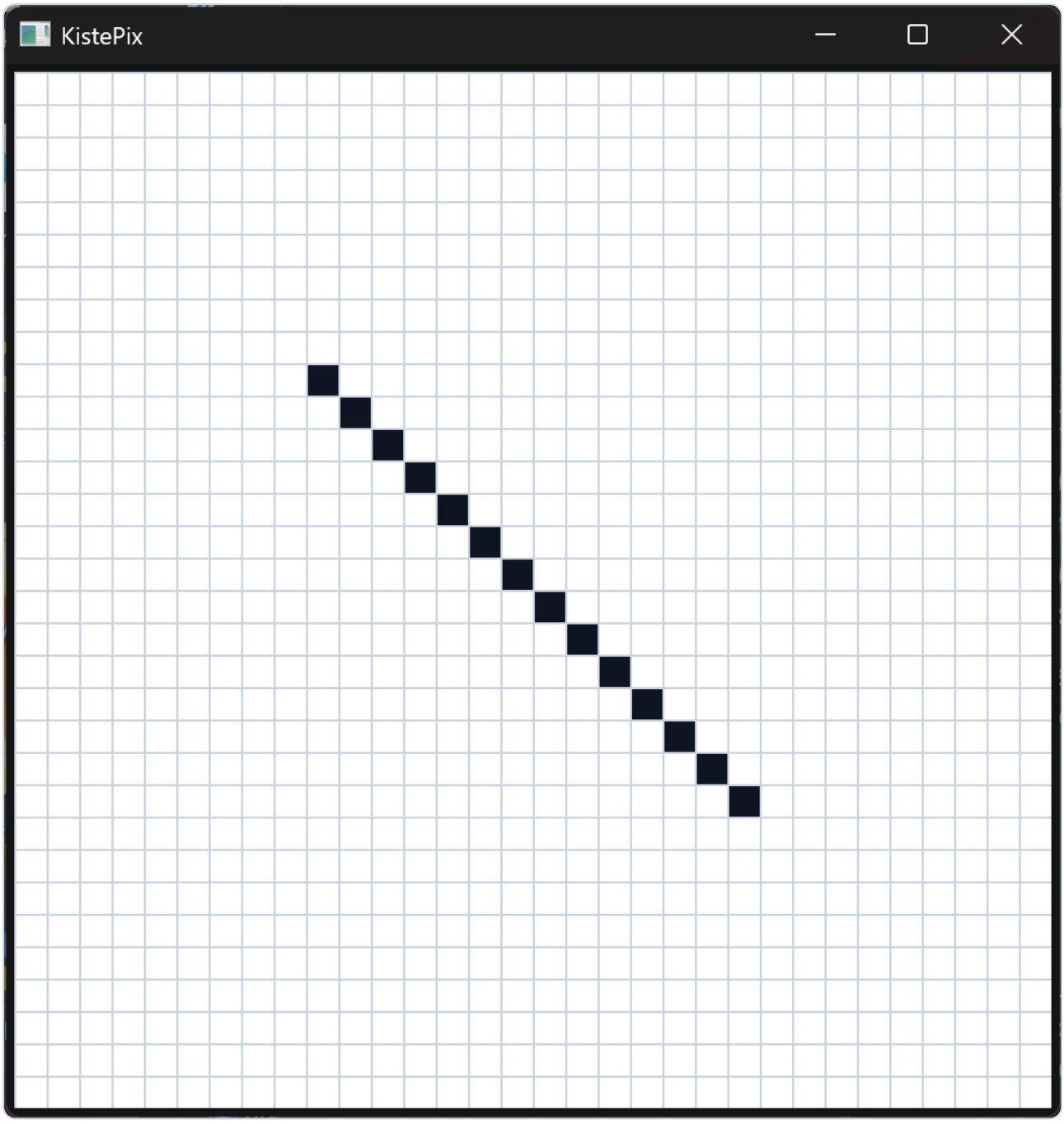
Das ganze Programm

Hier ist Stufe 2 komplett. Gegenüber Stufe 1 sind nur drei Dinge dazugekommen: die Farbe `STIFT`, die `zellfarbe`-Regel „nicht 0 → Stiftfarbe", die `male`-Funktion und der Klick-Behandler.

kistepix.ki (Stufe 2)

```
// KistePix – Stufe 2: Der Stift.
// Ein Klick auf die Leinwand färbt die getroffene Zelle. Neu gezeichnet wird
// nur DIESE eine Zelle – darum bleibt es auch bei vielen Klicks flüssig.
nutze gui
nutze liste

fest LEIN_B = 32
fest LEIN_H = 32
fest ZELLE = 16
fest RAND = "#cbd5e1"
fest LEER = "#ffffff"
fest STIFT = "#111827"    // die Farbe des Stifts (echte Farben kommen in Stufe 3)

nimm pixel = liste.gefüllt(LEIN_B * LEIN_H, 0)
nimm bild_gui = gui.leinwand(LEIN_B * ZELLE, LEIN_H * ZELLE)

// 0 = leer (weiß), alles andere = gemalt (Stiftfarbe).
funktion zellfarbe(wert) {
    wenn wert == 0 {
        gib LEER
    }
    gib STIFT
}

funktion zeichne_zelle(x, y) {
    nimm f = zellfarbe(pixel[y * LEIN_B + x])
    gui.rechteck(bild_gui, x * ZELLE + 1, y * ZELLE + 1, ZELLE - 1, ZELLE - 1, f)
}
```

```
funktion zeichne() {
    gui.rechteck(bild_gui, 0, 0, LEIN_B * ZELLE, LEIN_H * ZELLE, RAND)
    für y = 0 bis LEIN_H - 1 {
        für x = 0 bis LEIN_B - 1 {
            zeichne_zelle(x, y)
        }
    }
}

// Eine Zelle setzen (malen): Wert merken und nur diese eine Zelle neu zeichnen.
funktion male(x, y) {
    wenn x >= 0 und x < LEIN_B und y >= 0 und y < LEIN_H {
        pixel[y * LEIN_B + x] = 1
        zeichne_zelle(x, y)
    }
}

zeichne()

// Beim Klick: Bildpunkte in eine Rasterzelle umrechnen und dort malen.
gui.bei_maus_klick(bild_gui, funktion(b) {
    nimm x = gui.maus_x(b) div ZELLE
    nimm y = gui.maus_y(b) div ZELLE
    male(x, y)
})

nimm fenster = gui.fenster_öffne("KistePix", LEIN_B * ZELLE, LEIN_H * ZELLE)
gui.zeige(fenster, bild_gui)
gui.starte()
```



Ein paar Klicks später: jede angeklickte Zelle ist gefärbt. Deine ersten selbst gemalten Pixel.

Der Clou: nur eine Zelle neu zeichnen

Beim Klick rufen wir **nicht** `zeichne()` (die ganze Leinwand), sondern nur `zeichne_zelle(x, y)` — genau ein Rechteck. Egal, ob die Leinwand 32×32 oder größer ist: Ein Klick kostet immer *gleich wenig* Arbeit. Genau so bleibt ein Malprogramm flüssig, auch wenn du schnell viele Pixel setzt. Die volle `zeichne()`-Runde heben wir uns für später auf — für den Moment, in dem sich wirklich alles ändert (Füllen in Stufe 6).

Hier trifft sich ganz Teil I

Sieh dir den Klick-Behandler noch einmal an — in vier Zeilen steckt fast der ganze Grundlagenteil: **div** aus Kapitel 3 rechnet Punkte in Zellen um, die **Grenzprüfung** aus Kapitel 5 schützt den Rand, **y * LEIN_B + x** aus Kapitel 3 findet die Stelle im **Raster** (Kapitel 8), eine **Funktion** (Kapitel 7) bündelt das Malen, und ein **Ereignis-Behandler** (Kapitel 9) verbindet alles mit der Maus. Nichts davon war umsonst.

Teste dich

1. Ein Klick landet bei 150 Punkten von links, `ZELLE = 16`. In welcher Spalte?
2. Warum ruft der Klick-Behandler `zeichne_zelle` statt `zeichne`?
3. Was tut die `wenn`-Zeile in `male`, und warum ist sie wichtig?

Aufgabe ★

Wähle eine andere Stiftfarbe: Ändere `STIFT` auf ein kräftiges Rot (`"#ef4444"`), starte und male ein kleines Bild. Überlege dann: Was müsstest du ändern, damit der Stift *zwei* Farben zur Wahl hätte? (Genau das lösen wir in der nächsten Stufe.)

Lösung

Teste dich: (1) `150 div 16 = 9` — Spalte 9. (2) Weil sich nur *eine* Zelle geändert hat; alles neu zu zeichnen wäre unnötige Arbeit und würde bei vielen Klicks bremsen. (3) Sie prüft, ob `(x, y)` auf der Leinwand liegt — ohne sie könnte ein Klick knapp außerhalb eine Stelle im Raster ansprechen, die es gar nicht gibt.

Aufgabe: Du änderst nur die Zeile `fest STIFT = "#111827"` zu `fest STIFT = "#ef4444"`. Für zwei Farben bräuchtest du eine Möglichkeit, die aktive Farbe umzuschalten und im Raster nicht nur „gemalt/leer" (1/0), sondern *welche* Farbe zu merken — genau das baut Stufe 3.

Stufe 3 — Farben: Palette und aktive Farbe

Was du hier baust

Eine **Farbpalette** unter der Leinwand und eine **aktive Farbe**: Du klickst ein Farbfeld an, und ab da malt der Stift in dieser Farbe. Damit wird aus dem Schwarz-Weiß-Stift ein richtiges Buntmal-Werkzeug.

Bisher kannte eine Zelle nur zwei Zustände: leer (`0`) oder gemalt (`1`). Für Farben brauchen wir mehr — und der Trick ist elegant einfach: Wir legen eine **Liste von Farben** an, und der Zell-Wert ist einfach die **Nummer** der Farbe in dieser Liste.

Der Zell-Wert ist ein Palettenindex

`kistepix.ki (Ausschnitt)`

```
fest PALETTE = ["#ffffff", "#111827", "#ef4444", "#3b82f6", "#22c55e", "#f59e0b", "#a855f7", "#ec4899"]
nimm aktiv = 1
```

Die Idee

- `PALETTE` ist eine Liste von Farben (Kapitel 8). **Index 0 ist Weiß** — das bedeutet weiterhin „leer". Ab Index 1 kommen echte Farben.
- Der Wert einer Zelle im `pixel`-Raster ist jetzt genau dieser **Index**. Eine Zelle mit Wert `2` ist also `PALETTE[2]` = Rot.
- `aktiv` merkt sich, welche Farbe gerade gewählt ist. Das ändert sich beim Anklicken — darum `nimm` (veränderlich), nicht `fest`.

Dadurch wird `zellfarbe` sogar *kürzer* als vorher — sie schlägt die Farbe einfach in der Palette nach:

kistepix.ki (Ausschnitt)

```
funktion zellfarbe(wert) {
    gib PALETTE[wert]
}
```

Und beim Malen merkt sich die Zelle nun die **aktive** Farbe statt einer festen `1`:

kistepix.ki (Ausschnitt)

```
funktion male(x, y) {
    wenn x >= 0 und x < LEIN_B und y >= 0 und y < LEIN_H {
        pixel[y * LEIN_B + x] = aktiv
        zeichne_zelle(x, y)
    }
}
```

Die Palettenleiste

Die Palette ist eine zweite, schmale Leinwand unter der großen. `zeichne_palette` malt für jede Farbe ein Feld; das **aktive** bekommt einen dunklen Rahmen, damit du auf einen Blick siehst, womit du malst:

kistepix.ki (Ausschnitt)

```
funktion zeichne_palette() {
    gui.rechteck(palette_gui, 0, 0, LEIN_B * ZELLE, FELD, "#e2e8f0")
    für p = 0 bis länge(PALETTE) - 2 {
        nimm idx = p + 1
        nimm fx = p * FELD
        wenn idx == aktiv {
            gui.rechteck(palette_gui, fx, 0, FELD, FELD, "#0c4a6e")
            gui.rechteck(palette_gui, fx + 3, 3, FELD - 6, FELD - 6, PALETTE[idx])
        } sonst {
            gui.rechteck(palette_gui, fx + 1, 1, FELD - 2, FELD - 2, PALETTE[idx])
        }
    }
}
```

Wort für Wort

- Die Schleife läuft über die Positionen `p = 0` bis `länge(PALETTE) - 2` — also über alle Farben *ab* Index 1 (die leere Farbe 0 zeigen wir nicht als Feld).
- `idx = p + 1` ist der zugehörige Palettenindex, `fx = p * FELD` die Position des Feldes.
- Ist dieses Feld die `aktiv`e Farbe, malen wir zuerst einen dunklen Rahmen und die Farbe etwas kleiner hinein — das ergibt den Auswahl-Rand.

Zwei Behandler — Leinwand und Palette

Jetzt hat KistePix **zwei** anklickbare Flächen. Jede bekommt ihren eigenen Behandler:

kistepix.ki (Ausschnitt)

```
// Klick auf die Palette: das getroffene Feld wird die aktive Farbe.
gui.bei_maus_klick(palette_gui, funktion(b) {
    nimm p = gui.maus_x(b) div FELD
    wenn p >= 0 und p < länge(PALETTE) - 1 {
        aktiv = p + 1
        zeichne_palette()
    }
})
```

Derselbe `div`-Trick wie beim Malen: die Klickstelle geteilt durch die Feldbreite ergibt, welches Feld getroffen wurde. Wir setzen `aktiv` neu und zeichnen die Palette einmal neu, damit der Rahmen umspringt.

Beides ins Fenster: `gui.vertikal`

Damit Leinwand *und* Palette im selben Fenster erscheinen, stapeln wir sie mit `gui.vertikal` übereinander:

kistepix.ki (Ausschnitt)

```
nimm inhalt = gui.vertikal([bild_gui, palette_gui])
gui.zeige(fenster, inhalt)
```

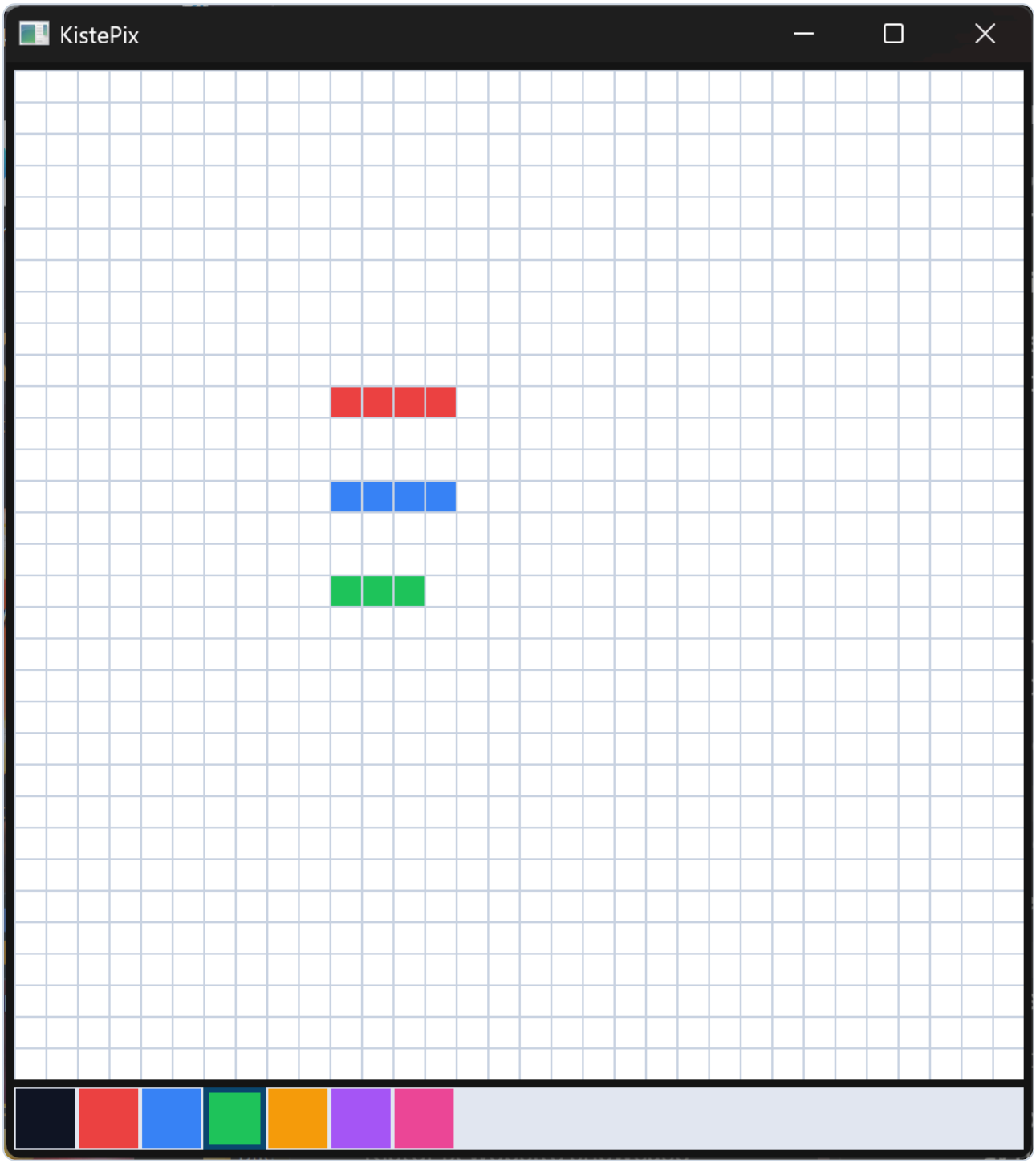
Das ganze Programm

kistepix.ki (Stufe 3)

```
// KistePix – Stufe 3: Farben.
// Das Raster merkt sich jetzt EINE FARBE je Zelle (als Palettenindex), und eine
// Palettenleiste unter der Leinwand lässt dich die aktive Farbe anklicken.
nutze gui
nutze liste

fest LEIN_B = 32
fest LEIN_H = 32
fest ZELLE = 16
fest FELD = 32            // Kantenlänge eines Palettenfeldes
fest RAND = "#cbd5e1"

// Die Palette: Index 0 = leer (weiß), ab 1 echte Farben. Ein Zell-Wert IST der
```

```
// Palettenindex — zellfarbe schlägt die Farbe einfach nach.
fest PALETTE = ["#ffffff", "#111827", "#ef4444", "#3b82f6", "#22c55e", "#f59e0b", "#a855f7", "#ec4899"]

// Zustand: welche Palettenfarbe ist aktiv? Ändert sich beim Klick → veränderlich.
nimm aktiv = 1

nimm pixel = liste.gefüllt(LEIN_B * LEIN_H, 0)
nimm bild_gui = gui.leinwand(LEIN_B * ZELLE, LEIN_H * ZELLE)
nimm palette_gui = gui.leinwand(LEIN_B * ZELLE, FELD)

funktion zellfarbe(wert) {
    gib PALETTE[wert]
}

funktion zeichne_zelle(x, y) {
    nimm f = zellfarbe(pixel[y * LEIN_B + x])
    gui.rechteck(bild_gui, x * ZELLE + 1, y * ZELLE + 1, ZELLE - 1, ZELLE - 1, f)
}

funktion zeichne() {
    gui.rechteck(bild_gui, 0, 0, LEIN_B * ZELLE, LEIN_H * ZELLE, RAND)
    für y = 0 bis LEIN_H - 1 {
        für x = 0 bis LEIN_B - 1 {
            zeichne_zelle(x, y)
        }
    }
}

// Die Palettenleiste: ein Feld je Farbe (ab Index 1); das aktive bekommt einen Rahmen.
funktion zeichne_palette() {
    gui.rechteck(palette_gui, 0, 0, LEIN_B * ZELLE, FELD, "#e2e8f0")
    für p = 0 bis länge(PALETTE) - 2 {
        nimm idx = p + 1
        nimm fx = p * FELD
        wenn idx == aktiv {
            gui.rechteck(palette_gui, fx, 0, FELD, FELD, "#0c4a6e")
            gui.rechteck(palette_gui, fx + 3, 3, FELD - 6, FELD - 6, PALETTE[idx])
        } sonst {
            gui.rechteck(palette_gui, fx + 1, 1, FELD - 2, FELD - 2, PALETTE[idx])
        }
    }
}

funktion male(x, y) {
    wenn x >= 0 und x < LEIN_B und y >= 0 und y < LEIN_H {
        pixel[y * LEIN_B + x] = aktiv
        zeichne_zelle(x, y)
    }
}

zeichne()
zeichne_palette()

// Klick auf die Leinwand: malen mit der aktiven Farbe.
gui.bei_maus_klick(bild_gui, funktion(b) {
    nimm x = gui.maus_x(b) div ZELLE
    nimm y = gui.maus_y(b) div ZELLE
    male(x, y)
})

// Klick auf die Palette: das getroffene Feld wird die aktive Farbe.
gui.bei_maus_klick(palette_gui, funktion(b) {
    nimm p = gui.maus_x(b) div FELD
```

```
    wenn p >= 0 und p < länge(PALETTE) - 1 {
        aktiv = p + 1
        zeichne_palette()
    }
})

nimm inhalt = gui.vertikal([bild_gui, palette_gui])
nimm fenster = gui.fenster_öffne("KistePix", LEIN_B * ZELLE, LEIN_H * ZELLE + FELD)
gui.zeige(fenster, inhalt)
gui.starte()
```



Drei Farben, eine Palette: Unten wählst du die Farbe (hier ist Grün aktiv — mit Rahmen), oben malst du damit.

Ein Werkzeug-Zustand, der bleibt

`aktiv` ist das erste Stück **Werkzeug-Gedächtnis** von KistePix: Einmal gewählt, bleibt die Farbe gewählt, bis du eine andere anklickst. Genau so verhalten sich echte Malprogramme. Und weil zwei gleich große Farbfelder sonst ununterscheidbar wären, zeigt der **Rahmen** sichtbar, welches gerade aktiv ist — eine kleine, aber wichtige Freundlichkeit gegenüber dem, der malt.

Teste dich

1. Welche Farbe hat eine Zelle mit dem Wert `3` (mit der Palette oben)?
2. Warum ist `PALETTE` als `fest`, `aktiv` aber als `nimm` angelegt?
3. Wie erkennt der Palette-Behandler, welches Farbfeld angeklickt wurde?

Aufgabe ★

Erweitere die Palette: Häng hinten zwei weitere Farben deiner Wahl an (z. B. `"#000000"` und `"#14b8a6"`) — einfach zwei Einträge mehr in der `PALETTE`-Liste. Starte und prüfe: Erscheinen zwei neue Felder, und kannst du damit malen? (Du musst sonst *nichts* ändern — überlege dir, warum.)

Lösung

Teste dich: (1) `PALETTE[3]` = `"#3b82f6"`, also Blau. (2) `PALETTE` ändert sich nie zur Laufzeit — die Farben stehen fest; `aktiv` wechselt bei jedem Palettenklick. (3) Über `gui.maus_x(b) div FELD` — die Klickposition geteilt durch die Feldbreite ergibt die Feldnummer.

Aufgabe: Du fügst der `PALETTE`-Liste einfach zwei Farben hinzu. Es funktioniert sofort, ohne weitere Änderung — weil `zeichne_palette` und der Behandler beide mit `länge(PALETTE)` rechnen, nicht mit einer festen Zahl. Genau dafür haben wir es so geschrieben: Die Palette darf beliebig lang sein.

Stufe 4 — Ziehen zum Malen und der Radierer

Was du hier baust

Zwei Bequemlichkeiten, die jedes Malprogramm braucht: **Ziehen** (mit gedrückter Maus einen durchgehenden Strich malen, statt jede Zelle einzeln zu klicken) und einen **Radierer**, der Zellen wieder auf „leer" setzt. Dazu kommt der erste **Werkzeug-Umschalter**.

Bisher musstest du jedes Pixel einzeln anklicken — für einen Strich mühsam. Jetzt soll die Maus, wenn du sie *gedrückt hältst und ziehst*, laufend malen. Und ein Radierer soll Fehler wieder wegnehmen können.

Ziehen: dieselbe Aktion, ein neues Ereignis

Das Schöne ist: Malen beim Ziehen ist genau dasselbe wie Malen beim Klicken — nur wird es *fortlaufend* ausgelöst, während du die Maus bewegst. Kiste hat dafür ein eigenes Ereignis: `gui.bei_maus_ziehen`.

Wir hängen einfach dieselbe Aktion auch daran:

kistepix.ki (Ausschnitt)

```
gui.bei_maus_klick(bild_gui, funktion(b) {
    benutze(gui.maus_x(b) div ZELLE, gui.maus_y(b) div ZELLE)
})
gui.bei_maus_ziehen(bild_gui, funktion(b) {
    benutze(gui.maus_x(b) div ZELLE, gui.maus_y(b) div ZELLE)
})
```

Weil während des Ziehens viele Ereignisse dicht hintereinander kommen — für jede Position, die die Maus überstreicht —, entsteht eine **durchgehende Spur**. Dass wir pro Ereignis nur *eine* Zelle neu zeichnen (aus Stufe 2), zahlt sich hier richtig aus: Auch ein schneller Strich bleibt flüssig.

Der Werkzeug-Zustand

Stift oder Radierer? Das merkt sich eine neue Zustands-Variable — genau wie `aktiv` für die Farbe:

kistepix.ki (Ausschnitt)

```
nimm werkzeug = "stift"   // "stift" oder "radierer"
```

Und die `male`-Funktion aus Stufe 2 wird zur allgemeineren `benutze`: Sie schaut nach, welches Werkzeug aktiv ist, und setzt die Zelle entsprechend — auf die aktive Farbe (Stift) oder auf `0`/leer (Radierer):

kistepix.ki (Ausschnitt)

```
funktion benutze(x, y) {
    wenn x >= 0 und x < LEIN_B und y >= 0 und y < LEIN_H {
        wenn werkzeug == "radierer" {
            pixel[y * LEIN_B + x] = 0
        } sonst {
            pixel[y * LEIN_B + x] = aktiv
        }
        zeichne_zelle(x, y)
    }
}
```

Wort für Wort

Die äußere `wenn`-Zeile ist wieder die **Grenzprüfung** (Kapitel 5). Die innere Entscheidung ist neu: Ist das Werkzeug der `"radierer"`, kommt `0` (leer) in die Zelle — sonst die `aktiv`e Farbe. Ob geklickt oder gezogen, ob Stift oder Radierer: alle Wege führen durch diese *eine* Funktion. Das hält den Code klein und ehrlich.

Die Werkzeugleiste — mit echten Knöpfen

Zum Umschalten bauen wir eine kleine Leiste aus zwei **Knöpfen**. Anders als die selbstgemalten Palettenfelder sind das *echte* Bedienknöpfe von Kiste — `gui.knopf_neu` erstellt sie, `gui.bei_klick` verdrahtet den Klick:

kistepix.ki (Ausschnitt)

```
nimm knopf_stift = gui.knopf_neu("Stift")
nimm knopf_radierer = gui.knopf_neu("Radierer")
gui.bei_klick(knopf_stift, funktion(k) { werkzeug = "stift" })
gui.bei_klick(knopf_radierer, funktion(k) { werkzeug = "radierer" })
nimm leiste = gui.horizontal([knopf_stift, knopf_radierer])
```

`gui.horizontal` ordnet die Knöpfe *nebeneinander* an (das Gegenstück zu `gui.vertikal`). Und ganz unten stapeln wir alles zusammen — Leiste, Leinwand, Palette:

kistepix.ki (Ausschnitt)

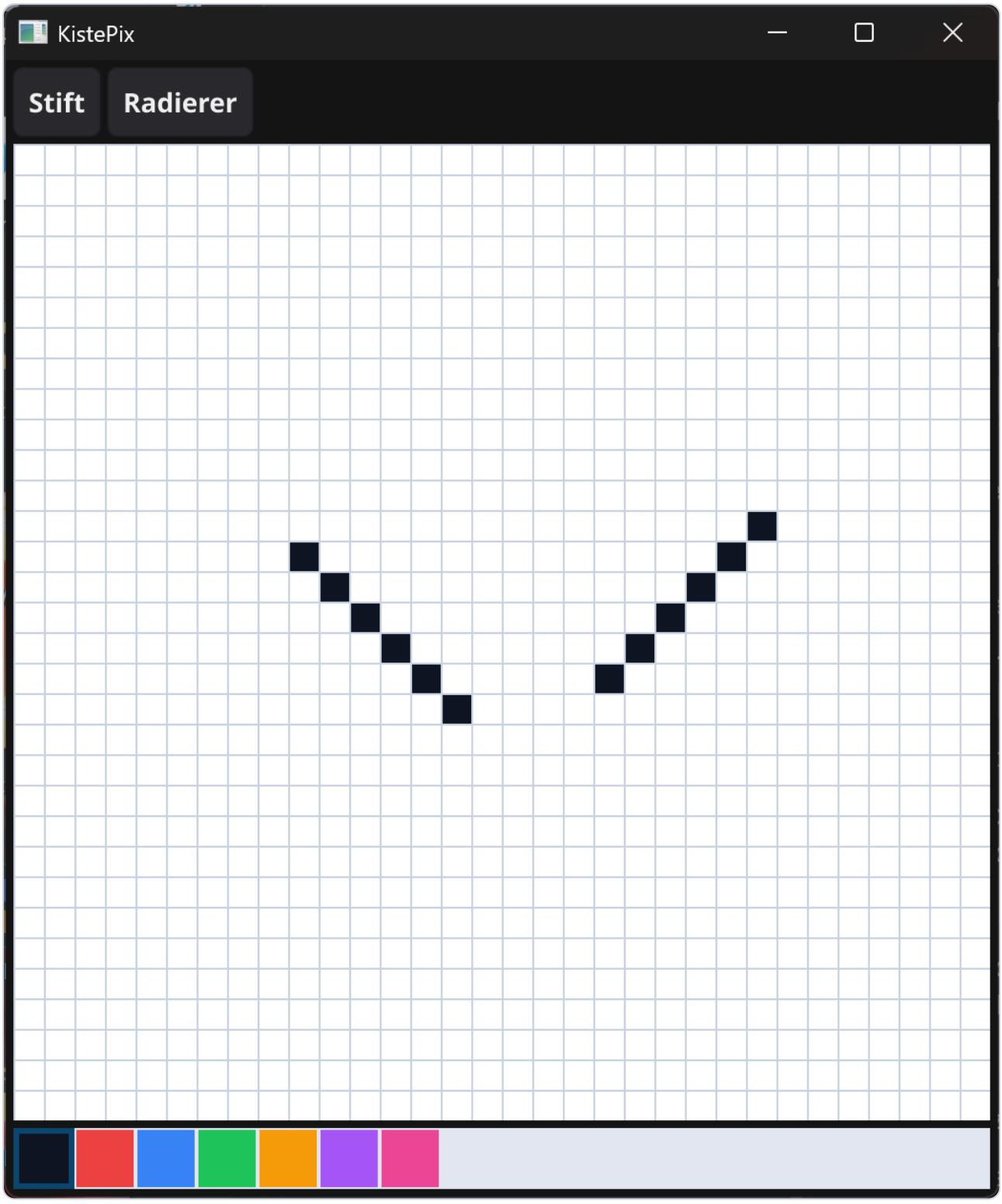
```
nimm inhalt = gui.vertikal([leiste, bild_gui, palette_gui])
```

Das ganze Programm

kistepix.ki (Stufe 4)

```
// KistePix – Stufe 4: Ziehen zum Malen und der Radierer.
// Mit gedrückter Maus ziehst du einen durchgehenden Strich. Zwei Knöpfe schalten
// zwischen Stift und Radierer um – der Radierer setzt Zellen wieder auf „leer".
nutze gui
nutze liste

fest LEIN_B = 32
fest LEIN_H = 32
fest ZELLE = 16
fest FELD = 32
fest RAND = "#cbd5e1"
fest PALETTE = ["#ffffff", "#111827", "#ef4444", "#3b82f6", "#22c55e", "#f59e0b", "#a855f7", "#ec4899"]

nimm aktiv = 1
nimm werkzeug = "stift"   // "stift" oder "radierer" – der zweite Werkzeug-Zustand

nimm pixel = liste.gefüllt(LEIN_B * LEIN_H, 0)
nimm bild_gui = gui.leinwand(LEIN_B * ZELLE, LEIN_H * ZELLE)
nimm palette_gui = gui.leinwand(LEIN_B * ZELLE, FELD)

funktion zellfarbe(wert) {
    gib PALETTE[wert]
}

funktion zeichne_zelle(x, y) {
    nimm f = zellfarbe(pixel[y * LEIN_B + x])
    gui.rechteck(bild_gui, x * ZELLE + 1, y * ZELLE + 1, ZELLE - 1, ZELLE - 1, f)
}

funktion zeichne() {
    gui.rechteck(bild_gui, 0, 0, LEIN_B * ZELLE, LEIN_H * ZELLE, RAND)
    für y = 0 bis LEIN_H - 1 {
        für x = 0 bis LEIN_B - 1 {
            zeichne_zelle(x, y)
        }
    }
}

funktion zeichne_palette() {
    gui.rechteck(palette_gui, 0, 0, LEIN_B * ZELLE, FELD, "#e2e8f0")
    für p = 0 bis länge(PALETTE) - 2 {
        nimm idx = p + 1
```

```
        nimm fx = p * FELD
        wenn idx == aktiv {
            gui.rechteck(palette_gui, fx, 0, FELD, FELD, "#0c4a6e")
            gui.rechteck(palette_gui, fx + 3, 3, FELD - 6, FELD - 6, PALETTE[idx])
        } sonst {
            gui.rechteck(palette_gui, fx + 1, 1, FELD - 2, FELD - 2, PALETTE[idx])
        }
    }
}

// Eine Zelle bearbeiten - je nach Werkzeug malen oder radieren.
funktion benutze(x, y) {
    wenn x >= 0 und x < LEIN_B und y >= 0 und y < LEIN_H {
        wenn werkzeug == "radierer" {
            pixel[y * LEIN_B + x] = 0
        } sonst {
            pixel[y * LEIN_B + x] = aktiv
        }
        zeichne_zelle(x, y)
    }
}

zeichne()
zeichne_palette()

// Klick UND Ziehen benutzen dieselbe Aktion - nur die Zelle unter der Maus zählt.
gui.bei_maus_klick(bild_gui, funktion(b) {
    benutze(gui.maus_x(b) div ZELLE, gui.maus_y(b) div ZELLE)
})
gui.bei_maus_ziehen(bild_gui, funktion(b) {
    benutze(gui.maus_x(b) div ZELLE, gui.maus_y(b) div ZELLE)
})

gui.bei_maus_klick(palette_gui, funktion(b) {
    nimm p = gui.maus_x(b) div FELD
    wenn p >= 0 und p < länge(PALETTE) - 1 {
        aktiv = p + 1
        zeichne_palette()
    }
})

// Werkzeugleiste: zwei echte Knöpfe schalten den Werkzeug-Zustand um.
nimm knopf_stift = gui.knopf_neu("Stift")
nimm knopf_radierer = gui.knopf_neu("Radierer")
gui.bei_klick(knopf_stift, funktion(k) { werkzeug = "stift" })
gui.bei_klick(knopf_radierer, funktion(k) { werkzeug = "radierer" })
nimm leiste = gui.horizontal([knopf_stift, knopf_radierer])

nimm inhalt = gui.vertikal([leiste, bild_gui, palette_gui])
nimm fenster = gui.fenster_öffne("KistePix", LEIN_B * ZELLE, LEIN_H * ZELLE + FELD + 48)
gui.zeige(fenster, inhalt)
gui.starte()
```



Ein durchgehender Zug (das „V") — und unten in der Spitze eine Lücke, dort hat der Radierer gewirkt.

Zwei Zustände, ein Muster

KistePix merkt sich jetzt **zwei** Dinge: die aktive Farbe (`aktiv`) und das aktive Werkzeug (`werkzeug`). Beide funktionieren gleich — eine Variable hält den Zustand, ein Klick ändert ihn, das Malen liest ihn. Dieses Muster — *Zustand merken, auf Klick ändern, beim Tun lesen* — ist der Bauplan für **jedes** weitere Werkzeug. In den nächsten Stufen kommen Rechteck und Füll-Eimer dazu, und beide reihen sich genau hier ein.

Teste dich

1. Warum entsteht beim Ziehen eine durchgehende Linie und nicht nur einzelne Punkte?
2. Welchen Wert setzt der Radierer in eine Zelle — und was bedeutet dieser Wert?
3. Was ist der Unterschied zwischen `gui.horizontal` und `gui.vertikal`?

Aufgabe ★

Der Radierer setzt eine Zelle immer auf `0` (Weiß/leer). Baue einen dritten Knopf `"Leeren"`, der die *ganze* Leinwand auf einmal löscht. Tipp: In einer Funktion mit einer Schleife über alle Zellen jede auf `0` setzen, danach einmal `zeichne()` aufrufen. Häng den Knopf mit in die `leiste`.

Lösung

Teste dich: (1) Weil `bei_maus_ziehen` laufend feuert — für jede Position, die die Maus beim Ziehen überstreicht —, und jede davon eine Zelle malt. (2) Den Wert `0`; das ist der Palettenindex für „leer" (Weiß). (3) `horizontal` ordnet Bedienelemente nebeneinander an, `vertikal` übereinander.

Aufgabe: So könnte der „Leeren"-Knopf aussehen — eine Funktion plus ein Knopf:

```
funktion leeren() {
    für i = 0 bis LEIN_B * LEIN_H - 1 {
        pixel[i] = 0
    }
    zeichne()
}
nimm knopf_leeren = gui.knopf_neu("Leeren")
gui.bei_klick(knopf_leeren, funktion(k) { leeren() })
// … und knopf_leeren mit in gui.horizontal([...]) aufnehmen.
```

Hier läuft `zeichne()` (die ganze Leinwand) statt `zeichne_zelle` — richtig so, denn diesmal ändert sich ja wirklich alles auf einen Schlag.

Stufe 5 — Das Rechteck

Was du hier baust

Ein **Rechteck-Werkzeug** mit Gummiband: Du drückst an einer Ecke, ziehst auf — und siehst dabei live eine **Vorschau** —, und beim Loslassen wird das Rechteck gefüllt. Damit malst du gefüllte Flächen in einem Zug, statt Zelle für Zelle.

Stift und Radierer wirken sofort unter der Maus. Ein Rechteck ist anders: Es entsteht zwischen *zwei* Ecken, und die zweite steht erst fest, wenn du die Maus loslässt. Dazwischen willst du sehen, was entstehen wird. Diese drei Momente — **drücken, ziehen, loslassen** — sind das Gummiband-Prinzip, das fast jedes Formwerkzeug benutzt.

Drei Ereignisse, ein Werkzeug

- **Drücken & erstes Ziehen:** Wir merken uns die **Start-Ecke** in `zieh_x` / `zieh_y`.
- **Weiterziehen:** Bei jedem Zug-Ereignis zeichnen wir die Leinwand neu und legen eine **Vorschau** des Rechtecks darüber — so wandert es mit der Maus.
- **Loslassen:** Jetzt schreiben wir das Rechteck **fest** ins Raster. Dafür gibt es `gui.bei_maus_los` — es feuert genau einmal am Ende eines Zuges.

Der Start wird als „läuft gerade ein Zug?" gemerkt. Solange keiner läuft, steht `zieh_x` auf `-1`:

kistepix.ki (Ausschnitt)

```
nimm werkzeug = "stift"    // "stift" | "radierer" | "rechteck"
nimm zieh_x = -1           // Start-Ecke des Gummibands (-1 = kein Zug läuft)
nimm zieh_y = -1
```

Über alle Zellen des Rechtecks: `rechteck_zellen`

Ob Vorschau oder Festschreiben — beides läuft über *dieselben* Zellen. Darum eine Funktion für beides, gesteuert durch einen Wahrheitswert `nur_vorschau`:

kistepix.ki (Ausschnitt)

```
funktion rechteck_zellen(x0, y0, x1, y1, nur_vorschau) {
    nimm lx = kleiner(x0, x1)
    nimm rx = größer(x0, x1)
    nimm oy = kleiner(y0, y1)
    nimm uy = größer(y0, y1)
    für y = oy bis uy {
        für x = lx bis rx {
            wenn x >= 0 und x < LEIN_B und y >= 0 und y < LEIN_H {
                wenn nur_vorschau {
                    gui.rechteck(bild_gui, x * ZELLE + 1, y * ZELLE + 1, ZELLE - 1, ZELLE - 1, PALETTE[aktiv])
                } sonst {
                    pixel[y * LEIN_B + x] = aktiv
                }
            }
        }
    }
}
```

Wort für Wort

- Die zwei Ecken können in beliebiger Richtung gezogen werden. `kleiner` und `größer` (zwei winzige Helfer) sortieren sie, sodass `lx` / `oy` immer links-oben und `rx` / `uy` rechts-unten sind.
- Die verschachtelte Schleife (Kapitel 6) läuft über jede Zelle im Rechteck.
- Bei `nur_vorschau` wird die Zelle nur *gezeichnet* (nichts gemerkt); sonst wird der Wert fest ins `pixel`-Raster geschrieben. Ein Flag, zwei Verwendungen.

Die drei Behandler

Jetzt hängen wir das Werkzeug an die drei Maus-Ereignisse. Wichtig: Stift und Radierer sollen wie bisher funktionieren — darum fragt jeder Behandler zuerst, welches Werkzeug aktiv ist.

kistepix.ki (Ausschnitt)

```
gui.bei_maus_ziehen(bild_gui, funktion(b) {
    nimm x = gui.maus_x(b) div ZELLE
    nimm y = gui.maus_y(b) div ZELLE
    wenn werkzeug == "rechteck" {
        wenn zieh_x == -1 {
            zieh_x = x
            zieh_y = y
        }
        zeichne()                                  // alte Vorschau wegräumen
        rechteck_zellen(zieh_x, zieh_y, x, y, wahr) // neue Vorschau zeigen
    } sonst {
        benutze(x, y)
    }
})

gui.bei_maus_los(bild_gui, funktion(b) {
    wenn werkzeug == "rechteck" und zieh_x != -1 {
        nimm x = gui.maus_x(b) div ZELLE
        nimm y = gui.maus_y(b) div ZELLE
        rechteck_zellen(zieh_x, zieh_y, x, y, falsch)
        zieh_x = -1
        zeichne()
    }
})
```

Warum hier die ganze Leinwand neu gezeichnet wird

Beim Stift zeichnen wir nur *eine* Zelle — hier rufen wir `zeichne()` (alles). Das ist richtig so: Die Vorschau muss bei jedem Zug-Schritt zuerst die *vorige* Vorschau wegräumen, sonst bliebe eine Spur stehen. Wir zeichnen also die echte Leinwand neu und malen die Vorschau frisch darüber. Bei unserer 32×32-Leinwand ist das flüssig. (Auf einer *sehr* großen Leinwand würde man auch hier das Kompositbild aus dem Ausblick nehmen — für uns bleibt es bei der klaren Variante.)

Das ganze Programm

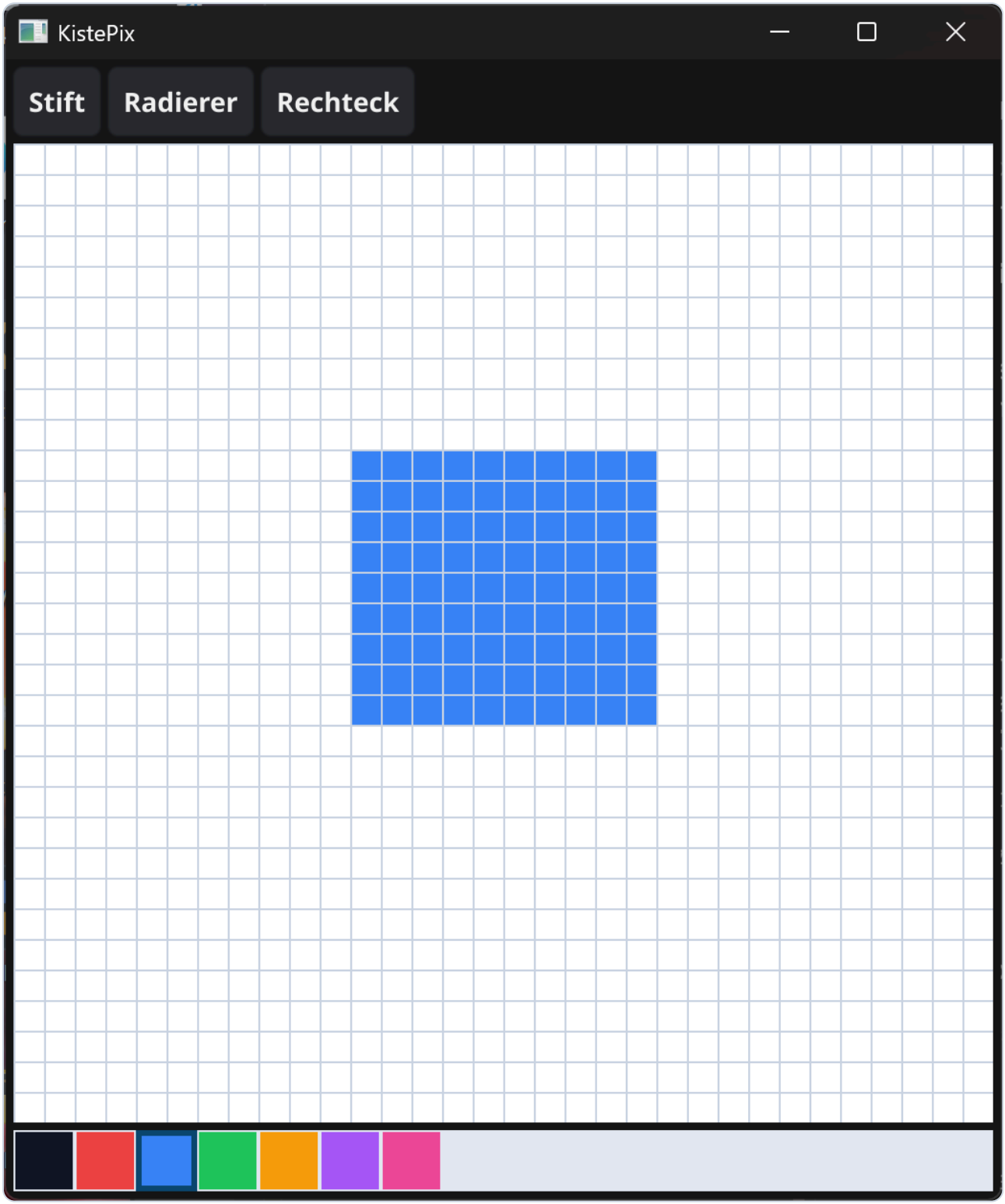
kistepix.ki (Stufe 5)

```
// KistePix – Stufe 5: Das Rechteck (Gummiband).
// Drücken setzt die eine Ecke, Ziehen zeigt eine Vorschau, Loslassen setzt das
// Rechteck fest. Ein drittes Werkzeug reiht sich in den Werkzeug-Zustand ein.
nutze gui
nutze liste

fest LEIN_B = 32
fest LEIN_H = 32
fest ZELLE = 16
fest FELD = 32
fest RAND = "#cbd5e1"
fest PALETTE = ["#ffffff", "#111827", "#ef4444", "#3b82f6", "#22c55e", "#f59e0b", "#a855f7", "#ec4899"]

nimm aktiv = 1
nimm werkzeug = "stift"    // "stift" | "radierer" | "rechteck"
nimm zieh_x = -1           // Start-Ecke des Gummibands (-1 = kein Zug läuft)
nimm zieh_y = -1

nimm pixel = liste.gefüllt(LEIN_B * LEIN_H, 0)
```

```
nimm bild_gui = gui.leinwand(LEIN_B * ZELLE, LEIN_H * ZELLE)
nimm palette_gui = gui.leinwand(LEIN_B * ZELLE, FELD)

funktion kleiner(a, b) {
    wenn a < b { gib a }
    gib b
}
funktion größer(a, b) {
    wenn a > b { gib a }
    gib b
}

funktion zellfarbe(wert) {
    gib PALETTE[wert]
}

funktion zeichne_zelle(x, y) {
    nimm f = zellfarbe(pixel[y * LEIN_B + x])
    gui.rechteck(bild_gui, x * ZELLE + 1, y * ZELLE + 1, ZELLE - 1, ZELLE - 1, f)
}

funktion zeichne() {
    gui.rechteck(bild_gui, 0, 0, LEIN_B * ZELLE, LEIN_H * ZELLE, RAND)
    für y = 0 bis LEIN_H - 1 {
        für x = 0 bis LEIN_B - 1 {
            zeichne_zelle(x, y)
        }
    }
}

funktion zeichne_palette() {
    gui.rechteck(palette_gui, 0, 0, LEIN_B * ZELLE, FELD, "#e2e8f0")
    für p = 0 bis länge(PALETTE) - 2 {
        nimm idx = p + 1
        nimm fx = p * FELD
        wenn idx == aktiv {
            gui.rechteck(palette_gui, fx, 0, FELD, FELD, "#0c4a6e")
            gui.rechteck(palette_gui, fx + 3, 3, FELD - 6, FELD - 6, PALETTE[idx])
        } sonst {
            gui.rechteck(palette_gui, fx + 1, 1, FELD - 2, FELD - 2, PALETTE[idx])
        }
    }
}

funktion benutze(x, y) {
    wenn x >= 0 und x < LEIN_B und y >= 0 und y < LEIN_H {
        wenn werkzeug == "radierer" {
            pixel[y * LEIN_B + x] = 0
        } sonst {
            pixel[y * LEIN_B + x] = aktiv
        }
        zeichne_zelle(x, y)
    }
}

// Über alle Zellen des Rechtecks (x0,y0)-(x1,y1) laufen. nur_vorschau = wahr
// zeigt es nur (Vorschau), sonst wird es fest ins Raster geschrieben.
funktion rechteck_zellen(x0, y0, x1, y1, nur_vorschau) {
    nimm lx = kleiner(x0, x1)
    nimm rx = größer(x0, x1)
    nimm oy = kleiner(y0, y1)
    nimm uy = größer(y0, y1)
    für y = oy bis uy {
```

```
        für x = lx bis rx {
            wenn x >= 0 und x < LEIN_B und y >= 0 und y < LEIN_H {
                wenn nur_vorschau {
                    gui.rechteck(bild_gui, x * ZELLE + 1, y * ZELLE + 1, ZELLE - 1, ZELLE - 1, PALETTE[aktiv])
                } sonst {
                    pixel[y * LEIN_B + x] = aktiv
                }
            }
        }
    }
}

zeichne()
zeichne_palette()

// Klick (ohne Ziehen): Stift/Radierer malen; Rechteck macht auf einen Klick nichts.
gui.bei_maus_klick(bild_gui, funktion(b) {
    wenn werkzeug != "rechteck" {
        benutze(gui.maus_x(b) div ZELLE, gui.maus_y(b) div ZELLE)
    }
})

// Ziehen: Stift/Radierer malen laufend; Rechteck merkt die Start-Ecke und zeigt Vorschau.
gui.bei_maus_ziehen(bild_gui, funktion(b) {
    nimm x = gui.maus_x(b) div ZELLE
    nimm y = gui.maus_y(b) div ZELLE
    wenn werkzeug == "rechteck" {
        wenn zieh_x == -1 {
            zieh_x = x
            zieh_y = y
        }
        zeichne()                                    // alte Vorschau wegräumen
        rechteck_zellen(zieh_x, zieh_y, x, y, wahr) // neue Vorschau zeigen
    } sonst {
        benutze(x, y)
    }
})

// Loslassen: das Rechteck fest ins Raster schreiben.
gui.bei_maus_los(bild_gui, funktion(b) {
    wenn werkzeug == "rechteck" und zieh_x != -1 {
        nimm x = gui.maus_x(b) div ZELLE
        nimm y = gui.maus_y(b) div ZELLE
        rechteck_zellen(zieh_x, zieh_y, x, y, falsch)
        zieh_x = -1
        zeichne()
    }
})

gui.bei_maus_klick(palette_gui, funktion(b) {
    nimm p = gui.maus_x(b) div FELD
    wenn p >= 0 und p < länge(PALETTE) - 1 {
        aktiv = p + 1
        zeichne_palette()
    }
})

nimm knopf_stift = gui.knopf_neu("Stift")
nimm knopf_radierer = gui.knopf_neu("Radierer")
nimm knopf_rechteck = gui.knopf_neu("Rechteck")
gui.bei_klick(knopf_stift, funktion(k) { werkzeug = "stift" })
gui.bei_klick(knopf_radierer, funktion(k) { werkzeug = "radierer" })
gui.bei_klick(knopf_rechteck, funktion(k) { werkzeug = "rechteck" })
```

```
nimm leiste = gui.horizontal([knopf_stift, knopf_radierer, knopf_rechteck])

nimm inhalt = gui.vertikal([leiste, bild_gui, palette_gui])
nimm fenster = gui.fenster_öffne("KistePix", LEIN_B * ZELLE, LEIN_H * ZELLE + FELD + 48)
gui.zeige(fenster, inhalt)
gui.starte()
```



Ein blaues Rechteck, in einem Zug aufgezogen. Der neue Knopf „Rechteck" wählt das Werkzeug; Farbe kommt wie immer aus der Palette.

Das Werkzeug-Muster trägt

Schau, wie mühelos sich das Rechteck einfügt: ein neuer Wert für `werkzeug`, ein neuer Knopf, ein paar Zeilen in den Behandlern — fertig. Genau das hatten wir in Stufe 4 versprochen: *Zustand merken, auf Klick ändern, beim Tun lesen.* Das nächste Werkzeug, der Füll-Eimer, reiht sich in Stufe 6 wieder genau hier ein — und bringt die spannendste Idee des ganzen Buches mit.

Teste dich

1. Welche drei Maus-Ereignisse machen das Gummiband aus?
2. Wozu dienen die Helfer `kleiner` und `größer` in `rechteck_zellen`?
3. Warum ruft die Vorschau `zeichne()` (die ganze Leinwand) und nicht nur `zeichne_zelle`?

Aufgabe ★

Das Rechteck ist *gefüllt*. Überlege (und probiere, wenn du magst): Wie müsstest du `rechteck_zellen` ändern, damit nur der **Rahmen** gemalt wird statt der ganzen Fläche? Tipp: Eine Zelle gehört zum Rahmen, wenn sie in der *ersten* oder *letzten* Spalte (`x == lx` oder `x == rx`) oder Zeile (`y == oy` oder `y == uy`) liegt.

Lösung

Teste dich: (1) Drücken (Start-Ecke), Ziehen (Vorschau), Loslassen (festsetzen). (2) Sie sortieren die zwei Ecken, damit die Schleife immer von links-oben nach rechts-unten läuft — egal, in welche Richtung du gezogen hast. (3) Weil die vorige Vorschau weg muss; ein einzelnes Neuzeichnen würde die alte Spur stehen lassen.

Aufgabe: Du würdest in der innersten Schleife nur dann malen, wenn die Zelle am Rand liegt — etwa so:

```
wenn x == lx oder x == rx oder y == oy oder y == uy {
    // … dieselbe Mal-/Merk-Zeile wie bisher …
}
```

So entsteht ein Rechteck-Umriss. Beides — gefüllt und Umriss — kannst du dir später sogar als zwei Werkzeuge nebeneinander vorstellen.

Stufe 6 — Füllen: eine ganze Fläche auf einen Klick

Was du hier baust

Den **Farbeimer**: Ein Klick färbt den ganzen *zusammenhängenden* Bereich der angeklickten Farbe um. Das ist das mächtigste Werkzeug von KistePix — und das mit der spannendsten Idee dahinter. Wir bauen es von Anfang an **richtig**, sodass es auch auf großen Bildern hält.

Ein Farbeimer klickt man in eine Fläche, und sie läuft voll — bis zu den Rändern, wo eine andere Farbe beginnt. Klingt einfach, ist aber die kniffligste Aufgabe im ganzen Buch: Der Computer muss von der angeklickten Zelle aus **alle Nachbarn gleicher Farbe** finden, und deren Nachbarn, und so weiter — ohne sich zu verheddern.

Die Idee: eine Warteschlange

Wir führen eine **Warteschlange** („to-do-Liste") von Zellen, die wir noch bearbeiten müssen. Der Ablauf:

- Die angeklickte Zelle kommt in die Warteschlange.
- Wir nehmen vorne eine Zelle heraus, färben sie um und schauen ihre **vier Nachbarn** an (oben, unten, links, rechts).
- Jeder Nachbar, der noch die alte Farbe hat, kommt *hinten* in die Warteschlange.
- Das wiederholen wir, bis die Warteschlange leer ist. Dann ist die ganze Fläche gefüllt.

So breitet sich die Farbe wie Wasser über die zusammenhängende Fläche aus. „Vorne heraus, hinten hinein" — das nennt man eine Warteschlange, genau wie an der Kasse.

> **Der eine Trick, auf den alles ankommt**
> Es gibt eine Falle, und sie ist der Grund, warum dieses Kapitel so sorgfältig gebaut ist. **Wir färben jede Zelle in genau dem Moment um, in dem wir sie in die Warteschlange legen** — nicht erst, wenn wir sie später herausnehmen. Warum? Eine Zelle hat mehrere Nachbarn, und jeder dieser Nachbarn würde sie „entdecken". Markieren wir sie erst später, käme dieselbe Zelle **mehrfach** in die Warteschlange — bei großen Flächen schwillt sie dann gewaltig an, das Programm wird langsam oder stürzt ab. Markieren wir sofort beim Einreihen, ist jede Zelle **garantiert genau einmal** drin. Deshalb reicht die Warteschlange auch nie über und wir können sie in fester Größe vorbereiten.

Die vorbereitete Warteschlange

Weil jede Zelle höchstens einmal in die Warteschlange kommt, ist die größtmögliche Zahl von Einträgen bekannt: die ganze Fläche. Also legen wir die Warteschlange **einmal** in voller Größe an (wie die Leinwand selbst, Kapitel 8) und verwenden sie bei jeder Füllung wieder:

kistepix.ki (Ausschnitt)

```
// Die Warteschlange fürs Füllen: EINMAL angelegt, Platz für die ganze Fläche.
nimm füll_q = liste.gefüllt(LEIN_B * LEIN_H, 0)
```

Statt zweier Positionen merken wir uns nur zwei Zahlen: `kopf` (wo wir als Nächstes herausnehmen) und `schwanz` (wo wir hinten anfügen). Solange `kopf` den `schwanz` nicht eingeholt hat, ist noch Arbeit da.

Die `fülle`-Funktion

kistepix.ki (Ausschnitt)

```
funktion fülle(zx, zy) {
    nimm start = zy * LEIN_B + zx
    nimm ziel = pixel[start]        // die Farbe, die wir überfluten
    wenn ziel != aktiv {            // schon die neue Farbe? Dann nichts tun.
```

```
        nimm kopf = 0
        nimm schwanz = 0
        füll_q[schwanz] = start
        schwanz = schwanz + 1
        pixel[start] = aktiv        // beim Einreihen SOFORT markieren
        solange kopf < schwanz {
            nimm nr = füll_q[kopf]
            kopf = kopf + 1
            nimm x = nr % LEIN_B
            nimm y = nr div LEIN_B
            wenn x + 1 < LEIN_B {
                nimm m = nr + 1
                wenn pixel[m] == ziel {
                    pixel[m] = aktiv
                    füll_q[schwanz] = m
                    schwanz = schwanz + 1
                }
            }
            wenn x - 1 >= 0 {
                nimm m = nr - 1
                wenn pixel[m] == ziel {
                    pixel[m] = aktiv
                    füll_q[schwanz] = m
                    schwanz = schwanz + 1
                }
            }
            wenn y + 1 < LEIN_H {
                nimm m = nr + LEIN_B
                wenn pixel[m] == ziel {
                    pixel[m] = aktiv
                    füll_q[schwanz] = m
                    schwanz = schwanz + 1
                }
            }
            wenn y - 1 >= 0 {
                nimm m = nr - LEIN_B
                wenn pixel[m] == ziel {
                    pixel[m] = aktiv
                    füll_q[schwanz] = m
                    schwanz = schwanz + 1
                }
            }
        }
        zeichne()
    }
}
```

Wort für Wort

- `ziel` ist die Farbe unter dem Klick — die, die wir überfluten. Ist sie schon die `aktiv`e Farbe, gibt es nichts zu tun.
- `füll_q[schwanz] = start` legt die Startzelle hinten an; `pixel[start] = aktiv` markiert sie **sofort**.
- Die `solange`-Schleife (Kapitel 6) arbeitet die Warteschlange ab. `nr % LEIN_B` und `nr div LEIN_B` holen Spalte und Zeile zurück (Kapitel 3 — der Weg *rückwärts* durch die Rasterformel).
- Die vier `wenn`-Blöcke prüfen die vier Nachbarn. Jeder Block hat *zuerst* die Grenzprüfung (bleibt der Nachbar auf der Leinwand?), dann: hat er noch die `ziel`-Farbe? Wenn ja — sofort umfärben und hinten anfügen.
- Am Ende einmal `zeichne()`: Diesmal hat sich wirklich viel geändert, also die ganze Leinwand neu — genau der Fall, für den wir `zeichne()` aufgehoben haben.

Das ganze Programm

Der Füller ist das vierte Werkzeug — ein neuer Knopf, ein Zweig im Klick-Behandler, sonst wie gehabt.

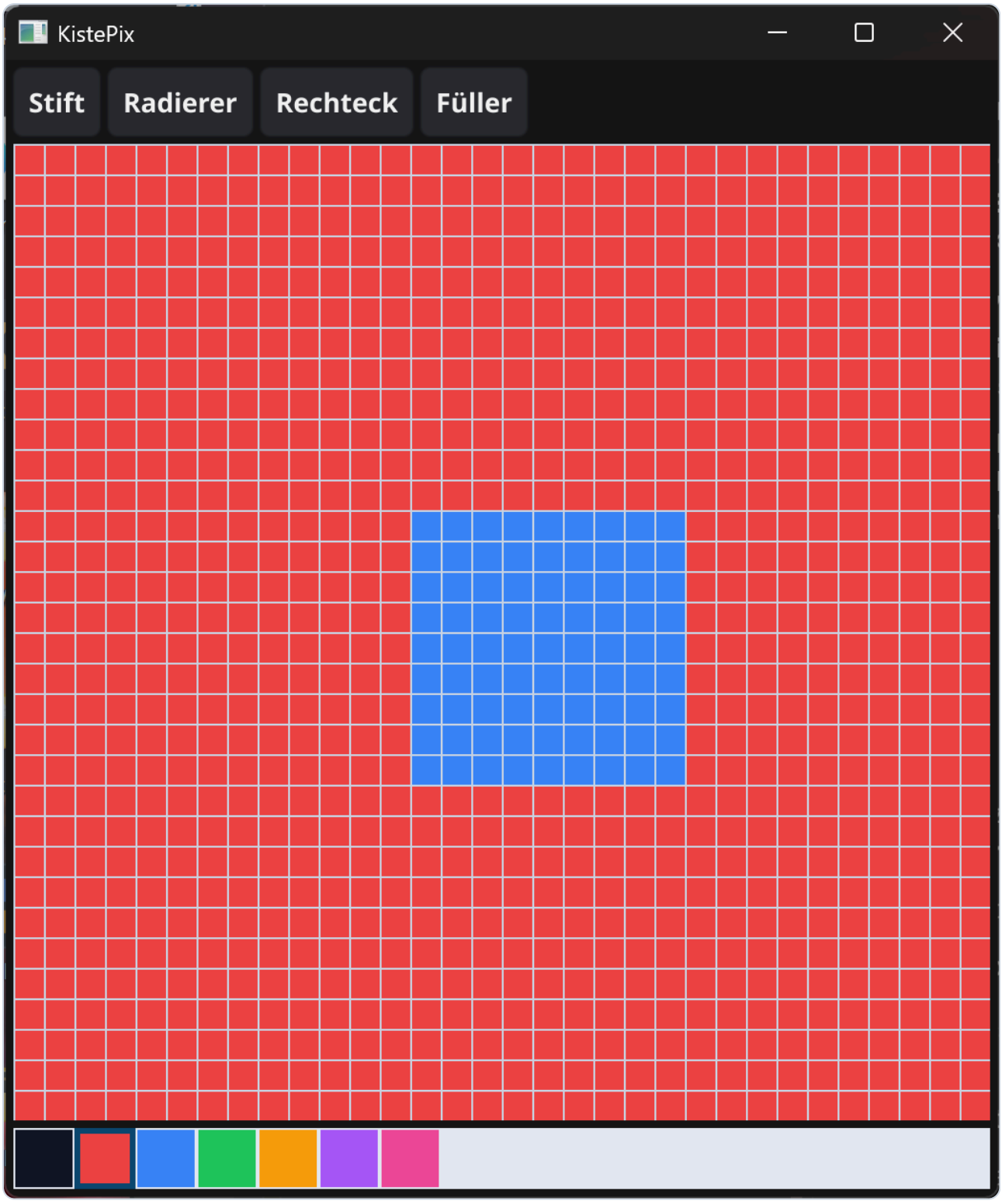
kistepix.ki (Stufe 6)

```
// KistePix – Stufe 6: Füllen (der Farbeimer).
// Ein Klick füllt den ganzen zusammenhängenden Bereich der angeklickten Farbe.
// WICHTIG: Wir markieren jede Zelle BEIM EINREIHEN sofort (setzen sie auf die
// neue Farbe) – so landet jede Zelle GENAU EINMAL in der Warteschlange. Die
// Warteschlange ist EINMAL vorbereitet (liste.gefüllt) und wird wiederverwendet.
nutze gui
nutze liste

fest LEIN_B = 32
fest LEIN_H = 32
fest ZELLE = 16
fest FELD = 32
fest RAND = "#cbd5e1"
fest PALETTE = ["#ffffff", "#111827", "#ef4444", "#3b82f6", "#22c55e", "#f59e0b", "#a855f7", "#ec4899"]

nimm aktiv = 1
nimm werkzeug = "stift"    // "stift" | "radierer" | "rechteck" | "füller"
nimm zieh_x = -1
nimm zieh_y = -1

nimm pixel = liste.gefüllt(LEIN_B * LEIN_H, 0)
// Die Warteschlange fürs Füllen: EINMAL angelegt, Platz für die ganze Fläche.
nimm füll_q = liste.gefüllt(LEIN_B * LEIN_H, 0)

nimm bild_gui = gui.leinwand(LEIN_B * ZELLE, LEIN_H * ZELLE)
nimm palette_gui = gui.leinwand(LEIN_B * ZELLE, FELD)

funktion kleiner(a, b) {
    wenn a < b { gib a }
    gib b
}
funktion größer(a, b) {
    wenn a > b { gib a }
    gib b
}
```

```
funktion zellfarbe(wert) {
    gib PALETTE[wert]
}

funktion zeichne_zelle(x, y) {
    nimm f = zellfarbe(pixel[y * LEIN_B + x])
    gui.rechteck(bild_gui, x * ZELLE + 1, y * ZELLE + 1, ZELLE - 1, ZELLE - 1, f)
}

funktion zeichne() {
    gui.rechteck(bild_gui, 0, 0, LEIN_B * ZELLE, LEIN_H * ZELLE, RAND)
    für y = 0 bis LEIN_H - 1 {
        für x = 0 bis LEIN_B - 1 {
            zeichne_zelle(x, y)
        }
    }
}

funktion zeichne_palette() {
    gui.rechteck(palette_gui, 0, 0, LEIN_B * ZELLE, FELD, "#e2e8f0")
    für p = 0 bis länge(PALETTE) - 2 {
        nimm idx = p + 1
        nimm fx = p * FELD
        wenn idx == aktiv {
            gui.rechteck(palette_gui, fx, 0, FELD, FELD, "#0c4a6e")
            gui.rechteck(palette_gui, fx + 3, 3, FELD - 6, FELD - 6, PALETTE[idx])
        } sonst {
            gui.rechteck(palette_gui, fx + 1, 1, FELD - 2, FELD - 2, PALETTE[idx])
        }
    }
}

funktion benutze(x, y) {
    wenn x >= 0 und x < LEIN_B und y >= 0 und y < LEIN_H {
        wenn werkzeug == "radierer" {
            pixel[y * LEIN_B + x] = 0
        } sonst {
            pixel[y * LEIN_B + x] = aktiv
        }
        zeichne_zelle(x, y)
    }
}

funktion rechteck_zellen(x0, y0, x1, y1, nur_vorschau) {
    nimm lx = kleiner(x0, x1)
    nimm rx = größer(x0, x1)
    nimm oy = kleiner(y0, y1)
    nimm uy = größer(y0, y1)
    für y = oy bis uy {
        für x = lx bis rx {
            wenn x >= 0 und x < LEIN_B und y >= 0 und y < LEIN_H {
                wenn nur_vorschau {
                    gui.rechteck(bild_gui, x * ZELLE + 1, y * ZELLE + 1, ZELLE - 1, ZELLE - 1, PALETTE[aktiv])
                } sonst {
                    pixel[y * LEIN_B + x] = aktiv
                }
            }
        }
    }
}

// Der Farbeimer: füllt den zusammenhängenden Bereich der Farbe unter (zx, zy).
```

```
funktion fülle(zx, zy) {
    nimm start = zy * LEIN_B + zx
    nimm ziel = pixel[start]        // die Farbe, die wir überfluten
    wenn ziel != aktiv {            // schon die neue Farbe? Dann nichts tun.
        nimm kopf = 0
        nimm schwanz = 0
        füll_q[schwanz] = start
        schwanz = schwanz + 1
        pixel[start] = aktiv        // beim Einreihen SOFORT markieren
        solange kopf < schwanz {
            nimm nr = füll_q[kopf]
            kopf = kopf + 1
            nimm x = nr % LEIN_B
            nimm y = nr div LEIN_B
            wenn x + 1 < LEIN_B {
                nimm m = nr + 1
                wenn pixel[m] == ziel {
                    pixel[m] = aktiv
                    füll_q[schwanz] = m
                    schwanz = schwanz + 1
                }
            }
            wenn x - 1 >= 0 {
                nimm m = nr - 1
                wenn pixel[m] == ziel {
                    pixel[m] = aktiv
                    füll_q[schwanz] = m
                    schwanz = schwanz + 1
                }
            }
            wenn y + 1 < LEIN_H {
                nimm m = nr + LEIN_B
                wenn pixel[m] == ziel {
                    pixel[m] = aktiv
                    füll_q[schwanz] = m
                    schwanz = schwanz + 1
                }
            }
            wenn y - 1 >= 0 {
                nimm m = nr - LEIN_B
                wenn pixel[m] == ziel {
                    pixel[m] = aktiv
                    füll_q[schwanz] = m
                    schwanz = schwanz + 1
                }
            }
        }
        zeichne()
    }
}

zeichne()
zeichne_palette()

gui.bei_maus_klick(bild_gui, funktion(b) {
    nimm x = gui.maus_x(b) div ZELLE
    nimm y = gui.maus_y(b) div ZELLE
    wenn werkzeug == "füller" {
        wenn x >= 0 und x < LEIN_B und y >= 0 und y < LEIN_H {
            fülle(x, y)
        }
    } sonstwenn werkzeug != "rechteck" {
        benutze(x, y)
```

```
    }
})

gui.bei_maus_ziehen(bild_gui, funktion(b) {
    nimm x = gui.maus_x(b) div ZELLE
    nimm y = gui.maus_y(b) div ZELLE
    wenn werkzeug == "rechteck" {
        wenn zieh_x == -1 {
            zieh_x = x
            zieh_y = y
        }
        zeichne()
        rechteck_zellen(zieh_x, zieh_y, x, y, wahr)
    } sonstwenn werkzeug == "stift" oder werkzeug == "radierer" {
        benutze(x, y)
    }
})

gui.bei_maus_los(bild_gui, funktion(b) {
    wenn werkzeug == "rechteck" und zieh_x != -1 {
        nimm x = gui.maus_x(b) div ZELLE
        nimm y = gui.maus_y(b) div ZELLE
        rechteck_zellen(zieh_x, zieh_y, x, y, falsch)
        zieh_x = -1
        zeichne()
    }
})

gui.bei_maus_klick(palette_gui, funktion(b) {
    nimm p = gui.maus_x(b) div FELD
    wenn p >= 0 und p < länge(PALETTE) - 1 {
        aktiv = p + 1
        zeichne_palette()
    }
})

nimm knopf_stift = gui.knopf_neu("Stift")
nimm knopf_radierer = gui.knopf_neu("Radierer")
nimm knopf_rechteck = gui.knopf_neu("Rechteck")
nimm knopf_füller = gui.knopf_neu("Füller")
gui.bei_klick(knopf_stift, funktion(k) { werkzeug = "stift" })
gui.bei_klick(knopf_radierer, funktion(k) { werkzeug = "radierer" })
gui.bei_klick(knopf_rechteck, funktion(k) { werkzeug = "rechteck" })
gui.bei_klick(knopf_füller, funktion(k) { werkzeug = "füller" })
nimm leiste = gui.horizontal([knopf_stift, knopf_radierer, knopf_rechteck, knopf_füller])

nimm inhalt = gui.vertikal([leiste, bild_gui, palette_gui])
nimm fenster = gui.fenster_öffne("KistePix", LEIN_B * ZELLE, LEIN_H * ZELLE + FELD + 48)
gui.zeige(fenster, inhalt)
gui.starte()
```



Ein Klick mit dem Füller in den weißen Hintergrund — und alles läuft rot voll, sauber bis an die Kanten des blauen Rechtecks. Der Eimer respektiert jede Grenze.

Warum dieses Kapitel so wichtig ist

Genau hier lag der Fehler in der ersten Auflage dieses Buches: Ein Farbeimer, der Zellen zu spät markierte, funktionierte auf kleinen Bildern, blähte aber auf großen die Warteschlange auf und stürzte ab — und das Buch behauptete trotzdem, er sei „sicher". Diese Version macht es richtig. Damit das keine leere Behauptung bleibt, haben wir **genau dieses Muster** an einer riesigen Fläche geprüft: 1500×1000 = **1,5 Millionen Zellen**, in einem Zug vollständig gefüllt, in Sekundenbruchteilen, ohne Absturz. Auf unserer 32×32-Leinwand merkst du davon nichts — aber das Fundament trägt bis dort hinauf.

Teste dich

1. In welchem Moment färben wir eine Zelle um — beim *Einreihen* in die Warteschlange oder beim *Herausnehmen*? Und warum genau dann?
2. Warum reicht für die Warteschlange die Größe „ganze Fläche" sicher aus?
3. Was bedeuten `kopf` und `schwanz`?

Aufgabe ★

Male mit dem Stift eine geschlossene Form (z. B. ein Kästchen-Rechteck als Umriss), wähle eine Farbe und klick mit dem Füller *hinein*. Läuft nur das Innere voll? Klick dann *außerhalb* — läuft der Rest voll, aber die Form bleibt eine Grenze? Überlege: Was passiert, wenn deine Umriss-Linie eine kleine Lücke hat?

Lösung

Teste dich: (1) Beim **Einreihen** — sofort. Sonst könnten mehrere Nachbarn dieselbe Zelle erneut einreihen, und die Warteschlange würde überlaufen. (2) Weil durch das Sofort-Markieren jede Zelle **höchstens einmal** eingereiht wird — mehr Einträge als Zellen kann es also nie geben. (3) `kopf` ist die Stelle, von der wir als Nächstes herausnehmen; `schwanz` die Stelle, an der wir hinten anfügen. Sind sie gleich, ist die Warteschlange leer.

Aufgabe: Innen füllen färbt nur die eingeschlossene Fläche; außen füllen färbt alles Übrige, wobei die Form als Grenze wirkt. Hat der Umriss eine **Lücke**, läuft die Farbe hindurch — innen und außen sind dann *zusammenhängend*, und der Eimer füllt beides. Das ist kein Fehler, sondern genau die Definition von „zusammenhängend": Der Eimer folgt jeder offenen Verbindung.

Stufe 7 — Das Bild als PNG exportieren

Was du hier baust

Den **Export**: Ein Knopf schreibt deine Leinwand als echte **PNG-Bilddatei**, die du überall öffnen, teilen oder in ein Spiel einbauen kannst. Aus deinem KistePix-Werk wird ein richtiges Bild.

Bisher lebt dein Bild nur im Fenster. Jetzt geben wir ihm ein Dateiformat, das die ganze Welt versteht: **PNG**. Dafür hat Kiste den Werkzeugkasten `bild` — er kann ein Bild im Speicher anlegen, mit Farben füllen und als Datei schreiben.

Farben als Zahlen: die gepackte RGB-Darstellung

Ein kleiner Unterschied zuerst: Zum *Zeichnen* auf dem Bildschirm haben wir Farben als Hex-Text benutzt (`"#ef4444"`). Zum *Schreiben* in eine Bilddatei braucht Kiste die Farbe als **Zahl**. Eine Farbe besteht aus drei Anteilen — Rot, Grün, Blau, je 0 bis 255 —, und die packt man mit einer einfachen Rechnung in eine einzige Zahl:

kistepix.ki (Ausschnitt)

```
// Dieselben Farben als Zahl fürs Speichern: r * 65536 + g * 256 + b packt Rot,
// Grün und Blau in eine einzige Ganzzahl (das braucht bild.setze_puffer).
fest PALETTE_RGB = [
    255 * 65536 + 255 * 256 + 255,   // #ffffff (weiß / leer)
    17 * 65536 + 24 * 256 + 39,      // #111827
    239 * 65536 + 68 * 256 + 68,     // #ef4444
    59 * 65536 + 130 * 256 + 246,    // #3b82f6
    34 * 65536 + 197 * 256 + 94,     // #22c55e
    245 * 65536 + 158 * 256 + 11,    // #f59e0b
    168 * 65536 + 85 * 256 + 247,    // #a855f7
    236 * 65536 + 72 * 256 + 153     // #ec4899
]
```

Warum `r * 65536 + g * 256 + b`?

Das ist derselbe Trick wie `y * breite + x` beim Raster: Man legt mehrere Zahlen hintereinander in *eine*. `b` belegt die untersten Stellen, `g` die mittleren (mal 256), `r` die obersten (mal 65536 = 256×256). Weil jeder Anteil höchstens 255 ist, überlappen sie sich nie — genau wie Einer, Hunderter und Zehntausender im Dezimalsystem. Du musst die Zahl nie selbst lesen; wichtig ist nur, dass `PALETTE` und `PALETTE_RGB` dieselben Farben in derselben Reihenfolge beschreiben.

Die `exportiere`-Funktion

Der Export baut ein Bild im Speicher, füllt für jede Rasterzelle einen Block von Bildpunkten und schreibt das Ganze als PNG:

kistepix.ki (Ausschnitt)

```
funktion exportiere() {
    nimm bw = LEIN_B * EXPORT_ZOOM
    nimm bh = LEIN_H * EXPORT_ZOOM
    nimm handle = bild.neu(bw, bh)
    nimm puffer = liste.gefüllt(bw * bh, 0)
    für oy = 0 bis bh - 1 {
        nimm zy = oy div EXPORT_ZOOM
        für ox = 0 bis bw - 1 {
            nimm zx = ox div EXPORT_ZOOM
            puffer[oy * bw + ox] = PALETTE_RGB[pixel[zy * LEIN_B + zx]]
        }
    }
    bild.setze_puffer(handle, puffer)
```

```
    nimm ziel = pfad.verbunden(umgebung.benutzer_ordner(), "kistepix.png")
    bild.schreibe_png(handle, ziel)
    gui.dialog_info(fenster, "Gespeichert", "Dein Bild wurde gespeichert als:\n{ziel}")
}
```

Wort für Wort

- `bild.neu(bw, bh)` legt ein leeres Bild im Speicher an. Mit `EXPORT_ZOOM = 16` wird jede Zelle zu einem 16×16-Block — aus 32×32 Zellen wird ein 512×512-Bild, handlich groß.
- Die zwei Schleifen laufen über *jeden Bildpunkt*. `ox div EXPORT_ZOOM` sagt, zu welcher Rasterzelle er gehört — wieder unser `div` aus Kapitel 3. Die Farbe der Zelle schlagen wir in `PALETTE_RGB` nach und legen sie in den `puffer`.
- `bild.setze_puffer(handle, puffer)` schreibt alle Bildpunkte in *einem* Zug ins Bild — schnell, selbst bei großen Bildern.
- `bild.schreibe_png(handle, ziel)` speichert die Datei. `ziel` ist dein Benutzer-Ordner plus `kistepix.png` — dort findest du sie sicher wieder.
- `gui.dialog_info(…)` zeigt dir zur Bestätigung, wohin gespeichert wurde.

Das ganze Programm

Der Export ist der fünfte Knopf. Alles andere kennst du.

kistepix.ki (Stufe 7)

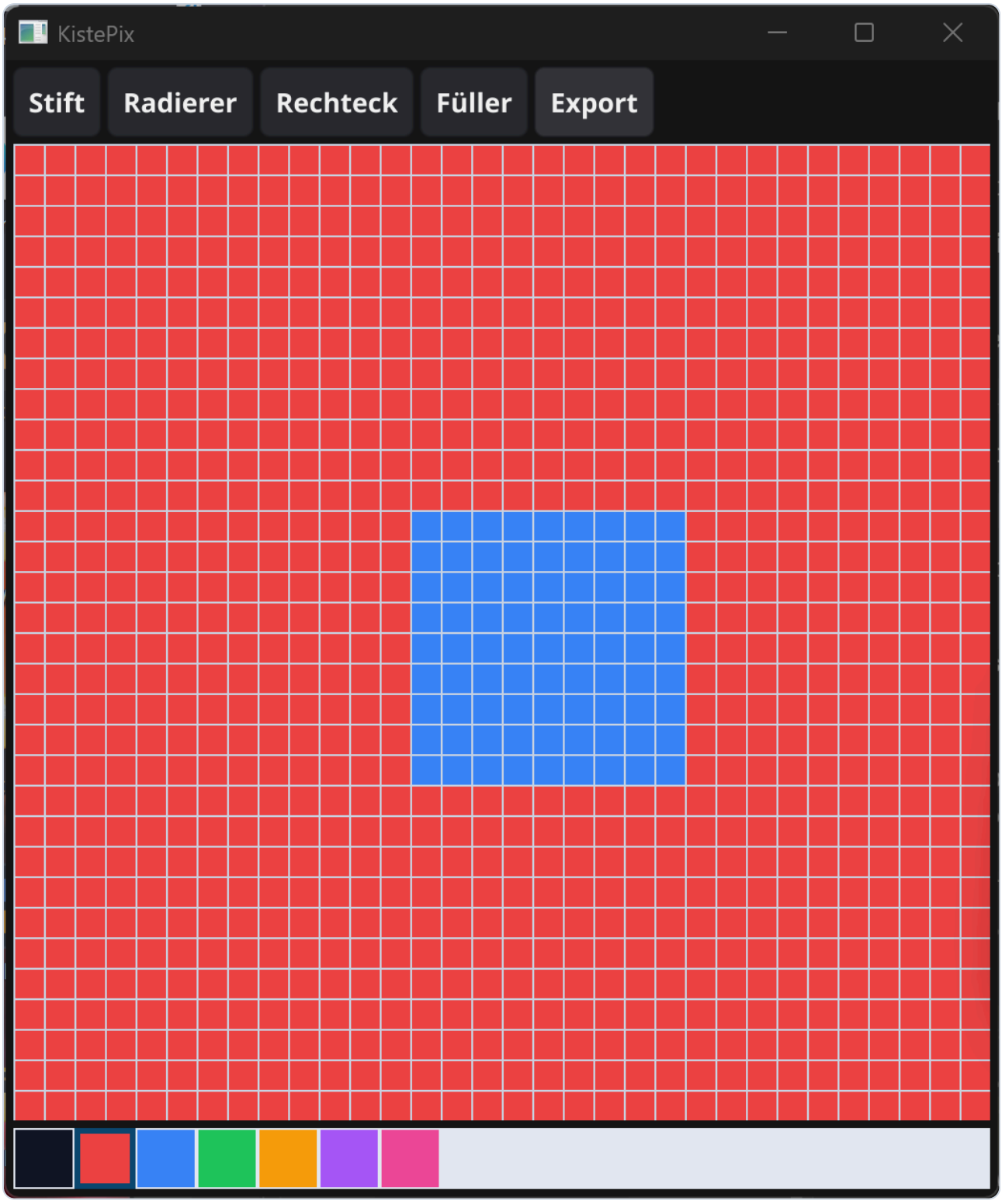
```
// KistePix – Stufe 7: Das Bild als PNG exportieren.
// Ein Knopf schreibt deine Leinwand als echte Bilddatei. Jede Zelle wird zu einem
// Block von Bildpunkten (EXPORT_ZOOM), damit das PNG eine handliche Größe hat.
nutze gui
nutze liste
nutze bild
nutze umgebung
nutze pfad

fest LEIN_B = 32
fest LEIN_H = 32
fest ZELLE = 16
fest FELD = 32
fest RAND = "#cbd5e1"
fest PALETTE = ["#ffffff", "#111827", "#ef4444", "#3b82f6", "#22c55e", "#f59e0b", "#a855f7", "#ec4899"]
// Dieselben Farben als Zahl fürs Speichern: r * 65536 + g * 256 + b packt Rot,
// Grün und Blau in eine einzige Ganzzahl (das braucht bild.setze_puffer).
fest PALETTE_RGB = [
    255 * 65536 + 255 * 256 + 255,   // #ffffff (weiß / leer)
    17 * 65536 + 24 * 256 + 39,      // #111827
    239 * 65536 + 68 * 256 + 68,     // #ef4444
    59 * 65536 + 130 * 256 + 246,    // #3b82f6
    34 * 65536 + 197 * 256 + 94,     // #22c55e
    245 * 65536 + 158 * 256 + 11,    // #f59e0b
    168 * 65536 + 85 * 256 + 247,    // #a855f7
    236 * 65536 + 72 * 256 + 153     // #ec4899
]
fest EXPORT_ZOOM = 16     // ein Zellen-Pixel wird im PNG zu 16×16 Bildpunkten

nimm aktiv = 1
nimm werkzeug = "stift"
```

```
nimm zieh_x = -1
nimm zieh_y = -1
nimm fenster = 0          // wird unten gesetzt; exportiere braucht es für den Dialog

nimm pixel = liste.gefüllt(LEIN_B * LEIN_H, 0)
nimm füll_q = liste.gefüllt(LEIN_B * LEIN_H, 0)
nimm bild_gui = gui.leinwand(LEIN_B * ZELLE, LEIN_H * ZELLE)
nimm palette_gui = gui.leinwand(LEIN_B * ZELLE, FELD)

funktion kleiner(a, b) {
    wenn a < b { gib a }
    gib b
}
funktion größer(a, b) {
    wenn a > b { gib a }
    gib b
}

funktion zellfarbe(wert) {
    gib PALETTE[wert]
}

funktion zeichne_zelle(x, y) {
    nimm f = zellfarbe(pixel[y * LEIN_B + x])
    gui.rechteck(bild_gui, x * ZELLE + 1, y * ZELLE + 1, ZELLE - 1, ZELLE - 1, f)
}

funktion zeichne() {
    gui.rechteck(bild_gui, 0, 0, LEIN_B * ZELLE, LEIN_H * ZELLE, RAND)
    für y = 0 bis LEIN_H - 1 {
        für x = 0 bis LEIN_B - 1 {
            zeichne_zelle(x, y)
        }
    }
}

funktion zeichne_palette() {
    gui.rechteck(palette_gui, 0, 0, LEIN_B * ZELLE, FELD, "#e2e8f0")
    für p = 0 bis länge(PALETTE) - 2 {
        nimm idx = p + 1
        nimm fx = p * FELD
        wenn idx == aktiv {
            gui.rechteck(palette_gui, fx, 0, FELD, FELD, "#0c4a6e")
            gui.rechteck(palette_gui, fx + 3, 3, FELD - 6, FELD - 6, PALETTE[idx])
        } sonst {
            gui.rechteck(palette_gui, fx + 1, 1, FELD - 2, FELD - 2, PALETTE[idx])
        }
    }
}

funktion benutze(x, y) {
    wenn x >= 0 und x < LEIN_B und y >= 0 und y < LEIN_H {
        wenn werkzeug == "radierer" {
            pixel[y * LEIN_B + x] = 0
        } sonst {
            pixel[y * LEIN_B + x] = aktiv
        }
        zeichne_zelle(x, y)
    }
}

funktion rechteck_zellen(x0, y0, x1, y1, nur_vorschau) {
    nimm lx = kleiner(x0, x1)
```

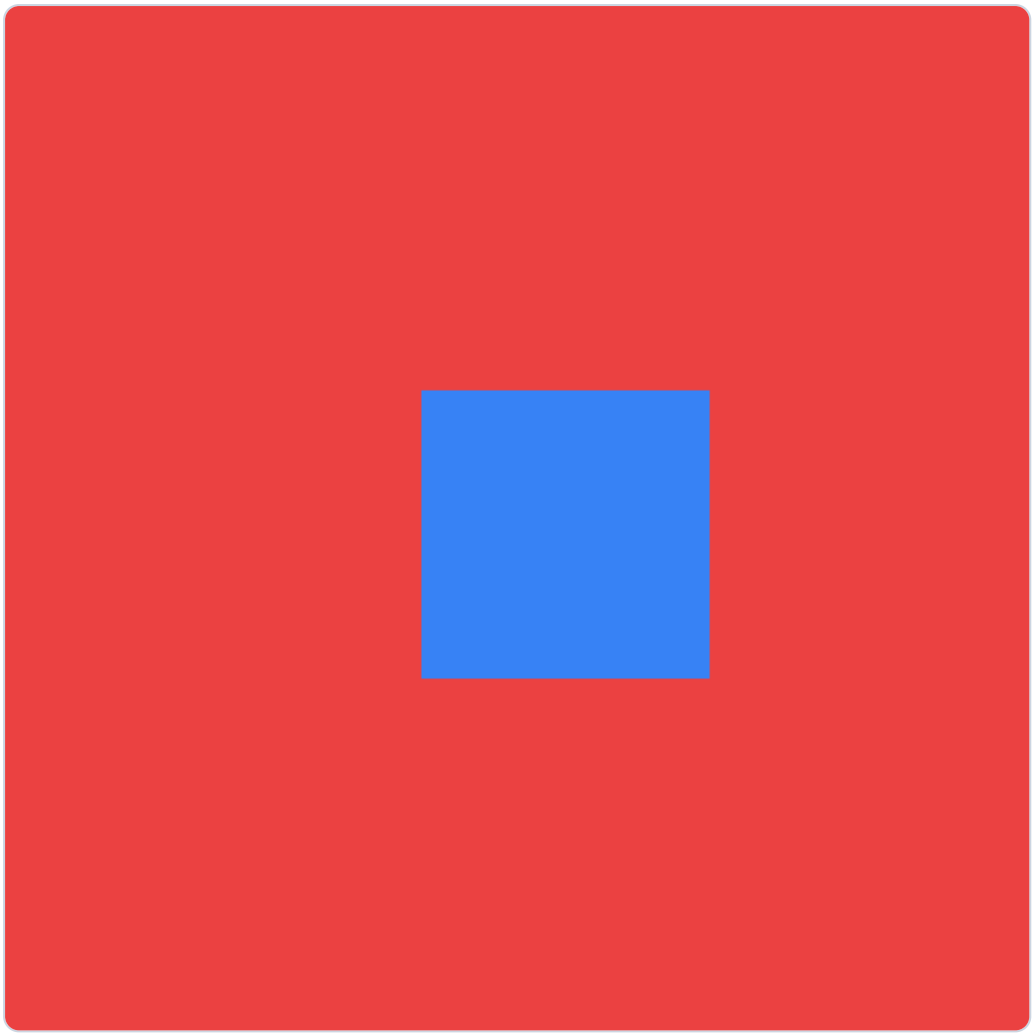
```
    nimm rx = größer(x0, x1)
    nimm oy = kleiner(y0, y1)
    nimm uy = größer(y0, y1)
    für y = oy bis uy {
        für x = lx bis rx {
            wenn x >= 0 und x < LEIN_B und y >= 0 und y < LEIN_H {
                wenn nur_vorschau {
                    gui.rechteck(bild_gui, x * ZELLE + 1, y * ZELLE + 1, ZELLE - 1, ZELLE - 1, PALETTE[aktiv])
                } sonst {
                    pixel[y * LEIN_B + x] = aktiv
                }
            }
        }
    }
}

funktion fülle(zx, zy) {
    nimm start = zy * LEIN_B + zx
    nimm ziel = pixel[start]
    wenn ziel != aktiv {
        nimm kopf = 0
        nimm schwanz = 0
        füll_q[schwanz] = start
        schwanz = schwanz + 1
        pixel[start] = aktiv
        solange kopf < schwanz {
            nimm nr = füll_q[kopf]
            kopf = kopf + 1
            nimm x = nr % LEIN_B
            nimm y = nr div LEIN_B
            wenn x + 1 < LEIN_B {
                nimm m = nr + 1
                wenn pixel[m] == ziel {
                    pixel[m] = aktiv
                    füll_q[schwanz] = m
                    schwanz = schwanz + 1
                }
            }
            wenn x - 1 >= 0 {
                nimm m = nr - 1
                wenn pixel[m] == ziel {
                    pixel[m] = aktiv
                    füll_q[schwanz] = m
                    schwanz = schwanz + 1
                }
            }
            wenn y + 1 < LEIN_H {
                nimm m = nr + LEIN_B
                wenn pixel[m] == ziel {
                    pixel[m] = aktiv
                    füll_q[schwanz] = m
                    schwanz = schwanz + 1
                }
            }
            wenn y - 1 >= 0 {
                nimm m = nr - LEIN_B
                wenn pixel[m] == ziel {
                    pixel[m] = aktiv
                    füll_q[schwanz] = m
                    schwanz = schwanz + 1
                }
            }
        }
```

```
        zeichne()
    }
}

// Die Leinwand als PNG schreiben: jede Zelle wird zu einem EXPORT_ZOOM-großen Block.
funktion exportiere() {
    nimm bw = LEIN_B * EXPORT_ZOOM
    nimm bh = LEIN_H * EXPORT_ZOOM
    nimm handle = bild.neu(bw, bh)
    nimm puffer = liste.gefüllt(bw * bh, 0)
    für oy = 0 bis bh - 1 {
        nimm zy = oy div EXPORT_ZOOM
        für ox = 0 bis bw - 1 {
            nimm zx = ox div EXPORT_ZOOM
            puffer[oy * bw + ox] = PALETTE_RGB[pixel[zy * LEIN_B + zx]]
        }
    }
    bild.setze_puffer(handle, puffer)
    nimm ziel = pfad.verbunden(umgebung.benutzer_ordner(), "kistepix.png")
    bild.schreibe_png(handle, ziel)
    gui.dialog_info(fenster, "Gespeichert", "Dein Bild wurde gespeichert als:\n{ziel}")
}

zeichne()
zeichne_palette()

gui.bei_maus_klick(bild_gui, funktion(b) {
    nimm x = gui.maus_x(b) div ZELLE
    nimm y = gui.maus_y(b) div ZELLE
    wenn werkzeug == "füller" {
        wenn x >= 0 und x < LEIN_B und y >= 0 und y < LEIN_H {
            fülle(x, y)
        }
    } sonstwenn werkzeug != "rechteck" {
        benutze(x, y)
    }
})

gui.bei_maus_ziehen(bild_gui, funktion(b) {
    nimm x = gui.maus_x(b) div ZELLE
    nimm y = gui.maus_y(b) div ZELLE
    wenn werkzeug == "rechteck" {
        wenn zieh_x == -1 {
            zieh_x = x
            zieh_y = y
        }
        zeichne()
        rechteck_zellen(zieh_x, zieh_y, x, y, wahr)
    } sonstwenn werkzeug == "stift" oder werkzeug == "radierer" {
        benutze(x, y)
    }
})

gui.bei_maus_los(bild_gui, funktion(b) {
    wenn werkzeug == "rechteck" und zieh_x != -1 {
        nimm x = gui.maus_x(b) div ZELLE
        nimm y = gui.maus_y(b) div ZELLE
        rechteck_zellen(zieh_x, zieh_y, x, y, falsch)
        zieh_x = -1
        zeichne()
    }
})
```

```
gui.bei_maus_klick(palette_gui, funktion(b) {
    nimm p = gui.maus_x(b) div FELD
    wenn p >= 0 und p < länge(PALETTE) - 1 {
        aktiv = p + 1
        zeichne_palette()
    }
})

nimm knopf_stift = gui.knopf_neu("Stift")
nimm knopf_radierer = gui.knopf_neu("Radierer")
nimm knopf_rechteck = gui.knopf_neu("Rechteck")
nimm knopf_füller = gui.knopf_neu("Füller")
nimm knopf_export = gui.knopf_neu("Export")
gui.bei_klick(knopf_stift, funktion(k) { werkzeug = "stift" })
gui.bei_klick(knopf_radierer, funktion(k) { werkzeug = "radierer" })
gui.bei_klick(knopf_rechteck, funktion(k) { werkzeug = "rechteck" })
gui.bei_klick(knopf_füller, funktion(k) { werkzeug = "füller" })
gui.bei_klick(knopf_export, funktion(k) { exportiere() })
nimm leiste = gui.horizontal([knopf_stift, knopf_radierer, knopf_rechteck, knopf_füller, knopf_export])

nimm inhalt = gui.vertikal([leiste, bild_gui, palette_gui])
fenster = gui.fenster_öffne("KistePix", LEIN_B * ZELLE, LEIN_H * ZELLE + FELD + 48)
gui.zeige(fenster, inhalt)
gui.starte()
```



Ein kleines Werk in KistePix — bereit zum Export mit dem neuen Knopf „Export".



Die exportierte Datei `kistepix.png` — dasselbe Bild, aber als echte Bilddatei und **ohne** Gitterlinien. Das Gitter war nur eine Hilfe am Bildschirm.

Am Bildschirm mit Gitter, in der Datei ohne

Vergleich die zwei Bilder: Auf der Leinwand helfen dir die Gitterlinien, Zellen zu treffen — in der PNG-Datei sind sie weg, dort zählen nur die echten Pixel. Genau richtig: Das Gitter ist ein Werkzeug fürs Malen, kein Teil deines Bildes.

Hält auch bei großen Bildern

Derselbe Weg — Puffer bauen, `bild.setze_puffer`, `bild.schreibe_png` — wurde für dieses Buch an einem **1500×1000-Bild (1,5 Millionen Bildpunkte)** geprüft: vollständig und in Sekundenbruchteilen geschrieben. Dein Export ist also nicht nur für kleine Bilder gebaut.

Teste dich

1. Warum braucht der Export die Farben als Zahl (`PALETTE_RGB`) und nicht als Hex-Text?
2. Wie groß wird die PNG-Datei bei `LEIN_B = 32` und `EXPORT_ZOOM = 16`?
3. Warum hat die exportierte Datei keine Gitterlinien?

Aufgabe ★

Male ein kleines Bild und exportiere es. Öffne dann `kistepix.png` aus deinem Benutzer-Ordner in einem beliebigen Bildbetrachter — es ist ein ganz normales Bild. Ändere danach `EXPORT_ZOOM` auf `8` und exportiere erneut: Wie groß ist die Datei jetzt (in Bildpunkten)?

Lösung

Teste dich: (1) Weil `bild.setze_puffer` Zahlen erwartet — die drei Farbanteile in eine Ganzzahl gepackt. (2) `32 * 16 = 512`, also 512×512 Bildpunkte. (3) Weil das Gitter nur beim Zeichnen auf die Leinwand gemalt wird, nicht in die `pixel`-Daten gehört — und exportiert werden nur die Pixel-Daten.

Aufgabe: Bei `EXPORT_ZOOM = 8` wird jede Zelle zu 8×8 Bildpunkten, also `32 * 8 = 256` — eine 256×256-Datei. Der Zoom bestimmt nur die Größe der Datei; das gemalte Bild bleibt dasselbe.

Ein Gedanke für später

Der Export schreibt immer dieselbe Datei `kistepix.png` und überschreibt die vorige. In einem „erwachsenen" Editor würdest du beim Speichern *fragen*, wohin und unter welchem Namen — dafür gibt es `gui.dialog_speichern`. Das ist ein schöner erster Ausbau, wenn du selbst weitermachst (Stufe 8).

Stufe 8 — Fertig: jetzt bist du dran

Geschafft — du hast einen Pixel-Editor gebaut

Von der leeren Leinwand bis zur fertigen Bilddatei: KistePix ist ein **vollständiges, funktionierendes Programm**, das du Zeile für Zeile selbst verstanden und aufgebaut hast. Das ist eine echte Leistung — die allermeisten Menschen, die täglich Programme benutzen, könnten so etwas nicht.

Was jetzt in deinem KistePix steckt

Sieben Stufen, ein Editor:

- Eine **Leinwand** als flaches Pixel-Raster (S1).
- Ein **Stift**, der auf Klick und Zug malt (S2, S4).
- Eine **Farbpalette** mit aktiver Farbe (S3).
- Ein **Radierer** (S4).
- Ein **Rechteck** mit Gummiband-Vorschau (S5).
- Ein **Füll-Eimer**, richtig und schnell gebaut (S6).
- Ein **PNG-Export** in eine echte Bilddatei (S7).

Und darunter, unsichtbar, das ganze Handwerk aus Teil I: Variablen und Konstanten, Zahlen und `div`, Entscheidungen und Grenzprüfungen, Schleifen über das Raster, Funktionen als Werkzeuge, Listen als

Datenmodell, Fenster und Maus. Nichts davon war Theorie — alles ist in KistePix zu etwas Sichtbarem geworden.

Jetzt bist du dran

Wir haben KistePix bewusst **klein und klar** gehalten. Das war kein Mangel, sondern der Plan: Ein Programm, das man *ganz* versteht, ist der beste Startpunkt zum Weiterbauen. Und genau dazu laden wir dich jetzt ein. Hier sind Ideen — von „an einem Nachmittag machbar" bis „richtig groß":

Kleine Schritte (du kennst schon alles dafür)

- **Leeren-Knopf** — die ganze Leinwand auf einmal löschen (war die Aufgabe in S4).
- **Rechteck-Umriss** statt gefüllt — nur den Rand malen (Aufgabe in S5).
- **Speichern-Dialog** — beim Export fragen, wohin (Hinweis in S7, `gui.dialog_speichern`).
- **Mehr Farben** — die Palette einfach verlängern (Aufgabe in S3).
- **Ein Linien-Werkzeug** — wie das Rechteck, aber nur eine gerade Linie zwischen zwei Punkten.

Mittlere Projekte (ein bisschen Neues nachschlagen)

- **Rückgängig** (Strg + Z) — sich vor jeder Änderung den alten Zustand merken.
- **Pipette** — eine Farbe von der Leinwand aufnehmen und aktiv machen.
- **Ein Bild laden** — mit `bild.lade` ein PNG öffnen und ins Raster übernehmen.
- **Kreis** und andere Formen — mit etwas Geometrie im selben Gummiband-Muster.
- **Zoom** — die Leinwand vergrößert anzeigen, um an einzelnen Pixeln zu feilen.

Große Vorhaben (dein eigenes „Deluxe")

- **Ebenen** — mehrere Raster übereinander, die zusammen ein Bild ergeben.
- **Animation** — viele Bilder nacheinander, als bewegtes GIF exportiert.
- **Auswählen und Verschieben** — einen Bereich markieren und umsetzen.
- **Riesige Leinwände** — hier lohnt sich die Technik aus dem Ausblick: die Leinwand als *ein* Bild im Speicher zeichnen (`bild.setze_puffer` + `gui.leinwand_bild`) statt Tausender Einzel-Rechtecke.

Der wichtigste Rat zum Weiterbauen

Geh vor wie in diesem Buch: **eine Fähigkeit nach der anderen**. Nimm dein fertiges KistePix, füg *ein* kleines Ding hinzu, bring es zum Laufen, freu dich — und dann das nächste. So ist KistePix entstanden, und so entsteht auch die große Version. Niemand baut ein ganzes Programm auf einmal; alle bauen es Stück für Stück.

Wie du weiter lernst

Programmieren lernt man, indem man programmiert — und indem man neugierig bleibt. Ein paar Wege:

- **Probier Dinge aus.** Ändere Werte, brich absichtlich etwas, schau, was passiert, und repariere es. Jeder Fehler, den du selbst findest, bringt dich weiter.
- **Lies deinen eigenen Code von gestern.** Verstehst du ihn noch? Wenn nicht, schreib bessere Namen und Kommentare — auch das ist Programmieren.
- **Bau etwas ganz anderes.** Ein kleines Spiel, einen Rechner, ein Quiz. Dieselben Bausteine aus Teil I tragen dich überall hin — nicht nur in KistePix.
- **Das Online-Lernbuch** auf **kiste-lang.org** geht bei einigen Themen noch tiefer und wird laufend erweitert.

Zum Schluss

Du bist mit einem leeren Editor gestartet und hast Kiste von Grund auf gelernt — und jetzt steht ein echtes Programm auf deinem Rechner, das du selbst gebaut hast und in- und auswendig kennst. Das nimmt dir niemand mehr.

Was du hier gelernt hast, ist kein Kiste-Wissen allein: Variablen, Schleifen, Funktionen, Entscheidungen, Datenmodelle, Ereignisse — das sind die Ideen *jeder* Programmiersprache. Du hast programmieren gelernt. Kiste war nur die freundliche Tür dorthin.

Viel Freude beim Weiterbauen — und beim Malen mit KistePix.

A Glossar

Hier findest du die wichtigsten Fachbegriffe aus diesem Buch — kurz und in einfachen Worten erklärt, alphabetisch geordnet. Neben den Begriffen, die dir beim Bauen von KistePix begegnet sind, sind auch allgemeine Programmier-Wörter dabei, die dir in jeder Programmiersprache wieder begegnen.

> **Schlüsselwörter und Funktionen im Detail**
> Dieses Glossar erklärt *Begriffe*. Die genauen Schreibweisen aller Schlüsselwörter, eingebauten Funktionen und Modul-Funktionen — mit Signatur und Kurzbeispiel — findest du in **Anhang B (Sprachreferenz)**.

A

Algorithmus — eine genaue Schritt-für-Schritt-Anleitung, die ein Problem löst. KistePix nutzt zum Beispiel einen Füll-Algorithmus, um eine Fläche auf einen Klick zu füllen.

Alphakanal — siehe **Deckkraft**.

Anweisung — ein einzelner Befehl, den Kiste ausführt; meist eine Zeile Programm.

AOT-Kompilierung (englisch *ahead of time*, „im Voraus") — Kiste übersetzt dein ganzes Programm schon *vor* dem Start in Maschinencode. Darum startet die fertige `.exe` sofort und läuft schnell.

Argument — der konkrete Wert, den du beim Aufruf an einen Parameter übergibst: bei `wurzel(81)` ist `81` das Argument.

Aufruf — eine Funktion benutzen, indem man ihren Namen mit Klammern schreibt: `mathe.wurzel(81)`.

Ausdruck — ein Stück Code, das einen Wert ergibt, etwa `3 + 4` oder `länge(name)`.

Auswahl (englisch *selection*) — ein aufgezogenes Rechteck, dessen Inhalt du verschieben kannst; der gestrichelte Rahmen darum heißt **Marquee**.

B

Bedingung — ein Wahrheits-Ausdruck, der entscheidet, ob ein `wenn`-Block läuft, z. B. `alter >= 18`.

Bit — die kleinste Informationseinheit: eine 0 oder eine 1. Acht Bit ergeben ein Byte.

Binärsystem — Zählen nur mit 0 und 1; so rechnet der Computer im Innersten.

Bresenham-Algorithmus — ein klassisches Verfahren, um mit reiner Ganzzahl-Mathematik eine gerade Linie (und verwandt: einen Kreis) sauber auf ein Pixelgitter zu zeichnen.

Bug — ein Fehler im Programm, der zu falschem Verhalten führt. Ihn zu finden und zu beheben heißt *Debugging*.

Byte — acht Bit zusammen; kann 256 verschiedene Werte (0–255) darstellen. Eine Farbkomponente (Rot, Grün oder Blau) ist genau ein Byte.

C

Callback (deutsch *Rückruf*) — eine Funktion, die du hinterlegst, damit Kiste sie später „zurückruft", wenn ein Ereignis passiert — etwa ein Mausklick.

Closure (Funktions-Abschluss) — eine Funktion, die sich Werte aus ihrer Umgebung merkt und später noch benutzen kann.

Compiler — siehe **Kompilieren**.

Container — ein GUI-Baustein, der andere Bausteine anordnet, z. B. untereinander (`vertikal`), nebeneinander (`horizontal`) oder im `raster`.

D

Datenmodell — die Art, wie ein Programm seine Daten organisiert. In KistePix ist die Leinwand ein flaches Raster: alle Pixel in *einer* Liste, Zeile für Zeile.

Deckkraft (Alpha) — wie undurchsichtig eine Ebene oder Farbe ist: 0 % = völlig durchsichtig, 100 % = voll deckend.

Dezimal (Typ `dezimal`) — Kistes exakter Nachkomma-Zahltyp für kaufmännisch genaues Rechnen, ohne Rundungsfehler.

div (Ganzzahl-Division) — teilt und wirft den Rest weg: `7 div 2` ergibt `3`. Damit rechnet man aus einer Pixelposition die Zellenspalte aus.

Drehen — ein Bild um 90° im Uhrzeigersinn kippen.

E

Ebene (englisch *layer*) — eine durchsichtige Malschicht. Mehrere Ebenen übereinander ergeben zusammen das fertige Bild.

Eingebaute Funktion — eine Funktion, die Kiste von sich aus kennt, ohne `nutze`, etwa `sag`, `länge` oder `typ`.

Emoji — siehe **Graphem-Cluster**.

Ereignis — etwas, das während des Programmlaufs passiert: ein Klick, ein Tastendruck, eine Mausrad-Drehung. Darauf reagierst du mit einem Callback.

Escape-Sequenz — ein Sonderzeichen im Text, mit Rückstrich geschrieben: `\n` (neue Zeile), `\t` (Tabulator), `\"` (Anführungszeichen).

F

Farbkanal — einer der drei Farb-Anteile Rot, Grün und Blau einer RGB-Farbe.

Feld — eine Variable, die zu einem Objekt gehört, z. B. `spieler.leben`.

Flood-Fill (Füllen) — ein Algorithmus, der eine zusammenhängende Fläche gleicher Farbe ab einem Startpunkt umfärbt.

Frame (Einzelbild) — ein einzelnes Bild einer Animation. Viele Frames schnell nacheinander gezeigt ergeben Bewegung.

Funktion — ein benannter Block Code, den du mit einem Aufruf immer wieder starten kannst. Sie nimmt Parameter und gibt oft einen Wert zurück.

G

Ganz (Typ `ganz`) — Kistes Typ für ganze Zahlen ohne Komma: …, −2, −1, 0, 1, 2, …

Game-Loop — der Takt, in dem ein Spiel viele Male pro Sekunde neu rechnet und zeichnet; in Kiste über `gui.bei_zeit`.

Geld (Typ `geld`) — ein eigener Typ für Beträge mit Währung, der die typischen Rundungsfehler beim Rechnen vermeidet.

GIF (animiert) — ein Bildformat, das mehrere Frames als kleine Animation in einer Datei speichert.

Graphem-Cluster — das, was ein Mensch als *ein* Zeichen sieht (ein Emoji, ein Buchstabe mit Akzent), auch wenn es innen aus mehreren Bausteinen besteht. Kiste zählt und schneidet Text nach Graphemen — darum bleibt ein Emoji immer heil.

Grenzprüfung — testen, ob eine Position noch im gültigen Bereich liegt (etwa ein Pixel im Raster), bevor man zugreift.

GUI (grafische Oberfläche, englisch *graphical user interface*) — Fenster, Knöpfe und Grafik statt reiner Textausgabe; in Kiste über `nutze gui`.

Gummiband — die Technik, beim Ziehen eine Vorschau (Linie, Rechteck, Kreis) mitwachsen zu lassen und sie erst beim Loslassen festzuschreiben.

H

Handler — siehe **Callback**.

Hexadezimal (Hex) — ein Zahlensystem zur Basis 16 mit den Ziffern 0–9 und a–f. Farben schreibt man als `#rrggbb`; je zwei Hex-Ziffern sind ein Byte.

Höhere Funktion (englisch *higher-order*) — eine Funktion, die eine andere Funktion als Wert nimmt oder zurückgibt, etwa `liste.abgebildet`.

I

Index — die Position eines Elements in einer Liste oder eines Zeichens im Text. Kiste zählt ab `0`.

Instanz — siehe **Objekt**.

Interpolation — Werte direkt in einen Text einsetzen mit geschweiften Klammern: `"Hallo {name}"`.

Iteration — ein Durchlauf einer Schleife; „iterieren" heißt, Element für Element durchzugehen.

K

Karte (Typ `karte`, englisch *dictionary*) — eine Sammlung, die Werte unter einem Schlüssel nachschlägt, z. B. `preise["Apfel"]`.

Klasse — ein Bauplan für Objekte. Sie legt fest, welche Felder und Methoden die Objekte haben.

Kommentar — Text im Programm, den Kiste ignoriert; er erklärt den Code für Menschen (mit `//`).

Komma (Typ `komma`) — Kistes Fließkomma-Typ für wissenschaftliches Rechnen: schnell, aber nicht exakt. Für Geld nimm `dezimal` oder `geld`.

Kompilieren (Übersetzen) — den Quelltext in Maschinencode umwandeln, den der Prozessor direkt ausführt. Kiste tut das vor dem Start (siehe **AOT-Kompilierung**).

Komposition (Zusammensetzen) — alle sichtbaren Ebenen von unten nach oben zu einem einzigen Bild verrechnen.

Konstante — ein benannter Wert, der sich nie ändert; per Konvention GROSS geschrieben, z. B. `RASTER`.

Konstruktor — die Methode, die ein frisches Objekt mit Startwerten einrichtet; läuft bei `neu`.

Koordinaten — ein Zahlenpaar `(x, y)`, das eine Position angibt. Auf der Leinwand ist `(0, 0)` oben links.

L

Layer — siehe **Ebene**.

Leinwand (englisch *canvas*) — die Zeichenfläche der GUI, auf der KistePix seine Pixel malt.

Liste (Typ `liste`) — eine geordnete Sammlung vieler Werte hinter einem Namen.

Logik-Operator — `und`, `oder`, `nicht` — verknüpft Wahrheitswerte.

M

Marquee — der gestrichelte, laufende Rahmen um eine **Auswahl**.

Maschinencode — die Befehle, die der Prozessor unmittelbar versteht; das Ergebnis des Kompilierens.

Methode — eine Funktion, die zu einer Klasse gehört und auf einem Objekt aufgerufen wird, z. B. `konto.einzahlen(50)`.

Mittelpunkt-Kreis-Algorithmus — zeichnet einen Pixelkreis mit reiner Ganzzahl-Mathematik, indem er nur einen Achtel-Bogen berechnet und ihn 8-fach spiegelt.

Modul — ein Werkzeugkasten von Funktionen zu einem Thema, geladen mit `nutze`, etwa `mathe`, `text` oder `gui`.

N

neu — das Schlüsselwort, das aus einer Klasse ein konkretes Objekt erzeugt.

nichts (Typ `nichts`) — der „leere" Wert; steht für „hier ist kein Wert".

O

Objekt (Instanz) — ein konkretes Ding, nach einer Klasse gebaut, mit eigenen Feldwerten.

Onion-Skinning — siehe **Zwiebelschale**.

Operator — ein Rechen- oder Vergleichszeichen wie `+`, `-`, `==` oder `und`.

P

Palette — die Sammlung wählbarer Farben. Jede Bildzelle speichert nur einen Palettenindex, nicht die Farbe selbst.

Palettenindex — die Nummer einer Farbe in der Palette. Im Bild steht pro Pixel bloß diese Nummer — das spart Platz und macht Umfärben leicht.

Parameter — der Platzhalter in einer Funktionsdefinition, der beim Aufruf einen Wert (das Argument) bekommt.

Pipette — ein Werkzeug, das die Farbe eines vorhandenen Pixels aufnimmt und zur aktiven Malfarbe macht.

Pixel — der kleinste Bildpunkt. In KistePix ist ein Pixel ein groß gezoomtes Rechteck im Raster.

PNG — ein verlustfreies Bildformat. KistePix exportiert Bilder als PNG.

Q

Quelltext (Quellcode) — der Programmtext, den du schreibst: die `.ki`-Datei.

R

Radierer — ein Werkzeug, das Pixel wieder auf „leer" zurücksetzt.

Raster (Gitter) — die gleichmäßige Einteilung der Leinwand in Zeilen und Spalten.

Rekursion — eine Funktion, die sich selbst aufruft, um ein Problem in kleinere gleiche Teile zu zerlegen.

RGB — das Farbmodell aus Rot, Grün und Blau (je 0–255). So mischt der Bildschirm jede Farbe.

Rückgabewert — der Wert, den eine Funktion mit `gib` an den Aufrufer zurückliefert.

Rückruf — siehe **Callback**.

S

Schleife — ein Block, der sich wiederholt: `solange`, `für` oder `wiederhole`.

Schlüssel — der Name, unter dem eine Karte einen Wert ablegt und wiederfindet.

Schlüsselwort — ein festes Wort der Sprache mit besonderer Bedeutung, z. B. `wenn`, `funktion`, `neu`. Die volle Liste steht in Anhang B.

Serialisierung — Daten in eine speicherbare Text- oder Byte-Form umwandeln (und später zurück), etwa zum Speichern und Laden.

Spiegeln — ein Bild waagerecht oder senkrecht umklappen.

Sprite — eine kleine, bewegliche Grafik oder Figur in einem Spiel.

Sprite-Sheet — ein Bild, das alle Frames nebeneinander enthält, damit Spiel-Engines wie Unity oder Godot sie in Einzelbilder zerschneiden können.

Stapel (englisch *stack*) — eine Sammlung nach dem Prinzip „zuletzt hinein, zuerst heraus". Rückgängig und Wiederholen arbeiten mit zwei Stapeln.

Symmetrie (8-fache) — beim Kreis genügt es, einen Achtel-Bogen zu berechnen und ihn achtmal zu spiegeln — der ganze Ring entsteht daraus.

Syntax — die Grammatikregeln, nach denen Kiste-Code geschrieben sein muss, damit er verständlich ist.

T

Tooltip — ein kleiner Text, der beim Verweilen der Maus über einem Knopf erklärt, was der Knopf tut.

Treffer-Test — prüfen, ob ein Punkt (etwa ein Mausklick) innerhalb eines Rechtecks liegt.

Typ — die Art eines Werts (`ganz`, `text`, `wahrheit`, `liste` …). Der Typ bestimmt, was man mit dem Wert tun darf.

U

Undo / Redo (Rückgängig / Wiederholen) — einen Schritt zurücknehmen bzw. wieder vorholen, umgesetzt mit zwei Stapeln.

Unicode — der weltweite Zeichensatz, der alle Schriften, Sonderzeichen und Emojis umfasst.

V

Variable — ein benannter Behälter für einen Wert, den man lesen und ändern kann.

Vererbung — eine Klasse übernimmt mit `erbt` die Felder und Methoden einer anderen und erweitert sie.

Vergleichsoperator — `==`, `!=`, `<`, `>`, `<=`, `>=`.

View-Transform — die Umrechnung zwischen den Koordinaten des Bildes und dem gezoomten, verschobenen Ausschnitt auf dem Bildschirm.

Viewport — der gerade sichtbare Ausschnitt der Leinwand beim Zoomen und Schwenken.

W

Wahrheit (Typ `wahrheit`) — ein Wert, der nur `wahr` oder `falsch` sein kann.

Werkzeug — der aktive Modus des Editors: Stift, Radierer, Füller, Linie, Rechteck, Kreis, Auswahl, Pipette, Hand.

Widget — ein einzelnes GUI-Bedienelement wie ein Knopf, ein Etikett oder ein Eingabefeld.

Z

Zustand — die Gesamtheit der Werte, die ein Programm gerade „im Kopf hat": in KistePix etwa die aktive Farbe und das aktive Werkzeug.

Zwiebelschale (englisch *onion-skinning*) — der vorige Frame blass durchscheinend im Hintergrund, damit man den nächsten passend darüber zeichnen kann.

B Sprachreferenz

Dieser Anhang ist zum **Nachschlagen** gedacht — nicht zum Durchlesen. Hier stehen kompakt alle Schlüsselwörter, Operatoren, eingebauten Funktionen und Typen von Kiste, dann die Standard-Bibliothek Modul für Modul und zum Schluss der große `gui`-Werkzeugkasten, mit dem du KistePix gebaut hast. Kurze, echte Beispiele zeigen die wichtigsten Stellen im Einsatz.

Immer die neueste Fassung

Kiste wächst weiter. Die *stets aktuelle* und noch ausführlichere Referenz — mit jeder neuen Funktion — findest du online im KISTE-Lernbuch unter **kiste-lang.org/de/buch/**. Dieser Anhang bildet den Stand zur gedruckten Kiste-Version (siehe Impressum) ab.

B.1 Schlüsselwörter

Schlüsselwörter sind feste Wörter der Sprache. Du kannst sie **nicht** als eigene Namen für Variablen, Funktionen oder Felder verwenden (z. B. sind `teile`, `neu` oder `gib` reserviert).

Schlüsselwort	Bedeutung
`nimm`	Eine veränderliche Variable deklarieren: `nimm x = 5`.
`fest`	Eine Konstante deklarieren (unveränderlich): `fest PI = 3.14159`.
`wahr, falsch`	Die beiden Wahrheitswerte.
`nichts`	Der „leere" Wert — steht für „kein Wert".
`wenn, sonstwenn, sonst`	Bedingungen — verschiedene Fälle abarbeiten.
`prüfe, fall, standard`	Mehrfach-Auswahl (switch): gegen viele Werte prüfen.
`solange`	Schleife, solange eine Bedingung wahr ist (while).
`für, bis, schritt`	Zähl-Schleife: `für i = 1 bis 10 schritt 2` (`bis` ist inklusiv).
`wiederhole, in`	Über eine Sammlung gehen: `wiederhole x in liste`.
`springe`	Eine Schleife sofort abbrechen (break).
`weiter`	Zum nächsten Schleifendurchlauf springen (continue).
`funktion`	Eine Funktion (oder Methode) definieren.
`gib`	Einen Wert aus einer Funktion zurückgeben und sie sofort beenden.
`nutze`	Ein Modul laden: `nutze gui`.
`teile`	Dinge aus einer eigenen Modul-Datei nach außen freigeben (exportieren).
`klasse`	Einen Bauplan für Objekte definieren.
`neu`	Ein Objekt aus einer Klasse erzeugen: `neu Person("Anna", 30)`.
`dies`	Das aktuelle Objekt innerhalb einer Methode (in anderen Sprachen `this`).
`erbt`	Eine Klasse von einer anderen erben lassen: `klasse Hund erbt Tier`.
`super`	Die gleichnamige Methode der Eltern-Klasse aufrufen.
`öffentlich, privat`	Sichtbarkeit von Klassen-Mitgliedern steuern.
`und, oder, nicht`	Logik-Operatoren.
`div`	Ganzzahl-Division (teilt und wirft den Rest weg): `7 div 2` = 3.
`versuche, fange`	Einen Fehler abfangen, statt das Programm abstürzen zu lassen.
`wirf`	Selbst einen Fehler auslösen.

B.2 Operatoren

Gruppe	Operatoren
Arithmetik	`+ - * / %` (Rest) `^` (Potenz) `div` (Ganzzahl-Division)
Vergleich	`== != < <= > >=`
Zuweisung	`= += -= *= /= ??=` (Coalescing-Zuweisung)
Logik	`und oder nicht`
Null-sicher	`?.` (sicherer Member-Zugriff) `??` (Standardwert bei `nichts`)
Zugriff	`.` (Feld/Methode) `[]` (Index) `()` (Aufruf)

B.3 Eingebaute Funktionen

Diese Funktionen kennt Kiste von sich aus — ohne `nutze`.

Ein- und Ausgabe

Signatur	Beschreibung
`sag(...)`	Gibt seine Argumente auf der Konsole aus.
`frage(text)`	Zeigt `text` und liest eine Zeile Nutzereingabe ein.

Reflexion — Werte untersuchen

Signatur	Beschreibung
`länge(x)`	Länge: Grapheme bei Text, Anzahl Elemente bei Liste/Karte/Bereich.
`bytes_länge(text)`	Anzahl der UTF-8-Bytes eines Texts.
`zeichen_länge(text)`	Anzahl der Unicode-Codepoints eines Texts.
`typ(x)`	Der Typ eines Werts (vergleichbar mit `Ganz`, `Text` …).
`doku(x)`	Der `///`-Doku-Kommentar einer Funktion/Klasse/eines Moduls.

Umwandlung zwischen Typen

Signatur	Beschreibung
`als_ganz(x)`	Zu `ganz` (schneidet Richtung Null ab).
`als_komma(x)`	Zu `komma` (kann Genauigkeit verlieren).
`als_dezimal(x)`	Zu `dezimal` (rundet `komma` auf 15 Stellen).
`als_dezimal_exakt(k)`	Zu `dezimal`, bit-genaue `komma`-Darstellung.
`als_text(x)`	Zu einem Anzeige-Text.
`als_wahrheit(x)`	Strikt zu `wahrheit` — nur `wahr`/`falsch` erlaubt.

Runden

Signatur	Beschreibung
`runden(x)`	Kaufmännisch runden (banker's rounding).
`runden(x, stellen)`	Auf N Nachkommastellen runden.
`aufrunden(x)`	Immer nach oben (Richtung +∞).
`abrunden(x)`	Immer nach unten (Richtung −∞).
`abschneiden(x)`	Nachkommastellen weglassen (Richtung Null).

Zahlen vergleichen · Sammlungen

Signatur	Beschreibung
`fast_gleich(a, b, t)`	`|a-b| <= t` — die richtige Wahl beim Vergleich von `komma`-Zahlen.
`kopiere(x)`	Tiefe Kopie einer Liste/Karte.
`füge_hinzu(liste, x)`	Ein Element an eine Liste anhängen.
`entferne_letztes(l)`	Letztes Element entfernen und zurückgeben.
`schluessel(karte)`	Liste aller Schlüssel einer Karte.

B.4 Die eingebauten Typen

Typ	Wofür
`ganz`	Ganze Zahlen ohne Komma: …, −1, 0, 1, 2, …
`komma`	Fließkomma-Zahlen für Wissenschaft (schnell, nicht exakt).
`dezimal`	Exakte Nachkomma-Zahlen (kaufmännisch, ohne Rundungsfehler).
`text`	Zeichenketten; Kiste zählt nach Graphemen (Emojis bleiben heil).
`wahrheit`	`wahr` oder `falsch`.
`liste`	Geordnete Sammlung: `[1, 2, 3]`.
`karte`	Schlüssel→Wert-Nachschlagewerk: `{"a": 1}`.
`funktion`	Eine Funktion ist selbst ein Wert (in Variablen speicherbar).
`nichts`	Der leere Wert.

Dazu kommen die **Wert-Typen** mancher Module: `Geld` (Modul `geld`), `Bytes` (Modul `bytes`) und `Zeit` (Modul `zeit`). Die **Typ-Konstanten** zum Vergleichen mit `typ(x)` lauten: `Ganz`, `Komma`, `Dezimal`, `Text`, `Wahrheit`, `Nichts`, `Liste`, `Karte`, `Bereich`, `Funktion`, `Klasse`, `Modul`, `Typ`.

B.5 Variablen und Konstanten

`nimm` legt eine veränderliche Variable an, `fest` eine unveränderliche Konstante. Die eingebauten Funktionen aus B.3 stehen überall bereit:

`b_kern.ki`

```
// Anhang B – Sprachkern: Variablen, Konstanten, eingebaute Funktionen
nimm name = "Kiste"        // nimm = veränderliche Variable
fest MAX = 100             // fest = Konstante (unveränderlich)

sag("Name: {name}")
sag("Länge von name: {länge(name)}")
sag("Typ von MAX: {typ(MAX)}")
sag("MAX als Text: {als_text(MAX)}")
sag("Pi gerundet: {runden(3.14159, 2)}")
sag("Aufgerundet: {aufrunden(2.1)}")
```

```
AUSGABE
Name: Kiste
Länge von name: 5
Typ von MAX: ganz
MAX als Text: 100
Pi gerundet: 3.14
Aufgerundet: 3
```

B.6 Steuerfluss

Form	Bedeutung
`wenn B { … } sonstwenn B { … } sonst { … }`	Fallunterscheidung nach Bedingungen.
`prüfe x { fall a, b { … } standard { … } }`	Mehrfach-Auswahl gegen konkrete Werte.
`solange B { … }`	Wiederholen, solange B wahr ist.
`für i = a bis b schritt s { … }`	Zählschleife; `bis` inklusiv, `schritt` optional.
`wiederhole x in sammlung { … }`	Jedes Element durchgehen (Liste, Text, Karte).
`springe / weiter`	Schleife abbrechen / zum nächsten Durchlauf springen.

b_steuerung.ki

```
// Anhang B – Steuerfluss: wenn/sonstwenn/sonst, prüfe, für, solange, wiederhole
nimm punkte = 75
wenn punkte >= 90 { sag("Note: sehr gut") }
sonstwenn punkte >= 60 { sag("Note: bestanden") }
sonst { sag("Note: nicht bestanden") }

nimm tag = "samstag"
prüfe tag {
    fall "samstag", "sonntag" { sag("Wochenende") }
    standard { sag("Arbeitstag") }
}

nimm summe = 0
für i = 1 bis 5 {          // bis ist inklusiv: 1,2,3,4,5
    summe += i
}
sag("Summe 1..5 = {summe}")

nimm n = 3
solange n > 0 {
    sag("Countdown {n}")
```

```
    n -= 1
}

wiederhole farbe in ["rot", "grün", "blau"] {
    sag(farbe)
}
```

```
AUSGABE
Note: bestanden
Wochenende
Summe 1..5 = 15
Countdown 3
Countdown 2
Countdown 1
rot
grün
blau
```

B.7 Funktionen

Eine Funktion bündelt Code unter einem Namen. Parameter dürfen **Standardwerte** haben; `gib` liefert ein Ergebnis zurück und beendet die Funktion sofort. Funktionen dürfen sich selbst aufrufen (**Rekursion**).

b_funktionen.ki

```
// Anhang B – Funktionen: Parameter, Standardwerte, gib, Rekursion
funktion begrüße(name = "Welt") {   // Standardwert
    gib "Hallo, " + name + "!"
}

funktion fakultät(n) {              // Rekursion
    wenn n <= 1 { gib 1 }
    gib n * fakultät(n - 1)
}

sag(begrüße())
sag(begrüße("Kiste"))
sag("5! = {fakultät(5)}")
```

```
AUSGABE
Hallo, Welt!
Hallo, Kiste!
5! = 120
```

B.8 Klassen und Objekte

Eine `klasse` ist ein Bauplan mit Feldern und Methoden. `funktion neu(...)` ist der Konstruktor, `dies` das aktuelle Objekt. Mit `erbt` übernimmt eine Klasse alles von einer Eltern-Klasse; `super` ruft deren Methode auf.

b_klassen.ki

```
// Anhang B — Klassen: Felder, Methoden, Vererbung (erbt) und super
klasse Tier {
    funktion neu(name) {
        dies.name = name
    }
    funktion vorstellen() {
        sag("Ich bin {dies.name}")
    }
}

klasse Hund erbt Tier {
    funktion neu(name, rasse) {
        super.neu(name)          // Eltern-Konstruktor
        dies.rasse = rasse
    }
    funktion vorstellen() {
        super.vorstellen()       // erst Eltern-Logik
        sag("  Rasse: {dies.rasse}")
    }
}

nimm rex = neu Hund("Rex", "Schäferhund")
rex.vorstellen()
sag("Name direkt gelesen: {rex.name}")
```

```
AUSGABE
Ich bin Rex
  Rasse: Schäferhund
Name direkt gelesen: Rex
```

Klassen dürfen außerdem **Magic-Methods** definieren, die Kiste automatisch benutzt: `als_text()` (eigene Anzeigeform), `gleicht(anderes)` (eigener `==`-Vergleich), `vergleicht_mit(anderes)` (Sortier-Reihenfolge) und `länge()` (eigene Länge).

B.9 Fehlerbehandlung

`wirf` löst einen Fehler aus; `versuche` / `fange` fängt ihn ab, statt das Programm abstürzen zu lassen. Im `fange`-Block steht der geworfene Wert in der genannten Variable.

b_fehler.ki

```
// Anhang B — Fehlerbehandlung: wirf, versuche/fange
funktion dividiere(a, b) {
    wenn b == 0 {
        wirf "Division durch Null nicht erlaubt"
    }
    gib a / b
}

versuche {
    sag("10 / 2 = {dividiere(10, 2)}")
    sag("10 / 0 = {dividiere(10, 0)}")   // wirft
} fange fehler {
    sag("Abgefangen: {fehler}")
}
```

```
AUSGABE
10 / 2 = 5
Abgefangen: Division durch Null nicht erlaubt
```

B.10 Module und die Standard-Bibliothek

Mit `nutze modul` lädst du einen Werkzeugkasten; danach rufst du seine Funktionen als `modul.funktion(…)` auf. Du darfst beliebig viele Module laden.

b_module.ki

```
// Anhang B – Module zusammen nutzen: text, liste, mathe
nutze text
nutze liste
nutze mathe

nimm farben = text.zerlegt("rot,grün,blau", ",")
sag("Anzahl Farben: {länge(farben)}")
sag("Erste groß: {text.großgeschrieben(farben[0])}")

nimm zahlen = [4, 9, 16, 25]
sag("Summe: {liste.summe(zahlen)}")
sag("Maximum: {liste.maximum(zahlen)}")
sag("Wurzel aus 16: {mathe.wurzel(16)}")
```

```
AUSGABE
Anzahl Farben: 3
Erste groß: ROT
Summe: 54
Maximum: 25
Wurzel aus 16: 4.0
```

Modul	Wofür
`mathe`	Wurzeln, Potenzen, Logarithmen, Trigonometrie, Min/Max, Konstanten.
`zufall`	Zufallszahlen, Mischen, Auswählen, reproduzierbare Saat.
`liste`	Listen filtern, abbilden, falten, sortieren, gruppieren.
`text`	Texte umformen, suchen, aufteilen, zusammenfügen.
`geld`	Exakt mit Beträgen und Währungen rechnen.
`datei`	Dateien und Ordner lesen, schreiben, verwalten.
`json`	JSON lesen (parsen) und schreiben (erzeugen).
`pfad`	Dateipfade OS-korrekt bauen und zerlegen.
`umgebung`	Betriebssystem-Infos, Umgebungsvariablen, Programm-Argumente.
`hash`	Prüfsummen (SHA-256), HMAC, Base64/Hex-Kodierung.
`regex`	Muster suchen, ersetzen, zerlegen (RE2).
`csv`	Tabellen im CSV-Format lesen und schreiben.
`zeit`	Datum und Uhrzeit, deutsche Formate, ISO-8601.
`prozess`	Externe Programme sicher starten.
`bytes`	Binäre Daten (unveränderlich), Hex/Base64, Datei-Ein/Ausgabe.
`netz`	HTTP-Anfragen (GET/POST/…), TLS.
`gui`	Fenster, Bedienelemente, Grafik — siehe B.11.
`bild`	Bilder erzeugen, laden, speichern — siehe B.12.

Hinweis: Ein paar Funktionen einzelner Module stehen im fertig gebauten Programm (`kiste build`) noch nicht bereit — Kiste sagt dir das dann klar. Die stets aktuelle Verfügbarkeit steht im Online-Lernbuch.

B.10.1 `mathe`

Signatur	Beschreibung
`mathe.PI, mathe.E, mathe.TAU`	Konstanten π, e und 2π.
`mathe.UNENDLICH, mathe.NEG_UNENDLICH, mathe.NAN`	IEEE-754-Sonderwerte (+∞, −∞, „keine Zahl").
`mathe.wurzel(x)`	Quadratwurzel.
`mathe.n_te_wurzel(x, n)`	n-te Wurzel.
`mathe.hoch(basis, exponent)`	Potenz basis^exponent.
`mathe.exp(x)`	e hoch x.
`mathe.ln(x), mathe.log10(x), mathe.log2(x)`	Logarithmen zur Basis e, 10, 2.
`mathe.log(x, basis)`	Logarithmus zur beliebigen Basis.
`mathe.sinus(x), mathe.kosinus(x), mathe.tangens(x)`	Winkelfunktionen (Eingabe in Radian).
`mathe.arkussinus(x), mathe.arkuskosinus(x), mathe.arkustangens(x)`	Umkehr-Winkelfunktionen.
`mathe.arkustangens2(y, x)`	Arkustangens mit Quadranten-Kenntnis.
`mathe.als_radian(grad), mathe.als_grad(radian)`	Grad ↔ Radian umrechnen.
`mathe.maximum(a, b, …), mathe.minimum(a, b, …)`	Größten/kleinsten von ≥ 2 Werten finden.
`mathe.absolut(x)`	Absolutbetrag.
`mathe.vorzeichen(x)`	−1, 0 oder 1.
`mathe.ist_nan(x), mathe.ist_unendlich(x), mathe.ist_endlich(x)`	Sonderwert-Prüfungen (für `komma`).

B.10.2 `zufall`

Signatur	Beschreibung
`zufall.zahl()`	Komma-Zahl im Bereich [0.0, 1.0).
`zufall.ganz(min, max)`	Ganzzahl im Bereich [min, max] (inklusiv).
`zufall.dezimal(min, max)`	Dezimal-Zahl im Bereich [min, max).
`zufall.element(liste)`	Zufälliges Element aus der Liste.
`zufall.gemischt(liste)`	Neue, durchgemischte Liste (Original bleibt).
`zufall.mische(liste)`	Mischt die Liste direkt (verändert sie).
`zufall.setze_saat(zahl)`	Setzt die Saat → reproduzierbare Zufallsfolge.

B.10.3 `liste`

Signatur	Beschreibung
`liste.länge(l)`, `liste.leer(l)`	Anzahl der Elemente / ob leer.
`liste.summe(l)`, `liste.mittel(l)`	Summe / Mittelwert der Elemente.
`liste.maximum(l)`, `liste.minimum(l)`	Größtes / kleinstes Element.
`liste.gefüllt(n, wert)`	Neue Liste aus n Kopien von `wert`.
`liste.gefiltert(l, fn)`	Nur die Elemente, für die `fn` wahr ergibt.
`liste.abgebildet(l, fn)`	Jedes Element durch `fn` umwandeln.
`liste.gesammelt(l, start, fn)`	Alle Elemente zu einem Wert zusammenfalten.
`liste.sortiert(l)`	Neue Liste, aufsteigend sortiert.
`liste.sortiere(l)`	Sortiert die Liste direkt.
`liste.umgekehrt(l)`, `liste.kehre_um(l)`	Umgekehrte Reihenfolge (neu / direkt).
`liste.angehängt(l, wert)`, `liste.hänge_an(l, wert)`	Wert am Ende ergänzen (neu / direkt).
`liste.vorangestellt(l, wert)`, `liste.stelle_voran(l, wert)`	Wert am Anfang ergänzen (neu / direkt).
`liste.eingefügt(l, index, wert)`, `liste.füge_ein(l, index, wert)`	An Position einfügen (neu / direkt).
`liste.entfernt(l, index)`, `liste.entferne(l, index)`	Element an Position löschen (neu / direkt).
`liste.einzigartig(l)`, `liste.entferne_duplikate(l)`	Doppelte entfernen (neu / direkt).
`liste.geflacht(l)`, `liste.flache(l)`	Verschachtelte Listen flach machen (neu / direkt).
`liste.scheibe(l, start, ende)`	Teilbereich [start, ende) (inklusiv).
`liste.erste_n(l, n)`, `liste.letzte_n(l, n)`	Die ersten / letzten n Elemente.
`liste.verbunden(l1, l2, …)`	Mehrere Listen aneinanderhängen.
`liste.gezippt(l1, l2)`, `liste.gezippt_mit(l1, l2, fn)`	Elemente paaren / paarweise verrechnen.

B.10.4 `text`

Signatur	Beschreibung
`text.länge(s)`, `text.bytes_länge(s)`, `text.zeichen_länge(s)`	Grapheme / Bytes / Codepoints zählen.
`text.leer(s)`	Ob der Text leer ist.
`text.enthält(s, teil)`, `text.beginnt_mit(s, p)`, `text.endet_mit(s, p)`	Enthalten / Anfang / Ende prüfen.
`text.finde_index(s, teil)`	Position von `teil`; `nichts` wenn nicht gefunden.
`text.zähle(s, teil)`	Vorkommen zählen.
`text.großgeschrieben(s)`, `text.kleingeschrieben(s)`, `text.kapitalisiert(s)`	Groß / klein / erster Buchstabe groß.
`text.beschnitten(s)`, `text.links_beschnitten(s)`, `text.rechts_beschnitten(s)`	Leerraum entfernen (beidseitig / links / rechts).
`text.umgekehrt(s)`, `text.wiederholt(s, n)`	Umdrehen / n-mal wiederholen.
`text.ersetzt(s, alt, neu)`, `text.ersetzt_erstes(s, alt, neu)`	Alle / nur das erste Vorkommen ersetzen.
`text.scheibe(s, start, ende)`, `text.zeichen_bei(s, index)`	Teilstück / einzelnes Graphem.
`text.erste_n(s, n)`, `text.letzte_n(s, n)`	Die ersten / letzten n Grapheme.
`text.gefüllt_links(s, länge, füller)`, `text.gefüllt_rechts(s, länge, füller)`	Auf Länge auffüllen.
`text.zerlegt(s, trenner)`, `text.zeilen(s)`, `text.wörter(s)`, `text.zeichen(s)`	In eine Liste aufteilen.
`text.verkettet(liste, trenner)`	Eine Liste von Texten mit Trenner verbinden.

B.10.5 `geld`

Signatur	Beschreibung
`geld.neu(betrag, währung)`	Einen Geld-Wert erzeugen, z. B. `geld.neu(19.99, "CHF")`.
`geld.betrag(g)`	Den reinen Betrag (Zahl) herausholen.
`geld.währung(g)`	Die Währung (Text) herausholen.
`geld.in_währung(g, zielwährung, kurs)`	In eine andere Währung umrechnen (mit Kurs).
`geld.gerundet_auf(g, schritt)`	Auf einen Schritt runden (z. B. 0.05).
`geld.formatiert(g, optionen)`	Mit eigenen Optionen formatieren (Symbol, Trenner …).

Die Operatoren `+ - * / == != < <= > >=` funktionieren direkt auf Geld. Bewusst *nicht* erlaubt sind unsinnige Rechnungen (Geld × Geld, Geld + Zahl, verschiedene Währungen addieren) — Kiste meldet das als klaren Fehler.

`b_geld.ki`

```
// Anhang B – Geld: exakt rechnen mit Währung
nutze geld

nimm preis = geld.neu(19.99, "CHF")
nimm rabatt = geld.neu(5.00, "CHF")
nimm summe = preis - rabatt        // Operatoren funktionieren direkt

sag("Preis:       {preis}")
sag("Nach Rabatt: {summe}")
sag("Betrag:      {geld.betrag(summe)}")
sag("Währung:     {geld.währung(summe)}")
```

```
AUSGABE
Preis:       19.99 CHF
Nach Rabatt: 14.99 CHF
Betrag:      14.99
Währung:     CHF
```

B.10.6 `datei`

Signatur	Beschreibung
`datei.inhalt(pfad)`	Ganze Datei als Text lesen.
`datei.zeilen(pfad)`	Datei zeilenweise in eine Liste lesen.
`datei.schreibe(pfad, inhalt)`	Text in eine Datei schreiben (überschreibt).
`datei.hänge_an(pfad, inhalt)`	Text am Ende anhängen.
`datei.existiert(pfad)`, `datei.ist_datei(pfad)`, `datei.ist_ordner(pfad)`	Vorhandensein / Art prüfen (wirft nie).
`datei.größe(pfad)`	Dateigröße in Bytes.
`datei.geändert_unix(pfad)`	Zeitpunkt der letzten Änderung (Unix-Sekunden).
`datei.lösche(pfad)`	Datei oder leeren Ordner löschen.
`datei.kopiere(quelle, ziel)`, `datei.verschiebe(alt, neu)`	Kopieren / verschieben (umbenennen).
`datei.erstelle_ordner(pfad)`, `datei.erstelle_ordner_rekursiv(pfad)`	Ordner anlegen (ein Schritt / rekursiv).
`datei.lösche_ordner_rekursiv(pfad)`	Ordner samt Inhalt löschen (Vorsicht!).
`datei.einträge(pfad)`, `datei.einträge_mit_details(pfad)`	Verzeichnis auflisten (Namen / mit Metadaten).

B.10.7 `json`

Signatur	Beschreibung
`json.geparst(text)`	JSON-Text in Kiste-Werte umwandeln (wirft bei Fehler).
`json.versucht_geparst(text)`	Wie oben, gibt aber `nichts` bei Fehler.
`json.erzeugt(wert)`	Kiste-Wert in kompakten JSON-Text.
`json.erzeugt_schön(wert, einrückung)`	In schön eingerückten JSON-Text.

B.10.8 `pfad`

Signatur	Beschreibung
`pfad.trenner, pfad.listen_trenner`	OS-Trennzeichen für Pfade / für die PATH-Liste.
`pfad.verbunden(teil1, teil2, …)`	Pfad-Teile OS-korrekt zusammensetzen.
`pfad.dateiname(p), pfad.ordner(p)`	Letzter Teil / übergeordneter Ordner.
`pfad.erweiterung(p), pfad.basisname(p)`	Endung / Name ohne Endung.
`pfad.zerlegt(p)`	Pfad in seine Bestandteile zerlegen.
`pfad.normalisiert(p)`	`.`, `..`, doppelte Trenner auflösen.
`pfad.absolut(p), pfad.relativ(basis, ziel)`	Absolut machen / relativen Pfad berechnen.
`pfad.ist_absolut(p), pfad.ist_relativ(p)`	Art des Pfads prüfen.

B.10.9 `umgebung`

Signatur	Beschreibung
`umgebung.betriebssystem(), umgebung.architektur()`	OS ("windows"/"linux"/"darwin") / CPU-Architektur.
`umgebung.benutzer(), umgebung.benutzer_ordner()`	Benutzername / Home-Verzeichnis.
`umgebung.wert(name), umgebung.wert_oder(name, default)`	Umgebungsvariable lesen (mit/ohne Fallback).
`umgebung.setze(name, wert), umgebung.entferne(name)`	Umgebungsvariable setzen / entfernen (nur dieser Prozess).
`umgebung.alle()`	Alle Umgebungsvariablen als Karte.
`umgebung.argumente()`	Kommandozeilen-Argumente als Liste.
`umgebung.arbeitsverzeichnis(), umgebung.wechsle_in(pfad)`	Arbeitsverzeichnis lesen / wechseln.
`umgebung.beende(code)`	Programm mit Exit-Code beenden.

B.10.10 `hash`

Signatur	Beschreibung
`hash.sha256(text), hash.sha512(text)`	Sichere Prüfsumme (Hex-Text).
`hash.md5(text)`	MD5 (gebrochen — nur für Caching/ETags).
`hash.hmac_sha256(daten, geheim)`	Authentifizierte Prüfsumme mit Schlüssel (API-Tokens).
`hash.base64_kodiert(text), hash.base64_dekodiert(text)`	Base64 kodieren / dekodieren.
`hash.hex_kodiert(text), hash.hex_dekodiert(text)`	Hex kodieren / dekodieren.

Sicherheit: SHA-256 ist **kein** Passwort-Hash. Für Passwörter braucht man absichtlich langsame Verfahren (Bcrypt/Argon2).

B.10.11 `regex`

Signatur	Beschreibung
`regex.passt(muster, text)`	Ob das Muster irgendwo passt (wahr/falsch).
`regex.treffer(muster, text)`, `regex.treffer_alle(muster, text)`	Ersten / alle Treffer.
`regex.gruppen(muster, text)`, `regex.treffer_alle_gruppen(muster, text)`	Capture-Gruppen des ersten / aller Treffer.
`regex.ersetzt(muster, text, ersatz)`, `regex.ersetzt_erstes(…)`	Alle / erstes Vorkommen ersetzen.
`regex.zerlegt(muster, text)`	Text an Regex-Trennern zerlegen.
`regex.maskiert(text)`, `regex.gültig(muster)`	Sonderzeichen maskieren / Muster prüfen.

Kiste nutzt RE2 (lineare Laufzeit, kein Lookahead/Backreferences im Muster) — sicher gegen „böse" Muster. Inline-Flags: `(?i)` ohne Groß/Klein, `(?m)` Zeilen-Anker, `(?s)` Punkt matcht Zeilenumbruch.

B.10.12 `csv`

Signatur	Beschreibung
`csv.zeilen(text, optionen?)`	CSV lesen → Liste von Listen.
`csv.karten(text, optionen?)`	CSV mit Kopfzeile lesen → Liste von Karten.
`csv.erzeugt(zeilen, optionen?)`	Liste von Listen → CSV-Text.
`csv.erzeugt_aus_karten(karten, optionen?)`	Liste von Karten → CSV-Text (Kopfzeile automatisch).

Optionen als Karte, z. B. `{"trenner": ";"}`. Alle Werte kommen als `text` zurück (keine automatische Typ-Erkennung).

B.10.13 `zeit`

Signatur	Beschreibung
`zeit.jetzt()`, `zeit.jetzt_lokal()`, `zeit.heute()`	Aktuelle UTC-Zeit / lokale Zeit / heutiges Datum.
`zeit.von_komponenten(jahr, monat, tag, …)`	`Zeit`-Wert aus Bestandteilen bauen.
`zeit.jahr(z)`, `zeit.monat(z)`, `zeit.tag(z)`, `zeit.stunde(z)`, `zeit.wochentag(z)`	Bestandteile auslesen.
`zeit.unix(z)`	Sekunden seit 1970 (UTC).
`zeit.formatiert(z, muster)`, `zeit.formatiert_iso(z)`	Nach Muster / als ISO-8601 formatieren.
`zeit.geparst(text, muster)`, `zeit.geparst_iso(text)`	Text zu `Zeit` einlesen.
`zeit.ist_vor(a, b)`, `zeit.ist_nach(a, b)`, `zeit.ist_gleich(a, b)`	Zwei Zeitpunkte vergleichen.
`zeit.schlafe_ms(ms)`	N Millisekunden warten.

Format-Tokens (deutsch): `JJJJ` Jahr, `MM` Monat, `TT` Tag, `hh:mm:ss` Uhrzeit. `zeit.jetzt()` liefert immer UTC.

B.10.14 `prozess`

Signatur	Beschreibung
`prozess.ausgeführt(args, optionen?)`	Programm starten → Karte mit exit_code, stdout, stderr.
`prozess.ausgabe(args, optionen?)`	Starten und nur die Ausgabe (stdout) als Text.
`prozess.exit_code(args, optionen?)`	Nur den Exit-Code zurückgeben.
`prozess.erfolgreich(args, optionen?)`	Nur wahr/falsch (Exit-Code 0?).

`args` ist **immer eine Liste** — z. B. `["git", "status"]` (schützt vor Shell-Injection).

B.10.15 `bytes`

Signatur	Beschreibung
`bytes.von_text(text)`, `bytes.von_ganzen(liste)`	`Bytes` aus Text / Zahlenliste (0–255).
`bytes.aus_hex(hex)`, `bytes.aus_base64(b64)`	`Bytes` aus Hex- / Base64-Text.
`bytes.als_text(b)`, `bytes.als_text_oder(b, default)`	Zu Text (wirft / mit Fallback).
`bytes.als_ganze(b)`, `bytes.als_hex(b)`, `bytes.als_base64(b)`	Zu Zahlenliste / Hex / Base64.
`bytes.byte_bei(b, index)`, `bytes.scheibe(b, von, bis)`	Einzelnes Byte / Teilbereich.
`bytes.enthält(b, teil)`, `bytes.ist_gleich(b1, b2)`, `bytes.verkettet(b1, b2, …)`	Enthalten / gleich / aneinanderhängen.
`datei.bytes_inhalt(pfad)`, `datei.schreibe_bytes(pfad, b)`	Rohe Bytes aus Datei lesen / schreiben.

B.10.16 `netz`

Signatur	Beschreibung
`netz.hole(url, optionen?)`	GET-Anfrage → `Antwort`-Wert.
`netz.hole_text(url, optionen?)`	GET, muss 2xx + UTF-8 sein → Text direkt.
`netz.sende(methode, url, körper, optionen?)`	POST/PUT/DELETE/PATCH mit Text-/Bytes-Körper.
`netz.sende_json(methode, url, daten, optionen?)`	JSON senden (Content-Type automatisch).
`netz.pinge(url)`	HEAD-Anfrage → nur HTTP-Status.

Der `Antwort`-Wert hat u. a. `status`, `inhalt`, `bytes`, `header`, `dauer_ms`, `url`, `erfolgreich`.

B.11 Der `gui`-Werkzeugkasten

Mit `nutze gui` baust du Fenster-Programme wie KistePix. Die App startet mit `gui.starte()` als **letzter Zeile**. Handler-Funktionen bekommen das Widget selbst; um andere Widgets zu ändern, nutzt du

globale Variablen. `gui` läuft nur im gebauten Programm (`kiste build`, F7).

Fenster und Start

Signatur	Beschreibung
`gui.fenster_öffne(titel, breite, höhe)`	Ein neues Fenster erzeugen.
`gui.zeige(fenster, inhalt)`	Ein Widget/Layout ins Fenster hängen.
`gui.vollbild(fenster, an)`	Vollbild ein (`an` = 1) oder aus (0).
`gui.starte()`	Die App starten — muss die letzte Zeile sein.

Beschriftung und Knöpfe

Signatur	Beschreibung
`gui.etikett_neu(text)`	Ein Text-Schild (nur Anzeige).
`gui.knopf_neu(text)`	Ein Knopf mit Beschriftung.
`gui.knopf_icon(text, icon)`	Ein Knopf mit Icon (Icon-Namen wie "speichern", "öffnen" …).
`gui.bei_klick(knopf, handler)`	Handler beim Knopfdruck.
`gui.setze_text(widget, text)`	Beschriftung von Knopf, Etikett oder Kontrollkästchen live ändern.

Eingabe-Elemente

Signatur	Beschreibung
`gui.eingabefeld(platzhalter)`	Einzeiliges Textfeld.
`gui.textbereich(platzhalter, zeilen?)`	Mehrzeiliges Textfeld.
`gui.wert(feld)`, `gui.setze_wert(feld, text)`	Text lesen / setzen.
`gui.bei_änderung(feld, handler)`	Handler bei jeder Textänderung.
`gui.kontrollkästchen(text)`	Ankreuzfeld (Checkbox).
`gui.angehakt(k)`, `gui.setze_angehakt(k, an)`	Häkchen lesen (1/0) / setzen.
`gui.bei_umschaltung(k, handler)`	Handler beim An/Aus-Schalten.
`gui.auswahl_box(optionen)`	Aufklapp-Auswahl (Dropdown).
`gui.gewählt(box)`, `gui.bei_wahl(box, handler)`	Gewählten Text lesen / Handler bei Wahl.
`gui.radio_knöpfe(optionen)`	Radio-Knöpfe (eine Option aus mehreren).
`gui.schieberegler(min, max)`	Schieberegler für eine Zahl.
`gui.schieber_wert(s)`, `gui.setze_schieber_wert(s, wert)`	Position lesen / setzen.
`gui.bei_schieben(s, handler)`	Handler beim Bewegen.
`gui.fortschritt_neu()`, `gui.setze_fortschritt(balken, prozent)`	Fortschrittsbalken erzeugen / setzen (0–100).

Layout und Container

Signatur	Beschreibung
`gui.vertikal(widgets)`, `gui.horizontal(widgets)`	Untereinander / nebeneinander anordnen.
`gui.raster(spalten, widgets)`	Raster mit N Spalten.
`gui.zentriert(widget)`	Widget mittig platzieren.
`gui.dehnbar()`	Dehnbarer Platzhalter (frisst freien Platz).
`gui.feste_größe(widget, breite, höhe)`	Exakte Größe geben.
`gui.teiler_h(a, b, pos)`, `gui.teiler_v(a, b, pos)`	Ziehbarer Teiler (waagerecht / senkrecht).
`gui.farb_rahmen(widget, hex)`, `gui.setze_rahmen_farbe(rahmen, hex)`	Farbige Karte um ein Widget / Farbe ändern.
`gui.reiter_neu()`, `gui.reiter_hinzufügen(reiter, titel, inhalt)`	Reiter (Tabs) anlegen / Seite hinzufügen.

Ereignisse — Tastatur und Maus

Signatur	Beschreibung
`gui.taste(fenster, "Strg+S", handler)`	Tastatur-Kürzel registrieren.
`gui.bei_taste(fenster, handler)`, `gui.letzte_taste(fenster)`	Handler bei Tastendruck / Name der Taste.
`gui.taste_gedrückt(fenster, taste)`	1, wenn die Taste gerade gehalten wird (für Spiele).
`gui.bei_maus_klick(leinwand, handler)`	Handler beim Klick auf die Leinwand.
`gui.maus_x(leinwand)`, `gui.maus_y(leinwand)`	Klick-Koordinaten (leinwand-lokal).
`gui.bei_maus_bewegung(leinwand, handler)`	Handler bei Mausbewegung (ohne Taste).
`gui.bei_maus_ziehen(leinwand, handler)`	Handler beim Ziehen (mit gedrückter Taste).
`gui.zieh_dx(leinwand)`, `gui.zieh_dy(leinwand)`	Verschiebung seit dem letzten Zieh-Schritt.
`gui.bei_maus_los(leinwand, handler)`	Handler beim Loslassen (Zug-Ende).
`gui.bei_maus_rad(leinwand, handler)`	Handler beim Mausrad-Drehen.
`gui.rad_dy(leinwand)`, `gui.rad_dx(leinwand)`	Mausrad-Bewegung senkrecht / waagerecht.

Leinwand (frei zeichnen)

Signatur	Beschreibung
`gui.leinwand(breite, höhe)`	Eine Zeichenfläche erzeugen.
`gui.leinwand_elastisch()`	Eine Zeichenfläche, die mit dem Fenster / im Vollbild mitwächst.
`gui.leinwand_breite(l)`, `gui.leinwand_höhe(l)`	Aktuelle Pixel-Größe der Leinwand (nach dem Layout).
`gui.bei_größe(l, handler)`	Handler bei jeder Größenänderung der Leinwand.
`gui.rechteck(l, x, y, breite, höhe, farbe)`	Gefülltes Rechteck malen (Farbe als `"#rrggbb"`).
`gui.kreis(l, x, y, radius, farbe)`	Gefüllten Kreis malen (x, y = Mittelpunkt).
`gui.linie(l, x1, y1, x2, y2, farbe, dicke)`	Linie mit Dicke malen.
`gui.text_an(l, x, y, text, farbe, größe)`	Text malen (x, y = obere linke Ecke).
`gui.bild_an(l, pfad, x, y, breite, höhe)`	Eine Bilddatei auf die Leinwand zeichnen.
`gui.leeren(l)`	Die ganze Leinwand löschen (transparent).

Leinwand-Koordinaten: (0, 0) ist oben links; x geht nach rechts, y nach unten.

Zeit und Animation

Signatur	Beschreibung
`gui.bei_zeit(ms, handler)`	Handler alle `ms` Millisekunden aufrufen (gibt ein Handle zurück). 16 ms ≈ 60 Bilder/s.
`gui.nach_zeit(ms, handler)`	Handler einmal nach `ms` aufrufen.
`gui.stoppe_zeit(handle)`	Einen laufenden Timer stoppen.

Dialoge, Menüs und Erscheinungsbild

Signatur	Beschreibung
`gui.dialog_info(fenster, titel, text)`	Info-Dialog mit OK.
`gui.dialog_frage(fenster, titel, text, handler)`	Ja/Nein-Dialog (Handler bekommt 1/0).
`gui.dialog_datei(fenster, handler)`, `gui.dialog_speichern(fenster, handler)`	Datei öffnen / speichern.
`gui.gewählter_pfad(fenster)`	Den im Dialog gewählten Pfad lesen.
`gui.menü_leiste(fenster)`, `gui.menü_hinzufügen(leiste, titel)`	Menüleiste / ein Menü hinzufügen.
`gui.menü_eintrag(menü, titel, handler)`, `gui.menü_trenner(menü)`	Menü-Eintrag / Trennstrich.
`gui.menü_anwenden(fenster)`	Menüleiste neu anwenden — nach dem Umbeschriften der Titel/Einträge per `gui.setze_text`.
`gui.setze_tooltip(widget, text)`	Hilfe-Text beim Verweilen (vor `gui.zeige` setzen).
`gui.thema("dunkel"/"hell")`, `gui.setze_farbe(rolle, hex)`	Design-Thema / Themen-Farbe setzen.
`gui.setze_wichtigkeit(knopf, "primär"/…)`, `gui.setze_knopf_farbe(knopf, hex)`	Knopf hervorheben / eigene Farbe geben.

B.12 Das `bild`-Modul

Mit `nutze bild` erzeugst und speicherst du echte Bilddateien — so exportiert KistePix (Bau-Stufe S7). Ein neues Bild ist zunächst komplett durchsichtig.

Signatur	Beschreibung
`bild.neu(breite, höhe)`	Neues (transparentes) Bild in Pixeln.
`bild.setze_pixel(b, x, y, r, g, b)`	Einen Pixel setzen (r, g, b je 0–255).
`bild.setze_block(b, x, y, breite, höhe, r, g, b)`	Ein ganzes Rechteck füllen.
`bild.lade(pfad)`	Bilddatei laden (PNG, JPEG, GIF, BMP, TIFF, WEBP).
`bild.breite(b)`, `bild.höhe(b)`	Maße in Pixeln.
`bild.hole_pixel(b, x, y)`	Farbe an (x, y) als Liste `[r, g, b, a]` lesen.
`bild.schreibe_png(b, pfad)`	Als PNG speichern (verlustfrei, mit Transparenz — beste Wahl).
`bild.schreibe_jpeg(b, pfad, güte)`	Als JPEG (Fotos; `güte` 1–100, keine Transparenz).
`bild.schreibe_gif(b, pfad)`, `bild.schreibe_bmp(b, pfad)`, `bild.schreibe_tiff(b, pfad)`	Als GIF / BMP / TIFF speichern.
`bild.schreibe_gif_animiert(pfad, bilder, verzögerung)`	Animiertes GIF aus mehreren Bildern (Verzögerung in Hundertstelsekunden).

B.13 Und wenn etwas fehlt?

Kiste bekommt laufend neue Funktionen. Wenn du hier etwas nicht findest oder die allerneueste Fassung brauchst, schau ins Online-Lernbuch unter **kiste-lang.org/de/buch/** — dort steht die komplette, stets aktuelle Referenz mit vielen weiteren Beispielen.