universität wien

DIPLOMARBEIT

Titel der Diplomarbeit

Schülergerechte Einführung in die Programmierung und in das algorithmische Denken anhand von visuellen Programmiersprachen

Verfasserin

Doris Mayr

angestrebter akademischer Grad

Magistra der Naturwissenschaften (Mag. rer. nat.)

Wien, 2014

Studienrichtung lt. Studienblatt:	UF Lehramt Informatik und Informatikmanagement
Studienkennzahl lt. Studienblatt:	A 190 406 884
Betreuer:	Ao. Univ. Prof. Dipl.-Ing. Dr. techn. Gerald Futschek

I. Eidesstattliche Erklärung

Hiermit erkläre ich, dass ich diese Arbeit selbstständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit - einschließlich Tabellen, Karten und Abbildungen -, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Stockerau, 13. Juni 2014

II. Abstract

Es wird immer wichtiger, dass Schüler und Schülerinnen Programmieren lernen, denn sie lernen beim Programmieren viel mehr als das bloße Erstellen von Programmen.

Daher sind die zentralen Fragestellungen dieser Diplomarbeit „Wie können SchülerInnen Programmieren lernen?“ und „Warum sollen SchülerInnen Programmieren lernen?“.

Es werden auf die Probleme von ProgrammieranfängerInnen eingegangen und daraus begründet, warum Schüler und Schülerinnen mit visuellen Programmiersprachen programmieren lernen sollen. Zusätzlich werden allgemeine Begründungen für das Programmieren erörtert.

Die Vor- und Nachteile von Scratch und Snap! gegenüber Java werden aufgezeigt, indem die Programmiersprachen anhand von einfachen Beispielen verglichen werden.

Zum Abschluss werden 40 Aufgabenstellungen für den Programmierunterricht erstellt, mit denen man die einzelnen Grundkonzepte der Programmierung erlernen kann.

Die Forschungsfragen werden mit Hilfe von Literaturrecherche beantwortet, aber ebenso sind eigene Ideen und Erläuterungen in diese Diplomarbeit eingearbeitet.

III. Abstract

Nowadays it is getting more and more important for students to learn how to program, because they learn a lot more than just creating programs.

Therefore the central questions of this master thesis are “How can students learn how to program?” and “Why should students learn programming?”.

The problems of beginners are pointed out and it is discussed why students should start learning how to program by using visual programming languages. Additionally, general arguments for learning how to program are shown.

Moreover, both advantages and disadvantages of Scratch and Snap! in comparison to Java are listed and explained with the help of simple examples.

Finally, 40 tasks which can be used in programming lessons are defined. With these tasks it is possible to learn the basic concepts of programming.

All research questions are answered with the help of literature research, own ideas and explanations.

IV. Danksagung

An dieser Stelle möchte ich mich bei allen bedanken, die mich während meines Studiums unterstützt haben. Vor allem bedanken möchte ich mich bei meinem Freund Oliver Fischer, der mich immer wieder motiviert hat zu lernen, an mich geglaubt hat und für das Verständnis, dass es auch Wochenenden gibt, die dem Lernen gewidmet sind.

Besonders bedanken möchte ich bei meinen Eltern, Ulrike Mayr und Leopold Mayr, die mich vielfältig unterstützt haben.

Meinem Diplomarbeitsbetreuer Dipl.-Ing. Dr. techn. Gerald Futschek möchte ich mich für die laufende Unterstützung bei der Diplomarbeit bedanken, vor allem bei der Ideenfindung für die Diplomarbeit gab er mir hilfreiche Denkanstöße.

Nicht zu vergessen ist meine liebe Studienkollegin, Tanja Worel, die ich im Laufe meines Studiums kennen lernen durfte. Gemeinsam haben wir viele Übungen und Seminare besucht. Sie war auch immer ein Motivator und meine treibende Kraft, die mich dazu bewegte noch ein Seminar im schon vollgestopften Stundenplan unterzubringen. Vielen Dank dafür!

Inhaltsverzeichnis

I. Eidesstattliche Erklärung 3
II. Abstract 5
III. Abstract 7
IV. Danksagung 9
1. Problemstellung 13
2. Methodische Vorgehensweise 14
3. Warum Programmieren lernen? 15
4. Schwierigkeiten der ProgrammieranfängerInnen 20
5. Visuelle Programmiersprachen 24
5.1. Scratch 24
5.2. Snap! Build your own Blocks 25
5.3. Visuelle Programmiersprachen minimieren die Schwierigkeiten 25
5.4. Wo sind die Grenzen? 27
6. Konzepte / Strukturen der Programmierung 28
7. Vergleich Scratch / Snap! mit Java 31
7.1. Variablen 31
7.2. Schleifen 33
7.3. Bedingte Anweisungen und Verzweigungen 36
7.4. Methoden und Funktionen 38
7.5. Listen 46
7.6. Sonstiges 47
8. Aufgaben 51
8.1. Strukturierte Vorgehensweise 54
8.2. Variablen 58
8.3. Schleifen 60

8.4. Bedingte Anweisungen und Verzweigungen 65
8.5. Methoden und Funktionen 71
8.6. Listen 80
8.7. Rekursion 95
9. Zusammenfassung 99
10. Literaturverzeichnis 101
11. Abbildungsverzeichnis 103
12. Tabellenverzeichnis 106

1. Problemstellung

In vielen Schulen reduziert sich der Informatikunterricht auf die Inhalte des Europäischen Computerführerscheins. Das Interesse der Schülerinnen und Schüler am Programmieren hat nachgelassen. Argumente gegen das Erlernen sind zum Bespiel, dass das Programmieren zu zeitaufwendig ist, zu schwierig ist und es liefert kein befriedigendes Ergebnis. (vgl. [Sc05])

Dennoch ist „Algorithmen, Datenstrukturen und Programmierung“ ein wesentlicher Bestandteil des Informatiklehrplans der 5. Klasse bzw. des neuen Kompetenzmodell Informatik.

Doch wie können Schülerinnen und Schüler wieder Begeisterung am Programmieren finden? Warum ist es wichtig algorithmisch denken zu können und wesentliche Programmierkonzepte zu verstehen, abgesehen davon, dass es ein Bestandteil des Lehrplans ist?

Im Zuge der Diplomarbeit werden folgende Forschungsfragen beantwortet:

- „Auf welche Weise können Schülerinnen und Schüler mit visuellen Programmiersprachen Programmieren lernen?
- „Warum sollen Schülerinnen und Schüler Programmieren lernen?“
- „Welche Konzepte der Programmierung können mittels Scratch und Snap! erlernt werden und wo gibt es Grenzen?“
- „Welche Vorteile und Nachteile, gegenüber anderen schulüblichen Programmiersprachen wie Java, haben die visuellen Programmiersprachen Scratch und Snap!?“

2. Methodische Vorgehensweise

Die wichtigste Methode dieser Diplomarbeit war die Literaturrecherche. Mit dieser wurden mehrere Begründungen für das Erlernen von Programmiersprachen gefunden. Die Erläuterungen dazu stammen ebenso aus wissenschaftlicher Literatur oder stammen von mir selbst. Um die Frage beantworten zu können, warum Schülerinnen und Schüler mittels visuellen Programmiersprachen programmieren lernen sollte, wurden zuerst auf die Probleme der ProgrammieranfängerInnen eingegangen, die sich aus mehreren Studien gezeigt haben. Darauf aufbauend wurden Begründungen gefunden, wieso visuelle Programmiersprachen helfen können diese Barrieren zu beseitigen.

Als visuelle Programmiersprachen wurden Scratch und Snap! ausgewählt, da sie die Möglichkeit bieten, einfache spielerische Szenarien zu programmieren.

Die wesentlichen Konzepte der Programmierung, die man mit den visuellen Programmiersprachen Scratch und Snap! erlernen kann, wurden schülergerecht beschrieben.

Aufbauend auf diese Konzepte wurden die visuellen Programmiersprachen Scratch/Snap! und Java verglichen. Daraus wurden die Vorteile und Nachteile erläutert.

Ausgehend von den Begründungen, die für ein Erlernen von Programmieren sprechen, und den zu erlernenden Konzepten wurden Unterrichtsbeispiele zusammengestellt. Sie bestehen aus einer Aufgabenstellung, den Lernzielen und eines möglichen Lösungswegs. Diese Unterrichtsbeispiele wurden den einzelnen Konzepten zugeordnet. Viele dieser Aufgaben bauen einander auf.

3. Warum Programmieren lernen?

In diesem Kapitel wird erläutert warum Schüler und Schülerinnen Programmieren lernen sollen. Die Auflistung wurde aus [Br09] übernommen. Die jeweiligen Erläuterungen dazu haben eine eigene Quellenangabe oder kommen von mir selbst.

Programmieren fördert die Problemlösefähigkeit und das logische Denken

Programmieren bedeutet nicht nur eine Übersetzung eines bekannten Algorithmus in die Sprache der Computer, sondern ist die Suche nach Lösungswegen für ein gegebenes Problem. Daher fordert und fördert Programmieren, so ähnlich wie der Mathematikunterricht, die Entwicklung des allgemeinen lösungsorientierten Denken bzw. der Problemlösefähigkeit. (vgl. [GH10])

Programmieren sorgt für eine klare präzise Sprache bei der Formulierung der Anweisungen

Programmiersprachen verlangen, dass man sich exakt ausdrückt und auch jeden Schritt angibt der zu tun ist. Das ist deswegen notwendig, da der Computer nicht in der Lage ist zu improvisieren. Er führt nur jene Befehle aus, die man Ihm aufträgt. Nicht mehr und auch nicht weniger. (vgl. [GH10])

Da der Rechner bei der Ausführung des implementierten Algorithmus nur Anweisungen Schritt für Schritt befolgt, ist absolute Präzision und Klarheit der Formulierungen für die Beschreibung des Lösungsweges unbedingt notwendig. Nur wenn dies beachtet wird, kann der Rechner sie verstehen und auch umsetzen. Diese präzise Sprache trägt zur Förderung der Kommunikationsfähigkeit bei. (vgl. [GH10])

Programmieren weckt das Verständnis für Abläufe im Rechner und macht die langlebigen Grundlagen der Informatik deutlich

Nicht nur Programmieren allein wird gelernt, sondern auch die Begriffe der Informatik, wie zum Beispiel Verifikation, Berechnungskomplexität / Rechenaufwand und Determiniertheit. Ebenso wird ersichtlich, dass nicht jedes System vollkommen ist, da man beim Programmieren auch lernt, Programme nach Effizienz, Länge, Verständlichkeit, modulare Struktur, Benutzerfreundlichkeit und Kompatibilität zu beurteilen. (vgl. [Hr12])

In anderen Fächern, wie Mathematik und Physik, beginnt man Grundkonzepte und Begriffsbildungen zu lernen bzw. zu lehren. Die wichtigsten Begriffe der Informatik sind „Algorithmus“ und „Programm“. Diese zwei Grundbegriffe werden ideal beim Programmieren vermittelt. (vgl. [Hr12])

Programmieren verbindet die mathematisch-naturwissenschaftliche Denkweise mit der Arbeitsweise der technischen Wissenschaften

Programmieren erfordert eine Problemspezifikation, die Suche nach einem Lösungsweg und auch eine formale Ausdruckweise zur Beschreibung einer gefunden Lösungsmethode. Diese drei Aspekte sind ebenso in der Mathematik zu finden. (vgl. [Hr12])

Die modulare Arbeitsweise ist weit verbreitet in den Ingenieurwissenschaften. Man löst ein Problem indem man es in kleine Teilprobleme aufgliedert und dann diese Teilprobleme einzeln und schrittweise löst. Man erstellt einfache kleine Systeme und Lösungen für Teilprobleme, die leichter auf Richtigkeit zu überprüfen sind. Nach dem Zusammenführen dieser kleineren Teilsysteme ist man in der Lage das gewünschte große Problem zu lösen. Diese modulare Vorgehensweise muss man auch bei komplexen Programmierungen vornehmen. Programmieren lehrt also diese modulare Vorgehensweise, die typisch in den Ingenieurwissenschaften ist. (vgl. [Hr12])

Programmieren unterstützt das fächerübergreifende Arbeiten

Nicht nur Informatikspezifische Probleme können mit Programmieren gelöst werden, sondern auch Probleme, die man in anderen Fachgruppen findet. Abgesehen von mathematischen Problemlösungen, die man leicht programmieren kann (z.B. Mittelwert einer Liste, geometrische Formen zeichnen, etc.) können auch Programme programmiert werden, die zu einem anderen Fachbereich, wie zum Beispiel der Biologie, Musik oder Deutsch zugeordnet werden können.

Programmieren leistet einen hochwertigen Beitrag zur Allgemeinbildung

Informatik ist die Leitwissenschaft der Informationsgesellschaft. Da wir gerade in einem Informationszeitalter leben und alle Prozesse informationsgesteuert sind, ist es wichtig auch die Hintergründe und die Abläufe von Prozessen zu verstehen. (vgl. [Kl14])

Wesentliche Bildungsziele sind im österreichischen Lehrplan zu finden, die uns eindeutig zeigen, wie wichtig es ist, dass Schüler und Schülerinnen Programmieren lernen:

„Die Schülerinnen und Schüler sollen lernen, in altersadäquater Form Problemstellungen zu definieren, zu bearbeiten und ihren Erfolg dabei zu kontrollieren.“ ([BMUKK04])

„Verständnis für Phänomene, Fragen und Problemstellungen aus den Bereichen Mathematik, Naturwissenschaft und Technik bilden die Grundlage für die Orientierung in der modernen, von Technologien geprägten Gesellschaft.“ ([BMUKK04])

„… Als für die Analyse und Lösung von Problemen wesentliche Voraussetzungen sind Formalisierung, Modellbildung, Abstraktions- und Raumvorstellungsvermögen zu vermitteln.“ ([BMUKK04])

Ebenso hat sich die österreichische Bildungspolitik zum Ziel gesetzt, dass jeder Schüler und jeder Schülerin gewisse digitale Kompetenzen erlernt.

Folgende Kompetenzen im Bereich Algorithmen sollen Kinder in Österreich nach der Volksschule besitzen: ([@Eb13])

- „Ich kann einfache Anleitungen verstehen und ausführen."
- „Ich kann einfache Anleitungen erstellen."
- „Ich weiß, dass ein Computerprogramm entsteht, indem Anweisungen aneinander gereiht werden."

Kinder nach der 8. Schulstufen sollen im Bereich Algorithmen folgende digitalen Kompetenzen besitzen: ([@An13a])

- „Ich kann eindeutige Handlungsanleitungen (Algorithmen) nachvollziehen und ausführen."
- „Ich kann einfache Handlungsanleitungen (Algorithmen) verbal und schriftlich formulieren."
- „Ich kann einfache Algorithmen aus dem Alltag nennen und beschreiben."
- „Ich kann einfache Programme in einer geeigneten Entwicklungsumgebung erstellen."

Nach dem Pflichtfach Informatik in der 5. Klasse Gymnasium sollen Schülerinnen und Schüler im Bereich Algorithmen folgende digitalen Kompetenzen besitzen: ([@An13b])

- „Ich kann den Algorithmusbegriff erklären."
- „Ich kann einfache Algorithmen nachvollziehen und erklären."
- „Ich kann die Umsetzung von Algorithmen mit einem Computer erklären."
- „Ich kann einfache Aufgaben mit Mitteln der Informatik modellieren."
- „Ich kann einfache Algorithmen entwerfen, diese formal darstellen, implementieren und testen.
- „Ich kann an Hand von einfachen Beispielen die Korrektheit von Programmen bewerten."

Programmieren schult das schrittweise Vorgehen

Programmiercode muss Schritt für Schritt angegeben werden, damit der Computer ihn versteht. Ein Überspringen eines logischen Schrittes kann nicht stattfinden, wenn man ein lauffähiges Programm programmieren möchte.

4. Schwierigkeiten der ProgrammieranfängerInnen

[KMA04] nennen sechs Barrieren, die Programmieranfänger zu Beginn zu überwinden haben, dabei wird der Begriff „Interface“ verwendet, um jedes Element eines Programmiersystem bzw. Programmiersprache (Funktionen, Variable, Events, …) zu beschreiben:

Designbarriere

„I don‘t know what I want the computer to do …“

Designbarrieren sind die kognitiven Schwierigkeiten, die die Darstellung der Problemlösung mit sich bringt. Die Schwierigkeit des Problemlösens besteht darin, dass man nicht genau weiß, wie man die Lösung darstellen soll bzw. in einen Algorithmus formulieren soll. Diese Barriere ist oft nicht überwindbar, da viele Problemlösungen kaum visualisierbar sind.

Beispiel: Ein Programmieranfänger soll eine Liste mit Namen alphabetisch sortieren. Die beste Lösung für ihn ist es einfach solange die Namen zu verschieben, bis das gewünschte Ergebnis vorliegt. Aus dieser Vorgehensweise ist es schwierig einen Algorithmus zu entwickeln, da der Programmieranfänger nicht konkret angeben kann, welche Schritte der Algorithmus durchlaufen muss.

Auswahlbarriere

„I think I know what I want the computer to do, but I don’t know what to use …“

ProgrammieranfängerInnen wissen oft nicht welche Möglichkeiten und Hilfsprogramme die Programmiersprachen bieten und welche sie für ihre Problemlösung einsetzen können. Sie suchen zwar nach Hilfsprogrammen, wissen aber nicht

nach welchen Schlüsselwörter sie suchen sollen. Finden sie dann doch Informationen / Hilfsprogramme, haben sie oft Schwierigkeiten die Anleitung der Hilfsprogramme zu verstehen.

Beispiel: Ein Programmieranfänger bekommt die Aufgabe, dass er eine Alarmglocke, die zu einer bestimmten Zeit läuten soll. Die erste Schwierigkeit des Programmieranfängers ist es die Zeitfunktionen der Programmiersprache Basic zu finden. Hat er diese Zeitfunktionen gefunden, kann es sein, dass der Programmieranfänger Schwierigkeiten dabei hat, die Zeitfunktionen zu verstehen und anzuwenden.

Koordinationsbarriere

„I think I know what things to use, but I don't know how to make them work together …"

Kennen nun die ProgrammieranfängerInnen diese Hilfsfunktionen, Interfaces und Bibliotheken und verstehen sie auch, ist nun die nächste Schwierigkeit, dass die ProgrammieranfängerInnen dieses Angebot der Programmierumgebung auch einsetzen und miteinander kombinieren können, so dass ein komplexeres Programm entsteht.

Beispiel: Versteht ein Programmieranfänger das Konzept der Formulare in Visual Basic, kann es trotzdem sein, dass er beim Versuch ein Unterformular zu erstellen, das mit dem Hauptformular kommunizieren soll, auf Probleme stößt.

Benutzungsbarriere

„I think I know what to use, but i don't know how to use it …"

Wenn ProgrammieranfängerInnen beim Anwenden der richtigen Interfaces Probleme haben spricht man von Benutzungsbarrieren. ProgrammieranfängerInnen wissen welches Interface sie verwenden müssen, aber nicht wie sie es verwenden sollen bzw. welchen Effekt es hat das Interface zu verwenden.

Beispiel: Ein Programmieranfänger hat als Aufgabe ein Textfeld interaktiv zu gestalten. Der Anfänger kennt die Möglichkeit über die Dokumentation Informationen über die benötigte Funktion herauszufinden. Jedoch passiert es, dass er die Informationen falsch versteht und es führt dazu, dass er Funktion mit falschen Parametern aufruft. Dies bedeutet, dass er auf keinen syntaktischen Fehler, jedoch auf einen semantischen Fehler trifft.

Verstehensbarriere

„I thought I know how to use this, but it didn't do what I expected …"

Die Verstehensbarriere bezieht sich auf das richtige Interpretieren von Fehlermeldungen. Dabei ist es unwesentlich ob es sich um Kompilierfehler oder Laufzeitfehler handelt. Aufgrund von schwer verständlichen Fehlermeldungen fällt es ProgrammieranfängerInnen oft schwer herauszufinden welche Teile ihres Programms richtig und welche fehlerhaft sind.

Ebenso handelt es sich um eine Verstehensbarriere, wenn aufgrund einer Benutzungsbarriere semantische Fehler entstehen und der Lernende nicht weiß warum das Programm ein Fehlverhalten aufweist.

Beispiel: Ein Programmieranfänger verwendet in VB das Timer Objekt und soll damit ein Textfeld verändern. Jedoch nimmt er fälschlicherweise an, dass das Timer Objekt beim Starten des Programms zu zählen beginnt. Er wundert sich warum sich sein Textfeld nicht aktualisiert und sucht den Fehler in der Aktualisierungsfunktion und nicht in der Verwendung der Timer Objekts.

Informationsbarriere

„I think I know why it didn't do what I expected, but I don't know how to check …"

Falls ein beliebiger Fehler in einer Programmiersprache auftritt gibt die Programmierumgebung Hilfestellungen um den Fehler auszubessern. Wenn diese Hilfestellungen jedoch nicht verständlich genug formuliert sind, spricht man von einer Informationsbarriere. Außerdem handelt es sich um Informationsbarrieren, wenn

es schwer ist Informationen über das interne Verhalten des Programmes herauszufinden.

Beispiel: Ein Programmieranfänger verwendet um einen Fehler auszubessern die Debugger Funktion seiner Programmierumgebung, dabei verändert sich die Oberfläche des verwendeten Tools und er trifft dabei auf neue Probleme.

Zusätzlich zu den von [KMA04] aufgezeigten und eben erläuterten sechs Problemen nennt [Ko09] noch zwei Probleme, die Programmieranfänger betreffen können:

Probleme mit der Syntax

Die Syntax einer Programmiersprache ist das größte Problem für Programmieranfänger. In einer Studie wurden in den Jahren 2003 und 2004 insgesamt 19008 Probleme bei ca. 470 Programmieranfänger festgestellt und kategorisierst. Die meisten Fehler sind Syntaxfehler. (vgl. [Ga05] und [Ro06])

Unmotivierende Probleme / Aufgaben

Die ersten Programmieraufgaben sind häufig mathematische Berechnungen zu lösen, wie zum Beispiel Berechnung von Fibnoacci Zahlen, durchschnittliche Tagestemperaturen, Body Mass Index oder die Summe von Zahlen (vgl. [FC03]) Da aber Schüler und Schülerinnen mit grafischen Oberflächen, Animationen, Spiele, etc. vertraut sind (vgl. [La06]), wirkt es nicht motivierend, wenn Schüler und Schülerinnen eine lange Zeit nur Programme entwickeln, die mathematische Berechnung durchführen oder Zeichenketten manipulieren. (vgl. [FC04]) Ebenso kann ein Problem sein, dass den Schülern und SchülerInnen die mathematischen Kenntnisse zur Problemlösung fehlen. Dadurch entsteht ein weiteres Problem.

5. Visuelle Programmiersprachen

Visuelle Programmiersprachen sind Programmiersprachen, die grafische Elemente und Figuren verwenden um ein Programm zu entwickeln. Sie ermöglichen Entwicklern von Programmen die textuelle Syntax auszublenden um mit Hilfe einer Vielzahl von grafischen Elementen zu programmieren. Die Grafiken bzw. Icons werden bei visuellen Programmiersprachen als Eingabe, Aktivität, Verbindung und/oder als Ausgabe des Programmes verwendet. (vgl. [@Ja14])

In dieser Diplomarbeit werden die visuellen Programmiersprachen Scratch und Snap! verwendet, da sie momentan (als die Diplomarbeit geschrieben wurde), zwei der weitverbreitetsten visuellen Programmiersprachen, die auch im Informatikunterricht verwendet werden, sind.

5.1. Scratch

Scratch ist eine visuelle Programmiersprache, die von der Lifelong Kindergarten Group, am MIT Media Lab entwickelt wurde. Das Ziel von der Lifelong Kindergarten Group ist es, dass junge Leute (vor allem im Alter von 8 bis 16) erste Erfahrungen im Programmieren sammeln, indem sie interaktive und medienreiche Projekte realisieren. (vgl. [Ma10])

Programmiert wird bei Scratch, mit Hilfe von bunten Kommando – Bausteinen, die mittels Drag & Drop zusammengebaut werden um damit Figuren auf der Bühne zu steuern. Die Kommando – Kacheln sind färbig und in acht Kategorien (wie z.B. „Bewegung“, „Aussehen“, „Steuerung“, …) eingeteilt. (vgl. [Ma10])

Scratch läuft im Internetbrowser. Man hat die Möglichkeit seine Projekte offline zu speichern, oder gleich online zu stellen und der Scratch – Communitiy zu präsentieren.

5.2. Snap! Build your own Blocks

Möchte man mehr als Figuren animieren und grundlegende Informatik Konzepte, wie zum Beispiel Funktionen programmieren oder mit Listen arbeiten, muss man auf die Erweiterung „Snap! Build your own Blocks“ von Scratch zurückgreifen. Snap! Build your own Blocks sieht auf den ersten Blick genauso aus wie Scratch. Dennoch gibt es hier mehr Möglichkeiten, auf die die Entwickler von Scratch bewusst verzichtet haben, um Programmieranfänger und Programmieranfängerinnen nicht von der Fülle an Kommando-Blöcke abzuschrecken.

Scratch bildet nur die Basis im Bereich Aussehen und Verhalten für Snap!, denn Snap! bietet um einiges mehr: (vgl. [Mo13])

- Snap! kann man erweitern, indem man neue Kontrollstrukturen schafft
- Mittels Snap! kann man objektorientiert programmieren
- Listen, Sprites, Blöcke und alle anderen Datentypen (wie zum Beispiel: Text, Zahlen und Wahrheitswerte) sind in Snap! sind first-class:
    - sie können Variablen zugewiesen werden
    - sie können als Paramater in Funktionen verwendet werden
    - sie können als Ergebnisse von Funktionen verwendet werden
    - sie können als Inhalte einer Datenstruktur verwendet werden.

5.3. Visuelle Programmiersprachen minimieren die Schwierigkeiten

Viele dieser Anfangsschwierigkeiten, die im Kapitel 4 erläutert wurden, können mit visuellen Programmiersprachen minimiert werden.

Die **Designbarriere** kann mit visuellen Programmiersprachen nicht komplett beseitigt werden. Denn auch bei visuellen Programmiersprachen muss man lernen algorithmisch zu denken. Die Designbarriere besteht aber auch deswegen, weil man einen Lösungsweg nicht visualisieren kann und manche Lösungswege sehr

abstrakt sind. Mit Hilfe von visuellen Programmiersprachen kann man den Programmcode visualisieren und sofort Resultate sehen. In Scratch und Snap! lassen sich die Programmabläufe und die Belegungen der Variablen am Bildschirm anzeigen und verfolgen. Das Problem der Designbarriere wird sich nie vollständig beseitigen lassen. Denn gerade diese Überwindung bedeutet einen großen Fortschritt für den Lernenden. Wenn man einmal verstanden hat wie man sein Programm aufbauen will, ist die Syntax der jeweiligen Programmiersprache nebensächlich.

Die **Auswahlbarriere** kann mit Scratch und Snap! kaum entstehen, da nur eine gewisse, leicht überschaubare und gut strukturierte Anzahl an Hilfsprogrammen zur Verfügung steht. Nach kurzer Zeit kennt man alle Hilfsprogramme / Strukturen, die Scratch und Snap! zur Verfügung stehen. Daher wird auch die **Benutzungs-** und **Koordinationsbarriere** minimiert. Hilfsprogramme und Elemente von Scratch und Snap! sind einfach und eindeutig benannt. Es wird sofort klar welche Funktion sich hinter welchem Befehl versteckt.

Zusätzlich kann es nicht vorkommen, dass man beim Aufrufen einer Funktion falsche Parameter setzt, da die benötigten Parameter genau benannt sind. Einen genauen Vergleich von Scratch und Snap! zu anderen Programmiersprachen findet man im Kapitel 7. „Vergleich Scratch / Snap! mit schulüblichen Programmiersprachen“.

In Scratch / Snap! sind Syntaxfehler ausgeschlossen, da man die bunten Bausteine nur so zusammenbauen kann, dass keine Fehler entstehen. So können keine **Verstehensbarrieren** und **Informationsbarrieren** bezüglich einer falschen Syntax auftreten. Dennoch kann die Verstehensbarriere hinsichtlich Semantikfehler auftreten.

Um die anfängliche Motivationsschwäche zu beseitigen, kann man mittels Scratch und Snap! in wenigen Schritten ein kleines Spiel programmieren. Zum Beispiel: Ein Ball fliegt zufällig durch die Bühne und die typische Scratch Katze ist steuerbar. Wenn der Ball die Katze trifft miaut die Katze. Dieses kurze kleine Spiel ist einer Unterrichtseinheit programmiert und die Schüler und Schülerinnen

sehen sofort erste Erfolge und sind motiviert weiteres zu Programmieren. Ebenso kann man immer wieder zwischendurch mit den Schüler und Schülerinnen kleine Spiele programmieren, die zur Auflockerung dienen können. (z.B. Vier gewinnt oder Ping Pong)

Man kann also sagen, dass der Hauptgrund, wieso man mit visuellen Programmiersprachen programmieren lernen sollte, darin besteht, dass die Anfangsschwierigkeiten beseitigt werden.

5.4. Wo sind die Grenzen?

[Mo11] befragte Programmieranfänger aus den Naturwissenschaften des dritten Semesters, die mit Java und Scratch programmierten. Ein großer Teil findet, dass man mit Scratch nicht „richtig programmieren“ kann, da wichtige Strukturen fehlen. Der Wunsch „richtig zu programmieren“ tritt nicht nur bei Naturwissenschaftsstudenten auf, sondern ebenso bei Schüler und Schülerinnen.

Beide Programmiersprachen sind keine Produktionssysteme. Ein eigenständiges Programm, das man vermarktet und verkauft oder einfach nur außerhalb der Scratch und Snap! Community verwenden kann, kann man mit Scratch oder Snap! nicht erstellen. Die beiden Programmiersprachen haben das Ziel, den Bereich Algorithmen und Datenstrukturen Kindern, Jugendlichen und Erwachsenen näher zu bringen. Der Hauptanwendungsbereich dieser beiden Programmiersprachen liegt im Bereich der Lehre. (vgl. [Mo13])

Scratch und Snap! werden laufend weiterentwickelt und verbessert. Ebenso ist zu beachten, dass Scratch und Snap! nicht schnell sind. (vgl. [Mo13])

6. Konzepte / Strukturen der Programmierung

Grundlegende Konzepte der Algorithmen und Datenstrukturen können mit Hilfe von Scratch und Snap! erlernt werden.

Sequenz

Ein Programm ist eine Folge von unterschiedlichen Anweisungen, die nach der Reihe abgearbeitet werden. Der Computer kennt wenige einfache Anweisungen. Wenn man also ein komplexeres Problem lösen möchte, muss man diese Anweisungen kombinieren. (vgl. [Hr12])

Variablen

Variablen sind Namen für Orte im Speicher. Schülerinnen und Schüler können sich Variablen als Box im Speicher vorstellen, die einen Namen trägt und gewisse Werte annehmen können. (vgl. [Ma14])

Schleifen / Iteration

Eine Schleife ist eine Struktur, die Anweisungen mehrmals wiederholt, solange oder bis eine gewisse Bedingung erfüllt ist. Man unterscheidet dabei zwischen mehreren Arten von Schleifen: Kopfgesteuerte Schleifen, Fußgesteuerte Schleifen und Endlosschleifen. Diese drei Arten werden im nächsten Kapitel genauer erläutert.

Schüler und Schülerinnen sollen erkennen, dass Schleifen eine wesentliche Kürzung des Programmcodes darstellen und sie sinnvoll eingesetzt werden können, wenn ein Programmcode mehrmals wiederholt werden soll.

Manche Probleme lassen sich ausschließlich nur durch Schleifen lösen. Zum Beispiel muss man Schleifen verwenden, wenn die Anzahl der Wiederholungen eines Codeabschnittes von Benutzereingaben oder von einem Ergebnis einer Berechnung abhängig ist.

Bedingte Anweisungen und Verzweigungen / Alternative

Bedingte Anweisungen und Verzweigungen bieten die Möglichkeit den Verlauf des Programmcodes abhängig von Parametern zu machen. Das Programm wird dadurch nicht strikt von oben nach unten abgearbeitet, sondern enthält Abzweigungen die abhängig von Programmparametern verarbeitet werden.

Soll z.B. ein bestimmter Programmcode nur unter einer bestimmten Bedingung ausgeführt werden (z.B. ein Ergebnis einer Berechnung ist gerade), benötigt man bedingte Anweisungen und Verzweigungen um diesen Fall gezielt zu behandeln.

Methoden und Funktionen / Abstraktionen

Methoden und Funktionen kann man als kleine Unterprogramme verstehen, die Ergebnisse zurückliefern oder Objekte direkt verändern. Schüler und Schülerinnen sollen erkennen, dass mit Methoden und Funktionen der Programmcode modular und einfacher aufgebaut werden kann, wenn man Teilproblemlösungen in Funktionen bzw. Methoden auslagert. Ebenso sollen sie erkennen, dass Methoden und Funktionen als Vorlagen für Problemlösungen dienen können und mit Übergabeparameter zu speziellen Problemlösungen werden können. Zum Beispiel schreibt man eine Funktion, die ein Quadrat mit Länge a zeichnet. Für den Programmcode ist es irrelevant wie groß a ist. Erst mit dem Aufruf und der Spezifikation des Parameters a wird ein konkretes Quadrat gezeichnet.

Listen

Schüler und Schülerinnen können sich Listen in Scratch / Snap! genauso wie Listen im alltäglichen Leben vorstellen. Dennoch müssen sie verstehen, wie man mit diesen Listen arbeiten kann. Schwierig ist es, geeignete Algorithmen für Listen zu entwickeln.

Rekursion

Wenn sich eine Funktion selbst wieder aufruft, spricht man von einem rekursiven Funktionsaufruf. Schüler und Schülerinnen müssen verstehen, dass es hierbei verschiedene Ebenen gibt und dass eine Abbruchbedingung gegeben sein muss.

Objektorientierung

Ebenso kann die Objektorientierung mittels Snap! realisiert werden. Dazu möchte ich auf die Diplomarbeit von Bernd Kurzmann, „Einstieg in das objektorientiere Programmieren mit Hilfe von Lernumgebungen, TU Wien, 2013“ verweisen.

7. Vergleich Scratch / Snap! mit Java

7.1. Variablen

Elementare Datentypen

In Scratch und Snap! gibt es drei elementare Datentypen: (vgl. [Ma14])

- Boolean

Eine Boolean Variable kann die Werte „falsch“ oder „wahr“ annehmen. Dieser Datentyp wird vor allem bei Bedingungen verwendet.

- Number

Scratch unterscheidet nicht zwischen Dezimalzahlen und ganze Zahlen.

- String

Eine String Variable kann die Werte von Wörtern, einzelner Buchstaben, Zahlen und Zeichen enthalten.

Hingegen gibt es in Java mehr elementare Datentypen, wie zum Beispiel byte, short, int, long, char, boolean, float, double und String.

Verwendung

Scratch und Snap! Programmierer müssen nicht auf den Datentyp beim Erstellen einer Variablen achten. Der Datentyp von Variablen wird in Scratch bzw. Snap! passend interpretiert um sinnvoll damit arbeiten zu können. Zum Beispiel: Wenn man den String „ 123“ mit 5 multiplizieren möchte, wandelt Scratch bzw. Snap! den String „ 123“ in die Zahl 123 um.

Dennoch ist zu beachten, dass manche Strukturen von Scratch/Snap! gewisse Datentypen verlangen. Das sieht man daran, dass die Platzhalter der einzelnen Parameter unterschiedliche Formen haben.

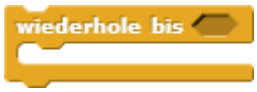
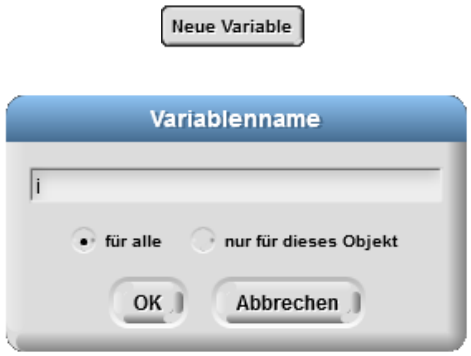
	In runde Platzhalter können nur Zahlen eingefügt werden. Variablen mit dem Datentyp Number (z.B. ) sind rund.
	In sechseckige Platzhalter können nur Variablen und Ausdrücke hineingezogen werden, die den Datentyp boolean haben. Eine boolean Konstante hat auch die Form eines Sechsecks (z.B. )
	In rechteckige Platzhalter passen alle Typen.

Tabelle 1: Datentypen Scratch / Snap!

Die Erstellung einer Variablen in Scratch ist einfacher als in Java, da in Scratch nicht auf den Datentyp geachtet werden muss. Ebenso ist es nicht textbasierend, sondern mit zwei Klicks erledigt. Erst nachdem man einen Wert gesetzt hat, ist klar, welcher Datentyp die Variable hat.

Die Erhöhung einer Variable in Java ist nicht intuitiv und ein Schüler bzw. eine Schülerinnen könnte `i = i+1;` auch falsch interpretieren, wenn sie bzw. er dies als nicht gültige mathematische Gleichung auffasst. Diese Fehlinterpretation kann in Scratch nicht stattfinden, da eine Erhöhung einer Variablen mittels  erfolgt.

	Scratch	Java
Erstellung		`int i;`
Wert setzen		`i = 0;`

Wert erhöhen		`i = i+1;`
		oder
		`i++;`

Tabelle 2: Vergleich Variablen Scratch / Java

7.2. Schleifen

In Scratch gibt es eine Zählschleife, eine kopfgesteuerte Schleife und eine Endlosschleife. Dennoch kann man die anderen Schleifenarten programmieren.

Die einzelnen Schleifarten von Scratch sollen anhand folgender Beispiele mit Java verglichen werden.

Beispiel: Gib die Zahlen 1 bis 10 in aufsteigender Reihenfolge aus.

Zählschleife

Scratch	Java

	`for(int i = 1; i<=10; i++){`
	`  System.out.println(i);`
	`}`

Tabelle 3: Vergleich Zählschleife Scratch / Java

Bei Scratch sind die Befehle leicht und ohne Vorkenntnisse zu verstehen. Mit ein paar Klicks findet man die Struktur „*wiederhole x mal*“. Hingegen bei Java muss man zuerst herausfinden, dass das Schlüsselwort *„for“* eine Schleife beginnt, und die einzelnen Elemente in der Klammer verstehen. Dies benötigt mehr Zeit und kann von Anfängern ohne Vorkenntnisse nicht so leicht verstanden werden.

Ebenso ist zu beachten, dass man bei Java darauf Acht geben muss, wie man die geschwungenen Klammern { } setzt. Die { - Klammer kennzeichnet einen Ausführungsblock und symbolisiert den Start der Schleife und die } – Klammer symbolisiert das Ende des Ausführungsblocks und somit der Schleife. Der Code innerhalb der geschwungen Klammern wird demnach mehrmals ausgeführt. Die Klammersetzung führt oft zu Syntaxfehlern und, vor allem bei einem schlechten Programmierstil (keine Einrückungen, etc.), zu Unübersichtlichkeit und erhöht die Fehleranfälligkeit.

Bei Scratch werden diese Verständnisprobleme und mögliche Fehlerquellen stark minimiert. Der Schleifenrumpf umfasst alle darin befindlichen Befehle, passt seine Größe selbstständig eventuellen neuen Befehlen an und erzwingt damit eine einheitliche und leicht verständliche Struktur des Programms. Lernende können keine Klammern, Semikolons o.Ä. vergessen und müssen sich nicht selbst um die Lesbarkeit ihres Codes kümmern.

Kopfgesteuerte Schleife

Scratch	Java

	`int i = 1;`
	`while(i<=10){`
	`  System.out.println(i++);`
	`}`

Tabelle 4: Vergleich Kopfgesteuerte Schleife Scratch / Java

In Java muss das Schlüsselwort *while* gefunden werden und verstanden werden, welche Bedingung *while* benötigt um zu funktionieren. Der größte Unterschied zu Scratch ist, dass in Java die Schleife solange ausgeführt wird, **solange** die Bedingung erfüllt ist. Hingegen bei Scratch wird die Schleife solange ausgeführt, **bis** die Bedingung erfüllt ist.

Endlosschleife

Scratch	Java

	`int i = 1;`
	`while(true){`
	`System.out.println(i++);`
	`if(i > 10) break;`
	`}`

Tabelle 5: Vergleich Endlosschleife Scratch / Java

Eine Endlosschleife als Struktur gibt es in Java nicht, dennoch kann man sich eine Endlosschleife selbst programmieren. In diesem Beispiel wird die Endlosschleife abgebrochen, wenn die Variable i größer als 10 ist. In Scratch gibt es die Endlosschliefe „wiederhole fortlaufend“. Sie wird zum Beispiel verwendet, wenn ein Ball bis zum Ende des Spieles sich bewegen soll.

Anhand dieses Beispiels sieht man nicht nur den Unterschied zwischen den beiden Schleifen, sondern auch, wie man eine Schleife abbrechen kann. In Java muss man wieder das Schlüsselwort „break“ kennen und anwenden können.

7.3. Bedingte Anweisungen und Verzweigungen

Vergleichs – Operatoren

	Scratch	Java
kleiner		<
gleich		==
größer		>
nicht		!
oder		||
und		&&

Tabelle 6: Vergleich Vergleichsoperatoren Scratch / Java

In Java muss man die Vergleichsoperatoren kennen um sie anwenden zu können. Vor allem sind die Funktionen der Operatoren „gleich“, „nicht“, „oder“ und „und“ nicht intuitiv erkennbar.

In Scratch gibt es die Vergleichsoperatoren „kleiner – gleich“ und „größer – gleich“ nicht standardmäßig. Diese sind in Java einfacher umzusetzen.

Scratch	Java
	`a <= b`
	`a >= b`

Tabelle 7: Vergleich kleiner-gleich, größer-gleich Scratch / Java

An diesem Beispiel sieht man auch, dass man in Scratch Operatoren verschachteln kann.

Bedingte Anweisungen

Eine bedingte Anweisung ist eine Anweisung, die nur ausgeführt wird wenn eine Bedingung erfüllt ist. So eine bedingte Anweisung haben wir zum Beispiel schon bei der Endlosschleife verwendet um die Schleife abzubrechen. Bedingte Anweisungen werden verwendet um auf spezielle Variablenstellungen / Zustände des Programms gezielt einzugehen.

Beispiel: Wenn der Wert der Variable kleiner ist als 12, dann gib aus, dass die Zahl zu klein ist.

Scratch	Java

	`int a = 10;`
	`if(a < 12){`
	`System.out.println("Zahl zu klein!");`
	`}`

Tabelle 8: Vergleich Bedingte Anweisung Scratch / Java

Wie man sieht muss in Java wieder eine, für Programmieranfänger komplizierte, Anordnung von Klammern verwendet werden um eine Bedingung zu kennzeichnen. In Scratch hingegen ist es, aufgrund der automatischen Strukturierung, wesentlich einfacher den Ablauf des Programms nachzuvollziehen.

Verzweigung

Eine Verzweigung ist eine Erweiterung einer bedingten Anweisung. Die Erweiterung besteht darin, dass man die Möglichkeit eines „sonst“-Zweiges hat, falls die bedingte Anweisung nicht ausgeführt wird.

Beispiel: Wenn der Wert der Variable kleiner ist als 12, dann gib aus, dass die Zahl zu klein ist, ansonsten sage, dass die Zahl die richtige Größe besitzt.

Scratch	Java

	`int a = 10;`
	`if(a < 12){`
	`System.out.println("Zahl zu klein!");`
	`}else{`
	`System.out.println("Zahl ist gross genug!");`
	`}`

Tabelle 9: Vergleich Verzweigung Scratch / Java

Bei diesem einfachen Beispiel ist der Vorteil der Darstellung von Scratch gegenüber Java nicht sehr groß. Wenn jedoch komplexere Anweisungen in den jeweiligen Zweigen benötigt werden, wird bei Ungeübten der Code in Java sehr schnell unübersichtlich. Hingegen bleibt bei Scratch, aufgrund der farblichen Hervorhebungen und automatischen Strukturierung der Code immer leicht lesbar.

7.4. Methoden und Funktionen

Ohne Rückgabewert und ohne Parameter

Wenn man eine Funktion / Methode schreiben möchte, die keinen Rückgabewert liefert, kann man Scratch verwenden, ansonsten muss man auf Snap! zurückgreifen.

Folgendes muss man vornehmen, um in Scratch eine neue Funktion / Methode zu erstellen. In Scratch heißen Funktionen / Methoden „Blöcke“.

Um einen neuen Block erstellen zu können, geht man auf den Punkt „Weitere Blöcke“ und anschließend auf „Neuer Block“. Anschließend kann man dem Block ein Namen geben.

Anhand eines Beispiels wird die Erstellung eines Blockes genauer erläutert.

Beispiel: Zeichne ein Quadrat mit Länge 10

1. **Schritt**

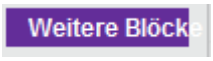


Abbildung 1: Weitere Blöcke

2. **Schritt: Neuen Block erstellen**

Abbildung 2: Neuer Block

3. **Schritt: Namen vergeben bzw. Übergabeparameter definieren**

Abbildung 3: New Block

4. **Schritt: Block definieren**

Der Block wird nun im Skriptenbereich angezeigt und man kann diesen definieren.

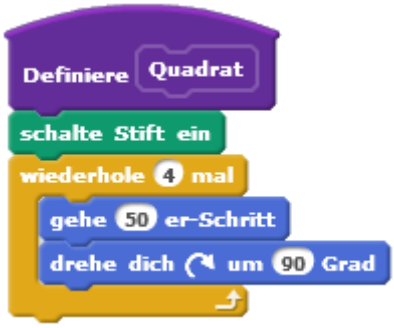


Abbildung 4: Block definieren

Anhand des Beispiels sieht man auch wieder die sinnvolle Nutzung von Schleifen.

5. **Schritt: Block einsetzen**

Der Block „Quadrat“ ist nun wie alle anderen Blöcke im Bereich „Skripten“ zu finden und kann ab jetzt genauso verwendet werden wie bereits vorhandene Blöcke.

Abbildung 5: Block verwenden

Ohne Rückgabewert und mit Parameter

Beispiel: Zeichne ein Quadrat mit Länge a.

Die ersten zwei Schritte sind ident, wie bei dem vorherigen Beispiel „Ohne Rückgabewert und ohne Parameter“. Ab Schritt 3 gibt es kleine Änderungen:

3. **Schritt: Namen vergeben bzw. Übergabeparameter definieren**

Um Übergabeparameter in seinem neuen Block verwenden zu können, muss man auf Einstellungen klicken und erhält eine Auswahl an möglichen Arten der Übergabeparameter.

Abbildung 6: Übergabeparameter

Es kann eine Zahl, ein Text oder ein boolescher Wert übergeben werden. Weiters kann man einen Beschriftungstext angeben, um die Parameter zu beschriften. In unserem Beispiel wird ein Zahlenwert übergeben, den wir mit „mit Laenge:“ beschriften und die Variable für den Zahlenwert nennen wir „laenge“

Abbildung 7: Übergabeparameter definieren

4. Schritt: Block definieren

Der Block erscheint nun wieder in der Bühne und man kann ihn nun definieren. Den Übergabeparameter den wir „laenge“ genannt haben, kann man nun als Variable verwenden.

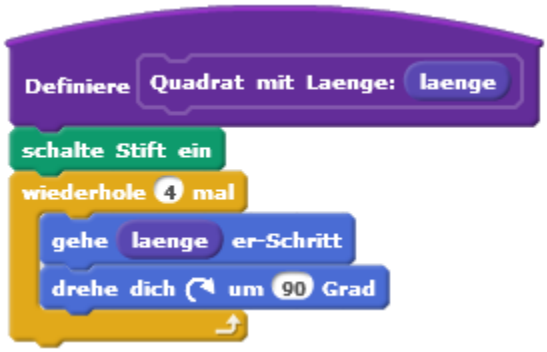


Abbildung 8: Block definieren

5. Schritt: Block verwenden

Den Block findet man nun wieder im Bereich „Skripten“. Hier sieht man deutlich, wie wichtig es war, den Beschriftungstext einzufügen, denn ansonsten hätte man beim Anwenden des Blockes eventuell nicht mehr gewusst welche Bedeutung der Parameter hat.

Abbildung 9: Block verwenden

Mit Rückgabewert

Sobald man Funktionen mit Rückgabewerten programmieren möchte, muss man auf Snap! zurückgreifen.

In Snap! ist das Definieren von Funktionen ähnlich aufgebaut wie in Scratch. Dennoch gibt es paar wenige Änderungen:

1. Blöcke kann man nun zu den einzelnen Rubriken zuordnen und sie werden dort wieder gefunden

2. Blöcke können nun Zahlenwerte, Text oder boolesche Werte zurückgeben.
3. Die Übergabeparameter können verschiedene Datentypen besitzen. So hat man die Möglichkeit ganze Objekte, Befehle oder Liste zu übergeben. Dadurch wird eine Vielzahl neuer, komplexerer Programme möglich.

Zahl oder Text als Rückgabewert

Beispiel: Ermittle das Maximum zweier Zahlen.

Snap!:

1. **Schritt**

   Auswählen in welche Kategorie der neue Block erstellt werden soll und bestimmen ob es sich:

   - um einen **Befehl** (kein Rückgabewert)
   - um eine **Funktion** (Rückgabewert: Zahl oder Text)
   - um ein **Prädikat** (boolescher Rückgabewert)

   handelt.

Abbildung 10: Neuer Block Maximum

2. Schritt

Übergabeparameter definiert man indem man auf das Plus neben dem Funktionsname klickt. Anschließend wählt man den Datentyp des Übergabeparameters aus. Abschließend gibt man diesem Übergabeparameter einen Namen.

Abbildung 11: Übergabeparameter definieren

3. Schritt:

Nun kann der erstellte Block definiert werden. Die Übergabeparameter stehen nun, äquivalent zu Scratch, als Variablen zur Verfügung.

Abbildung 12: Block definieren

4. Schritt

Den erstellten Block findet man nun unter der Kategorie, die wir zu Beginn

zugeordnet haben. Nun kann der Block so wie die anderen Standardblöcke verwendet werden.

Abbildung 13: Block verwenden

Java:

In Java gibt es keinen so großen Unterschied zwischen Methoden mit Rückgabewert und jenen ohne Rückgabewert. Um in Java eine Methode zu erstellen sind mit Sicherheit weniger Klicks erforderlich als in Scratch / Snap!. Jedoch führt dieser minimalistische Ansatz in Java dazu, dass die Erstellung einer Funktion alles andere als intuitiv ist.

In Scratch / Snap! wird man Schritt für Schritt seinem Ziel näher gebracht und man hat als Lernender somit die Chance ohne viele Vorkenntnisse eine Funktion zu erstellen. In Java muss man die Syntax wissen, ansonsten hat man keine Chance eine Methode zu erstellen. Die Syntax wird anhand des bereits oben in Snap! implementieren Beispiels einer Maximum-Funktion erklärt:

```
public static int max(int a, int b)        ← Methodenkopf
{                                           ← Anfang Methodenrumpf
        if(a < b) return b;                 ← Anweisung
        return a;                           ← Anweisung
}                                           ← Ende Methodenrumpf
```

Der Methodenkopf hat eine strikt definierte Syntax. Allgemein kann man einen Methodenkopf in Java wie folgt einteilen:

Modifikatoren	Rückgabedatentyp	Methodenname	(	Datentyp Argument 1	Datentyp Argument 2	)
public static	int	max	(	int a	int b	)

Wenn man den Vergleich mit Snap! zieht, sieht man, dass genau diese Informationen (Rückgabewert, Methodenname, Übergabeparameter,…) bei der Erstellung einer Funktion in Snap! Schritt für Schritt abgefragt werden. In Java muss man diese Information in einer Zeile liefern.

7.5. Listen

In Snap! können Listen beliebige Länge annehmen und sind auch nicht auf einen Datentyp gebunden. In Java wird zwischen verketteten Listen und Arrays unterschieden. In weiterer Folge werden Listen in Snap! mit Arrays in Java verglichen, da beide die Basisstruktur für die Zusammenfassung mehrerer Variablen darstellen.

Ein grundlegender Nachteil von Arrays in Java ist, dass jedes Element des Arrays vom gleichen Datentyp sein muss. Zusätzlich muss die Länge des Arrays zum Zeitpunkt des Kompilierens bekannt sein und kann nicht dynamisch zur Laufzeit verändert werden.

Als weiteren Vorteil bietet Snap! bereits mehrere Funktionen, die die Arbeit mit Listen vereinfachen. Diese Funktionen müssen zuvor in Java selbst geschrieben werden oder eine passende Bibliothek eingebunden werden. Dies stellt wiederum für ProgrammieranfängerInnen eine weitere Hürde dar.

Beim Zugriff auf Elemente der Liste gibt es erneut einen Unterschied zwischen Snap! und Java. Elemente einer Liste haben immer einen eindeutigen Index (=fortlaufende Nummerierung). Dem ersten Element einer Liste in Snap! wird der Index 1 zugeordnet. Hingegen wird dem ersten Element eines Arrays in Java der Index 0 zugeordnet. Aufgrund dieser Indexverschiebung haben ProgrammieranfängerInnen oftmals Probleme beim Ansprechen der Listenelemente.

Greift man in Snap! auf einen Index zu, den es nicht gibt, so wird der Wert 0 zurückgeliefert. Es kommt aber nicht, so wie in Java, zu einem Fehler und das Programm wird nicht abgebrochen.

	Snap!	Java
Erstellung		`int[] zahlen ={1,4,7,8,9};`
Werte hinzufügen		-
Auf Werte zugreifen		`int wert = zahlen[1];`
Länge von Liste		`zahlen.length;`
Enthält die Liste dieses Element		-
Element ersetzen		-

Tabelle 10: Vergleich Liste/Array Snap! / Java

7.6. Sonstiges

Ausgabe

Die Ausgabe in Scratch ist auf die Sprechblase begrenzt. Hingegen gibt es in Java eine Vielzahl an unterschiedlichen Ausgabemöglichkeiten, wie z.B. die Konsole, Textdatei, GUI-Elemente, etc.

Scratch	Java
	`System.out.println("Hello");`

Tabelle 11: Vergleich Ausgabe Scratch / Java

Eingabe

In Java gibt es mehrere Varianten Benutzereingaben zu programmieren: (vgl. [UI03])

- *System.in* lässt sich als Eingabestrom nutzen. Dieser liest aber nur Bytes ein.
- Die Klasse *Console* kann man für Ausgaben und Eingaben verwenden. Sie ist aber nur praktisch wenn passwortgeschützte Eingaben notwendig sind.
- Über einen grafischen Eingabedialog *JOptionPane* kann ebenso eine Benutzereingabe erfolgen.

Beispiel: Benutzer wird nach dem Namen gefragt.

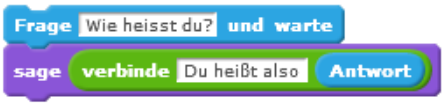
Scratch	Java
	`String s = javax.swing.JOption-Pane.showInputDialog("Wie heisst du?");` `System.out.println("Du heisst also"+s);`

Tabelle 12: Vergleich Eingabe Scratch / Java

Dieses Beispiel spiegelt eindrucksvoll wieder wie einfach und intuitiv Scratch an manche Bereiche der Programmierung, im Vergleich zu Java, herangeht. Das damit Anfangsbarrieren wie die Auswahlbarriere, Designbarriere, Informationsbarriere wesentlich minimiert werden, liegt offensichtlich auf der Hand.

Ein Programmieranfänger muss zuerst herausfinden, dass in Java mehrere Möglichkeiten zur Benutzereingabe zur Verfügung stehen und anschließend die richtige Möglichkeit auswählen und auch richtig anwenden.

Solange die Benutzereingabe ein Text ist, kann die gezeigte Variante in Java verwendet werden. Sobald die Eingabe einer Zahl erforderlich ist, mit der weitergearbeitet wird, wird es in Java komplizierter. Denn die Eingabe muss in eine Zahl umgewandelt werden. In Scratch muss hingegen nichts Weiteres beachtet werden, da die Datentypen automatisch angepasst werden.

Beispiel: Multipliziere die Eingabe mit 5.

Scratch	Java

	`String s = javax.swing.JOptionPane.showInput-Dialog("Gib eine Zahl ein:");`
	`int i = Integer.parseInt(s);`
	`System.out.println(5*i);`

Tabelle 13: Vergleich Eingabe multiplizieren Scratch/Java

Mit der Zeile `int i = Integer.parseInt(s);` zwingt man Java dazu die Eingabe als eine ganze Zahl zu interpretieren. Falls diese Konversation schief geht, z.B. aufgrund einer ungültigen („abc“) Benutzereingabe, kommt zu einem Laufzeitfehler und das Programm bricht ab.

Man müsste eine aufwändige Absicherung der Benutzereingabe implementieren wenn man das Programm stabiler programmieren möchte. Scratch hingegegen nimmt einem diese Probleme ab und ermöglicht eine komfortable und sichere Benutzereingabe.

Zufallszahlen

In Java ist nur eine reelle Zufallszahl zwischen 0 und 1 (ohne 1) möglich. Um Zufallszahlen in einem beliebigen Intervall zu bekommen, muss man folgendes anwenden:

Beispiel: Zufallszahl zwischen 5 und 15.

Scratch	Java
	`int i = (int) (Math.random()*11+5);`

Tabelle 14: Vergleich Zufallszahlen Scratch / Java

Ohne ein gewisses rechnerisches Geschick kommt man in Java also nicht aus.

Modulorechnung

Scratch	Java

	`int i = 10 % 5;`

Tabelle 15: Vergleich Modulorechnung Scratch / Java

Wenn ein Schüler das Zeichen „%“ für Modulo-Rechnung in Java weiß, dann ist die Anwendung ähnlich jener in Scratch.

Syntax

Das folgende Beispiel zeigt wie einfach die Syntax von Scratch im Vergleich zu Java sein kann:

Beispiel: Stoppuhr

Bei diesem Beispiel soll gestoppt werden wie lange ein Benutzer braucht um die Eingabe zu tätigen.

Scratch

Java

```
long startZeit = System.currentTimeMillis();
String s = javax.swing.JOptionPane.showInputDialog("5*3 =");
long endZeit = System.currentTimeMillis();
System.out.println("Du hast " + (endZeit - startZeit)/1000 +
"Sek. benoetigt.");
```

Tabelle 16: Vergleich Stoppuhr Scratch / Java

8. Aufgaben[1]

In diesem Kapitel werden Unterrichtsbeispiele aufgeführt, die sich zu einem oder mehreren Konzepten der Programmierung zuordnen lassen.

Diese Unterrichtsbeispiele sollen als Anleitung für eine strukturierte, aufbauende Einführung in das Programmieren mit Scratch/Snap! dienen. Dabei wurden die Aufgaben in Konzepte der Programmierung (z.B. Schleifen, bedingte Anweisung,…) gruppiert.

Ziel ist es die einzelnen Konzepte anhand von praktischen Aufgaben zu erlernen. Vor allem werden die Kompetenzen des Kompetenzmodelles Informatik 5. Klasse als Ziele dieser Aufgaben gesehen: ([@An13b])

- „Ich kann den Algorithmusbegriff erklären."
- „Ich kann einfache Algorithmen nachvollziehen und erklären."
- „Ich kann die Umsetzung von Algorithmen mit einem Computer erklären."
- „Ich kann einfache Aufgaben mit Mitteln der Informatik modellieren."
- „Ich kann einfache Algorithmen entwerfen, diese formal darstellen, implementieren und testen."
- „Ich kann an Hand von einfachen Beispielen die Korrektheit von Programmen bewerten."

Es werden die Programme aufbauend entwickelt. Das bedeutet, ein Konzept wird erst dann erlernt, wenn alle Konzepte, die man dafür benötigt, zuvor gelernt wurden. Beispiel: Man benötigt für die Verwendung von bedingten Anweisungen ein Verständnis über Variablen, aber um das Konzept der Variablen zu verstehen benötigt man keine Kenntnis über bedingte Anweisungen. Deswegen werden Variablen vor Verzweigungen erlernt. Bei Schleifen benötigt man zu Beginn nicht

[1] Teilweise wurden Aufgaben von Büchern übernommen die nicht für Scratch oder Snap! bestimmt sind. Die Aufgaben wurden auf Scratch oder Snap! angepasst, dennoch bleibt der Sinn der Aufgabe derselbe. (zum Beispiel wurde statt der Schildkröte in LOGO die Katze in Scratch verwendet)

das Konzept der Variablen. Dennoch wird das Konzept der Variablen irgendwann bei den Schleifen benötigt. Deswegen werden Variablen vor den Schleifen erlernt.

Dieser Grundidee folgend wird zu Beginn das strukturierte Vorgehen erlernt. Wichtig ist es, dass die SchülerInnen und Schülerinnen verstehen, dass diese „Schritt für Schritt Anweisungen“ beim Programmieren unerlässlich sind.

Variablen werden als zweites Konzept erlernt, da kein anderes Konzept Voraussetzung für das Erlernen ist. Ebenso kennen Schüler und Schülerinnen das Konzept der Variablen aus dem Mathematikunterricht. Ein weiterer Grund ist, dass die Komplexität der Programme schnell ansteigt und die Verwendung von Variablen die Lesbarkeit erhöhen kann.

Anschließend werden Aufgaben erstellt bei denen Schüler und Schülerinnen erkennen sollen, dass Programmteile öfters wiederholt werden. Ab diesem Zeitpunkt werden die Schleifen eingeführt.

Als nächstes Konzept werden die bedingten Anweisungen erlernt. Mit ihnen vergrößern sich die möglichen Aufgabenstellungen und es wird das Konzept der Variablen angewendet und vertieft.

Funktionen und Methoden werden erlernt, damit die Schüler und Schülerinnen den modularen Aufbau eines Programmes kennen lernen und Probleme in Teilprobleme lösen können. Es werden alle anderen Konzepte zuvor gelernt, da diese Konzepte bei Funktionen und Methoden Anwendung finden.

Viele Funktionen und Methoden kann man auf Listen anwenden. Deswegen werden Listen nach den Funktionen/Methoden behandelt.

Da Rekursion ein schwieriges Konzept der Programmierung ist und Schüler und Schülerinnen schon Erfahrungen mit dem Programmieren gesammelt haben sollten, bevor sie das Konzept der Rekursion lernen, wird dieses Konzept am Ende behandelt.

Mit dieser Einteilung wird ein „Hin und Her“ beim Erlernen von Konzepten vermeiden.

Zusätzlich werden passende Aufgaben zu Einheiten zusammengefasst. Jede Einheit besteht aus mehreren Aufgaben, die auf einander aufbauend sind. Ziel jeder Einheit ist es mehrere Konzepte der Programmierung anhand eines größeren Beispiels zu lernen. Dabei erkennt der Schüler wie wichtig es ist ein komplexes Problem in kleinere Teilprobleme/Aufgaben zu gliedern.

Die Aufgaben wurden wie folgt zu Einheiten zusammengefasst:

Einheit 1 (Spiel: Fang den Ball): Aufgabe 1, Aufgabe 4, Aufgabe 12
Einheit 2 (Quadrate zeichnen): Aufgabe 2, Aufgabe 7, Aufgabe 17
Einheit 3 (Zinsrechnung): Aufgabe 5, Aufgabe 9, Aufgabe 19, Aufgabe 22, Aufgabe 25
Einheit 4 (Monate): Aufgabe 13, Aufgabe 26

8.1. Strukturierte Vorgehensweise

Aufgabe 1

Spiel: Fang den Ball 1

Lass die Katze 100 Schritte nach vorne gehen und anschließend 100 Schritte rückwärtsgehen.

Lernziel

Schüler und Schülerinnen sollen

- Scratch kennen lernen.
- wissen, dass die Katze in verschiedene Richtungen gehen kann.

Möglicher Lösungsweg

Abbildung 14: Lösungsweg Aufgabe 1

Bemerkungen

Den Drehtyp muss man auf „links-rechts“ setzen, ansonsten würde sich die Katze beim Drehen auf den Kopf stellen.

Aufgabe 2

Quadrat 1

Zeichne ein Quadrat mit Länge 100.

Lernziel

Schüler und Schülerinnen sollen

- strukturiert vorgehen können.
- mit Drehungen umgehen können.

Möglicher Lösungsweg

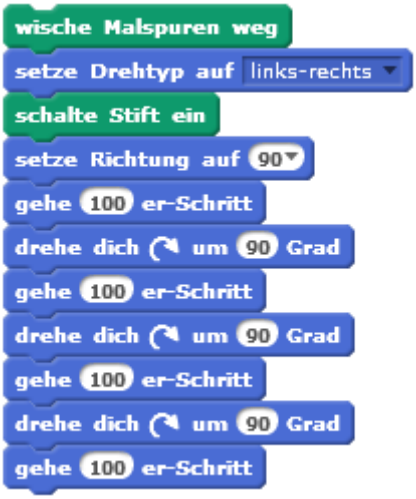


Abbildung 15: Lösungsweg Aufgabe 2

Bemerkung

Hier könnte man bereits mit guten SchülerInnen erarbeiten, wie das Programm kürzer programmiert werden kann. Sie könnten selbst die Schleifen entdecken.

Aufgabe 3

Schreibe Programme, die die folgenden Bilder zeichnen. Bei allen Bildern darfst du die Startposition der Katze bezüglich des zu zeichnenden Bildes selbst wählen. (vgl. [Hr12])

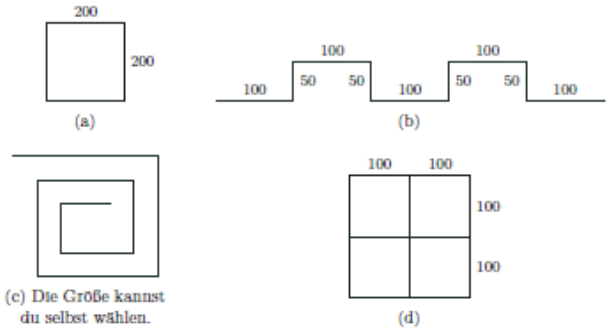


Abbildung 16: Bild aus [Hr12]

Lernziel

Schüler und Schülerinnen sollen

- strukturiert vorgehen können.
- mit Drehungen umgehen können.

Möglicher Lösungsweg

Es wird Beispiel (b) (rechts) vorgezeigt.

Abbildung 17: Lösungsweg Aufgabe 3

Bemerkung

Auch hier könnte der Programmcode verkürzt werden, wenn die SchülerInnen das Konzept der Schleife bereits erlernt und verstanden haben. Bessere SchülerInnen können bereits hier mit Schleifern experimentieren.

Alternativ könnte man den Code den Schüler und Schülerinnen bereitstellen und sie müssen per Hand auf ein Blatt Papier die Zeichnung zeichnen.

Aufgabe 4

Spiel: Fang den Ball 2

Steuere die Katze mittels der Tasten „oben“, „unten“, „rechts“ und „links“. Erweitere dein Programm sodass es aussieht als ob die Katze ihre Füße bewegen würde. (Hinweis: Kostüme)

Lernziel

Schüler und Schülerinnen sollen

- den Befehlsregister durchstöbern und dadurch kennen lernen.
- lernen, wie die Katze durch die Tastatur gesteuert werden kann.
- die Bühne als Koordinatensystem interpretieren können.
- verschiedene Kostüme kennenlernen und wissen wie man ein Kostüm wechselt und einsetzt.

Möglicher Lösungsweg

Abbildung 18: Lösungsweg Aufgabe 4

8.2. Variablen

Aufgabe 5

Zinsrechnung 1

Der Benutzer möchte Geld sparen und möchte es auf ein Sparbuch legen. Er weiß den Zinssatz und möchte nun wissen, wie groß sein Endkapital sein wird Es handelt sich um eine jährliche Verzinsung ohne Zinseszinsen. (lineare Verzinsung) Schreibe ein Programm, das den Benutzer nach dem Startkapital, nach dem Zinssatz und nach der Laufzeigt fragt und berechne das Endkapital.

Lernziel

Schüler und Schülerinnen sollen

- die Zinsrechnung wiederholen.
- Benutzereingaben anwenden und verarbeiten können.
- Variablen erstellen können.
- Benutzereingaben in Variablen speichern können.
- Berechnungen mit Variablen durchführen können.

Bemerkung

Hier wäre es vorteilhaft, wenn die Benutzereingaben überprüft werden. Eine Überprüfung findet man in Aufgabe 12.

Möglicher Lösungsweg

Abbildung 19: Lösungsweg Aufgabe 5

Aufgabe 6

Zwei Variablen a und b haben gewisse Werte. Programmiere ein Programm, das b den Wert von a zuweist und a den Wert von b zuweist. Die Werte der Variablen werden also vertauscht.

Lernziel

Schüler und Schülerinnen sollen

- Variablen erstellen können.
- Variablen Werte zuweisen können und Werte auslesen können.
- erkennen, dass Hilfsvariablen notwendig sind.

Möglicher Lösungsweg

Abbildung 20: Lösungsweg Aufgabe 6

8.3. Schleifen

Aufgabe 7

Quadrat 2

Vereinfache den Programmcode der Aufgabe 2, indem du Schleifen benutzt.

Problemstellung der Aufgabe 2:

Zeichne ein Quadrat mit Länge 100.

Lernziel

Schüler und Schülerinnen sollen

- einfache Schleifen verwenden können.
- Programme mittels Schleifen vereinfachen können.

Möglicher Lösungsweg

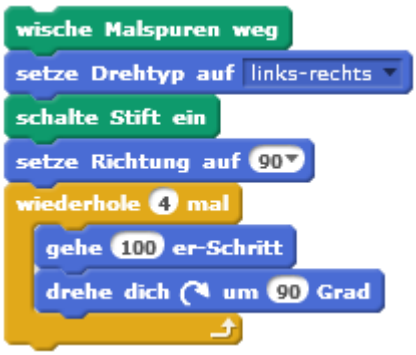


Abbildung 21: Lösungsweg Aufgabe 7

Aufgabe 8

Vereinfache den Programmcode der Aufgabe 3, indem du Schleifen benutzt.

Problemstellung der Aufgabe 3:

Schreibe Programme, die die folgenden Bilder zeichnen. Bei allen Bildern darfst du die Startposition der Katze bezüglich des zu zeichnenden Bildes selbst wählen. (vgl. [Hr12])

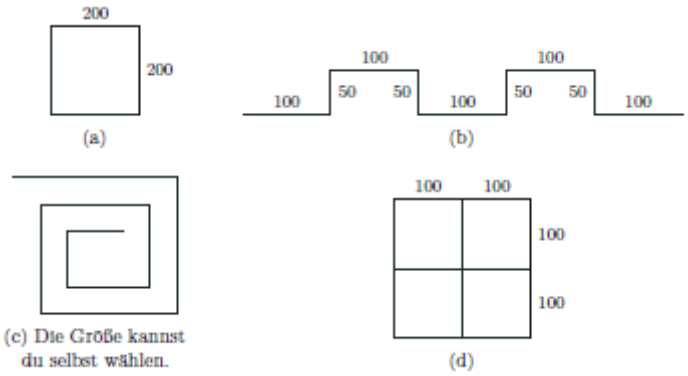


Abbildung 22: Bild aus [Hr12]

Lernziel

Schüler und Schülerinnen sollen

- einfache Schleifen verwenden können.
- Programme mittels Schleifen vereinfachen können.
- erkennen wann Schleifen sinnvoll sind und erkennen wann sich ein Muster wiederholt und man daher Schleifen einsetzen kann.

Möglicher Lösungsweg

Hier wird wieder Beispiel b vorgezeigt.

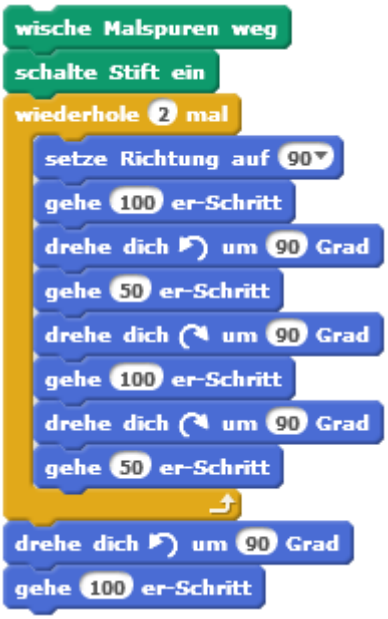


Abbildung 23: Lösungsweg Aufgabe 8

Bemerkungen

Damit Schüler und Schülerinnen auch lernen Programmcode zu lesen, könnte man diesen Code den Schülern und Schülerinnen zur Verfügung stellen und sie müssen die Anweisungen nachvollziehen und die Zeichnung auf ein Blatt Papier zeichnen.

Aufgabe 9

Zinsrechnung 2

Forme die Aufgabe 5 nun so um, so dass die schon erhaltenen Zinsen bei der nächsten Verzinsung berücksichtigt werden. (Zinseszinsen)

Lernziel

Die Schüler und Schülerinnen sollen

- mathematische Berechnungen programmieren können.
- Schleifen richtig anwenden können.

Möglicher Lösungsweg

Abbildung 24: Lösungsweg Aufgabe 9

Aufgabe 10

Verwandle den Mauszeiger in einen Ball. Der Ball soll sich permanent drehen. Lass eine Katze den Ball verfolgen, wenn sie den Ball erwischt, miaut sie. (vgl. [Ma14])

Bemerkung

Dieses Beispiel könnte auch zur strukturierten Vorgehensweise hinzugefügt werden, da aber Endlosschleifen benötigt werden, wurde es zum Konzept der Schleifen hinzugefügt.

Lernziel

Schüler und Schülerinnen sollen

- Endlosschleifen sinnvoll anwenden können.
- zwei Objekte miteinander agieren lassen können.

Möglicher Lösungsweg

Abbildung 25: Skript Ball

Abbildung 26: Bühne Aufgabe 10

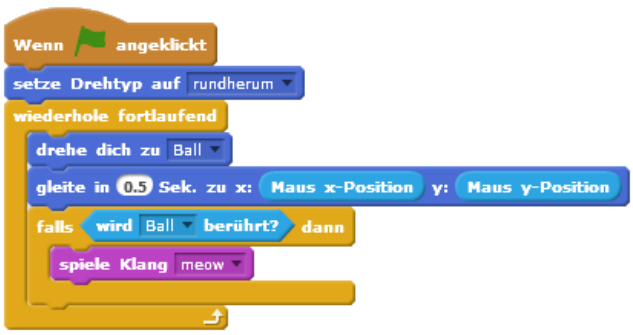


Abbildung 27: Skript Katze

Aufgabe 11

Zeichne folgende Figur. (vgl. [Ma14])

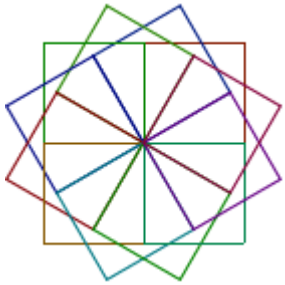


Abbildung 28: Angabe Beispiel 11

Lernziel

Schüler und Schülerinnen sollen

- aus einer gegebenen Figur Teilfiguren erkennen können.
- aus kleinen Figuren große Figuren zusammenbauen können.
- verschachtelte Schleifen anwenden können.

Möglicher Lösungsweg

Abbildung 29: Lösungsweg Aufgabe 11

8.4. Bedingte Anweisungen und Verzweigungen

Aufgabe 12

Spiel: Fang den Ball 3

Lass einen Ball zufällig auf der Bühne gleiten. Der Ball soll vom Rand abprallen. Die Katze soll mittels der Tasten „oben“, „unten“, „rechts“ und „links“ steuerbar sein. Sobald die Katze den Bald berührt, soll sie „miauen“.

Lernziel

Schüler und Schülerinnen sollen

- wissen wie man ein zweites Objekt in Scratch hinzufügt.
- wissen wie zwei Objekte in Scratch miteinander agieren können.
- wissen, dass die Skripte objektbezogen sind.
- mit Zufallszahlen arbeiten können.
- den Sinn von Endlosschleifen in Scratch verstehen.
- die Operation „wird … berührt?“ kennen lernen.

Möglicher Lösungsweg

Die Steuerung mittels Tasten ist in Aufgabe 4 zu finden.

Das Skript der Katze muss noch folgendes enthalten:

Abbildung 30: Erweiterung Katze

Abbildung 31: Skript Ball

Aufgabe 13

Monate 1

Der Benutzer gibt eine Zahl zwischen 1 und 12 ein. Schreibe ein Programm, das den dazugehörigen Monat ausgibt. Überprüfe dabei, ob die Benutzereingabe korrekt ist.

Lernziel

Die Schüler und Schülerinnen sollen

- Benutzereingaben auf Richtigkeit überprüfen.
- Zahlen zu anderen Werten zuordnen können.

Möglicher Lösungsweg

Die Lösung wird nur bis zur Zuordnung „Februar“ gezeigt. „März“ bis „Dezember“ gehen analog.

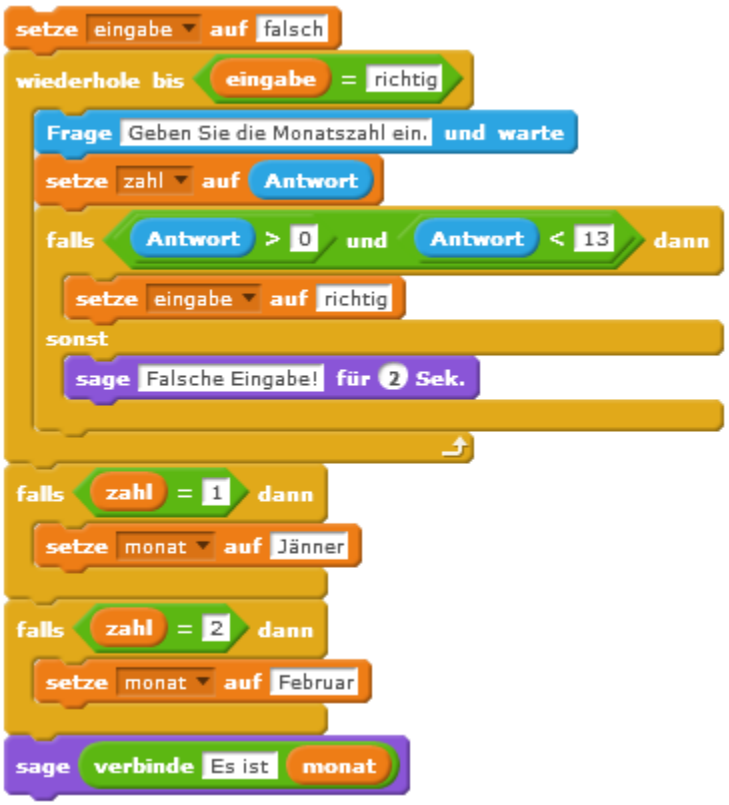


Abbildung 32: Lösungsweg Aufgabe 13

Aufgabe 14

Ein Jahr ist ein Schaltjahr, wenn die Jahreszahl durch 4 teilbar ist, aber nicht auch durch 100. Ausnahmen sind die Jahre, die auch durch 400 teilbar sind. Schreibe ein Programm, das ausgibt, ob das eingegebene Jahr ein Schaltjahr war bzw. wird. Überprüfe wieder die Benutzereingabe. (Jahreszahl > 0)

Lernziel

Die Schüler und Schülerinnen sollen

- mit den Modulo Operator in Scratch umgehen können.
- die Operatoren „und“ und „oder“ richtig anwenden können.
- verschachtelte bedingte Anweisungen anwenden können.

Möglicher Lösungsweg

Siehe rechts

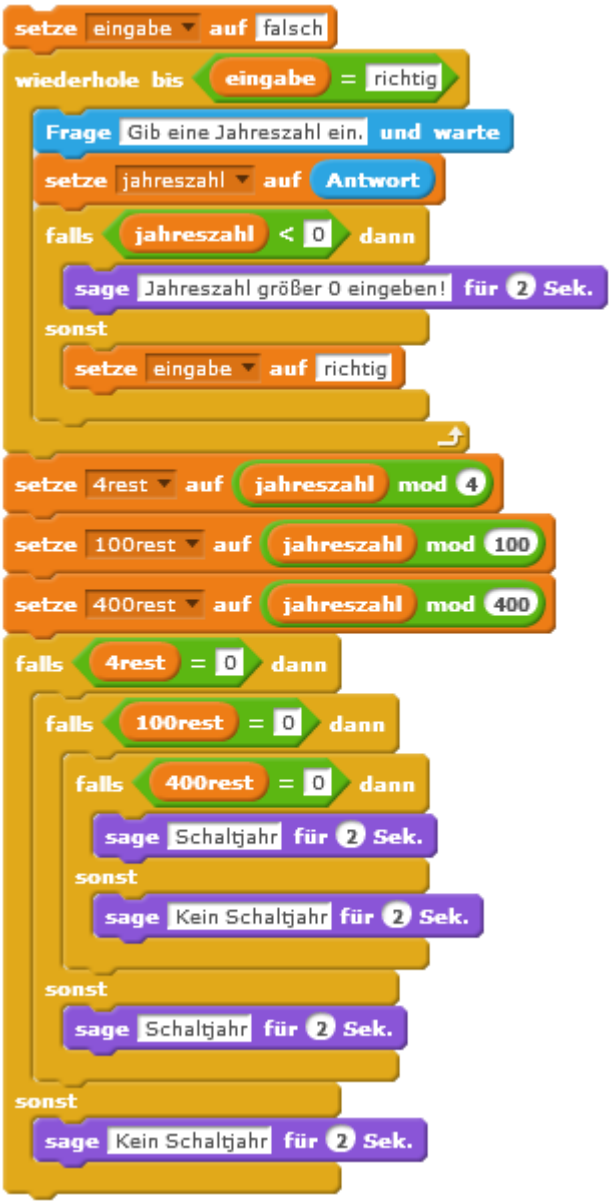


Abbildung 33: Lösungsweg Aufgabe 14

Aufgabe 15

Der Body – Mass – Index (BMI) ist eine Maßzahl für die Bewertung des Körpergewichtes eines Menschen in Relation zu seiner Körpergröße. Der BMI berechnet sich mittels BMI = Masse in Kilogramm / Körpergröße in Metern zum Quadrat. Liegt der BMI unter 19 spricht man von Untergewicht, ist der BMI zwischen 19 und 25 spricht man von Normalgewicht, ist der BMI über 25 und unter 30 spricht man von Übergewicht und beträgt der BMI über 30 spricht man von Adipositas. Schreibe ein Programm, das den Benutzer sagt, ob er Untergewicht, Normalgewicht, Übergewicht oder Adipositas besitzt.

Lernziel

Schüler und Schülerinnen sollen

- mathematische Berechnung in Scratch durchführen.
- einer Zahl einen Wert zuordnen können.

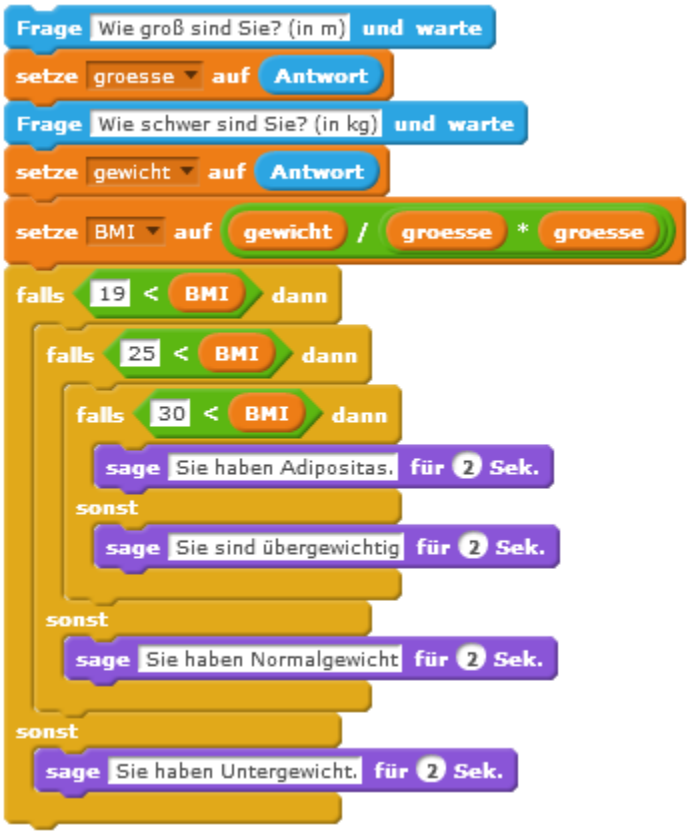


Abbildung 34: Lösungsweg Aufgabe 15

Aufgabe 16

Lass Äpfel von oben nach unten fallen. Ein Korb, der sich am unteren Rand der Bühne befindet, wird mit der linken und rechten Pfeiltaste gesteuert. Ziel des Spieles ist es mittels des Korbes die Äpfel zu fangen. Fängt der Korb einen Apfel, gibt es einen Punkt. (vgl. [Ma14])

Lernziel

Schülerinnen und Schüler sollen

- zwei Objekte miteinander interagieren lassen können.
- Positionen von Objekten abfragen können und darauf reagieren können.
- ein komplexes Problem aufteilen können.

Bemerkung

Es sollte die Möglichkeit des Klonens mit den Schülerinnen und Schülern besprochen werden.

Möglicher Lösungsweg

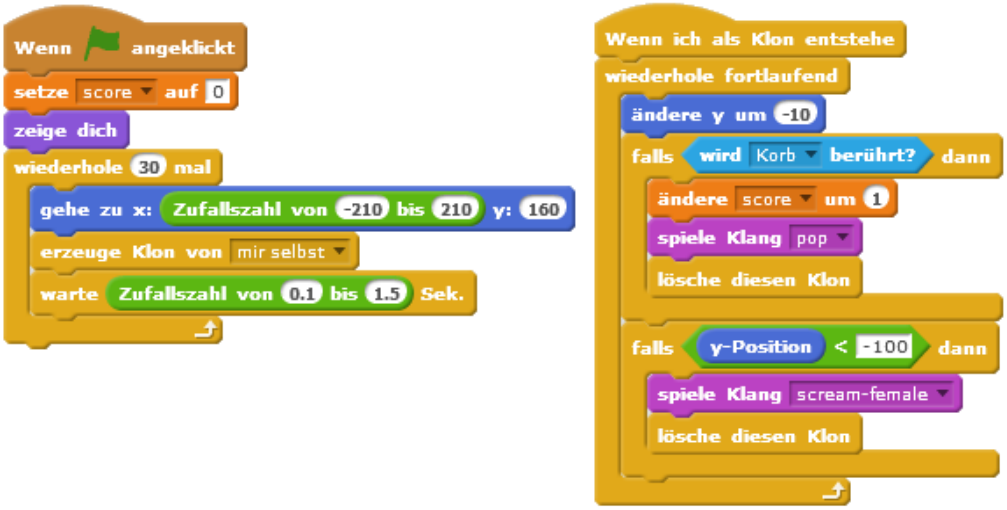


Abbildung 35: Skriptbild Apfel

Abbildung 36:Skriptbild Korb

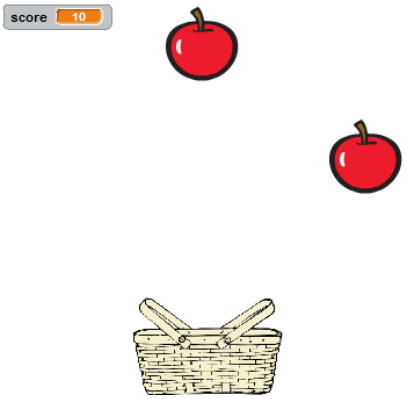


Abbildung 37: Bühnenbild Aufgabe 16, Korb aus http://openclipart.org

8.5. Methoden und Funktionen

Aufgabe 17

Quadrat 3

Zeichne 5 verschiedene Quadrate irgendwo auf der Bühne. Der Benutzer soll entscheiden wie groß das größte Quadrat ist. Achte darauf, dass die Quadrate nicht außerhalb der Bühne gezeichnet werden.

Lernziel

Schüler und Schülerinnen sollen

- Blöcke entwickeln können.
- die Bühne als Koordinatensystem interpretieren können und dadurch richtige Grenzen setzen können.

Möglicher Lösungsweg

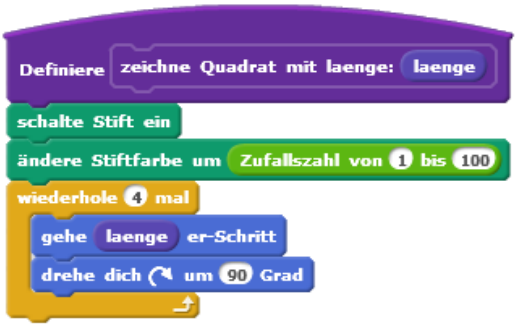


Abbildung 38: Definition Block Quadrat zeichnen

Abbildung 39: Lösung Aufgabe 17

Aufgabe 18

In Snap! gibt es den Operator <= bzw. >= nicht. Schreibe dafür eine passende Funktion.

Lernziel

Schüler und Schülerinnen sollen

- Funktionen entwickeln können und anwenden können.
- sehen, dass Funktionen den Programmcode vereinfachen können.

Bemerkung

Da nun Funktionen mit Rückgabewerte erstellt werden muss man auf Snap! zurückgreifen.

Möglicher Lösungsweg

Es wird der Operator „<=“ vorgezeigt. „>=“ geht analog.

Abbildung 40: Lösungsweg Aufgabe 18

Abbildung 41: größer gleich in Anwendung

Aufgabe 19

Zinsrechnung 3

Snap! und Scratch stellen viele mathematischen Operatoren zur Verfügung. Dennoch gibt es nicht die Möglichkeit eine Zahl zu potenzieren. Schreibe dafür eine Funktion. In Aufgabe 9 (Zinseszinsen) hast du die Potenz mittels Schleifen lösen müssen. Verwende nun deine selbstgeschriebene Funktion dazu und kürze dadurch deinen Programmcode.

Lernziel

Schüler und Schülerinnen sollen

- Funktionen entwickeln können und anwenden können.
- sehen, dass Funktionen den Programmcode vereinfachen können.

Möglicher Lösungsweg

Abbildung 42: Lösungsweg Aufgabe 19

Abbildung 43: Kürzung Aufgabe 9

Aufgabe 20

Überprüfe ob die Eingabe des Benutzers eine natürliche Zahl ist.

Lernziel

Schüler und Schülerinnen sollen

- Funktionen entwickeln können und anwenden können.
- nicht triviale Lösungswege selbst finden.

Möglicher Lösungsweg

Abbildung 44: ist a natürliche Zahl?

Bemerkung

Dieses Problem ist nicht trivial, denn die Schüler und Schülerinnen müssen bereits viele Aspekte der Programmierung bzw. von Snap! kennen und einige Tricks anwenden können. Es ist hilfreich den Schülerinnen und Schülern zu sagen, dass eine natürliche Zahl keine Nachkommastellen besitzt und dadurch sich die Zahl beim Runden auf die Einerstelle nicht ändert.

Aufgabe 21

Die Fakultät / Das Faktorielle ist eine Funktion, die einer natürlichen Zahl n das Produkt aller natürlichen Zahlen kleiner oder gleich n zuordnet. Sie wird durch ein „!“ abgekürzt. Beispiel: 3! = 1·2·3 = 6 5! = 1·2·3·4·5 = 120 Beachte, dass 0! = 1. Das Faktorielle einer Zahl wird häufig in der Mathematik verwendet. Schreibe dazu eine Funktion, die das Faktorielle einer Zahl zurückgibt. Wenn die Zahl keine natürliche Zahl ist, gib -1 zurück.

Lernziel

Schüler und Schülerinnen sollen

- mathematische Funktionen in Snap! erstellen können.
- den Umgang mit Schleifen üben.
- die Rückgabe des Wertes -1 als Fehler interpretieren können.

Möglicher Lösungsweg

Abbildung 45: Lösungsweg Aufgabe 21

Abbildung 46: Anwendung Faktorielle, liefert 120 zurück

Aufgabe 22

Zinsrechnung 4

Aufbauend auf die Aufgabe 17, schreibe nun eine Funktion, die das Endkapital nach einer exponentiellen Verzinsung zurückgibt.

Lernziel

Schüler und Schülerinnen sollen

- bereits gelöste Problemstellungen in Funktionen verpacken

Möglicher Lösungsweg

Abbildung 47: Lösungsweg Aufgabe 22

Bemerkung

Hier sieht man wieder eine Grenze von Snap!. Wenn die Funktion und die Parameterübergaben selbsterklärend sein sollen, wird der Funktionskopf sehr lange.

Aufgabe 23

Lasse eine Biene folgende Bienenwabe zeichnen. (vlg. [Hr12])

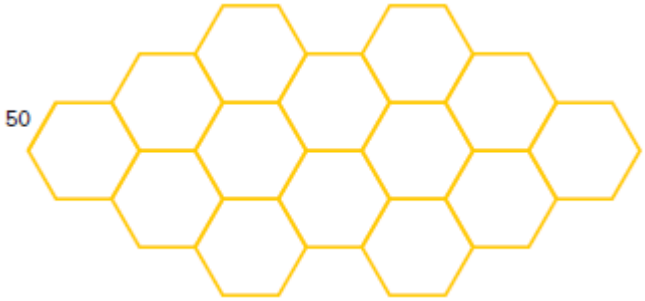


Abbildung 48: Bild aus [Hr12]

Lernziel

Schüler und Schülerinnen sollen

- geometrische Figuren nachzeichnen können.
- Aufgabenstellung in kleinere Probleme aufteilen und modular aufbauen können.

Möglicher Lösungsweg

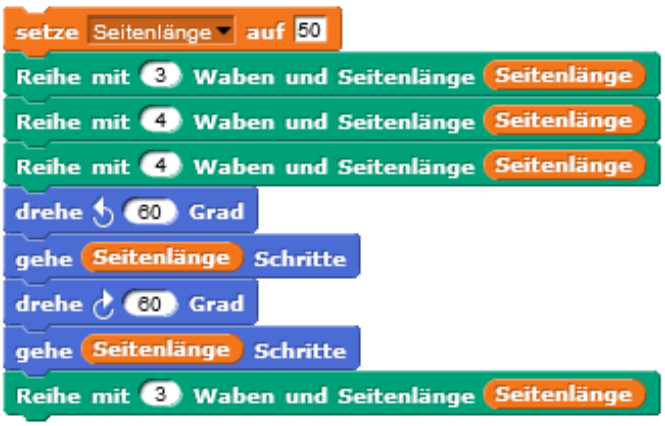


Abbildung 49: Lösungsweg Aufgabe 23

Abbildung 50: Waben zeichnen, geht wieder zum Anfang zurück

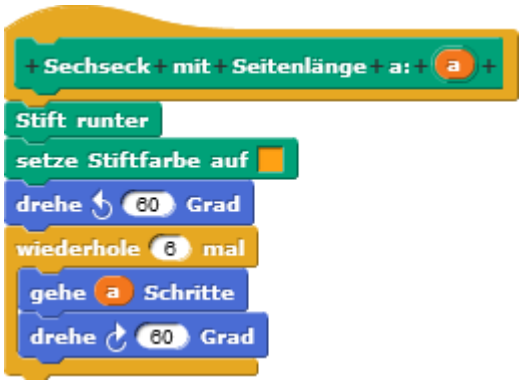


Abbildung 51: einzelne Wabe

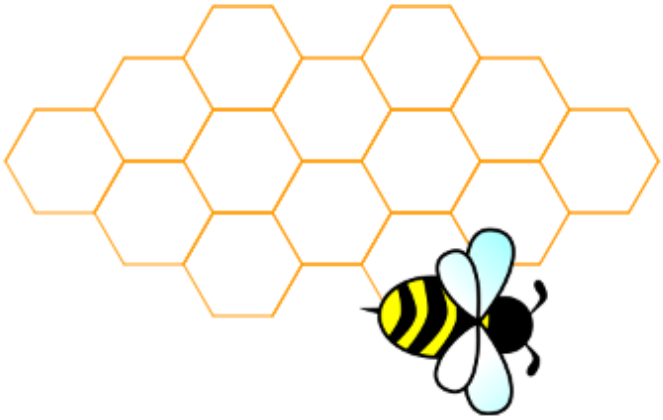


Abbildung 52: Ausgabe Aufgabe 23, Biene aus http://openclipart.org

Aufgabe 24

Erstelle einen Block, der ein beliebiges n-Eck zeichnet.

Lernziel

Schülerinnen und Schüler sollen

- mathematische Formeln anwenden können.
- erkennen, dass mittels Parameter eine Vielzahl an Problemen gelöst werden können.

Möglicher Lösungsweg

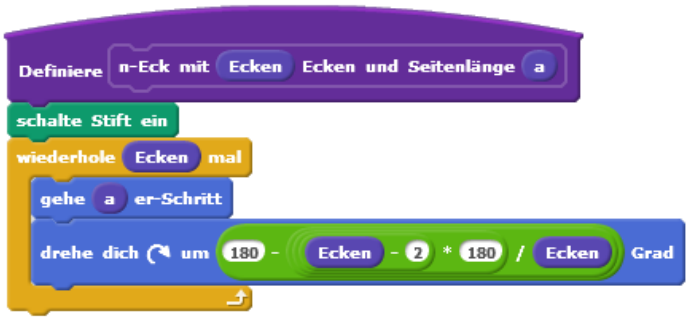


Abbildung 53: Lösungsmöglichkeit Aufgabe 24

8.6. Listen

Aufgabe 25

Zinsrechnung 5

Frau Huber legt einen gewissen Betrag auf ihr Sparbuch. Sie möchte wissen, wie sich ihr Kapital pro Jahr entwickelt. Es handelt sich um eine exponentielle Verzinsung. Schreibe ein Programm das die Entwicklung des Kapitals pro Jahr darstellt. Frage zuerst ab, wie viel und wie lange Frau Huber sparen möchte und wie groß der Zinssatz ist. Stelle die Kapitalsentwicklung in einer Liste dar, wobei an erster Stelle das Kapital nach einem Jahr, an zweiter das Kapital nach zwei Jahren, usw. dargestellt werden soll.

Lernziel

Schüler und Schülerinnen sollen

- selbstgeschriebene Funktionen verwenden können.
- Listen erstellen können und auf Listen zugreifen können.
- erkennen, dass Listen zur Datenspeicherung verwendet werden können.

Möglicher Lösungsweg

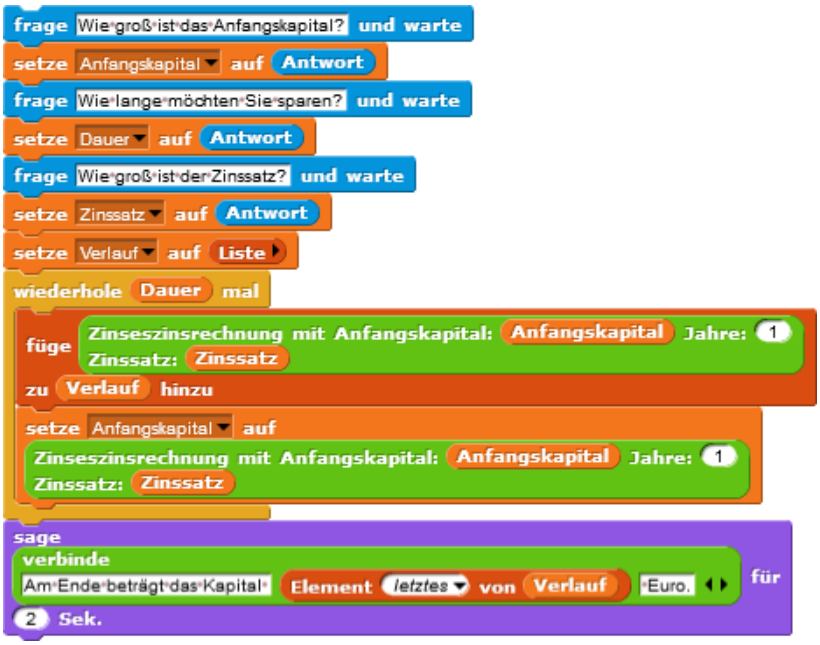


Abbildung 54: Lösungsweg Aufgabe 25

Aufgabe 26

Monate 2

Vereinfache die Aufgabe 12 (Monatszahl zu Monat zuordnen) indem du Listen verwendest.

Lernziel

Schüler und Schülerinnen sollen

- Listen erstellen können.
- auf Listen zugreifen können.
- schon erstellte Problemlösungen mit Hilfe von Listen vereinfachen können.

Möglicher Lösungsweg

Abbildung 55: Lösungsweg Aufgabe 26

Bemerkung

Die Schülerinnen und Schüler sollen diesen Lösungsweg mit dem Lösungsweg der Aufgabe 12 vergleichen.

Aufgabe 27

Schreibe eine Funktion, die eine Liste mit Länge n zurückliefert. Die Liste soll ganzzahlige Zufallszahlen im Bereich a bis b enthalten. Duplikate sind erlaubt.

Lernziel

Schüler und Schülerinnen sollen

- zufällige Listen erstellen können.
- mit Zufallszahlen umgehen können.

Möglicher Lösungsweg

Abbildung 56: Lösungsweg Aufgabe 27

Aufgabe 28

Schreibe eine Funktion, die überprüft, ob eine Liste Duplikate enthält oder nicht.

Lernziel

Schüler und Schülerinnen sollen

- verschachtelte Schleifen anwenden können.
- wissen, wie man Elemente der Liste miteinander vergleichen kann.

Möglicher Lösungsweg

Abbildung 57: Lösungsweg Aufgabe 28

Aufgabe 29

Schreibe eine Funktion, die Duplikate in einer Liste löscht.

Lernziel

Schüler und Schülerinnen sollen

- bereits programmierte Funktionen verwenden können.
- erkennen, dass sich Indizes und Grenzen einer Liste innerhalb einer Schleife verändern können.

Möglicher Lösungsweg

Abbildung 58: Lösungsweg Aufgabe 29

Aufgabe 30

Schreibe eine Funktion, die eine zufällige Liste der Länge n ohne Duplikate zurückliefert. In der Liste sollen Zahlen im Bereich a bis b enthalten sein.

Lernziel

Schüler und Schülerinnen sollen

- eine Listen mit Zufallszahlen erstellen können.
- überprüfen können ob ein Element bereits in der Liste vorkommt.
- mit Zufallszahlen umgehen können.
- erkennen, dass die Länge der Liste nicht größer, als die Intervallslänge der Zufallszahlen sein kann.

Möglicher Lösungsweg

Abbildung 59: Lösungsweg Aufgabe 30

Aufgabe 31

Schreibe eine Funktion, die eine Liste in umgekehrter Reihenfolge zurückgibt. Die ursprüngliche Liste darf nicht verändert werden.

Lernziel

Schüler und Schülerinnen sollen

- Wege finden, auf einzelne Elemente einer Liste zugreifen können.
- erkennen, dass man Listen auch von Ende bis zum Anfang durchgehen kann,
- erkennen, dass man Variablen auch verringern kann.

Möglicher Lösungsweg

Abbildung 60: Lösungsweg Aufgabe 31

Aufgabe 32

Schreibe eine Funktion, die die Reihenfolge der Elemente einer Liste umkehrt. Dabei darf keine Kopie der Liste erstellt werden.

Lernziel

Schüler und Schülerinnen sollen

- bestehende Listen verändern können.
- erkennen, wie man Listenelemente tauschen kann.

Möglicher Lösungsweg

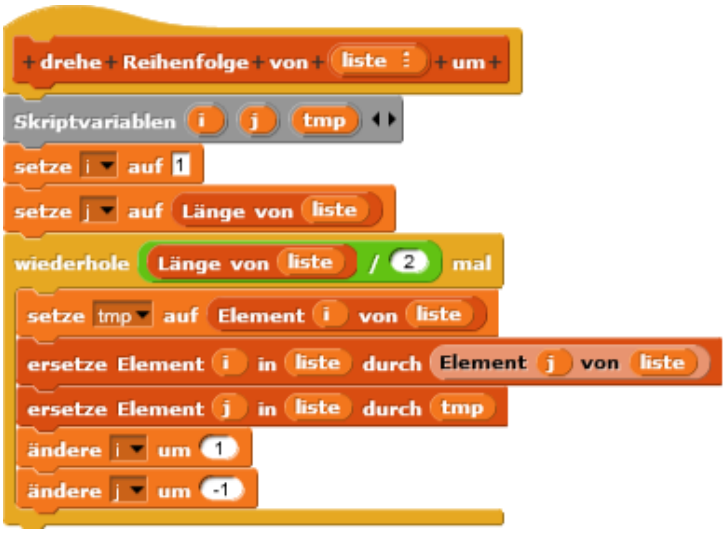


Abbildung 61: Lösungsweg Aufgabe 32

Bemerkung

Bei einer ungeraden Länge funktioniert die Anweisung „wiederhole Länge von Liste / 2“ dennoch.

Aufgabe 33

Schreibe eine Funktion, die überprüft, ob eine Liste aufsteigend sortiert ist.

Lernziel

Schüler und Schülerinnen sollen

- Funktionen schreiben können, die einen Wahrheitswert zurückliefert.
- Listenelemente miteinander vergleichen können.

Möglicher Lösungsweg

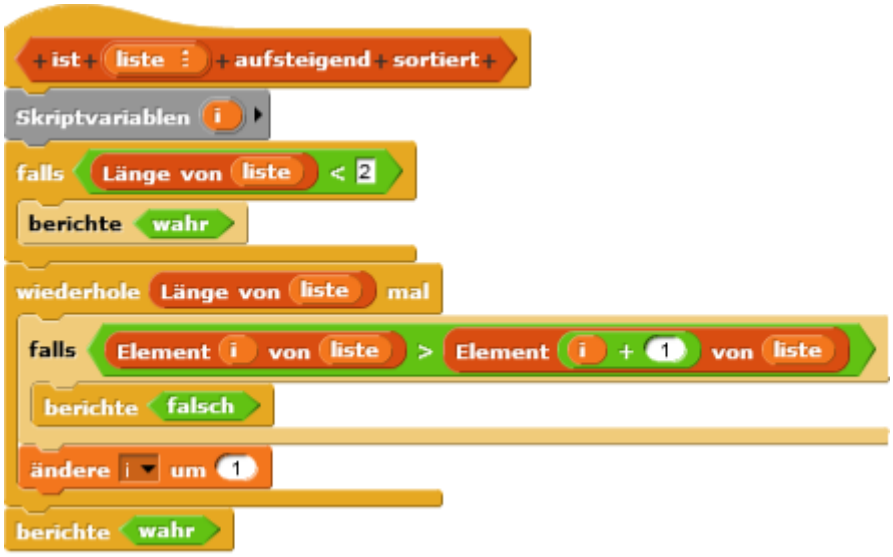


Abbildung 62: Lösungsweg Aufgabe 33

Aufgabe 34

Schreibe eine Funktion, die das Minimum einer Liste zurückliefert.

Lernziel

Schüler und Schülerinnen sollen

- Listenelemente miteinander vergleichen können.
- einfache Suchalgorithmen (lineare Suche) einsetzen können.

Möglicher Lösungsweg

Abbildung 63: Lösungsweg Aufgabe 34

Aufgabe 35

Schreibe eine Funktion, die den Index des kleinsten Elementes einer Liste zurückgibt.

Lernziel

Schüler und Schülerinnen sollen

- den Unterschied zwischen dem Index und dem Wert eines Listenelementes vertiefen.
- Den Umgang mit Indizes vertiefen.

Möglicher Lösungsweg

Abbildung 64: Lösungsweg Aufgabe 35

Aufgabe 36

Schreibe eine Funktion, die eine Liste aufsteigend sortiert.

Lernziele

Schüler und Schülerinnen sollen

- Einfache Sortieralgorithmen anwenden können.
- das Vertauschen von Listenelementen anhand einer Anwendung üben
- erkennen, dass das Sortieren bei vielen Listenelementen schnell rechenaufwendig werden kann.

Möglicher Lösungsweg

Abbildung 65: Hilfsfunktion: Ermittelt den Index des Minimums einer Liste in einem bestimmen Bereich

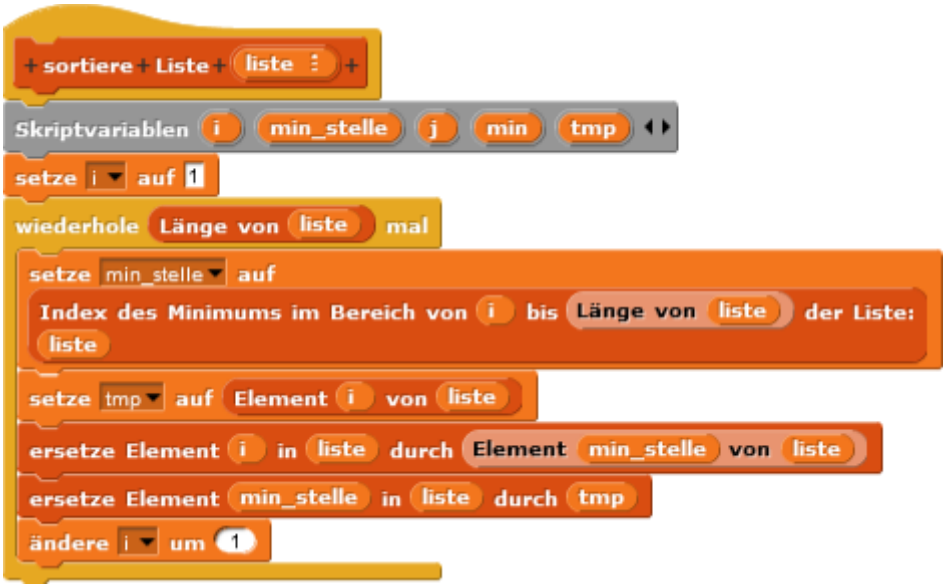


Abbildung 66: Lösungsweg Aufgabe 36

Bemerkung

Hier wurde der Sortieralgorithmus „Selectionsort“ verwendet. Dazu wurde die Funktion „Index des Minimums von Liste“ verändert. Bei der veränderten Funktion kann man angeben in welchem Bereich der Index gesucht werden soll. Schülerinnen und Schüler sollen sich mit mehreren Sortieralgorithmen beschäftigen. Im Zuge einer Gruppenarbeit können die Schüler und Schülerinnen mehrere Sortieralgorithmen kennenlernen. Jede Gruppe versucht einen eigenen Sortieralgorithmus zu entwickeln oder greift auf bekannte Algorithmen (z.B. Bubblesort, Linear Sort, etc.) zurück und stellt sie vor.

Das Tauschen von zwei Listenelementen könnte ebenso als eigene Aufgabe formuliert werden.

Aufgabe 37

„Dass jeder Mensch einzigartig ist, zeigt nicht nur sein Äußeres oder seine Fingerabdrücke. Auch auf molekularer Ebene gibt es Belege: die Blutgruppen. Jeder Mensch gehört einer besonderen, ererbten Blutgruppe an. Für die Transfusionsmedizin wichtig ist das AB0-Blutgruppensystem, das in den Jahren 1901 und 1902 vom österreichischen Arzt Karl Landsteiner entdeckt wurde. Man unterscheidet dabei vier Blutgruppen: 0, A, B und AB.“ ([@Eg14])

Erkundige dich darüber, welche Blutgruppe ein Patienten abhängig von seiner eignen empfangen darf. Erstelle anschließend ein Quiz. Der Computer sucht zufällig eine Blutgruppe aus, und der Benutzer soll eine Blutgruppe angeben, die der Patient empfangen kann. Der Computer sagt dir anschließend, ob die Wahl der Blutgruppe richtig war.

Lernziel

Schülerinnen und Schüler sollen

- Sachverhalte sinnvoll in Listen darstellen können.
- Listen als Listenelemente verwenden können und darauf zugreifen können.

Möglicher Lösungsweg

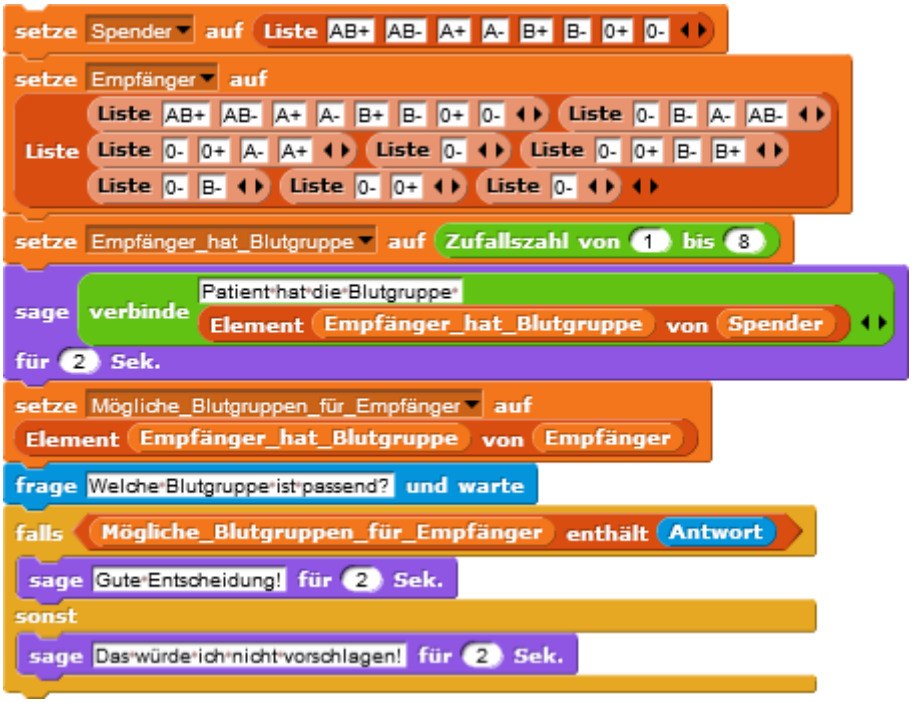


Abbildung 67: Lösungsweg Aufgabe 37

Abbildung 68: Bühnenbild Aufgabe 37, Tux aus http://openclipart.org

8.7. Rekursion

Aufgabe 38

Löse die Aufgabe 21 (Faktorielle) mit Hilfe von Rekursion.

Lernziel

Schülerinnen und Schüler sollen

- bereits gelöste Probleme rekursiv lösen können.
- die richtige Abbruchbedingung finden.
- erkennen, dass rekursive Lösungen oftmals weniger Code erfordern.

Möglicher Lösungsweg

Abbildung 69: Lösungsweg Aufgabe 38

Aufgabe 39

Eine Schneeflocke ist ein Fraktal. Schreibe eine Funktion, die eine Schneeflocke zeichnet.

Lernziel

Schüler und Schülerinnen sollen

- verstehen was ein Fraktal ist.
- selbst ein Fraktal zeichnen können und anschließend programmieren können.
- das Problem selbstständig in 2 Teilprobleme aufteilen können.

Möglicher Lösungsweg

Bei diesem Lösungsweg wird zuerst ein Davidstern gezeichnet. Von diesem ausgehend wird die Schneeflocke gebildet.

Abbildung 70: Davidstern

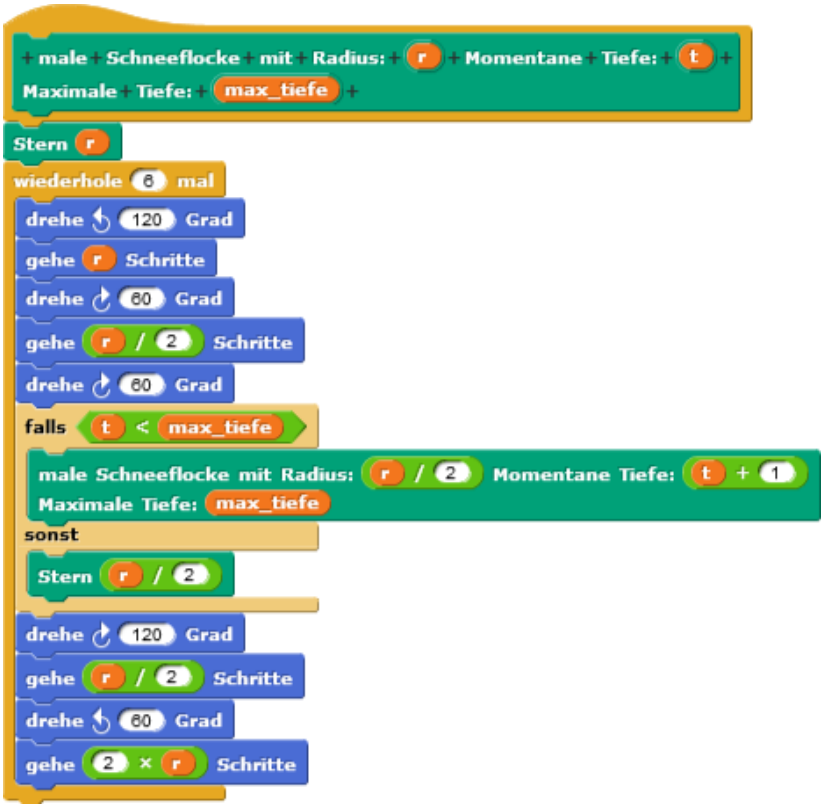


Abbildung 71: Lösungsweg Aufgabe 39

Mit dem Aufruf 

wird folgende Schneeflocke gezeichnet.

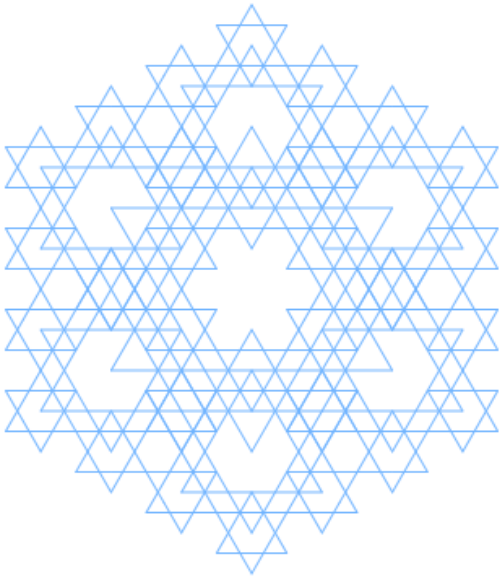


Abbildung 72: Gezeichnete Schneeflocke

Aufgabe 40

Schreibe einen Block, der von einer gewissen natürlichen Zahl bis 0 herunterzählt. Es soll daraus ein Countdown entstehen. Löse diese Aufgabe rekursiv und mittels einer Schleife. Vergleiche beide Blöcke. (vgl. [Ma14])

Lernziel

Schülerinnen und Schüler sollen

- eine rekursive Funktion entwickeln können.
- die richtige Abbruchbedingung finden können.
- rekursiven und iterativen Ausdruck vergleichen können.

Möglicher Lösungsweg

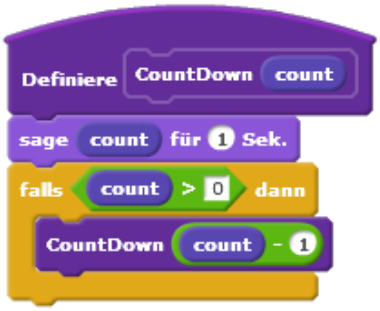


Abbildung 73: Lösungsweg Aufgabe 40 rekursiv

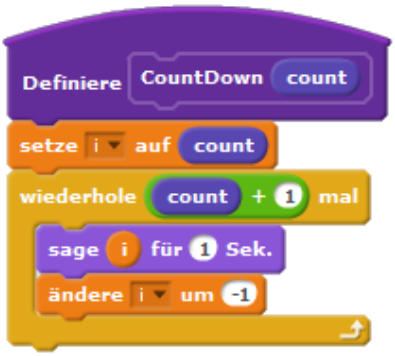


Abbildung 74: Lösungsweg Aufgabe 40 mittels Schleife

9. Zusammenfassung

Es stellte sich die Frage, warum Schüler und Schülerinnen Programmieren lernen sollten. Es wurden einige Gründe erläutert:

- Programmieren fördert die Problemlösefähigkeit und das logische Denken
- Programmieren sorgt für eine klare präzise Sprache bei der Formulierung der Anweisungen
- Programmieren weckt das Verständnis für Abläufe im Rechner und macht die langlebigen Grundlagen der Informatik deutlich
- Programmieren verbindet die mathematisch-naturwissenschaftliche Denkweise mit der Arbeitsweise der technischen Wissenschaften
- Programmieren unterstützt das fächerübergreifende Arbeiten
- Programmieren leistet einen hochwertigen Beitrag zur Allgemeinbildung
- Programmieren schult das schrittweise Vorgehen

Diese Gründe kann man für das Erlernen des Programmierens im Allgemeinen betrachten. Es gibt aber ebenso Gründe, wieso Schüler und Schülerinnen mit visuellen Programmiersprachen beginnen sollten.

ProgrammieranfängerInnen stehen am Anfang vor einigen Hürden, die sie beim Lernen überwinden müssen, damit ein effektiver Lernerfolg stattfinden kann. In diesem Zusammenhang spricht man von 7 unterschiedlichen Barrieren:

- Designbarriere
- Auswahlbarriere
- Koordinationsbarriere
- Benutzungsbarriere
- Verstehensbarriere
- Informationsbarriere

Ebenso haben viele Schüler und Schülerinnen Probleme mit der Syntax und zugleich finden sie auch die Aufgabenstellungen nicht motivierend.

Mit Hilfe von Scratch und Snap! können diese sieben Barrieren leicht überwunden werden. Die Probleme mit der Syntax sind nicht vorhanden, denn in Scratch und Snap! können keine Syntaxfehler entstehen. Unmotivierte Problemstellungen können mit Scratch und Snap! beseitigt werden, indem man im Unterricht medienreiche Spiele programmiert.

Es gibt aber dennoch Grenzen bei Scratch und Snap!. Mit Hilfe von Scratch und Snap! können alle Lerninhalte vom Bereich Algorithmen und Datenstrukturen (wie z.B. Methoden, Funktionen, Liste, etc.) erlernt werden.

Hingegen klare Vorteile gegenüber anderen Programmiersprachen liegt in Scratch und Snap! darin, dass keine Syntaxfehler entstehen können, die Elemente selbsterklärend benannt sind und man schnell erste Erfolge sieht.

Die Aufgaben wurden aufeinander aufbauend entwickelt. Es ist wichtig für Schüler und Schülerinnen immer wieder kleinere Problemlösungen zu finden, die miteinander kombiniert eine große Problemlösung ergeben.

10. Literaturverzeichnis

[@An13a] Andraschko, M.: http://digikomp.at/praxis/portale/digitale-kompetenzen/digikomp8nms-ahs-unterstufe/kompetenzmodell.html, letzter Aufruf 01.06.2014

[@An13b] Andraschko, M.: http://digikomp.at/praxis/portale/digitale-kompetenzen/digikomp12ahs/kompetenzmodelle/informatik-5-klasse.html, letzter Aufruf: 01.06.2014

[BMUKK04] Bundesministerium für Unterricht, Kunst und Kultur (BMUKK) (Hrsg.). 2004. Verordnung der Bundesministerien für Bildung, Wissenschaft und Kultur, mit der die Verordnung über die Lehrpläne der allgemein bildenden höheren Schulen geändert wird. Wien

[Br09] Bruderer, H. (2009). *Programmieren fördert die Problemlösungsfähigkeit. Plädoyer für den Programmierunterricht*. Zürich: ETH, Eidgenössische Technische Hochschule Zürich, Departement Informatik, Ausbildungs- und Beratungszentrum für Informatikunterricht

[@Eb13] Ebenhofer, M.: http://digikomp.at/praxis/portale/digitale-kompetenzen/digikomp4-volksschule/kompetenzmodell.html, letzter Aufruf: 01.06.2014

[@Eg14] Egg, J.: http://www.edugroup.at/praxis/portale/digitale-kompetenzen/digikomp8nms-ahs-unterstufe/unterrichtsbeispiele/detail/blutgruppen-3.html, letzter Aufruf: 30.05.2014

[FC03] Feldgen, M., & Clua, O. (2003). New motivations are required for freshman introductory programming. In *Frontiers in Education. FIE 2003 33rd Annual* (Vol. 1, pp. T3C-T24).

[FC04] Feldgen, M., & Clúa, O. (2004). *Games as a motivation for freshman students learn programming*. In *Frontiers in Education,. FIE 2004. 34th Annual* (pp. S1H-11)

[Ga05] Garner, S., et al. (2005). My program is correct but it doesn't run: a preliminary investigation of novice programmers' problems. In: *ACE '05: Proceedings of the 7th Australasian conference on Computing education*. Darlinghurst, Australia : Australian Computer Society (pp. 173–180)

[GH10] Gutknecht, J. Hromkovic, J. (2010). Plädoyer für den Programmierunterricht. In Springer-Verlag (Hrsg.) *Informatik-Spektrum 33(2)*, (pp. 230-238)

[Hr12] Hromkovic, J. (2010). *Einführung in die Programmierung mit Logo. Lehrbuch für Unterricht und Selbststudium.* Wiesbaden: Vieweg + Teubner Verlag

[@Ja14] Janssen, C.: http://www.techopedia.com/definition/22855/visual-programming-language-vpl, letzter Aufruf: 01.06.2014

[Kl14] Kleiner, P.: Was ist Informatik. (2014). In: Hasler Stiftung (Hrsg.): Bern

[KMA04] Ko A.; Myers B.; Aung H.(2004). Six learning barriers in end-user programming systems. In *Visual Languages and Human Centric Computing*, *2004 IEEE Symposium on Visual Languages*, (pp. 199-206)

[Ko09] Kohl, L. (2009). *Kompetenzorientierter Informatikunterricht in der Sekundarstufe I unter Verwendung der visuellen Programmiersprache Puck.* Dissertation. Jena: Universität

[La06] Landerer, C.. (2006) *Die Nintendo-Generation lernt Programmieren*, Diplomarbeit. Salzburg: Universität

[Ma10] Maloney, J., et al. (2010). The scratch programming language and environment. *ACM Transactions on Computing Education* (Vol. 10, Article 16)

[Ma14] Majed, M. (2014). *Learn to program with Scratch. A visual introduction to programming with games, art, science and Math.* San Francisco: No Starch Press

[Mo11] Modrow, E. (2011). Visuelle Programmierung-oder: Was lernt man aus Syntaxfehlern?. In Thomas Marco (Hrsg.), *Informatik in Bildung und Beruf, INFOS 2011, 14. GI-Fachtagung Informatik und Schule* (pp. 27-36). Münster: Köllen Druck + Verlag GmbH

[Mo13] Modrow, E. (2013). *Informatik mit BYOB / Snap!*. Göttingen: Universität

[Ro06] Robins, A. et al. (2006). Problem distributions in a CS1 course. In: *ACE '06: Proceedings of the 8th Austalian conference on Computing education*. Darlinghurst, Australia: Australian Computer (pp. 173–180)

[Sc05] Schwarz, G. (2005) Jenseits des ECDL: Lego Roboter Programmierung, In *CD Austria: Informatikunterricht an den AHS*, (Vol 3, pp.23)

[Ul03] Ullenboom, C. (2003). *Java ist auch eine Insel*. Bonn: Galileo Press

11. Abbildungsverzeichnis

Abbildung 1: Weitere Blöcke 39
Abbildung 2: Neuer Block 39
Abbildung 3: New Block 39
Abbildung 4: Block definieren 39
Abbildung 5: Block verwenden 40
Abbildung 6: Übergabeparameter 41
Abbildung 7: Übergabeparameter definieren 41
Abbildung 8: Block definieren 42
Abbildung 9: Block verwenden 42
Abbildung 10: Neuer Block Maximum 43
Abbildung 11: Übergabeparameter definieren 44
Abbildung 12: Block definieren 44
Abbildung 13: Block verwenden 45
Abbildung 14: Lösungsweg Aufgabe 1 54
Abbildung 15: Lösungsweg Aufgabe 2 55
Abbildung 16: Bild aus [Hr12] 56
Abbildung 17: Lösungsweg Aufgabe 3 56
Abbildung 18: Lösungsweg Aufgabe 4 57
Abbildung 19: Lösungsweg Aufgabe 5 58
Abbildung 20: Lösungsweg Aufgabe 6 59
Abbildung 21: Lösungsweg Aufgabe 7 60
Abbildung 22: Bild aus [Hr12] 61
Abbildung 23: Lösungsweg Aufgabe 8 61
Abbildung 24: Lösungsweg Aufgabe 9 62
Abbildung 25: Skript Ball 63
Abbildung 26: Bühne Aufgabe 10 63
Abbildung 27: Skript Katze 63
Abbildung 28: Angabe Beispiel 11 64
Abbildung 29: Lösungsweg Aufgabe 11 64

Abbildung 30: Erweiterung Katze 65
Abbildung 31: Skript Ball 65
Abbildung 32: Lösungsweg Aufgabe 13 66
Abbildung 33: Lösungsweg Aufgabe 14 67
Abbildung 34: Lösungsweg Aufgabe 15 68
Abbildung 35: Skriptbild Apfel 69
Abbildung 36:Skriptbild Korb 70
Abbildung 37: Bühnenbild Aufgabe 16, Korb aus http://openclipart.org 70
Abbildung 38: Definition Block Quadrat zeichnen 71
Abbildung 39: Lösung Aufgabe 17 71
Abbildung 40: Lösungsweg Aufgabe 18 72
Abbildung 41: größer gleich in Anwendung 72
Abbildung 42: Lösungsweg Aufgabe 19 73
Abbildung 43: Kürzung Aufgabe 9 73
Abbildung 44: ist a natürliche Zahl? 74
Abbildung 45: Lösungsweg Aufgabe 21 75
Abbildung 46: Anwendung Faktorielle, liefert 120 zurück 75
Abbildung 47: Lösungsweg Aufgabe 22 76
Abbildung 48: Bild aus [Hr12] 77
Abbildung 49: Lösungsweg Aufgabe 23 77
Abbildung 50: Waben zeichnen, geht wieder zum Anfang zurück 78
Abbildung 51: einzelne Wabe 78
Abbildung 52: Ausgabe Aufgabe 23, Biene aus http://openclipart.org 79
Abbildung 53: Lösungsmöglichkeit Aufgabe 24 79
Abbildung 54: Lösungsweg Aufgabe 25 80
Abbildung 55: Lösungsweg Aufgabe 26 81
Abbildung 56: Lösungsweg Aufgabe 27 82
Abbildung 57: Lösungsweg Aufgabe 28 83
Abbildung 58: Lösungsweg Aufgabe 29 84
Abbildung 59: Lösungsweg Aufgabe 30 85
Abbildung 60: Lösungsweg Aufgabe 31 86
Abbildung 61: Lösungsweg Aufgabe 32 87

Abbildung 62: Lösungsweg Aufgabe 33 88
Abbildung 63: Lösungsweg Aufgabe 34 89
Abbildung 64: Lösungsweg Aufgabe 35 90
Abbildung 65: Hilfsfunktion: Ermittelt den Index des Minimums einer Liste in einem bestimmen Bereich 91
Abbildung 66: Lösungsweg Aufgabe 36 92
Abbildung 67: Lösungsweg Aufgabe 37 94
Abbildung 68: Bühnenbild Aufgabe 37, Tux aus http://openclipart.org 94
Abbildung 69: Lösungsweg Aufgabe 38 95
Abbildung 70: Davidstern 96
Abbildung 71: Lösungsweg Aufgabe 39 97
Abbildung 72: Gezeichnete Schneeflocke 97
Abbildung 73: Lösungsweg Aufgabe 40 rekursiv 98
Abbildung 74: Lösungsweg Aufgabe 40 mittels Schleife 98

12. Tabellenverzeichnis

Tabelle 1: Datentypen Scratch / Snap! 32
Tabelle 2: Vergleich Variablen Scratch / Java 33
Tabelle 3: Vergleich Zählschleife Scratch / Java 33
Tabelle 4: Vergleich Kopfgesteuerte Schleife Scratch / Java 34
Tabelle 5: Vergleich Endlosschleife Scratch / Java 35
Tabelle 6: Vergleich Vergleichsoperatoren Scratch / Java 36
Tabelle 7: Vergleich kleiner-gleich, größer-gleich Scratch / Java 36
Tabelle 8: Vergleich Bedingte Anweisung Scratch / Java 37
Tabelle 9: Vergleich Verzweigung Scratch / Java 38
Tabelle 10: Vergleich Liste/Array Snap! / Java 47
Tabelle 11: Vergleich Ausgabe Scratch / Java 47
Tabelle 12: Vergleich Eingabe Scratch / Java 48
Tabelle 13: Vergleich Eingabe multiplizieren Scratch/Java 49
Tabelle 14: Vergleich Zufallszahlen Scratch / Java 49
Tabelle 15: Vergleich Modulorechnung Scratch / Java 50
Tabelle 16: Vergleich Stoppuhr Scratch / Java 50

DORIS MAYR

Prinz Eugen - Str. 3/3/24
2000 Stockerau

PERSÖNLICHE INFORMATIONEN

Geburtsdatum:	24.04.1990
Geburtsort:	Stockerau
Staatsangehörigkeit:	Österreich

AUSBILDUNG

seit Oktober 2009:	Universität Wien / Technische Universität Wien Lehramtsstudium Mathematik und Informatik & Informatikmanagement
2004 – 2009:	HTL Rennweg
2000 – 2004:	Hauptschule Ernstbrunn
1996 – 2000:	Volksschule Großmugl

PRAKTISCHE TÄTIGKEITEN

seit September 2013:	Lehrerin am BG und BRG Hollabrunn
seit Februar 2013:	Lehrerin am Erzbischöflichen Gymnasium Hollabrunn
Sommer 2012:	edugroup, Mitarbeit Portalaufbau www.sqa.at
Oktober 2011 - Februar 2013:	Tutorin für die Übungen Algorithmen, Datenstrukturen und Programmieren I & II Technische Universität Wien

QUALIFIKATIONEN

Jänner 2014 & 2011:	**Leistungsstipendium,** Universität Wien
Juli 2011:	**1. Diplomprüfungszeugnis**, Universität Wien
Juni 2009:	**Reife- und Diplomprüfung** mit Auszeichnung bestanden