Second year of programming in K-12 education - Influence of prior knowledge and working behavior with autograders on learning success.

Valentin Herrmann
valentin.herrmann@tum.de
Technical University of Munich
Munich, Germany

Abstract

While autograding systems are prevalent in universities, their application in K-12 education remains underexplored. Building on automated feedback research, this study investigates how K-12 students' prior programming knowledge and working behaviors within the Artemis autograder influence learning success. Data from 76 10th-grade computer science students were analyzed, comparing their performance on repetition exercises (testing 9th-grade prior knowledge) with that on advanced exercises introducing reference attributes, inheritance, and arrays. Using multiple linear regression and Spearman's rank correlation, student success metrics (score and error quotient) and working behaviors (submissions and build failures) were evaluated against a paper-and-pencil posttest. Results indicate that prior knowledge strongly correlates with success in advanced object-oriented topics, though this effect diminishes for algorithmic concepts like arrays. Furthermore, behavior metrics revealed complex interactions: identical measures yielded both positive and negative learning outcomes, depending on the specific topic. These findings suggest that future investigations require a complete reworking of exercises tailored to research requirements, rather than adapting existing materials.

CCS Concepts

• **Social and professional topics → K-12 education**.

Keywords

K-12 education, Programming Education, Autograders

ACM Reference Format:
Valentin Herrmann. 2026. Second year of programming in K-12 education - Influence of prior knowledge and working behavior with autograders on learning success.. In *Scientific work in computer science education (Advanced Seminar)*. ACM, New York, NY, USA, 19 pages.

1 Introduction

The author's bachelor's thesis [8] examined different ways to design automated feedback for programming exercises in a K-12 educational context, using the autograding system Artemis [14]. However, it revealed that this is an even more complex topic than expected, whereas large sample sizes and as much contextual information as possible are inevitable.

This study's objective is to begin addressing the limitations mentioned, especially the lack of context. This will be done by evaluating students' general work habits and learning outcomes when using autograding systems.

While the original study focused only on programming exercises on new topics in the students' second year of learning to program, the scope is now extended to include refresher exercises on previous-year topics before starting new topics, and measures learning outcomes afterward.

Some changes were made to the course concept for pedagogical reasons, which make data analysis easier and increase sample sizes as side effect.

2 Related Work

2.1 The Attachment Point

As mentioned before, this work immediately builds on the author's Bachelor's Thesis [8], which investigated the design of automated feedback in K-12 computer science education. The future work section suggests collecting and analyzing additional data on context and working behavior to gain a more comprehensive understanding of the situation. Additionally, it's recommended to increase the sample size. The following school year, changes were initially made for pedagogical reasons but also helped reduce impediments, such as small group sizes and the urge to cheat on practice exercises to achieve better grades. More details can be found in the section 4.1. The decision to analyze this data was made after the end of this course cycle, whereas the data had to be used as collected, without influencing its structure.

Prior Knowledge. Brod [3] and Kennedy et al. [11] highlight the importance of prior knowledge for learning about new topics. The course concept already includes exercises to repeat knowledge from the previous year. We will use the results of these repetition exercises to gain more context for new-topic exercises and reduce the impediment of missing context. Results of Kennedy et al. [11] indicate that students with better prior knowledge can more easily learn about topics that build on it. Brod [3] goes one step further and specifies that students with higher prior knowledge benefit more from active learning activities - in this case, programming exercises with automated feedback - learn fast and keep the new knowledge better in mind.

Advanced Seminar, Computing Education Research Group Munich

2.2 Automated Feedback and Autograders

Autograding systems like TUM's Artemis [14] provide student with automated feedback on their submissions for programming exercises. They have become irreplaceable in most university courses [2] and massive open online courses (MOOC) [11], but are only rarely used in K-12 computer science education.

Advantages. Autograding systems have several advantages for teachers and students. On the one hand, programming is highly frustrating for novices, and helping students individually requires huge time investments by teachers, which they usually cannot provide [12, 14]. Automated feedback reduces teachers' workload during class and allows students to receive more and immediate feedback. As a result, students learn better, engage better with exercises, solve more of them, and stay motivated to keep programming [6, 18, 19]. Besides that, data collected by autograding systems can serve for learning analytics [20].

Problems. Unfortunately, all that glitters is not gold. Some students tend to keep submitting new solutions without thinking and fall into a behavior of trying to break the autograding system. Besides, most autograding platforms meet universities' needs for the greatest possible flexibility in exercise creation, which makes exercise creation severely complex [2, 16, 19]. Unfortunately, most K-12 teachers do not have the skills and time to create exercises with such a high degree of freedom and complexity. Additionally, the use of autograding systems makes it easy for instructors to introduce continuous assessment and grading of practice exercises, which is counterproductive due to high pressure, decreased motivation, and the urge to cheat for better grades. Besides lower learning success, the latter also reduces data quality for subsequent analyses like this one. One of the pedagogical changes to the concept was to remove continuous assessment, and besides other advantages, improve data quality.

3 Research Questions and Hypotheses

3.1 Research Questions

As outlined in the previous section 2, there are several aspects from which we can gain more contextual knowledge about programming tasks with automated feedback and tackle impediments of the original study [8]. To formulate the research questions, the following aspects seem promising:

- Controlling prior knowledge to isolate actual effects of the exercises with automated feedback.
- Observe not only the successful solving of exercises but also the behavior during that work.
- Include a measure for learning outcome for being able to determine the effectiveness of the process.

Following these approaches, the following research questions are formulated:

RQ1. What is the relation between prior knowledge about basic topics of programming and…
  (a) …the working behavior on…
  (b) …the success of solving…
  …advanced exercises with automated feedback?

RQ2. What is the relation between…
  (a) …the working behavior on…
  (b) …the success of solving…
  …advanced programming exercises with automated feedback and the learning outcome on their (new) topics?

RQ3. What is the relation between…
  (a) …the working behavior on…
  (b) …the success of solving…
  …programming exercises about prior-knowledge-topics and the learning outcome on new topics?

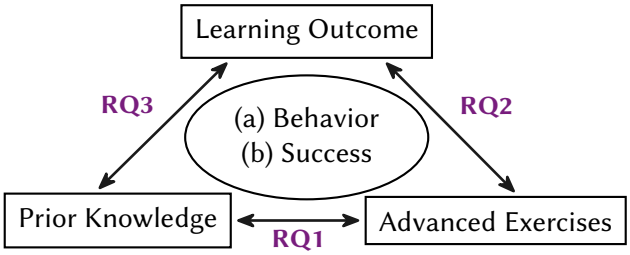


Figure 1: Overview of the Research Question Structure

3.2 Hypotheses

Literature. Following the results of Brod [3], it is expected that the successful solving of exercises about prior knowledge has the biggest impact on the successful solving of advanced exercises and the learning outcome (RQ1(b), RQ3(b)).

Kennedy et al. [11] explains that students with low prior knowledge find it hard to successfully engage with programming exercises. In combination with Baniassad et al. [2], this leads to the hypothesis that low prior knowledge is expected to correlate with bad working behavior (RQ1(a), RQ3(a)).

Practical Experience. For RQ2, intuition and practical experience suggest that higher scores in the advanced exercises and better working behavior lead to better results in the learning outcome.

Besides that, practical experience suggests that prior knowledge has a greater impact than working behavior in all cases.

4 Research Methods

To answer the research questions, data from three 10th-grade computer science courses were analyzed. This section explains the general circumstances, the pedagogical concept, and how it attaches to Herrmann [8], which constructs are measured and how they are analyzed.

4.1 Circumstances, Curriculum and Pedagogical Concept

Population. The data were collected within the scope of the regular computer science class in three courses in the STEM branch of a Bavarian Grammar School. These courses will later be referred to as a, b, and c. Each course has 90 minutes of class per week.

Curriculum. In 9th grade, the students learned basic Java programming concepts and object-oriented modeling. Therefore, they are already familiar with classes, methods, attributes, and

Table 1: Student Population by Course and Gender

	female	male	total
a	13	15	28
b	6	15	21
c	8	19	27
Total	27	49	76

constructors, as well as return values, parameters, local variables, primitive data types with arithmetic operations, conditional statements with if-else, and for- and while-loops [8, 15]

Before continuing with the 10th-grade curriculum, exercises focused on reintroducing programming language, object-oriented modeling, methods with parameters and return values, loops, and conditional branching are conducted to give students a chance to refresh and assess (ungraded) their prior knowledge. As explained later in 4.2 and 4.3, these exercises will serve as a proxy for measuring prior knowledge of basic programming topics.

The new topics in 10th grade are classes as data types/object references (later referred to as reference attributes), inheritance, and arrays. For each of these topics, students can choose to learn about the theory using written material, videos, or teacher-led mini-lectures. Afterward, apply this theoretical knowledge to two mandatory and one optional exercise per topic with automated feedback.

Tools and Technology. BlueJ [10] was chosen as development environment because it is the only option for using Git in the school's computer network due to firewall restrictions. Besides that, about half of the students were already familiar with it, while the others were using Greenfoot [1] before. As a fallback in case of problems with the interaction of the IDE and the autograder, an online code editor integrated into Artemis is available; however, it does not allow running code online.

TUM's autograder, Artemis [14], was used to deploy feedback. None of the students had ever used the autograding platform for Java programming. To mitigate the negative effects of students' unfamiliarity with autograders during learning new topics [17], the first repetition exercise was combined with precise instruction on how to use the autograding system. Figure 2 visualizes the style of exercises for new topics using the problem statement of the first exercise on inheritance. Together with the problem statement, students receive a source code template as a starting point for the exercise.

Adaptation of Pedagogical Concept. The analyzed data were collected in the school year following the one in which the study of Herrmann [8] was conducted in. Compared to the original concept, some changes were made for pedagogical reasons, which are useful for this analysis, as they help reduce impediments to the original study.

- All exercises on new topics were uniformly carried out, with instructive feedback that told the student what to do next to fix the current mistake. This increases the number of participations and submissions that can be compared.
- Practice exercises were not included in the grading based on their scores to ease the pressure of constant grading as suggested by Jenkins [9]. However, the project, which builds on the skills learned in practice exercises, was still graded. This reduces the urge to copy from other students or to let AI solve the exercises just to get a better grade, thereby increasing the overall validity of the results.
- Even though none of the exercises were mandatory in the sense of punishment if not solved, the third exercise on each new topic was tagged as optional for additional practice, while the first two remained mandatory. Unfortunately, as observed in the data, removing grading for practice exercises led several students to skip even mandatory exercises.
- Besides that, every student had to complete the exercises alone, even though they were encouraged to cooperate, what most did. This also increases participation and submissions

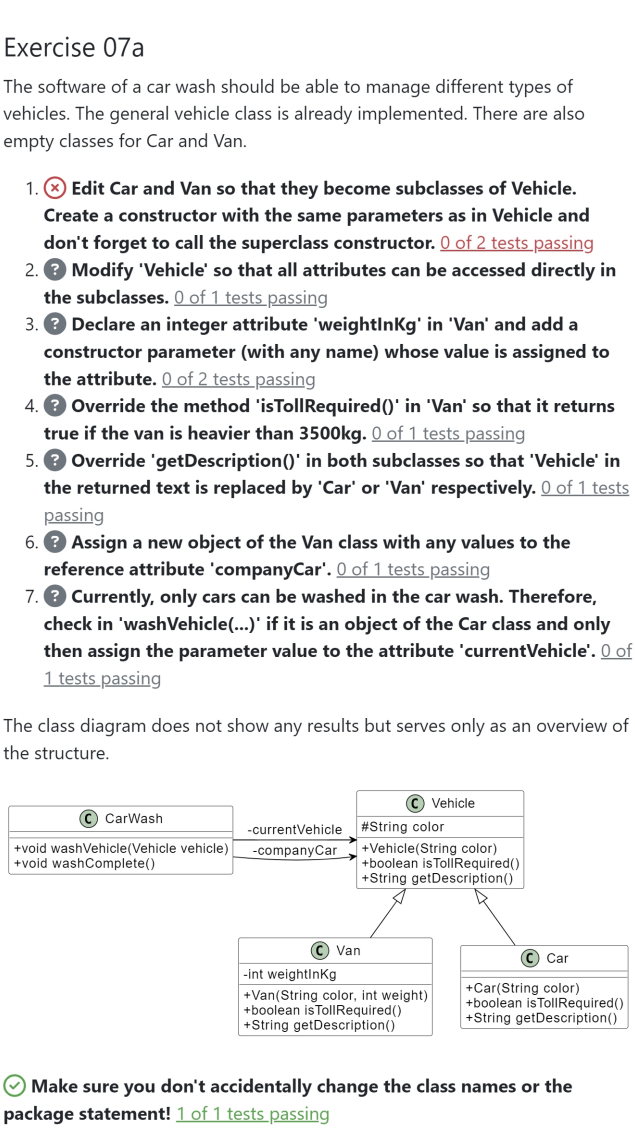


Figure 2: First Exercise about Inheritance (translated)

and allows for comparison of written exam results as a posttest with data from the exercises.

4.2 Measured Constructs

Prior Knowledge. For determining the relation between the students' prior knowledge and the working behavior/success (RQ1), respectively, the learning outcome (RQ3), compared to the prior knowledge, the prior knowledge obviously has to be measured. However, even though RQ2 does not immediately compare the prior knowledge, it is still necessary to measure it for dividing the effect of prior knowledge and exercise about advanced topics to be able to precisely measure the latter. Otherwise, only an absolute outcome value would be taken into account, which would be an enormous threat to the interval validity. The repetition exercises follow the pre-test model, even though they aren't a formally developed research tool; rather, they are designed with pedagogical considerations in mind.

The prior knowledge is measured based on the working behavior and success in the repetition exercises about topics of the previous year (see 4.1). Hereby, a higher overall score, faster problem-solving, and fewer submissions per exercise are interpreted as higher prior knowledge.

More details about how working behavior and success are operationalized can be found in the following sections.

Exercise Success. All research questions measure the success in working on programming exercises. For repetition and new-topic exercises, this measure is calculated identically. On the one hand, exercise success is measured by the overall score a student achieved; on the other hand, the performance when solving mistakes is also an important measure of how successful a student was in an exercise. Especially since students can submit as often as they like until they reach a score that satisfies them, it is important to consider the process when measuring success. To achieve this, an error quotient is calculated, which is explained in section 4.3.

Working Behavior. As explained in section 2, autograders may negatively affect students' work behavior. Therefore, one construct to record is how students interact with the autograding system, and to draw conclusions alongside other measures. The working behavior is observed by tracking the submission count for exercises and by accounting for the error quotient. The influences of this measure may provide insights into the usefulness of autograding systems in K-12 computer science education.

Learning Outcome. The superordinate goal of all teaching is the final outcome in terms of learning success. Therefore, all measured constructs are compared to this final measure. It can be measured overall or by specific (new) topics. To determine the learning outcome, a graded paper-and-pencil exam was carried out. Similar to the exercise on repetition topics, whose success is interpreted using the idea of a pre-test, the exam results are interpreted with the idea of a post-test in mind, even though it's not a formally created and evaluated research instrument but an exam designed following pedagogical considerations.

4.3 Operationalization

Exercise Success. To measure if a student completed an exercise successfully, two metric measures will be taken into account: The overall score and the error quotient. Both are normalized to the $[0, 100]$ interval for a more illustrative comparison. While the overall score considers only the final submission to an exercise (with 100 percent as the best possible value), the error quotient measures the number of mistakes made and the number of tries it takes the student to resolve them.

Figure 3 describes how the error quotient is calculated in the interval of $[0, 100]$ with 100 as the best possible value. For more intuitive analysis and visualization of results, this calculation differs from the one in the original study [8]. The error quotient punishes submissions that contain any mistakes, and if none of the mistakes of the previous submission were solved, the punishment is twice as hard. A perfect value of 100 would indicate that a student solved the exercise correctly in the first submission. The worst possible value, 0, would indicate that a student submitted often and kept making the same mistakes, or even more. 0 is never reached, but it is approached asymptotically as the number of submissions increases. For example, the worst value for one submission would be 50; for ten submissions, it would be 5; for 20 submissions, it would be 2.5; and the worst possible value for 50 submissions would be 1. This takes into consideration that it is worse to repeat mistakes over and over again than to solve one and make a new one afterwards. If a mistake reappears after it disappeared, it is assumed it was overshadowed by other mistakes and wasn't actually solved.

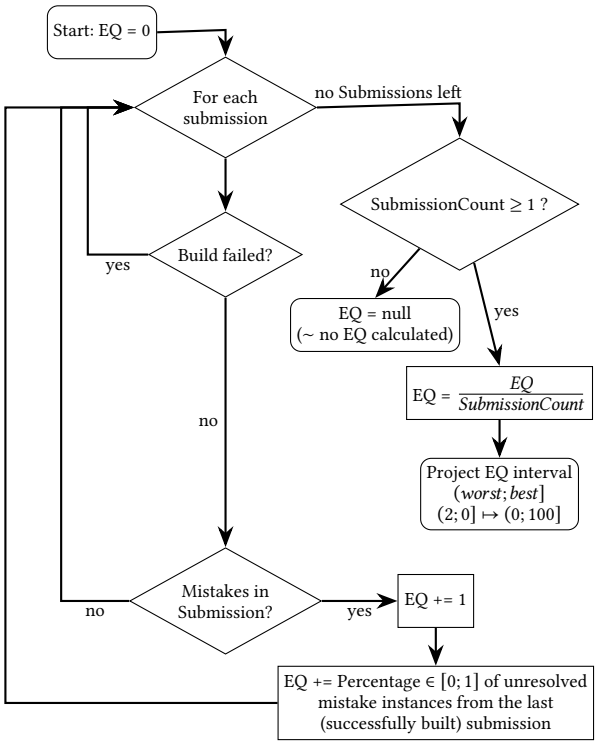


Figure 3: Calculation of error quotient values based on [8, 21]

When looking at these measures on exercise level, it is important to consider that even though the measure might be perfect for the first exercise on a topic, it could be very bad for the topic's second exercise, which might point out that the student still struggled on the first exercise but solved it with enormous time required. As the time-on-task for each exercise cannot be calculated properly [8], looking at the score and error quotient are the only ways to detect such anomalies. For the purpose of answering our research questions, it is sufficient to break down the measure to the topic level rather than the exercise level. Therefore, the values are aggregated to one average per topic, and the effects of major differences across the exercises of a given topic are sufficiently eliminated.

Working Behavior in Exercises. To determine a student's working behavior, with special focus on potential autograder misuse, the submission count and error quotient are analyzed. Higher submission counts and worse error quotients might indicate poor working behavior, such as brute-forcing the auto-grader by changing and resubmitting without proper debugging or thought process [2]. When working with text-based languages like Java, build failures and syntax errors are common problems and would be an interesting measure for both working behavior and success, and will be reported in all results. However, students know that the autograder will give zero points and provide no useful feedback for such a submission, which is why they focused on submitting only properly compiling code. Non-compiling code is easy to spot in BlueJ before submitting and can be fixed. Thus, most syntax problems are not visible in the autograder's data and, therefore, are not a useful measure for this analysis.

Another interesting measure to gain insights into working behavior would be time-on-task [21]. Unfortunately, for this type of exercise and course structure, a valid time-on-task cannot be quantitatively calculated [8] and therefore cannot be included in the operationalization of working behavior.

Learning Outcome. The learning outcome is calculated based on the posttest score, which is assessed through a paper-and-pencil exam. Due to scheduling and examination constraints, not all topics could be covered during the assessment of the learning outcome. For the courses a and b, the focus was on inheritance and arrays, while for course c, the focus was on inheritance and reference attributes. This leads to the drawback that there are fewer posttest results for arrays, and even fewer for reference attributes. All variants had mixed topics, multiple-choice exercises at the beginning, and spot-the-mistake exercises. The latter were counted towards the topic to which the mistake belongs. If multiple exercises on a topic were in a variant, the topic's score was calculated as the overall percentage.

To ensure a fair assessment, variants of posttest exercises used across multiple courses were created. The variants were created solely by renaming elements and exchanging conceptually interchangeable datatypes or operations (e.g., arrays of integers versus arrays of doubles).

The full set of posttest exercises is available in Appendix C. For better clarity, only one variant of each exercise is shown.

4.4 Data and Statistical Methods

Due to the multi-dimensional structure of the research questions, with many dependencies and relations among variables, statistical methods must be chosen that can properly handle confounding factors and make the actual effects visible. To achieve this, multiple statistical analyses have been performed, building on each other.

Data Cleansing. Following the approach of the original study [8], the following steps were taken to ensure undistorted data for analysis:

- Double submissions where students submitted twice in 10 seconds without making any changes are treated as one submission.
- Submissions without a generated result are treated as non-existent. This is a bug in the auto-grading platform, and students usually resubmitted right after and received the correct result.
- For all calculations, only values are considered where students actually participated in an exercise. If a student skipped an exercise, his values aren't included in any calculation. Students may not participate for several reasons, such as sickness, deciding they have practiced enough, running out of time, etc. Therefore, treating these as 0 points (or equivalent for other measures) would significantly distort the data.

Normal Distribution. was not expected for most variables due to the still relatively small sample (max. n=76) and the nature of the observed metrics (submission count, error quotient, and score). During analysis, this was verified for most values, especially those calculated from subsets for a specific gender or topic. For posttest results and values calculated for the full group on all exercises, this assumption could not be verified. Nevertheless, nonparametric test procedures were used in subsequent analyses for all groups to achieve consistent results across all cases. Nonparametric test procedures work with normally and non-normally distributed data, but have the minor drawback of having less statistical power [7].

Multiple (linear) Regression. is used for being able to determine which effects on the observed variable come from which controlled variable. This is especially helpful when trying to isolate the effect of the advanced exercises while controlling for prior knowledge, and vice versa [7, 13]. Therefore, this will be the main method used in answering RQ2 and RQ3.

To determine the degree of the function class to approximate using polynomial regression, polynomial regressions of degrees 1, 2, and 3 have been performed on all relevant combinations of variables. Thereafter, the root-mean-square errors (RMSE) were compared. The results strongly indicated a linear relationship between the variables, and only a few edge cases showed weak tendencies toward quadratic relationships (and even fewer for cubic relationships). More precisely, weak tendencies mean that the difference between the RMSEs of the linear and quadratic approximations was never greater than 10 in favor of a quadratic relation, whereas most differences favored a linear relation, reaching values as large as 2000 in favor of a linear relation. To ensure comparability of the results, multiple linear regression was used for all regression analyses [5, 7].

Spearman-Correlation. was used to gain detailed insights into how single metrics for advanced exercises and base-topic exercises depend on each other. This method was mainly used to answer RQ1. As described in section 4.3, behavior and success are operationalized using two metrics each. To achieve more differentiated insights into how these metrics interact, the exercise metrics were also compared using multiple linear regression, though only a few yielded significant results.

Identification of relevant relations for analysis. To identify existing, significant relations between the investigated measures, the previously described statistical methods were applied to all combinations of measures with and without grouping by gender, course, single topics, repetition/new topics. All combinations and groups that yielded statistically significant values at a confidence level of at least 5% (in multiple regression analysis, up to 10% if at least one value was better than 5%) were then further analyzed.

5 Results

5.1 Key Figures of the Collected Data

The following histograms provide a first insight into the structure of the collected data, grouped by repetition and new-topic exercises.

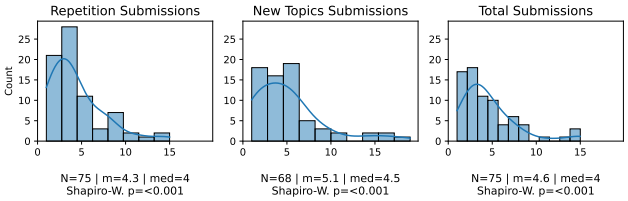


Figure 4: Submissions by new/repetition exercises and total

Submission Count. Looking at the submission counts per exercise and student (Figure 4), having mean and median values around four to five supports the assumption that most students maintain low submission counts. Some students nevertheless have significantly higher submission counts up to 15, which might point to autograder-breaking behavior. When interpreting this value, it is important to keep in mind that most exercises have multiple sub-tasks, which tend to result in more frequent submission to verify intermediate results before proceeding to advanced tasks.

For exercises on new topics, the submission counts are, all in all, a bit higher than for exercises about already known topics, which supports the intuition from practice that it is harder and requires more (instructive) feedback to solve exercises while learning the topic compared to solving exercises whose topics were already familiar in the past.

Build Failures. As expected only few submissions contain build failures (Figure 5). The students were instructed that submitting non-compiling code will result in 0 points and no useful feedback. Although Artemis displays complete build logs, these are basically unreadable for beginners. Because of this and the ability to easily see build failures in BlueJ before submitting, only very few students

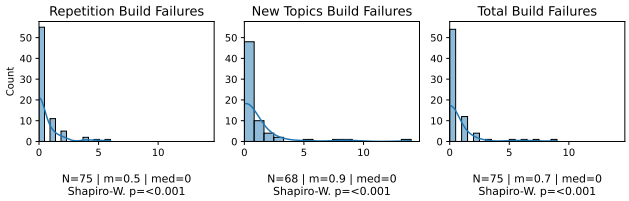


Figure 5: Failed builds by new/repetition exercises and total

submitted uncompilable code, even though they might have had syntax issues before.

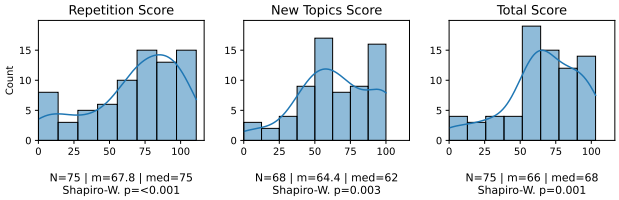


Figure 6: Score by new/repetition exercises and total

Exercise Score. Surprisingly, although the scores were well above 50%, one would have expected them to rise further toward 100% because they did not have any limit on how often they could try and were provided with instructive feedback on every submission. Using unstructured observation from the classroom, this can be explained by three factors:

- running out of time or motivation due to missing class or distraction during class
- self-responsible decision that a topic has been practiced enough (especially for repetition exercises)
- starting but not finishing optional exercises (this might especially explain the peak at <10% for repetition exercises).

Nevertheless, in the total score per student, they performed well, with a median of 68% and only a few students with total scores below 50%.

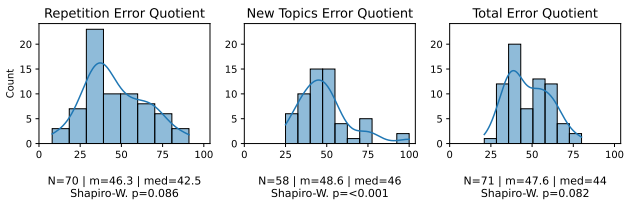


Figure 7: Error quotient by new/repetition exercises and total

Error Quotient. With a significance value in the Shapiro-Wilk test of >0.05 for repetition exercises and all exercises, the error quotient is the only exercise measure that can be seen as normally distributed. Interestingly, this does not apply to the error quotient of exercises on new topics, where two students reached an average EQ of 100 across exercises, meaning they solved all exercises on advanced topics they participated in immediately correctly. This wasn't the case for the average EQ in repetition exercises for any student, even though one might expect they would be easier to solve and yield better EQ values. However, as explained before, many exercises have multiple sub-tasks, and especially the loops exercise (see 20) with multiple fully independent sub-tasks encourages students to submit unfinished solutions, after which it is not possible to achieve the perfect error quotient of 100. Therefore, the error quotient must be seen as a relative progress measure rather than an absolute success measure, for which the absolute score is available. Overall, the error quotients are centered on a median of about 45 across all groups, yielding a consistent measure.

Posttest Scores. Unusually for examination in a computer science course in K-12 education, according to practical experience, the posttest results were normally distributed. The exceptions were exercises on arrays, which showed a tendency toward an equal distribution with a slight shift toward lower values. Overall, a median of 50% was achieved, with the median for inheritance exercises (57%) above the total median and the median for array exercises (38%) below it. Besides the differences in the quality of knowledge about these topics, the following aspects should be kept in mind when interpreting these absolute values:

- Students only have a limited time to complete the test. Therefore, the exercises that come last in the order may yield worse results because students may run out of time. For courses a and b, the array exercise was the last one in the order; for course c, the exercise about reference attributes was the last one. Spot-the-mistake exercises, which come earlier and contain arrays for some and inheritance for others, make up only a minority of points for the respective topic.
- To constructively align practice and examination, different styles of exercises were chosen (e.g. inheritance focused on the duality of diagram and code, while arrays had a deeper focus on algorithms and implementation). Practical experience shows that coding exercises are usually harder to solve on paper than diagram and modification exercises.

As with the other measures, the posttest results are much more interpretable with a relative rather than an absolute approach.

5.2 Results for RQ1

To analyze the relation between students' prior knowledge, their behavior when working on repetition exercises, and their working behavior and success when working on new topic exercises, the correlations among the available measures have been calculated. As described before, most measures weren't normally distributed. Therefore, Spearman's rank correlation was applied, and the results can be interpreted as follows according to Cohen [4]:

effect	$\lvert\rho\rvert$
minor/weak	≥ 0.1
medium/moderate	≥ 0.3
major/strong	≥ 0.5

For all correlations described below, values for the full group and for gender groups have been calculated. For male and female students, the correlation values did not deviate strongly from those of the full group. The values differed slightly in intensity, and the significance values were considerably worse due to the smaller groups. Figures 9, 10, 11, and 12 give an overview of correlation values for the whole group and will be analyzed in the following paragraphs.

Internal Correlation of Repetition Measures. For each exercise group, the score, error quotient, submission count, and build failure count are related. Before comparing the repetition and new-topic exercises, examining the internal correlation for the repetition exercises will provide an initial intuition for the data. There we see a moderate effect ($\rho = 0.44$) of students with high scores also having higher error quotient values while having fewer build failures ($\rho = -0.42$). A significant correlation between the score and submission count within repetition exercises does not exist, which is easily explained by the fact that students can submit as often as they like to improve their scores. Students with higher EQ values in repetition exercises also had moderately lower submission counts ($\rho = -0.33$) and build failures ($\rho = -0.42$), indicating that students with higher submission counts kept making similar mistakes, thereby raising their error quotient. The weak correlation ($\rho = 0.26$) between submission and build failure counts was expected, as more submissions increase the likelihood of build failures.

Internal Correlation compared. When comparing the internal correlation values across exercises grouped by advanced topics and repetition, it's noticeable that some measures show completely different correlations between topics, while others are consistent across them. Only two of the six correlation values are consistent across all topics. Those are a moderate correlation between submission count and build failures, and a moderate-to-strong correlation between high scores and the error quotient, peaking at $\rho = 0.74$ for inheritance exercises. For inheritance and reference attribute exercises, the score and submission count are positively correlated, with strong effects for inheritance exercises ($\rho = 0.59$) and weak effects for reference attribute exercises ($\rho = 0.28$), while no significant correlation between these measures was found for repetition and array exercises. Surprisingly, only repetition exercises showed a significant (negative) correlation between score and build failure count. While repetition exercises also showed a moderate negative correlation between submission count and error quotient, among the advanced topics, only reference attribute exercises showed a similar relation. Inheritance and array exercises did not show a significant correlation. The latter also applies to a correlation between build failures and error quotient values. However, comparing repetition exercises and reference attribute exercises to focus on correlations between these two measures, we see contrasting moderate effects: higher error quotients for higher submission counts in reference attribute exercises, and exactly the opposite in repetition exercises. Based on the given data, no

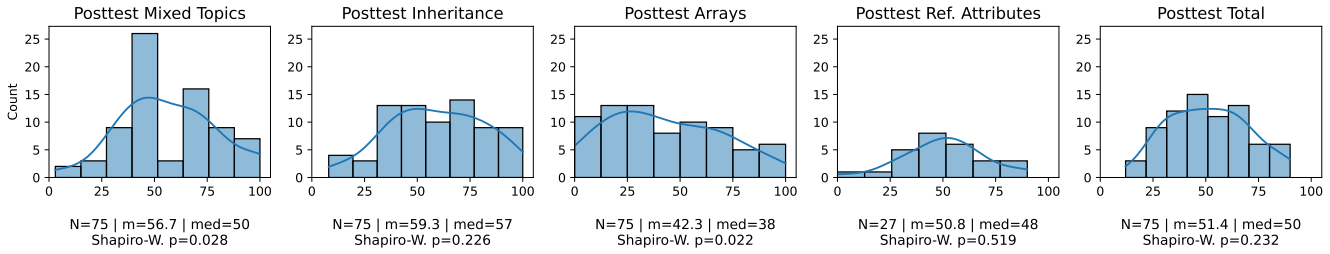


Figure 8: Posttest results by topic and total

explanation could be found for this. As seen in section 5.1, only very few students submitted build failures, which might distort this relation when one student had a very high build-fail count on one of the exercises. Figure 5 shows that for advanced topic exercises (which reference attribute exercises belong to), some students had significantly higher build fail counts than others, which supports this thesis.

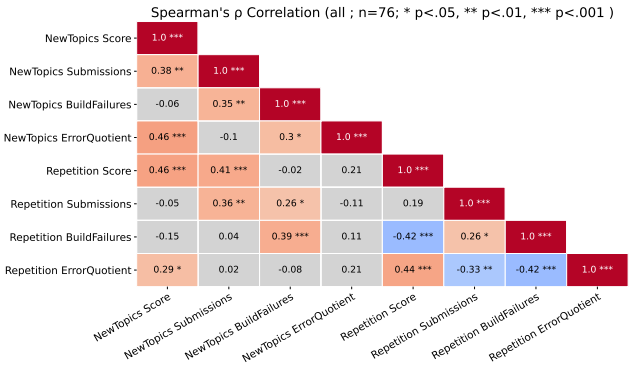


Figure 9: Correlation: Repetition and New Topics

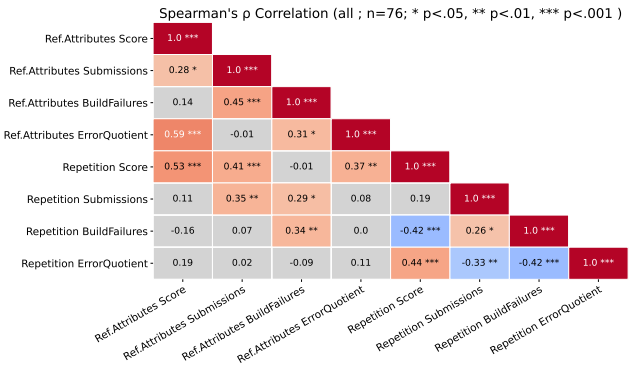


Figure 10: Correlation: Repetition and Ref. Attributes

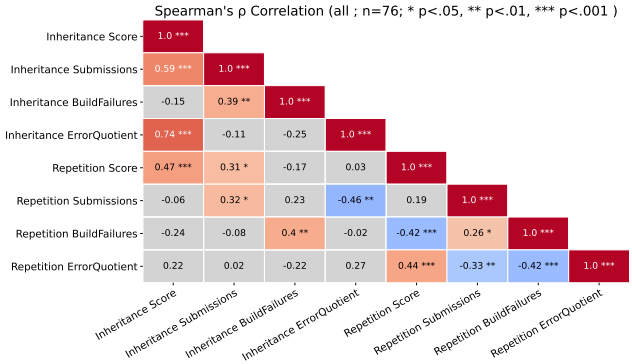


Figure 11: Correlation: Repetition and Inheritance

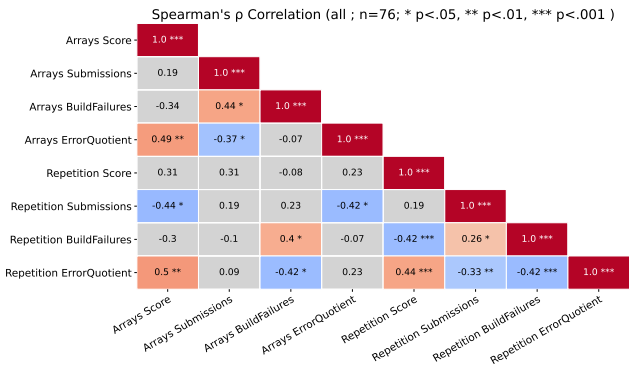


Figure 12: Correlation: Repetition and Arrays

Exercises on New Topics vs. Repetition. When looking for a correlation between success and working behavior of repetition exercises and the ones on new topics, one measure has similar moderate effects around $\rho = 0.4$ for repetition exercises with the three new topics and the overall average of the new topic exercises: Students who had many build failures in repetition exercises kept making those mistakes in advanced exercises, too. As mentioned before, the autograder does not provide helpful feedback on syntax errors and does not support students in improving in this area.

Besides, syntax usually gets more complicated in advanced topics, whereas students who have trouble with basic syntax will have even more trouble in the advanced exercises.

All other correlations appear to be highly dependent on the exercises' topics. Array exercises have a much more algorithmic approach, while reference attribute and inheritance exercises are more object-oriented. These differences also show in similar structured correlations between reference attributes/inheritance exercises with repetition exercises, whereas array exercises exhibit completely different correlations with repetition exercises. While higher scores on repetition exercises are strongly correlated with high scores on the first two advanced topics, no such effect was observed for arrays. Similar for submission counts in the advanced exercises: No correlation with any measure of the repetition exercises was observed for arrays, while higher submission counts in reference attribute and inheritance exercises were moderately correlated with higher scores and submission counts in repetition exercises. The correlation between high scores/submission counts in prior knowledge exercises and high scores/submission counts in advanced exercises supports our hypotheses, as topics build on each other, and students usually maintain a consistent work behavior. However, the correlation between high scores on repetition exercises and high submission counts in the first two advanced topics is surprising but can be explained by good students who want to verify that they are working correctly before proceeding to the next subtask, and maybe even rely on the instructive feedback.

Unfortunately, the multiple linear regression analysis did not yield any results that provide deeper insights into the relations or add value in any other way for answering RQ1.

5.3 Results for RQ2 and RQ3

To answer research questions 2 and 3, which investigate the influence of prior knowledge and advanced topic exercises on the learning outcomes of advanced topics, about 700 plausible combinations of variables were analyzed using multiple linear regression to identify significant influences on the learning outcome. Most results were not significant, especially those with more than 2 independent variables. Nevertheless, some significant influences could be identified. All regression coefficients are standardized, meaning distortion caused by the variable's scale (e.g., $EQ \in (0, 100]$ vs. submission count usually $\in [0, 15]$) is removed, allowing direct comparison of the coefficients. As described in previous sections, the validity of statistical calculations based on build failure counts is doubted. Even though some regression calculations yielded significant coefficient values for build failure values, they are left out.

Total outcomes. For the total posttest results, two regressions had significant coefficients: The one comparing total and repetition submission counts (figure 13) and the one comparing the influence of scores in exercises about advanced topics with scores in repetition exercises (14). Comparing scores in advanced versus repetition exercises shows a slightly better learning outcome when students achieve higher scores in repetition exercises than in advanced ones. However, the absolute coefficient values are close to each other, having moderate effects around $R = 0.3$. This is especially highlighted when comparing with the influence of the submission

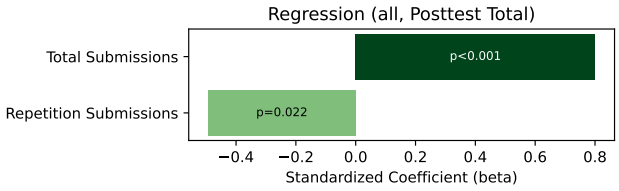


Figure 13: Regression: PT Total vs. Submission Count

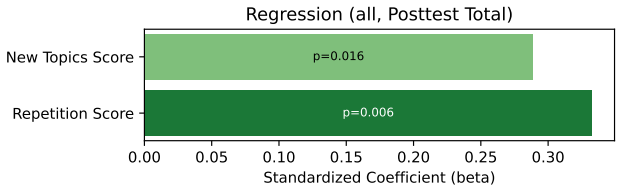


Figure 14: Regression: PT Total vs. Score

counts. The total submission count has a strong positive influence ($R = 0.8$) while submission counts in repetition exercises have a moderate to strong negative effect ($R \approx 0.8$) on the total posttest results. For interpreting this regression result, it is important to consider that the average submission count per repetition exercise is calculated from values that are also part of the average submission count of all exercises. However, as the influences work in opposite directions and a mutual reinforcement of the values is unlikely, the results can still be used to answer the research questions.

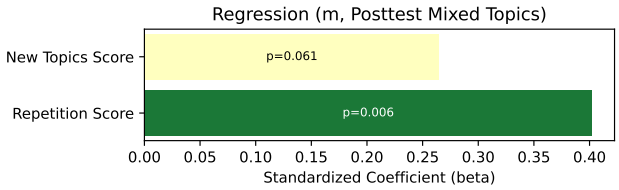


Figure 15: Regression: PT Mixed vs. Score (male)

Mixed Topic Outcomes. Among male students, scores on advanced and repetition exercises both have a positive influence on posttest results for mixed topics (figure 15). For female students, this combination of variables did not yield significant results. The mixed topics exercise is a multiple-choice exercise that covers array indexing, the types of polymorphic objects, and the iteration count of for-loops. Even though the scores of both exercise groups have a positive effect on this posttest category, the effect of repetition exercise scores is almost twice as high as that of the advanced exercise scores.

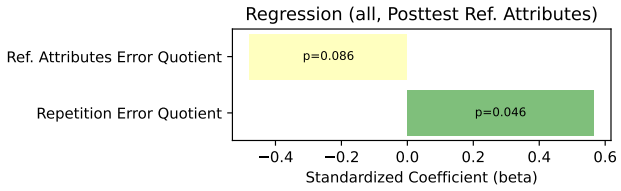


Figure 16: Regression: PT Ref-Attr vs. EQ

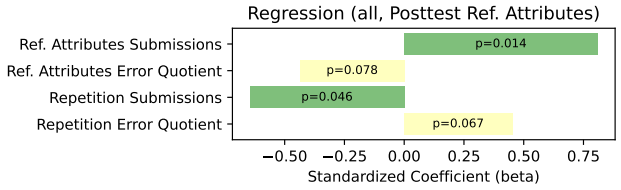


Figure 17: Regression: PT Ref-Attr vs. Behavior

Reference Attribute outcomes. The most expressive results were achieved for the influence on reference attribute results. Figure 16 shows a moderate to strong positive influence of error quotient values in repetition exercises, while high error quotient values in reference attribute exercises have a negative moderate to strong effect on the posttest results about reference attributes. Even more expressive are the results in figure 17, which include all behavior measures, enabling us to directly analyze the influence of working behavior in repetition vs. advanced exercises on posttest results. The results show a strong positive effect of submission counts in reference attribute exercises, whereas the same measure has a strong negative effect in repetition exercises. For the error quotient, it is the other way around. High error quotient values - indicating fast mistake-solving - in exercises about reference attributes have a moderate negative effect on the posttest results, while higher error quotient values in repetition exercises have a moderate positive effect on the results.

6 Discussion

6.1 RQ1

For the object-oriented topics, Reference Attributes and Inheritance, the hypothesis that prior knowledge has the biggest impact on successfully solving advanced exercises was found to be correct. The data shows a strong correlation between those values. For advanced exercises on arrays, the expected correlation was not observed. This might suggest a programming-paradigm-dependent correlation, or it might be because arrays are the last topic, and students have more experience with the other advanced topics before even starting exercises on arrays.

The hypothesis for RQ1(a) was that low prior knowledge correlates with bad working habits. This hypothesis was not supported by the results. This might have different causes. One might be that it was simply wrong. Another possibility is that bad behavior was defined incorrectly. Initially, many submissions were seen as trying to break the autograder for solving the exercise. However, many commits may also come from students after each subtask to verify that they did the correct thing. As many of the exercises have several subtasks, a more detailed, almost qualitative look into the data would be required to find a proper explanation for this phenomenon and answer the research question.

6.2 RQ2 & RQ3

Only a few statistical tests for research questions 2 and 3 led to significant results. This may be due to too few students, which can limit the ability to achieve significant results in multiple regression. Nevertheless, some tests yielded significant results. Again, this seems to be topic-dependent.

For mixed topic exercises in the posttest of male students, the first hypothesis of prior knowledge having the biggest effect on the learning outcome is strongly supported, while for the total score, this is only barely the case, as the influence of success in repetition and advanced exercises is almost the same.

Similar to the corresponding hypothesis for RQ1(a), the hypothesis that high submission counts lead to bad posttest results could not be verified. As explained before, the definition of bad working behavior must be remodeled for future work. High average submission counts over all exercises had a strong positive influence on the total posttest results, while high submission counts in repetition exercises had a negative influence. The latter would support the hypothesis; however, the first would not. Given this inconsistency, the hypothesis must be rejected in its entirety.

One exception to the non-significant missing insights is the topic of Reference Attributes. The results clearly show that high submission counts with low error quotients on the Reference Attribute exercises, and vice versa for the repetition exercises, lead to good learning outcomes for this topic. One possible explanation is that students with good prior knowledge (low submission count and high error quotient in repetition) receive a lot of (different) feedback on the new topic and therefore have many opportunities to learn from their mistakes. This idea might be worth investigating in future work.

6.3 Limitations

Missing Data for Build Failures. Several results showed interesting starting points for an analysis of the influence of build failures and their resolution.

Different Posttests. Due to scheduling, different courses had different posttest exercises and, in some cases, even different topics (e.g., reference attributes were covered only in course c). This significantly weakens the results' power.

Long Exercises. Many of the exercises consist of many subtasks, which makes it hard to determine if students simply submitted after each subtask or actually made mistakes. The design of the error quotient tries to address this limitation but cannot fully resolve it.

Students skipping Exercises. As mentioned before, several students skipped or did not finish exercises for an unknown reason. Knowledge of the reason for this would be important for properly interpreting these values and not just leaving them out.

7 Summary and Future Work

Similar to the original study, from which this one was adapted, many results lack significance and are highly topic-dependent. Nevertheless, some good starting points emerge from it, showing that prior knowledge is extremely important for successfully learning slightly advanced programming topics. However, the role of autograders in K-12 computer science education and their influence are topics that need further investigation. While this study immediately adapted the exercises from the original study [8], future work must fundamentally remodel the exercises to resolve the limitations mentioned before by creating more compact and easier-to-interpret exercises. Nevertheless, autograding in K-12 CS education remains a highly interesting and promising topic.

Declaration of the use of artificial intelligence

In preparing this thesis, I used Grammarly[1] for grammar and style corrections throughout all sections, ensuring clarity and coherence in my writing. I used DeepL[2] to enhance language quality and translate minor parts of the literature. Additionally, I used GitHub Copilot [3] to speed up the creation of Python and SQL code snippets for data analysis. Besides that, I used Consensus [4] and Google Scholar Labs [5] for literature research. Concluding, I used Gemini [6] to *verify* the paper's consistency.

References

[1] 3/28/2026. Greenfoot | About Greenfoot. https://www.greenfoot.org/overview

[2] Elisa Baniassad, Lucas Zamprogno, Braxton Hall, and Reid Holmes. 2021. STOP THE (AUTOGRADER) INSANITY: Regression Penalties to Deter Autograder Overreliance. In *Proceedings of the 52nd ACM Technical Symposium on Computer Science Education (ACM Digital Library)*, Mark Sherriff (Ed.). Association for Computing Machinery, 1062–1068.

[3] Garvin Brod. 2021. How Can We Make Active Learning Work in K-12 Education? Considering Prerequisites for a Successful Construction of Understanding. *Psychological science in the public interest : a journal of the American Psychological Society* 22, 1 (2021), 1–7.

[4] Jacob Cohen. 1988. *Statistical power analysis for the behavioral sciences (2nd ed.).* Hillsdale, N.J.: L. Erlbaum Associates.

[5] Priska Flandorfer. 2023. Durchführung und Interpretation der Regressionsanalyse. https://www.scribbr.de/statistik/regressionsanalyse/

[6] Hagit Gabbay and Anat Cohen (Eds.). 2022. *Investigating the effect of Automated Feedback on learning behavior in MOOCs for programming: Zenodo.*

[7] Jürgen Hedderich and Lothar Sachs. 2020. *Angewandte Statistik: Methodensammlung mit R* (17., überarbeitete und ergänzte auflage ed.). Springer Spektrum, Berlin and Heidelberg. https://link.springer.com/content/pdf/10.1007/978-3-662-62294-0.pdf

[8] Valentin Herrmann. 2024. *Comparing different ways to give automated feedback on programming exercises in K-12 education.* Ph. D. Dissertation. Technical University of Munich. https://valentin-herrmann.com/BEd-Thesis/

[9] Tony Jenkins. 2002. ON THE DIFFICULTY OF LEARNING TO PROGRAM. (2002).

[10] Journal of Computing Sciences in Colleges. 2001. Introduction to BlueJ: a Java development environment: Journal of Computing Sciences in Colleges: Vol 16, No 3.

[11] Gregor Kennedy, Carleton Coffrin, Paula de Barba, and Linda Corrin. 2015. Predicting success: how learners' prior knowledge, skills and activities predict MOOC performance. (2015), 136–140.

[12] Hieke Keuning, Jeuring, and Bastiaan Heeren. 2018. A Systematic Literature Review of Automated Feedback Generation for Programming Exercises. (2018). https://dl.acm.org/doi/pdf/10.1145/3231711

[13] Wolfgang Kohn. 2005. *Statistik: Datenanalyse und Wahrscheinlichkeitsrechnung.* Springer Berlin Heidelberg, Berlin, Heidelberg.

[14] Stephan Krusche and Andreas Seitz. 2018. ArTEMiS - An Automatic Assessment Management System for Interactive Learning. In *Proceedings of the 49th ACM Technical Symposium on Computer Science Education*, Tiffany Barnes, Daniel Garcia, Elizabeth K. Hawthorne, and Manuel A. Pérez-Quiñones (Eds.). ACM, 284–289.

[15] LehrplanPLUS Bayern. 10/13/2025. LehrplanPLUS - Gymnasium - 9 - Informatik - Fachlehrpläne. https://www.lehrplanplus.bayern.de/fachlehrplan/gymnasium/9/informatik

[16] Joan Lu, Bhavya Krishna Balasubramanian, Mike Joy, and Qiang Xu. 2026. Survey and Analysis for the Challenges in Computer Science to the Automation of Grading Systems. *ACM Computing Surveys* 58, 1 (2026), 1–37.

[17] Andrew Luxton-Reilly, Ewan Tempero, Nalin Arachchilage, Angela Chang, Paul Denny, Allan Fowler, Nasser Giacaman, Igor Kontorovich, Danielle Lottridge, Sathiamoorthy Manoharan, Shyamli Sindhwani, Paramvir Singh, Ulrich Speidel, Sudeep Stephen, Valerio Terragni, Jacqueline Whalley, Burkhard Wuensche, and Xinfeng Ye. 2023. Automated Assessment: Experiences From the Trenches. In *ACE '23: Proceedings of the 25th Australasian Computing Education Conference.* 1–10.

[18] Samiha Marwan, Ge Gao, Susan Fisk, Thomas W. Price, and Tiffany Barnes. 2020. Adaptive Immediate Feedback Can Improve Novice Programming Engagement and Intention to Persist in Computer Science. (2020), 194–203.

[19] José Carlos Paiva, José Paulo Leal, and Álvaro Figueira. 2022. Automated Assessment in Computer Science Education: A State-of-the-Art Review. *ACM Transactions on Computing Education* 22, 3 (2022), 1–40.

[20] Yizhou Qian and James Lehman. 2017. Students' Misconceptions and Other Difficulties in Introductory Programming. (2017). https://dl.acm.org/doi/pdf/10.1145/3077618

[21] Johan Snider. 2024. Edit, Run, Error, Repeat: Learning Analytics To Identify Most Improved Programming Student. (2024). https://www2.it.uu.se/student/thesis_project/links2/cer_thesis.pdf

[1] rammarly.com
[2] deepl.com
[3] github.com/features/copilot
[4] consensus.app
[5] scholar.google.com/scholar_labs
[6] gemini.google.com

A More Raw Analysis Plots

Raw plots from the statistical analysis can be found online under the following address:

https://valentin-herrmann.com/Hauptseminar/

If the page becomes accidentally unavailable in the future, do not hesitate to contact the author.

B Practice Exercises with automated Feedback (original, German)

Exercise 02 is a modeling/diagram exercise without automated feedback and was therefore left out.

Figure 18: Ex. 01 - Basics

Lade dir die Vorlage herunter.

Teil 1: Der selbe Song nochmal Keine Ergebnisse

1. Übernimm für die Klasse `Hello` den Code von den Folien aus dem Unterricht in die leere Klasse. Achtung: Die erste Zeile mit `package` muss unbedingt stehen bleiben!
2. Probiere aus, ob der Code korrekt funktioniert.
3. Sende deinen Programmcode zu Artemis ab.
4. Wird der Text oben grün, hast du alles richtig gemacht. Ansonsten kann du auf den roten unterstrichenen Text klicken und dir wird angezeigt was falsch ist. Nach dem Korrigieren, kannst du den Code erneut hochladen.

Noch nicht so richtig durchgestiegen, was das jetzt alles bedeutet? Schaut euch die folgenden SimpleClub-Videos an:

- Java Tutorial 2 Programm-Elemente Einstieg
- Erste eigene Klasse - Objektorientierte Programmierung in Java Teil 2

Teil 2: Neuer Song Keine Ergebnisse

1. Erstelle in der leeren Klasse `MeineKlasse` die Methoden und Attribute, die im Klassendiagramm beschrieben sind. Den Methodenrumpf lässt du erstmal leer. Tipp: Ein `+` vor dem Attribut/der Methode steht für `public`, ein `-` für `private`.
2. Dem Attribut `zahl` soll zu Beginn der Wert 0 zugewiesen werden.
3. Lade deinen Code wieder hoch und schaue, überprüfe die Ergebnisse.

4. Jetzt kommt der Inhalt der Methoden:
   - Die Methode `neueTextAusgabe()` soll zwei beliebige Zeilen Text auf der Konsole ausgeben. Du bestimmst also, was jeweils ausgegebene wird.
   - Die Methode `zahlAnpassen()` soll "Die Zahl ist jetzt " und anschließend den Wert des Attributs `zahl` ausgeben. Dann soll der Wert auf den des Parameters gesetzt werden und wieder der entsprechende Text in einer neuen Zeile ausgegeben werden. Die Ausgabe soll also in etwa so sehen:

```
Die Zahl ist jetzt 0
Die Zahl ist jetzt 5
```

Figure 19: Ex. 03 - Methods

Parameter und Rückgabewerte

Für die meisten Programme ist es unpraktisch, die errechneten Daten jedes Mal in der Konsole auszugeben (`print`) und anschließend wieder mit `Scanner` einzulesen. Daher können Methoden Übergabewerte (Parameter) und Rückgabewerte (Return-Values) besitzen.

Nicht mehr sich, wie das funktioniert? Unten findest du Beispiele und Erklärungen oder schaue das Simpleclub Video zu Methoden mit Parameter und/oder das Video zu Rückgabewerten

Beispiele

▸ Auf-/Zuklappen

Aufgabe

Taschenrechner Keine Ergebnisse

Setze folgendes Klassendiagramm um und implementiere dabei auch den Inhalt der Methoden. Vergiss nicht, deine Methoden auszuprobieren, bevor du hochlädst!

Da alle Methoden static (also Klassemethoden) sind, musst du keine Objekt erstellen, sondern kannst die Methoden direkt mit Rechtklick auf die Klasse starten.

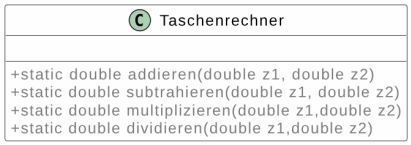


Figure 20: Ex. 04 (1) - Loops

Schleifen

Schleifen gehören zu den wichtigsten Elementen beim Programmieren. In dieser Aufgabe verwenden wir die beiden wichtigsten Arten: For-Schleifen (auch als Zählschleife bekannt) und While-Schleifen (Schleifen mit Bedingung).

Für jede Aufgabe ist bereits eine Methode vorbereitet. Bearbeite die Aufgaben immer in der richtigen Methode, sonst findet Artemis deine Bearbeitung nicht! In den meisten Fällen haben die Methoden Parameter, deren Werte erst beim Ausführen festgelegt werden. Wenn du da noch nicht ganz fit bist, hilft dieser Teil von Jonas Keils Video zu Methoden (die Wiedergabe beginnt automatisch bei 6:22 Min.) weiter.

While-Schleifen

Schaue bevor du mit der Aufgabe beginnst wie immer zuerst das Video von Jonas-Keil zu While-Schleifen.

Löse nun folgende Aufgaben mit einer While-Schleife:

Aufgabe 1: Zählen von 5 bis zum Parameter (1,5 BE) Keine Ergebnisse

Gib alle ganzen Zahlen von 5 bis einschließlich dem Wert des Parameters `limit` in jeweils einer eigenen Zeile auf der Konsole aus.

Beispielsweise für den Aufruf `Aufgaben.aufgabe1(11);` wäre die Ausgabe:

```
5
6
7
8
9
10
11
```

Aufgabe 2: Quadratzahlenliste (2 BE) Keine Ergebnisse

Gib bei 0 beginnend so viele aufeinanderfolgende Quadratzahlen wie im Parameter `limit` angegeben in jeweils einer eigenen Zeile auf der Konsole aus. Achte hierbei genau auf die Formatierung der Ausgabe!

Beispielsweise für den Aufruf `Aufgaben.aufgabe2(4);` wäre die Ausgabe:

```
0 * 0 = 0
1 * 1 = 1
2 * 2 = 4
3 * 3 = 9
```

Figure 21: Ex. 04 (2) - Loops

For-Schleifen

Schaue bevor du mit Abschnitt 2 der Aufgabe beginnst zuerst das Video von Jonas-Keil zu For-Schleifen.

Löse nun folgende Aufgaben mit einer For-Schleife:

Aufgabe 3: Countdown (1,5 BE) Keine Ergebnisse
Schreibe ein Programm, dass vom Parameter `start` rückwärts bis einschließlich zum Parameter `ende` zählt und jede Zahl in einer eigenen Konsolenzeile ausgibt. Überlege dir hierbei auch, was dein Programm tun soll, wenn `start < ende`.

Beispielsweise für den Aufruf `Aufgaben.aufgabe3(5,-1);` wäre die Ausgabe:

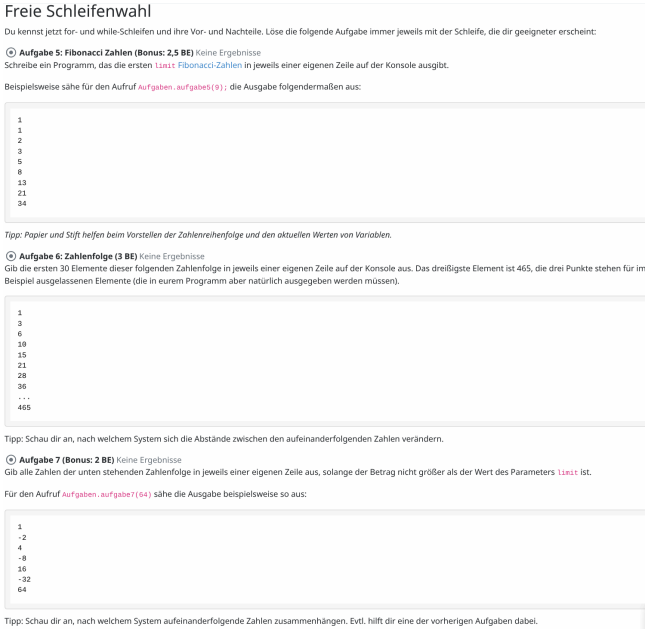
```
5
4
3
2
1
0
-1
```

Aufgabe 4: Nochmal von 5 zählen (1,5 BE) Keine Ergebnisse
Gib alle ganzen Zahlen von 5 bis ausschließlich dem Wert des Parameters `limit` (also nur alles was kleiner ist) in jeweils einer eigenen Zeile auf der Konsole aus. Vergleiche deinen Code auch mit `aufgabe1` und suche Gemeinsamkeiten.

Beispielsweise für den Aufruf `Aufgaben.aufgabe4(11);` wäre die Ausgabe:

```
5
6
7
8
9
10
```

Figure 22: Ex. 04 (3) - Loops



Freie Schleifenwahl

Du kennst jetzt for- und while-Schleifen und ihre Vor- und Nachteile. Löse die folgende Aufgabe immer jeweils mit der Schleife, die dir geeigneter erscheint:

Aufgabe 5: Fibonacci Zahlen (Bonus: 2,5 BE) Keine Ergebnisse
Schreibe ein Programm, das die ersten `limit` Fibonacci-Zahlen in jeweils einer eigenen Zeile auf der Konsole ausgibt.

Beispielsweise sähe für den Aufruf `Aufgaben.aufgabe5(9);` die Ausgabe folgendermaßen aus:

```
1
1
2
3
5
8
13
21
34
```

Tipp: Papier und Stift helfen beim Vorstellen der Zahlenreihenfolge und den aktuellen Werten von Variablen.

Aufgabe 6: Zahlenfolge (3 BE) Keine Ergebnisse
Gib die ersten 30 Elemente dieser folgenden Zahlenfolge in jeweils einer eigenen Zeile auf der Konsole aus. Das dreißigste Element ist 465, die drei Punkte stehen für im
Beispiel ausgelassenen Elemente (die in eurem Programm aber natürlich ausgegeben werden müssen).

```
1
3
6
10
15
21
28
36
...
465
```

Tipp: Schau dir an, nach welchem System sich die Abstände zwischen den aufeinanderfolgenden Zahlen verändern.

Aufgabe 7 (Bonus: 2 BE) Keine Ergebnisse
Gib alle Zahlen der unten stehenden Zahlenfolge in jeweils einer eigenen Zeile aus, solange der Betrag nicht größer als der Wert des Parameters `limit` ist.

Für den Aufruf `Aufgaben.aufgabe7(64)` sähe die Ausgabe beispielsweise so aus:

```
1
-2
4
-8
16
-32
64
```

Tipp: Schau dir an, nach welchem System aufeinanderfolgende Zahlen zusammenhängen. Evtl. hilft dir eine der vorherigen Aufgaben dabei.

Figure 23: Ex. 05a - Conditions

Zufallszahlen vergleichen Keine Ergebnisse
Die drei bestehenden Zeilen in der Vorlage generieren zwei zufällige Ganzzahlen. Deine Aufgabe ist, diese miteinander zu vergleichen und dein Ergebnis in der Konsole auszugeben. Verwende hierfür die (Un)Gleichheitszeichen `=`,`<`,`>`. Du musst keine Leerzeichen verwenden.

Ausgabe wenn die Zufallszahl `15` und `26` wären:

```
15<26
```

Figure 24: Ex. 05b - Conditions

Einmal-Eins-Tabelle mit Blocksatz Keine Ergebnisse
Gib für das 25er-Einmal-Eins eine Ergebnistabelle aus. Hierbei sollen alle Zeilen exakt gleich lang sein. Bei Zahlen mit weniger Ziffern als die größte Zahl müssen daher führende Nuller eingefügt werden. Die einzelnen Zahlen werden dabei durch genau ein Leerzeichen getrennt. Für das 5er-Einmal-Eins würde die Tabelle folgendermaßen aussehen:

```
01 02 03 04 05
02 04 06 08 10
03 06 09 12 15
04 08 12 16 20
05 10 15 20 25
```

Tipp: Eine Ausgabe, nach der **keine neue Zeile** begonnen wird, macht man mit folgender Methode: `System.out.print("Ausgabetext")`

Figure 25: Ex. 05c (optional) - Conditions

Alle Teiler bis 200 Keine Ergebnisse
Gibt für alle positiven ganzen Zahlen bis 200 alle ihre ganzzahligen Teiler aus. Beginne hierbei für jede neue Zahl eine neue Zeile und trennen die Teiler mit einem Leerzeichen. Die ersten vier Zeilen deiner Ausgabe sollen folgendermaßen aussehen:

```
1 ist teilbar durch: 1
2 ist teilbar durch: 1 2
3 ist teilbar durch: 1 3
4 ist teilbar durch: 1 2 4
...
```

Figure 26: Ex. 05d (optional) - Conditions

Random-Chatbot Keine Ergebnisse

Programmiere einen einfachen Chatbot nach folgenden Vorgaben:

- Der Nutzer beginnt mit der ersten Eingabe. Vorher gibt der Chatbot keinen Text aus.
- Man kann dem Chatbot drei verschiedene Fragen stellen (auf Tippfehler achten!):
  - "Was ist dein Name?"
  - "Wie alt bist du?"
  - "Was machst du gerade?"
- Für jede dieser Fragen kennt der Chatbot jeweils 3 verschiedene Antworten. Insgesamt muss es also **9 verschiedene Antworten** geben.
- Welche Antwort der Chatbot auf die jeweilige Frage gibt, wird jedes Mal zufällig ausgewählt. (Tipp: Zufallszahlen)
- Bei allen anderen Nutzereingaben, soll folgender Text ausgegeben werden: "Tut mir leid, diese Frage kann ich leider nicht beantworten."
- Nach der Antwort des Chatbots endet das Programm. Für eine neue Eingabe, muss die Methode händisch neugestartet werden.

In der Vorlage werden bereits ein Scanner für die Nutzereingaben und ein Zufallszahlengenerator angelegt.

- Nicht mehr sicher wie der Scanner funktioniert? --> Hier gibt es ein Video dazu
- Der Zufallszahlen-Generator wird in den ersten 2 Minuten dieses Videos erklärt

```
public static void aufgabe4()
{
    // Diese zwei Zeilen gehoeren zur Vorlage und duerfen
keinesfalls geaendert werden!
    Random random = new Random();
    Scanner scanner = new Scanner(System.in);

    // TODO: Unter diesem Kommentar die Aufgabe bearbeiten.

}
```

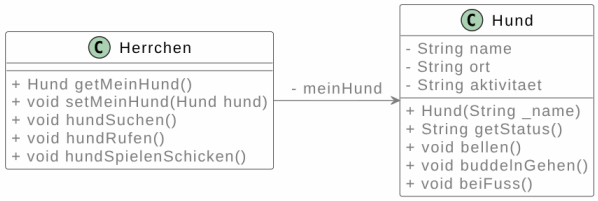
Figure 27: Ex. 06a - Reference Attributes

Der Hund besitzt einige (schon fertig programmierte) Fähigkeiten (Methoden), die sein Besitzer verwenden kann, indem er sie über das Referenzattribut aufruft.

Zur Erinnerung, das geht beispielsweise so: `meinHund.beiFuss();`

Hierzu benötigt die Klasse `Herrchen` ein Referenzattribut `meinHund` mit Datentyp `Hund` (siehe auch Beziehungspfeil im Klassendiagramm).

1. **Implementiere das Referenzattribut und weise ihm ein neues Objekt der Klasse Hund als Wert zu.** Keine Ergebnisse
2. **Implementiere den Methodenrumpf für getMeinHund() als Getter für das Attribut meinHund** Keine Ergebnisse
3. **Implementiere den Methodenrumpf für setMeinHund(Hund hund) als Setter für das Attribut meinHund** Keine Ergebnisse
4. **Implementiere den Inhalt der Methoden der Klassen Herrchen** Keine Ergebnisse



Ändere nichts an der Klasse Hund oder den Methodenköpfen von Herrchen! Keine Ergebnisse

Achutng: Manche Testergebnisse werden erst angezeigt, wenn alle Voraussetzungenn erfüllt wurden!

Figure 28: Ex. 06b - Reference Attributes

1. **Klassenstruktur implementieren** Keine Ergebnisse
   Implementiere die Methoden (zunächst leer) und die Attribute von Fahrer (siehe Klassendiagrammsunten).
2. **Methoden implementieren** Keine Ergebnisse
   Im Konstruktor von Fahrer soll hierbei ein neues Objekt der Klasse Fahrzeug angelegt und dem entsprechenden Attribut zugewiesen werden. Der Konstruktor von Fahrzeug benötigt als Parameter ein Objekt der Klasse Fahrer. Das aktuelle Fahrerobjekt, kann in Methoden der Klasse mit `this` verwendet (und auch als Parameter übergeben) werden. Die Benennung der restlichen Methoden beschreibt bereits, was getan werden soll.

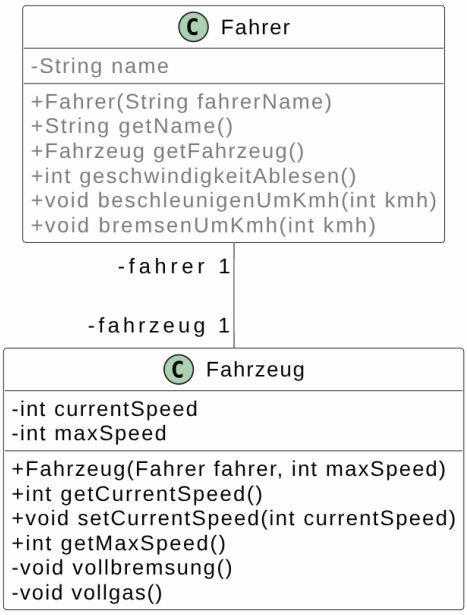


Figure 29: Ex. 06c (optional) - Reference Attributes

Ein `Kunde` kann bei seiner Bank zwei Konten besitzen: ein Girokonto und ein Sparkonto. Möchte er mit diesen Konten interargieren, nutzt er hierfür einen `Geldautomat`. Jeder Kunde hat einen `lieblingsautomat`, der im entsprechenden Referenzattribut gespeichert wird. Die Klassen `Geldautomat` und `Konto` sind bereits fertig. Die Klasse `Kunde` musst du noch fertigstellen. Wie die Klassen miteinander zusammenhängen, siehst du unten im Klassendiagramm.

1. **Deklariere das Referenzattribut lieblingsautomat und weise ihm ein neues Objekt der Klasse Geldautomat als Wert zu.** Keine Ergebnisse
2. **Deklariere die Referenzattribute Sparkonto und Girokonto, weise ihnen aber noch keine Werte zu** Keine Ergebnisse
3. **Weise den beiden Konto-Referenzattributen in der Methode kontenAnlegen() jeweils ein neues Objekt mit der Art 'Girokonto' bzw. 'Sparkonto' zu.** Keine Ergebnisse
4. **Implementiere den Methodenrumpf für setLieblingsautomat(Geldautomat automat) als Setter für das Attribut lieblingsautomat** Keine Ergebnisse
5. **Implementiere den Methodenrumpf für die Getter-Methoden getLieblingsautomat(), getGiroKonto() und getSparKonto()** Keine Ergebnisse
6. **Implementiere den Inhalt der Methoden der Klassen Kunde, indem du Methoden deine Lieblingsautomats aufrufst.** Keine Ergebnisse
   Tipp: Direkte Abhebungen und Einzahlungen sind hierbei nur vom Girokonto möglich. Eine Interaktion mit dem Sparkonto ist nur per Überweisung möglich.

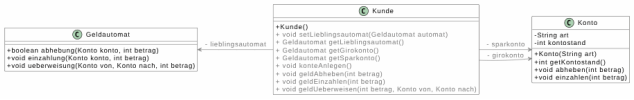
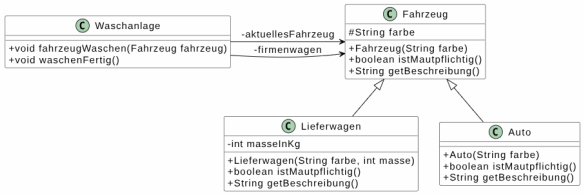


Figure 30: Ex. 07a - Inheritance

Die Software einer Waschanlage soll in der Lage sein, verschiedene Arten von Fahrzeugen zu verwalten. Die allgemeine Fahrzeugklasse ist bereits fertig implementiert. Ebenso gibt es bereits die leeren Klassen Auto und Lieferwagen.

1. **Bearbeite Auto und Lieferwagen so, dass sie Unterklassen von Fahrzeug werden. Lege hierbei jeweils einen Konstruktor mit den gleichen Parametern wie bei Fahrzeug an und vergiss nicht den Konstruktor der Oberklasse aufzurufen.** Keine Ergebnisse
2. **Verändere 'Fahrzeug' so, dass in den Unterklassen auf alle Attribute direkt zugegriffen werden kann.** Keine Ergebnisse
3. **Deklariere in 'Lieferwagen' ein integer-Attribut 'masseInKg' und ergänze einen Konstruktor-Parameter (mit beliebigem Namen), dessen Wert zu dem Attribut zugewiesen wird.** Keine Ergebnisse
4. **Überschreibe in 'Lieferwagen' die Methode 'istMautpflichtig(), sodass true zurückgegeben wird, wenn der Lieferwagen schwerer als 3500kg ist.** Keine Ergebnisse
5. **Überschreibe 'getBeschreibung()' in beiden Unterklassen, sodass 'Fahrzeug' im jeweils zurückgegebenen Text durch 'Auto' bzw. 'Lieferwagen' ersetzt wird.** Keine Ergebnisse
6. **Weise dem Referanzattribut 'firmenwagen' ein neues Objekt der Klasse Lieferwagen mit beliebigen Werten zu** Keine Ergebnisse
7. **In der Waschanlage aktuell nur Autos gewaschn werden. Prüfe daher in 'fahrzeugWaschen(...)', ob es sich um ein Objekt der Klasse Auto handelt und weise den Paramterwert nur dann dem Attribut 'aktuellesFahrzeug' zu.** Keine Ergebnisse

Das Klassendiagramm zeigt keine Ergebnisse an, sondern dient nur zur Übersicht über die Struktur.



Pass auf, dass du nicht versehentlich die Klassennamen oder die Package Anweisung änderst! Keine Ergebnisse

Figure 31: Ex. 07b (1) - Inheritance

Kontext

Wir programmieren die Grundlagen eines Computerspiels. Hierzu benötigen wir:

- ein Gameboard (=das Spielfeld)
- ein allgemeines Entity, das die grundlegenden Funktionen zum Anzeigen auf dem Gameboard umsetzt (Position, Bild, etc.).
- und verschiedene Arten von Entities (z.B. Spieler, Wände, Gegner, ...), die hiervon erben und das jeweilige Verhalten umsetzen.

Das allgemeine Entity enthält zudem mehrere standardmäßig leere Methoden als Vorlage für die Unterklassen. Deren genaue Funktion ist im Programmcode als Kommentar beschrieben.

Ein sehr ähnliches System wird in unserem Projekt verwendet werden!

Aufgaben

1. **Spezielle Entities** Keine Ergebnisse
   Bearbeite `PlayerEntity` und `BlockEntity` so, dass sie Unterklassen von `Entity` werden. Lege hierbei jeweils einen Konstruktor, wie im Klassendiagramm unten beschrieben, an und vergiss nicht den Konstruktor der Oberklasse aufzurufen. Objekte der Klasse `BlockEntity` sollen hierbei immer "resources/cobblestone.png" als Bild nutzen.
2. **Welt erstellen** Keine Ergebnisse
   Die `Main`-Klasse ist dafür zuständig, die Welt zu erstellen und das Spiel zu starten. Die Methode `main(String[] args)` ist bereits fertig. Programmiere in der Methode `createWorld()` folgendes:

- Erstelle eine neues `Gameboard`-Objekt mit `"resources/background.jpg"` als Hintergrundbild und speichere es im Referenzattribut `gameboard`.
- Erstelle ein neues `PlayerEntity`-Objekt mit `"resources/car.gif"` als Bild und beliebigen Koordinaten.
- Das `gameboard` hat eine Methode `add(Entity entity)`, mit der Objekte zu ihm hinzugefügt werden können. Füge dein Spieler-Objekt zum Game`gameboard` hinzu.
- Füge dem Gameboard mindestens 5 neue `BlockEntity`-Objekte an beliebigen Positionen hinzu (dieser letzte Teil wird nicht automatisch überprüft, aber du siehst beim Starten des Spiels sofort, ob es geklappt hat).

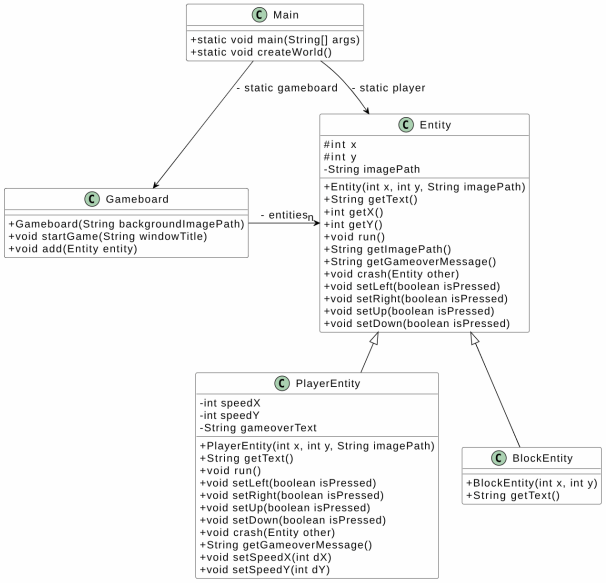
Ab sofort kannst du dein Spiel testen, indem du die main-Methode startest.

Figure 32: Ex. 07b (2) - Inheritance

3. **Koordinaten** Keine Ergebnisse
   Verändere `Entity` so, dass (nur) in den Unterklassen auf die aktuellen x- und y-Koordinaten direkt zugegriffen werden kann.
4. **Bewegung** Keine Ergebnisse
   Überschreibe `run()` in `PlayerEntity`, sodass bei jedem Ausführen der Methode (zur Erinnerung: ca. 25x pro Sekunde) die Koordinaten um die Geschwindigkeit in die jeweilige Richtung geändert werden. Die Geschwindigkeiten in x- bzw. y-Richtung sind jeweils in einem Attribut gespeichert und werden beim Drücken der Pfeiltasten automatisch gesetzt.
5. **Beschriftung** Keine Ergebnisse
   Überschreibe in `PlayerEntity` die Methode `getText()` so, dass immer der Text `"Player"` zurückgegeben wird und in `BlockEntity` so, dass immer `"Block"` zurückgegeben wird.
6. **Game Over** Keine Ergebnisse
   Deklariere in `PlayerEntity` ein String-Attribut `gameoverText` und setze seinen Wert bei der Deklaration auf `""` (leerer Text). Überschreibe außerdem die Methode `getGameoverMessage` in `PlayerEntity` so, dass sie ein Standard-Getter für `gameoverText` ist.
7. **Crash** Keine Ergebnisse
   Überschreibe die Methode `crash(Entity other)` in `PlayerEntity` so, dass bei einem Zusammenstoß mit einem `BlockEntity` (also insbesondere nicht mit einem `PlayerEntity`!) das Attribut `gameoverText` auf einen beliebigen nicht-leere Text gesetzt und das Spiel damit beendet wird.

Figure 33: Ex. 07b (3) - Inheritance

Das Klassendiagramm zeigt keine Ergebnisse an, sondern dient nur zur Übersicht über die Struktur. Manche Klassen sind zudem vereinfacht dargestellt. Sie haben in der Realität wesentlich mehr (für uns aber irrelevante) Attribute und Methoden.

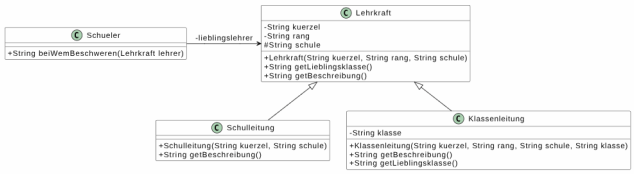


Pass auf, dass du nicht versehentlich die Klassennamen oder die Package Anweisung änderst! Keine Ergebnisse

Figure 34: Ex. 07c (optional) - Inheritance

1. **Setze unten stehendes Klassendiagramm im Programmcode um. Die Klasse 'Lehrkraft' muss dafür NICHT geändert werden! Vergiss nicht, dem Attribut 'klasse' von Klassenleitung den Wert des entsprechenden Konstruktor-Parameters zuzuweisen. Eine Schulleitung hat außerdem immer den Rang 'OStD'.** Keine Ergebnisse
2. **Überschreibe in 'Klassenleitung' die Methode 'getLieblingsklasse()' so, dass die eigene Klasse als Lieblingsklasse zurückgegeben wird.** Keine Ergebnisse
3. **Überschreibe 'getBeschreibung()' in beiden Unterklassen, sodass an den Text der Methode aus der Oberklasse entweder "und ist Klassenleitung der Klasse ..." oder "und ist Schulleitung des ..." mit einem jeweils passenden Wert statt '...' angehängt wird.** Keine Ergebnisse
4. **Weise dem Referanzattribut 'lieblingslehrer' ein neues Objekt der Klasse Klasseleitung mit beliebigen Werten zu** Keine Ergebnisse
5. **Die Methode 'beiWemBeschweren(Lehrkraft lehrer)' gibt zurück, bei wem man sich über die Lehrkraft, die Parameterwert ist, beschweren kann. Das ist bei einer Schulleitung "Ministerialbeauftragter", bei einer Klassenleitung "Schulleitung" und bei einer normalen Lehrkraft "Klassenleitung".** Keine Ergebnisse

Das Klassendiagramm zeigt keine Ergebnisse an, sondern dient nur zur Übersicht über die Struktur.



Pass auf, dass du nicht versehentlich die Klassennamen oder die Package Anweisung änderst! Keine Ergebnisse

Figure 35: Ex. 08a - Arrays

Achte darauf, nichts an den Methodenköpfen zu ändern! Keine Ergebnisse

Aufgaben

Achtung: Dir werden bei den einzelnen Teilaufgaben erst Ergebnisse angezeigt, wenn du begonnen hast, sie zu bearbeiten.

1. **Gib ein int-Array mit den ersten 11 2er-Potenzen (beginnend bei 2^0 und bis 2^10) zurück.** Keine Ergebnisse
2. **Bearbeite das Array, das du als Parameter bekommst, so, dass es an jedem Index genau das doppelte wie vorher enthält und gib es anschließend zurück. Multiplipziere also jedes Element mit 2.** Keine Ergebnisse
3. **Gib das erste Element des Arrays, dass du als Parameter bekommst, zurück.** Keine Ergebnisse
4. **Gib das letzte Element des Arrays, dass du als Parameter bekommst, zurück.** Keine Ergebnisse
5. **Gib die Summe aller Elemente im Array, dass du als Parameter bekommst, zurück.** Keine Ergebnisse

Figure 36: Ex. 08b - Arrays

Achte darauf, nicht an den Methodenköpfen zu ändern! Keine Ergebnisse

Aufgaben

Schulklasse erstellen Keine Ergebnisse
Diese Methode soll ein Array mit `Schueler`-Objekten als Klassenliste erzeugen und zurückgeben. Die Klasse soll 3 Schüler (in diese Reihenfolge!) haben: "Tick","Trick","Track".

Anwesenheit erfassen Keine Ergebnisse
Mit dieser Methode soll die Anwesenheit der Schüler erfasst werden. Hierbei wird ein Schueler-Array als Klassenliste übergeben. Für jedes Schueler-Objekt im Array soll die Anwesenheit gesetzt (Methode: `setAnwesend(boolean anwesend)`) werden und das Array anschließend zurückgegeben werden. Wir gehen davon aus, dass jeder zweite Schüler beginnend mit dem ersten anwesend ist. Also sind z.B. der 1., 3., 5., 7., ... Schüler anwesend.

Tipp: Achte auf den Unterschied zwischen Nummer in der Klassenliste und dem Index im Array.

Schüler anhand der Nummer in der Klassenliste Keine Ergebnisse
Diese Methode soll den x-ten Schüler im übergeben Array zurückgeben. Achte hierbei besonders auf ungültige Nummern. In diesen Fällen soll `null` anstatt eines Objekts zurückgegeben werden. Beachte auch hier wieder den unterschied zwischen Nummer und Index!

Anzahl anwesender Schüler Keine Ergebnisse
Zähle die anwesenden Schüler (anwesend bedeutet `schueler.getAnwesend() == true`) im Array und gib das Ergebnis zurück.

Figure 37: Ex. 08c (optional) - Arrays

Achte darauf, nicht an den Methodenköpfen zu ändern! Keine Ergebnisse

Aufgaben

Für ein Computerspiel möchtest du verwalten, welche Tasten auf der Tastatur gedrückt wurden. Jede Taste hat dafür einen Integer als ID. Die gedrückten Tasten werden in einem Integer-Array anhand der ID gespeichert. Du darfst außerdem davon ausgehen, dass in den als Parameter übergebenen Arrays keine Tasten-IDs doppelt vorkommen.

Array initialisieren Keine Ergebnisse
Erstelle ein leeres Array mit 20 Plätzen (leer = ohne eingefügte Werte = `-1` an jeder Position) und gib es als Return-Wert zurück.

Ist Taste gedrückt? Keine Ergebnisse
Diese Methode soll überprüfen, ob eine bestimmte Taste aktuell gedrückt ist. Überprüfe hierfür, ob die übergebene ID irgendwo im Array ist und gib einen entsprechenden Wahrheitswert zurück.

Taste losgelassen Keine Ergebnisse
Wird eine Taste losgelassen, soll das entsprechende Objekt aus dem Array entfernt werden. Die macht man, indem man die entsprechende Position im Array mit `-1` ersetzt. War die Taste nicht im Array soll nichts gemacht werden. Anschließend wird das Array wieder zurückgegeben.

Anzahl gedrückte Tasten Keine Ergebnisse
Gib die Anzahl der gedrückten Tasten im Array zurück.

C Posttest Exercises

To ensure clarity of the appendix, only one variant of each exercise is shown if the variants do not differ significantly.

Figure 38: PT: Mixed Basics

Grundlagen Kreuze jeweils die korrekten Lösungen an. Mehrere können, mind. eine ist jeweils richtig. / 6

a) Die Klassen *Lehrer* und *Schueler* erben von der Klasse *Mensch*. Die Objekte *mensch1, lehrer1* und *schueler1* sind Objekte der jeweiligen Klasse. Welche der folgenden if-Blöcke werden betreten (=Bedingung wird zu *true* ausgewertet)?

- X if(lehrer1 instanceof Mensch) { ... }
- ○ if(mensch1 instanceof Lehrer) { ... }
- X if(schueler1 instanceof Schueler) { ... }
- X if(schueler1 instanceof Mensch) { ... }

b) Was ist kein zulässiger Zugriff auf ein Element eines dieses Arrays: char[] zeichen = new char[20];

- ○ zeichen[0];
- X zeichen[-1];
- X zeichen[zeichen.length];
- ○ zeichen[1];

c) Wie oft wird der Inhalt einer For-Schleife mit dem folgenden Kopf durchlaufen? *for(int i = -1; i > 10; i -= 2)*

- X 0
- ○ unendlich oft
- ○ 5
- ○ 4

Für jedes korrekte Kreuz 1BE; für jedes falsche Kreuz -0,5BE. Pro Teilaufgabe aber immer ≥0BE

Figure 39: PT: Spot Mistake (Var.: Inheritance)

Fehlersuche In den Klassen *Mensch* und *Lehrer* haben sich 2 Syntax- bzw. Compiler-Fehler eingeschlichen. Beschreibe auf den Zeilen unterhalb kurz, was, wieso und wo der Fehler ist und korrigiere den Java-Code. / 5

```
public class Mensch {

  private String name;

  public Mensch(String _name) {

    name = _name   ;[1BE]

  }
}
```

```
public class Lehrer extends Mensch {

  public Lehrer(String _name) {
      super(_name); [1,5BE]

  }
  public void exKorrigieren() {
     // ...
  }
}
```

Semikolon fehlt in Klasse Mensch, Zeile 4. [1BE]
Konstruktor der Unterklasse muss Konstruktor der Oberklasse aufrufen, da dieser Parameter hat [1,5BE]

Figure 40: PT: Spot Mistake (Var.: Arrays)

Fehlersuche In der Klasse Schueler haben sich 2 Compiler-Fehler eingeschlichen. Beschreibe auf den Zeilen unterhalb kurz, was (und falls relevant, wo) der Fehler ist und korrigiere den Java-Code. / 4

```
public class Schueler {

   public void notenAusgeben(String[] noten) {//Einzelwerte zB."Mathe:2"

      for(int i=0; i < noten.length; i++) {

         System.out.println(noten(i));    statt noten(i): noten[i]   [1BE]
       }[1BE]
   }
}
```

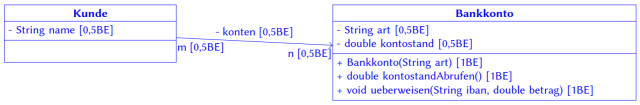
Falsche Klammerart bei Index-Zugriff [1BE]
Eine geschweifte Klammer zu wenig geschlossen [1BE]

Figure 41: PT: Reference Attributes

Modellierung: Referenzattribute Für das Online-Portal einer Bank soll ein Programm erstellt werden, über das Kunden mit ihren Bankkonten interagieren können. Da die Bank keine eigenen Experten für Softwarearchitektur hat, wirst du gebeten, den Kern des Programms zu erstellen: Die Klassen **Kunde** und **Bankkonto**. / 10

- Das Bankkonto soll speichern, was seine Art ist und wie hoch der aktuelle Kontostand ist, während der Kunde nur speichert, wie erst selbst heißt.
- Ein Kunde kann mehrere Bankkonten haben, auf die er zugreifen kann. Ebenso kann jedes Bankkonto mehrere Besitzer haben (z.B. bei Gemeinschaftskonten von Ehepaaren). Ein Bankkonto soll aber nicht auf seine Besitzer zugreifen können.
- Mit ihren Konten sollen Kunden folgendes tun können: Kontostand abrufen, Geld überweisen und beim Erstellen eingeben, wie es heißen soll. Für eine Überweisung muss ein bestimmter Betrag und die IBAN des Empfängers (z.B.: DE12 3456 7891 0111 21) eingegeben werden. Was der Kunde ansonsten ohne das Bankkonto tun kann, ist für das OnlinePortal nicht relevant.
- Soweit nicht anders gefordert, müssen für Attribute keine Getter- und Setter-Methoden enthalten sein.

a) Zeichne für den beschriebenen Einsatz ein vollständiges (UML-)Klassendiagramm mit, unter anderem, Datentypen, Beziehungen und Kardinalitäten. Nutze beim Zeichnen ein Lineal, achte auf korrektes Format und halte dich an (im Unterricht besprochene) Konventionen! Außerdem müssen alle Kardinalitäten explizit angegeben werden.



[0,5BE] für korrekten Beziehungspfeil; [1BE] für Gesamtformat; [0,5BE] pro Klasse
Wenn Methoden in falscher Klasse: jeweils halbe Punktzahl

b) Gibt für alle Referenzattribute an, in welcher Klasse und mit welchem Java-Programmcode du sie deklarieren würdest. Die Zuweisung eines Werts soll nicht enthalten sein. *Hinweis: Es gibt im Folgenden mehr Zeilen, als zu deklarierende Referenzattribute.*

Klasse: Kunde [0,5BE] Deklaration: private Konto[] konten; [1BE]

Klasse: - Deklaration: -

Klasse: - Deklaration: -

Figure 42: PT: Arrays 1

Arrays mit Zahlen
a) Vervollständige die Methode *arrayRechnen()* so, dass zu Beginn der Methode ein Array mit 4 beliebigen Ganzzahlen erzeugt, in der Variable *arr* gespeichert und am Ende zurückgegeben wird. / 5

```
public int[] arrayRechnen() {

    int[] arr = {4, 3, 2, 1}[1BE];
    for(int i=0; i < arr.length; i++) {
        arr[i] = i * arr[i];
    }
    return arr;
}
```

b) Fülle folgende Kästchen so aus, dass sie den Rückgabewert der Methode *arrayRechnen()* visualisieren.
Es soll in jedem Kästchen außerdem dargestellt werden, wie der Wert entstanden ist!

Beispiel: Ein Kästchen enthält den Wert 25, der in der Methode durch $5 * 5$ entstanden ist, würde man eintragen: $5 * 5 = 25$

Ersatzergebnis: Wenn du dein erzeugtes Array aus a) nicht nutzen kannst/möchtest, gehe davon aus, dass anfangs ein Array mit den Werten 11, 42, 420 und 69 erzeugt wurde und **kreuze hier unterhalb an**, dass du das Ersatzergebnis genutzt hast.

- ○ **Ersatzergebnis genutzt**

Lösung mit Musterlösung aus a)

$0 * 4 = 0$ [1BE]	$1 * 3 = 3$ [1BE]	$2 * 2 = 4$ [1BE]	$3 * 1 = 3$ [1BE]

Zusätzlicher Platz, falls du dich verschrieben hast. Bitte deutlich durchstreichen, welches **nicht** gewertet werden soll!
Lösung mit Ersatzergebnis

$0 * 11 = 0$ [1BE]	$1 * 42 = 42$ [1BE]	$2 * 420 = 840$ [1BE]	$3 * 69 = 207$ [1BE]

Figure 43: PT: Arrays 2

Arrays Gegeben ist die fertige Klasse *Schueler* mit unten stehendem Programmcode. / 5

```
public class Schueler {
    private String name;
    private int alter;
    public Schueler(String name, int alter) {
        this.name = name;
        this.alter = alter;
    }
    public String getName() {
        return name;
    }
    public int getAlter() {
        return alter;
    }
}
```

Ergänze die Methode anzahlUnter16 so, dass am Ende die Anzahl der Schüler im übergebenen Array, die **jünger** als 16 sind, zurückgegeben wird.

```
public int anzahlUnter16(Schueler[] arr) {

    int anzahl = 0 [0,5BE] ;

    for( int i = 0; i < arr.length; i++ [1,5BE] ) {

        if( arr[i].getAlter() < 16 [1,5BE] ) {

            anzahl++ [1BE] ;

        }
    }

    return anzahl [0,5BE] ;
}
```

Figure 44: PT: Inheritance 1

Vererbung und Polymorphie Gegeben ist der fertige Programmcode der vier Klassen Fahrzeug, Auto, LKW und Sattelzug. Beide Teilaufgaben a) und b) beziehen sich auf diesen Programmcode. / 10

```
public class Fahrzeug {
    protected String marke;
    public Fahrzeug(String marke) {
        this.marke = marke;
    }
}
```

```
public class LKW extends Fahrzeug {
    public LKW(String marke) {
        super(marke);
    }
    public String getDaten() {
        return "LKW:" + marke;
    }
}
```

```
public class Sattelzug extends LKW {
    public Sattelzug(String marke) {
        super(marke);
    }
    public String getDaten() {
        return "Sattelzug:" + marke;
    }
    public void ankuppeln() {
        System.out.print("angekuppelt");
    }
}
```

```
public class Auto extends Fahrzeug {
    public Auto(String marke) {
        super(marke);
    }
    public String getDaten() {
        return "Auto:" + marke;
    }
}
```

a) Ergänze das Klassendiagramm mit Beziehungen, Attributen und Methoden so, dass es mit dem Programmcode übereinstimmt. Achte hierbei insb. auf korrektes Format!

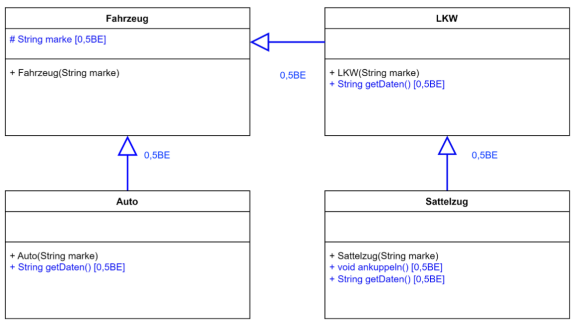


Abbildung falsch: [1BE]pro Vererbungspfeil

Figure 45: PT: Inheritance 2

b) Trage in den Kommentaren jeweils den Rückgabewert des Methodenaufrufs in der gleichen Zeile ein.
- Gibt die Methode keinen Wert zurück, notiere als Rückgabewert *void*.
- Ergibt der Methodenaufruf einen Fehler oder ist nicht möglich, notiere als Rückgabewert *Fehler:* und gib dahinter eine kurze Begründung an, wieso hier ein Fehler auftritt. Verwende dabei Fachbegriffe und schreibe ggf. auf den Zeilen darunter weiter.

```
Fahrzeug fahrzeug1 = new Auto("VW");
Auto auto1 = new Auto("BMW");
LKW lkw1 = new LKW("Scania");
LKW lkw2 = new Sattelzug("MAN");
Sattelzug sattelzug1 = new Sattelzug("DAF");

lkw1.getDaten(); // LKW:Scania [1BE]

lkw2.getDaten(); // Sattelzug:MAN [1BE]

sattelzug1.ankuppeln(); // void [0,5BE]

auto1.getDaten(); // Auto:BMW [1BE]

fahrzeug1.getDaten(); // Fehler [0,5BE]: Methode in Klasse, die statischer Typ (=Typ der Variable) ist, nicht verfügbar. [0,5BE](bei Auto:VW [0,5BE])
```