

Programmierung 1

im Wintersemester 2025/26

Praktikumsaufgabe 5

Ein Programm schreiben (Kap. 5)

Review: 2. Dezember 2025 (DFIW) und 3. Dezember 2025 (PI)

Diese Praktikumsaufgabe baut auf den Inhalten und Lernzielen der vorhergehenden Praktikumsaufgaben 1 bis 4 auf und führt das dort erworbene Wissen in einer umfassenden Aufgabe zusammen, in der Sie den Prozess der Erstellung eines Java-Programms durchlaufen. Dabei wenden Sie die zuvor isoliert betrachteten Konzepte und Techniken in einem größeren Kontext an und lernen, wie diese zusammenwirken, um ein vollständiges Programm zu erstellen. Das praktische Durchlaufen dieses Prozesses zur Entwicklung qualitativ hochwertiger Programme (die einerseits robust, gut strukturiert und leicht erweiterbar sind und andererseits schnell und früh einen hohen Nutzen liefern) wird durch die anschließende Vertiefung verschiedener Schleifen, Operatoren und Kontrollstrukturen ergänzt.

In **Aufgabe 1** implementieren Sie einen Mini-Taschenrechner und durchlaufen dabei den gesamten Programmierprozess einer professionellen Softwareentwicklung.

In **Aufgabe 2** verwenden Sie verschiedene Schleifentypen, um ein Array zu durchlaufen. Dadurch können Sie die Vor- und Nachteile der verschiedenen Typen von Schleifen analysieren und herausfinden, in welchen Szenarien ein bestimmter Typ besonders geeignet ist.

In **Aufgabe 3** verwenden Sie die Operatoren Inkrement (++) und Dekrement (--) in Schleifen, um Zähler zu erhöhen bzw. zu verringern.

In **Aufgabe 4** beenden Sie eine Schleife vorzeitig mit **break**, wenn eine bestimmte Bedingung erfüllt ist.

Heads up!

Für diese und die folgenden Praktikumsaufgaben gilt:

Konventionen und Praktiken: Halten Sie sich an die Konventionen der professionellen Softwareentwicklung und folgen Sie den bewährten Verfahren. Wählen Sie aussagekräftige Namen. Strukturieren Sie Ihren Code durch Einrückungen, damit er übersichtlich und leicht nachvollziehbar ist. Verwenden Sie Pseudo-UML, um Ihre Klassen zu entwerfen.

Testgetriebene Entwicklung: Arbeiten Sie iterativ! Schreiben Sie zuerst einen Test, der eine Funktionalität überprüft, und entwickeln Sie nur so viel Code, wie nötig ist, um den Test zu bestehen. Optimieren Sie anschließend Ihren Code und führen Sie alle Tests erneut aus, um sicherzustellen, dass Ihr Programm weiterhin korrekt funktioniert. Diese Vorgehensweise ermöglicht es Ihnen, sicher und schrittweise zu entwickeln, während die Tests als Sicherheitsnetz dienen.

Trennung von Produktions- und Testcode: Schreiben Sie den Code, der den "richtigen" *Produktionscode* ausführt und testet, in einer separaten Klasse mit der Namensendung `TestDrive`. Dieser Code wird *Testcode* genannt.

Kontinuierliches Lernen: Nutzen Sie diese Aufgaben, um das Ergebnis Ihrer Reflexion des Feedbacks aus den vorherigen Aufgaben umzusetzen. So lernen Sie kontinuierlich und können Ihre Fähigkeiten als professionelle Softwareentwickler:in ausbauen.

Und nach wie vor gilt: Lösen Sie die Aufgaben nicht allein, sondern mindestens zu zweit! Zum Beispiel mit Ihrer Coding-Partner:in. Denn in der Gruppe lernt es sich besser.

Wir als Ihre Lernbegleiter:innen unterstützen Sie gerne bei der Lösung der Aufgaben.

Hinweise zu testgetriebener Entwicklung (kurz: TDD¹):

Entwickeln Sie in Entwicklungszyklen, um schrittweise und systematisch zu arbeiten!

1. **Test first:** Schreiben Sie zuerst den Testcode, bevor Sie den “richtigen” Produktionscode schreiben.
 - Schreiben Sie in jedem Zyklus *nur einen Test*, der eine bestimmte, kleine und klar definierte neue Funktionalität überprüft.
 - Dieser neue Test sollte zunächst fehlschlagen, da die Funktionalität noch nicht implementiert ist.
2. **Implementieren:** Schreiben Sie nur so viel Code, wie unbedingt nötig ist, um den neuen Test zu bestehen.
 - Konzentrieren Sie sich darauf, *diesen einen Test* zu bestehen.
 - Halten Sie sich an kleine, klar definierte Änderungen.
 - Führen Sie alle Tests als Sicherheitsnetz immer wieder aus, um sicherzustellen, dass die bestehende und zuvor erfolgreich getestete Funktionalität nicht beeinträchtigt wird.
3. **Refaktorisieren:** Optimieren Sie den Code, sobald der Test erfolgreich war.
 - Bereinigen Sie den Code, ohne die eigentliche Funktionalität zu verändern.
 - Verwenden Sie die Tests erneut als Sicherheitsnetz, um sicherzustellen, dass der geänderte Code weiterhin korrekt funktioniert.

Warum TDD? Tests sind Ihr Sicherheitsnetz. Sie helfen, Fehler frühzeitig zu erkennen und machen Änderungen sicherer. Arbeiten Sie in kleinen Schritten, um den Überblick zu behalten und Ihren Code kontinuierlich zu verbessern.

¹Test-Driven Development, erfunden von Kent Beck und wunderbar zusammengefasst in seinem Artikel [Canon TDD](#).

1 Programmierprozess durchleben

Lernziele

- Sie erstellen einen abstrakten Programmentwurf in Form eines Flussdiagramms.
- Sie entwickeln Vorcode, der die Struktur und Logik des Programms beschreibt.
- Sie schreiben Testcode, bevor Sie die eigentliche Programmlogik implementieren.
- Sie wenden testgetriebene Entwicklung an, um in kleinen, abgesicherten Schritten ein lauffähiges Java-Programm zu entwickeln.
- Sie implementieren ein lauffähiges Java-Programm, das Benutzereingaben verarbeitet und Ergebnisse ausgibt.

In dieser Aufgabe entwickeln Sie einen Mini-Taschenrechner, der zwei Zahlen addieren, subtrahieren, multiplizieren oder dividieren kann. Der Taschenrechner soll Benutzereingaben verarbeiten und Ergebnisse ausgeben. Das Programm soll auch auf ungültige Eingaben reagieren, z. B. wenn die Benutzer:in einen ungültigen Operator eingibt oder versucht, durch Null zu teilen.

Heads up! Mit der testgetriebenen Entwicklung durchlaufen Sie einen iterativen Programmierprozess, der Ihnen hilft, Ihre Implementierung schrittweise zu verbessern und Fehler frühzeitig zu erkennen. Damit machen Sie einen großen Schritt in Richtung professioneller Softwareentwicklung. Herzlichen Glückwunsch!

Sie können sich an den folgenden Beispielausgaben orientieren.

Beispiel für eine Addition:

```
Geben Sie die erste Zahl ein: 10
Geben Sie die zweite Zahl ein: 5
Geben Sie den Operator ein (+, -, *, /): +
Ergebnis: 15.0
```

Beispiel für eine fehlerhafte Eingabe der Rechenoperation:

```
Geben Sie die erste Zahl ein: -3
Geben Sie die zweite Zahl ein: 7
Geben Sie den Operator ein (+, -, *, /): ^
Ungültiger Operator. Wählen Sie einen der folgenden Operatoren: +, -, *, /
```

Beispiel einer abgefangenen Division durch Null:

```
Geben Sie die erste Zahl ein: 4
Geben Sie die zweite Zahl ein: 0
Geben Sie den Operator ein (+, -, *, /): /
Eine Division durch Null ist nicht möglich. Geben Sie einen gültigen
→ Teiler ein.
```

1.1 Abstrakter Entwurf

Erstellen Sie einen abstrakten Entwurf für einen Mini-Taschenrechner!

Erstellen Sie dazu ein **Flussdiagramm**, das die Logik des Programms beschreibt. Das Programm soll:

- Zwei Zahlen von der Benutzer:in einlesen.
- Einen arithmetischen Operator (+, -, *, /) abfragen.
- Die entsprechende Berechnung durchführen und das Ergebnis ausgeben.
- Eine für die Benutzer:in hilfreiche Fehlermeldung ausgeben, wenn der Operator ungültig ist oder eine Division durch Null vorliegt.
- In beiden Fällen, d. h. bei gültigen und ungültigen Eingaben, kann das Programm beendet werden.²

Berücksichtigen Sie in Ihrem Entwurf verschiedene Pfade, z. B. für gültige und ungültige Eingaben.

²Wenn Sie möchten, können Sie das Programm so erweitern, dass es nach der Ausgabe erneut nach einer Berechnung fragt. Dazu kann eine while-Schleife und ein spezieller Operator wie q (Quit) zum Beenden des Programms verwendet werden.

1.2 Vorcode

Schreiben Sie den Vorcode für den Mini-Taschenrechner!

Übersetzen Sie dazu die Struktur und die Hauptlogik des Flussdiagramms Ihres Programms in Pseudocode. Konzentrieren Sie sich auf das *Was* und nicht auf das *Wie* (z. B. “Prüfe, ob der Operator gültig ist” statt “Benutze einen if-else-Block”).

1.3 Initiale Testliste

Erstellen Sie eine erste Liste der Testszenarien, die Sie mit Ihrem Mini-Taschenrechner abdecken möchten!

Ein solches Testszenario beschreibt, was getestet werden soll, ohne auf detaillierte Schritte einzugehen. Jedes Szenario sollte einen kleinen, klar definierten Aspekt des Programms abdecken. Beispiele für Testszenarien:

1. Addition zweier positiver Zahlen: $2 + 3 \rightarrow 5$
2. Division zweier positiver Zahlen: $10 / 2 \rightarrow 5$
3. Ungültiger Operator: $2 ? 3 \rightarrow \text{Fehler}$
4. Division durch Null: $10 / 0 \rightarrow \text{Fehler}$

Erweitern Sie die Liste um weitere Operatoren und Sonderfälle, die Ihnen einfallen (z. B. negative Zahlen, große Zahlen).³ Ihre Testszenarien sollten mindestens alle Pfade Ihres Flussdiagramms abdecken.

Wichtig: Die Liste muss zu diesem Zeitpunkt nicht vollständig sein und kann im Laufe der folgenden Implementierung erweitert werden.

³Wie $-5 - (-3) \rightarrow -2$ und $1000000 * 1000000 \rightarrow -727379968$

1.4 Testgetriebene Implementierung

Implementieren Sie den Mini-Taschenrechner in Java und verwenden Sie testgetriebene Entwicklung!

Warum TDD? Durch den testgetriebenen Ansatz konzentrieren Sie sich auf die aktuellen Anforderungen, minimieren Fehler und erhöhen die Codequalität – eine wichtige Fähigkeit für professionelle Softwareentwicklung.

1.4.1 Schritt-für-Schritt-Anleitung

Befolgen Sie dazu die folgenden Schritte in sich wiederholenden Zyklen:

1. Wählen Sie *genau ein* Testszenario aus der Liste aus und implementieren Sie einen konkreten, ausführbaren Test in Java, der das Testszenario abbildet.

Zum Beispiel der Test “Addition zweier positiver Zahlen”, der überprüft, ob die Addition von 2 und 3 korrekt 5 ergibt.

Wichtig: Da die Addition zu Beginn noch nicht existiert, wird dieser Test fehlschlagen. Das ist in Ordnung und so gewollt!

Tipp: Damit der Test kompiliert wird, können Sie in Ihrer Taschenrechner-Klasse die Methode `add` zunächst als **leere Methode** implementieren. D. h. die Methode gibt einfach `0` zurück.

2. Ändern und erweitern Sie den Code Ihrer Taschenrechner-Klasse, *um den in Schritt 1 implementierten Test (und alle vorherigen Tests, die als Sicherheitsnetz dienen) erfolgreich ausführen zu können.*

Wenn Sie dabei neue Testszenarien entdecken, fügen Sie diese der Liste als neue Szenarien hinzu.

Auf diese Weise konzentrieren Sie sich ausschließlich auf die erfolgreiche Erfüllung des in Schritt 1 implementierten Tests, in unserem Beispiel also auf die korrekte Addition

von 2 und 3.

3. Optional können Sie Ihre Implementierung in der Taschenrechner-Klasse aufräumen und verbessern. Ihre Tests dienen wieder als Sicherheitsnetz.

Beispielsweise könnten Sie die Parameter der Methode `add` umbenennen, um den Code lesbarer zu machen.

4. Kehren Sie zu Schritt 1 zurück, bis die Liste der Testszenarien leer ist.

Durch die schrittweise Umsetzung eines Testszenarios nach dem anderen wird der Entwicklungsprozess sicherer und übersichtlicher. Sie können sich auf ein kleines, klar definiertes Ziel konzentrieren und Fehler frühzeitig erkennen.

1.4.2 Implementierung des Mini-Taschenrechners

Implementieren Sie dazu zwei Klassen: Die erste enthält den Mini-Taschenrechner, die zweite die Tests. Sie können die Klasse für den Taschenrechner beispielsweise `MiniCalculator` nennen und die Testklasse entsprechend `MiniCalculatorTestDrive`. Die Testklasse enthält die `main`-Methode, in der Sie die Testszenarien aus Ihrer Liste implementieren.

Die `main`-Methode wird dabei *nach und nach* um die *einzelnen Testszenarien* aus Ihrer Liste erweitert (Schritt 1). Die Taschenrechner-Klasse wird *nach und nach* um die einzelnen *Methoden* und *“Mini-Funktionalitäten”* erweitert, die Sie für die erfolgreiche Durchführung der einzelnen Tests benötigen (Schritt 2).

Heads up! Dies kann bedeuten, zunächst eine leere Methode zu implementieren, diese dann mit einer Rechenoperation zu füllen und schließlich die Methode so zu erweitern, dass sie die gewünschte Funktionalität wie z. B. eine Fehlerbehandlung enthält. Das Ganze in Minischritten über mehrere Zyklen (*Iterationen*).

Wiederholen Sie die vier oben genannten Schritte Zyklus für Zyklus (*iterativ*), bis Ihre Liste leer ist. So entwickeln Sie Ihren Mini-Rechner in kleinen, sicheren Schritten – eben **testgetrieben**.

1.4.3 Tipps zur Reihenfolge der Testszenarien und zum Schnitt der Iterationen

Eine geeignete Reihenfolge für die testgetriebene Abarbeitung der Testszenarien ist:⁴

Iteration 1 – Einfacher Aufruf einer Methode: Addition Beginnen Sie mit der grundlegenden Funktionalität Ihres Taschenrechners: der Addition zweier Zahlen. In dieser Iteration implementieren Sie die Methode `add` und testen sie mit vorgegebenen Zahlen. Der Schwerpunkt liegt darauf, die Grundlage für weitere Funktionen zu legen und den TDD-Ansatz kennenzulernen, bei dem Sie zuerst den Test und dann die Implementierung schreiben.

- Methode `add` zur Addition zweier vorgegebener Zahlen (vgl. oben).

Iteration 2 – Eingabe von Zahlen und Addition Erweitern Sie Ihren Taschenrechner um Benutzereingaben. Ziel ist es, zwei Zahlen mit dem Scanner einzulesen, die Methode `add` aufzurufen und das Ergebnis auszugeben. Diese Iteration zeigt, wie Benutzerinteraktionen in die Programmlogik integriert werden können.

- Zwei Zahlen mit Scanner einlesen.
- Aufruf der Methode `add` mit den eingelesenen Zahlen.
- Ausgabe des Ergebnisses der Addition der beiden eingelesenen Zahlen.

Iteration 3 – Eingabe eines Operators: Addition Erweitern Sie Ihren Taschenrechner um die Fähigkeit, in Abhängigkeit von einem eingegebenen Operator zu rechnen. In dieser Iteration lesen Sie das Zeichen `+` ein, rufen die Methode `add` auf und geben das Ergebnis aus. Der Schwerpunkt liegt auf der Verknüpfung von Operatoren mit den entsprechenden Rechenmethoden.

- Das Zeichen `+` mit Scanner einlesen.
- Aufruf der Methode `add` mit den eingelesenen Zahlen, wenn `+` eingegeben wurde.⁵
- Ausgabe des Ergebnisses der Addition der beiden eingelesenen Zahlen.

⁴Vergessen Sie nicht, spätestens am Ende jeder Iteration alle bis dahin implementierten Testszenarien erneut auszuführen, um sicherzustellen, dass Sie nicht versehentlich bestehende Funktionalität verändert haben.

⁵Bei allen anderen Zeichen wird das Programm einfach beendet.

Iteration 4 – Methode zur Division Implementieren Sie die Methode `divide`, um zwei Zahlen zu teilen. Diese Iteration konzentriert sich auf die Entwicklung einer weiteren arithmetischen Grundfunktion. Ziel ist es, die Methode unabhängig von Benutzereingaben zu testen und zu validieren.

- Methode `divide` zur Division zweier vorgegebener Zahlen (vgl. `add`).

Iteration 5 – Division: Gültige Eingaben Integrieren Sie die Methode `divide` in die Benutzerinteraktion. Ziel ist es, das Zeichen `/` einzulesen, die Methode `divide` mit den eingelesenen Zahlen aufzurufen und das Ergebnis auszugeben. Diese Iteration zeigt, wie mehrere arithmetische Operationen schrittweise in das Programm integriert werden.

- Das Zeichen `/` mit `Scanner` einlesen.
- Aufruf der Methode `divide` mit den eingelesenen Zahlen, wenn `/` eingegeben wurde.⁶
- Ausgabe des Ergebnisses der Division der beiden eingelesenen Zahlen.

Iteration 6 – Division: Fehlerbehandlung Machen Sie Ihren Taschenrechner robuster, indem Sie den Fehlerfall “Division durch Null” abfangen. Ziel ist es, eine sinnvolle Fehlermeldung auszugeben, ohne das Programm zum Absturz zu bringen. Diese Iteration veranschaulicht die Bedeutung der Fehlerbehandlung in der Softwareentwicklung.

- Fehlerfallbehandlung “Division durch Null”.

Und so weiter...

Die grundsätzliche Idee ist, dass Sie schrittweise einen lauffähigen Mini-Taschenrechner entwickeln, der möglichst schnell von einer Benutzer:in verwendet werden kann.

Ein Taschenrechner, der “nur” die Addition beherrscht (bereits nach Iteration 2), ist aus Benutzersicht wertvoller als ein Taschenrechner, der zwar alle Operationen als interne Methoden implementiert hat, aber noch keine Benutzereingaben verarbeiten kann. Die nächste große Verbesserung ist, dass die Benutzer:in eine Division durchführen kann (Iteration 5), auch wenn der Taschenrechner bei der Eingabe von 0 abstürzt. Als nächstes wird die Fehlerbehandlung verbessert, um die Anwendung benutzerfreundlicher zu machen (Iteration 6).

⁶Wird `+` eingegeben, wird die Addition durchgeführt.

Ihre Tests aus früheren Iterationen – sei es durch automatisierte Aufrufe der Methoden `add` und `divide` oder durch manuelle Benutzereingaben Ihrerseits⁷ – stellen immer sicher, dass bestehende Funktionalität nicht unbeabsichtigt verändert wird. Ein beruhigendes Gefühl für Sie als professionelle Softwareentwickler:in und später auch für Ihre Auftraggeber:innen.

Hinweis: Es steht Ihnen frei, nach diesen 6 Iterationen aufzuhören oder weitere Testszenarien zu implementieren, wenn Sie dies für Ihren Lernerfolg als sinnvoll erachten.

Nutzen Sie die unten folgende Reflexion im Coding-Team, um Ihre Erfahrungen mit TDD und die Herausforderungen bei der Umsetzung zu diskutieren.

1.4.4 Ausblick

In dieser Aufgabe lernen Sie die *Denkweise* und den *Rhythmus* von TDD kennen: Zuerst einen Test schreiben, der fehlschlägt, und dann erst Code schreiben, damit der Test besteht.

Ihre `TestDrive`-Klasse ist ein wichtiger erster Schritt, um dieses Prinzip zu verinnerlichen. In der professionellen Softwareentwicklung wird dieser Prozess mit einem Test-Framework (wie `JUnit`) *automatisiert*. Diese Frameworks prüfen selbstständig, ob der Code das korrekte Ergebnis liefert, sodass Sie die Ausgabe nicht mehr manuell auf der Konsole vergleichen müssen.

1.5 Reflexion im Coding-Team

Diskutieren Sie im Coding-Team, um Ihr Verständnis zu vertiefen!

1.5.1 Abstrakter Entwurf

Welche Herausforderungen gab es bei der Umsetzung des Flussdiagramms in Vorcode und Java-Code?

⁷Sie werden schnell die Vorteile der Automatisierung in der Softwareentwicklung schätzen lernen, wie z. B. die der `Testautomatisierung`.

- Haben Sie die im Flussdiagramm entworfene Programmlogik wie geplant umgesetzt oder mussten Sie Änderungen vornehmen? Warum?
- Wie hilfreich war das Flussdiagramm für das Verständnis der Anforderungen und für die Strukturierung des Vorcodes und des Java-Codes?
- Glauben Sie, dass Flussdiagramme auch bei größeren Projekten oder in der Teamarbeit hilfreich wären? Warum?

1.5.2 Testgetriebene Entwicklung

Wie hat das Schreiben von Testcode vor der Implementierung Ihre Vorgehensweise beeinflusst?

- Hat das Schreiben von Tests Ihre Implementierung vereinfacht oder verkompliziert? Warum?
- Welche Vorteile oder Herausforderungen sind Ihnen beim Schreiben der Tests aufgefallen?
- Haben Ihnen die Tests geholfen, Fehler frühzeitig zu erkennen oder die Struktur Ihres Codes zu verbessern? Wenn ja, wie?

1.5.3 Schnitt und Reihenfolge der Iterationen

Wie sinnvoll war die gewählte Reihenfolge der Iterationen?

- Welche Iteration(en) hätten Sie früher oder später durchgeführt?
- Welche Reihenfolge ermöglicht es, möglichst schnell ein funktionsfähiges Programm bereitzustellen, das für die Benutzer:innen direkt nutzbar ist?
- Welche Reihenfolge ermöglicht es, möglichst schnell Erkenntnisse über die Gesamtstruktur des Programms zu gewinnen?

- Haben Sie das Gefühl, dass die gewählte Reihenfolge eher auf schnelle Ergebnisse oder eher auf langfristige Qualität und Struktur ausgerichtet war? Warum?
- Gab es Iterationen, die Ihrer Meinung nach einen besonders hohen “Lernwert” hatten, unabhängig davon, ob sie direkt zum Nutzen des Programms beigetragen haben?

1.5.4 Iteratives Vorgehen und schrittweise Entwicklung

Was sind die Vor- und Nachteile eines schrittweisen und iterativen Vorgehens?

- Wann war die Iteration klar und zielgerichtet und wann nicht?
- Würden Sie diesen Ansatz in einem zukünftigen Projekt wieder anwenden? In welchen Situationen wäre es besonders hilfreich?
- Gibt es Situationen, in denen ein weniger iteratives Vorgehen sinnvoller wäre?

2 Verschiedene Schleifen vergleichen

Lernziele

- Sie verwenden **for**-Schleifen, um Arrays zu durchlaufen.
- Sie verwenden **while**-Schleifen, um Arrays zu durchlaufen.
- Sie verwenden die erweiterte **for**-Schleife, um Arrays zu durchlaufen.
- Sie vergleichen die Vor- und Nachteile der verschiedenen Schleifentypen und analysieren, in welchen Szenarien ein bestimmter Schleifentyp besonders geeignet ist.

In dieser Aufgabe verwenden Sie verschiedene Schleifentypen, um ein Array zu durchlaufen. Sie verwenden dabei die klassische⁸ **for**-Schleife, die **while**-Schleife und die erweiterte **for**-Schleife, um die Elemente des Arrays auf der Konsole auszugeben.

2.1 Iteration mit **for**-Schleifen

Implementieren Sie eine klassische **for**-Schleife, die ein Array durchläuft und dessen Elemente auf der Konsole ausgibt!

Erstellen Sie dazu eine Klasse `ForLoopDemo` mit der Methode `main`. Implementieren Sie darin eine **for**-Schleife, die die Zahlwörter `"eins"`, `"zwei"`, `"drei"`, `"vier"` und `"fünf"` durchläuft und jeden Wert auf der Konsole ausgibt.

Ändern Sie dann die Schleife so ab, dass die Ausgabe rückwärts erfolgt (von `"fünf"` nach `"eins"`).

Sie können sich an den folgenden Beispielausgaben orientieren.

```
eins
zwei
drei
vier
fünf
```

⁸In Abgrenzung zur *erweiterten for*-Schleife.

fünf
vier
drei
zwei
eins

2.2 Iteration mit **while**-Schleifen

Implementieren Sie die gleiche Funktionalität wie in der vorherigen Aufgabe, aber verwenden Sie diesmal eine **while**-Schleife!

Erstellen Sie dazu eine Klasse `WhileLoopDemo` mit der Methode `main`. Implementieren Sie darin eine **while**-Schleife, die die Elemente des Arrays durchläuft und auf der Konsole ausgibt. Ändern Sie die Schleife so ab, dass die Werte in umgekehrter Reihenfolge ausgegeben werden.

2.3 Iteration mit der erweiterten **for**-Schleife

Implementieren Sie die gleiche Funktionalität noch einmal, aber verwenden Sie diesmal eine erweiterte⁹ **for**-Schleife!

Erstellen Sie dazu eine Klasse `EnhancedForLoopDemo` mit einer Methode `main`. Implementieren Sie eine erweiterte **for**-Schleife, die das Array durchläuft und dessen Werte auf der Konsole ausgibt.

Hinweis: Beachten Sie, dass die erweiterte **for**-Schleife nur in eine Richtung iterieren kann.

2.4 Reflexion im Coding-Team

Diskutieren Sie im Coding-Team, um Ihr Verständnis zu vertiefen!

⁹Auch bekannt als *enhanced for-loop*.

2.4.1 Klassische `for`-Schleife

Wann ist die klassische `for`-Schleife vorteilhafter als die `while`-Schleife oder die erweiterte `for`-Schleife?

Diskutieren Sie anhand der folgenden Szenarien:

- Sie möchten ein Array rückwärts durchlaufen und nur die geraden Werte ausgeben (z. B. 6, 4, 2, 0).
- Zusätzlich zu den Werten des Arrays sollen auch die Indizes der Elemente ausgegeben werden (z. B. "Index: 0, Wert: eins").

Überlegen Sie, warum die klassische `for`-Schleife hier besonders nützlich sein könnte.

2.4.2 `while`-Schleife

Wann ist die `while`-Schleife besonders nützlich?

Diskutieren Sie die folgenden Situationen:

- Ein Sensor gibt Daten in Echtzeit aus und Sie möchten diese so lange verarbeiten, bis ein bestimmter Grenzwert überschritten wird.
- Sie zählen von 1 bis 10, möchten aber die Schleife beenden, wenn die Summe der Werte 20 erreicht.

Überlegen Sie, wie die Flexibilität der `while`-Schleife Ihnen in solchen Fällen helfen kann.

2.4.3 Erweiterte `for`-Schleife

Welche Vorteile bietet die erweiterte `for`-Schleife?

Überlegen Sie, warum diese Schleife für den folgenden Fall ideal ist:

- Sie möchten ein Array von Objekten durchlaufen und nur auf bestimmte Eigenschaften zugreifen, z. B. auf das name-Attribut einer Liste von Person-Objekten.

Diskutieren Sie, wie die reduzierte Syntax hier die Lesbarkeit verbessert und den Fokus auf die Logik lenkt.

2.4.4 Persönliche Präferenzen

Welche Art von Schleife bevorzugen Sie persönlich und warum?

Testen Sie den Code mit den oben genannten Szenarien und überlegen Sie, welche Schleife für Sie in welchem Fall intuitiver oder einfacher war.

3 Operatoren in Schleifen nutzen

Lernziele

- Sie verwenden die Operatoren Inkrement und Dekrement, um Variablenwerte zu ändern.
- Sie implementieren Schleifen, die durch diese beiden Operatoren gesteuert werden.
- Sie analysieren die Unterschiede zwischen den Operatoren und alternativen Ansätzen.

In dieser Aufgabe verwenden Sie die Inkrement- und Dekrement-Operatoren, um Schleifen zu implementieren, die aufwärts und abwärts zählen

3.1 Zähler mit Inkrement

Implementieren Sie einen Zähler mit dem Inkrementoperator!

Erstellen Sie dazu eine Klasse `IncrementDemo` mit einer Methode `main` und implementieren Sie darin eine `for`-Schleife und eine `while`-Schleife, die jeweils von 1 bis 10 zählt und jeden Wert auf der Konsole ausgibt. Verwenden Sie den Inkrement-Operator `++`.

Sie können sich an der folgenden Beispielausgabe orientieren.

1
2
3
4
5
6
7
8
9
10

Ändern Sie Ihre Implementierung so, dass jede Schleife die folgende Ausgabe erzeugt:

-3

```
-2  
-1  
0  
1  
2  
3
```

3.2 Countdown mit Dekrement

Implementieren Sie einen Countdown mit dem Dekrementoperator!

Erstellen Sie dazu eine Klasse `DecrementDemo` mit einer Methode `main` und implementieren Sie darin eine `for`-Schleife und eine `while`-Schleife, die jeweils von 10 bis 0 zählt und jeden Wert auf der Konsole ausgibt. Verwenden Sie den Dekrement-Operator `--`.

Sie können sich an der folgenden Beispielausgabe orientieren.

```
10  
9  
8  
7  
6  
5  
4  
3  
2  
1  
0
```

Ändern Sie Ihre Implementierung so, dass jede Schleife die folgende Ausgabe erzeugt:

```
0  
-1  
-2  
-3
```

-4
-5

3.3 Reflexion im Coding-Team

Diskutieren Sie im Coding-Team, um Ihr Verständnis zu vertiefen!

Heads up! Es kann sehr hilfreich sein, die folgenden Fragen nicht nur theoretisch zu diskutieren, sondern auch Code zu schreiben und dessen Verhalten zu beobachten. Dies gilt insbesondere, aber *nicht ausschließlich*, für die beiden Fragestellungen in den Aufgaben 3.3.3 und 3.3.4.

3.3.1 Nutzen von Inkrement- / Dekrement-Operatoren

Wann sind `x++` und `x--` nützlicher als alternative Ansätze wie z. B. `x = x + 1` oder `x = x - 1`?

Diskutieren und überlegen Sie anhand der folgenden Szenarien, warum die diese Operatoren in diesen Fällen bevorzugt werden und ob es Situationen gibt, in denen die Alternativen besser geeignet wären.

- Sie implementieren eine `for`-Schleife, die eine festgelegte Anzahl von Iterationen durchläuft.
- Sie verwenden eine `while`-Schleife, um einen Zähler zu steuern, der dynamisch angepasst wird.

3.3.2 Lesbarkeit und Verständlichkeit

Wie beeinflussen `x++` und `x--` die Lesbarkeit des Codes?

Überlegen Sie, wie sich die Lesbarkeit von `for (int i = 0; i < 10; i++)` im Vergleich zu `for (int i = 0; i < 10; i = i + 1)` unterscheidet.

- Diskutieren Sie, ob und wie die Verwendung von `x++` oder `x--` in Schleifen die Lesbarkeit erschweren kann, z. B. wie in `for (int i = 0; i++ < 10;)`.
- Überlegen Sie, welche Konventionen im Team verwendet werden sollten, um Missverständnisse zu vermeiden.

3.3.3 Fehlerquellen und Vermeidungsstrategien

Was sind mögliche Fehlerquellen bei der Verwendung solcher Operatoren?

Diskutieren Sie die folgenden Situationen und überlegen Sie, wie solche Fehler vermieden werden können, z. B. durch Kommentare oder durch Umstrukturierung des Codes.

- Was passiert, wenn Sie `x++` statt `++x10` in einem komplexen Ausdruck verwenden, z. B. in einer Bedingung wie `if (array[i++] > 10)`?
- Wie kann die Verwendung von `x--` in einer Schleifenbedingung wie beispielsweise `while (i-- > 0)` zu unvorhergesehenem Verhalten führen?

3.3.4 Alternative Ansätze

In welchen Situationen wären diese Operatoren nicht geeignet?

Überlegen Sie, welche Alternativen Sie in den folgenden Szenarien verwenden würden, und diskutieren Sie, warum explizite Zuweisungen oder andere Ansätze in diesen Fällen klarer oder flexibler sein könnten.

- Ein Zähler soll in nichtlinearen Schritten inkrementiert oder dekrementiert werden (z. B. in sich ändernden Schritten).

¹⁰Beachten Sie den Unterschied zwischen dem Postinkrement- und dem Präinkrement-Operator.

- Eine Schleife soll durch eine komplexe Bedingung gesteuert werden, die mehr als eine einfache Anpassung des Zählers erfordert.

4 Schleifen frühzeitig beenden

Lernziele

- Sie verwenden eine Bedingung, um den Ablauf einer Schleife zu steuern.
- Sie implementieren eine Schleife mit der Anweisung **break**, um die Schleife vorzeitig zu beenden.
- Sie analysieren die Vor- und Nachteile der Verwendung von **break** in verschiedenen Kontexten.

In dieser Aufgabe beenden Sie eine Schleife vorzeitig, wenn eine bestimmte Bedingung erfüllt ist. Dazu verwenden Sie die **break**-Anweisung.

4.1 Schleife mit **break** verlassen

Schreiben Sie eine Methode, die durch ein Array iteriert und stoppt, wenn eine bestimmte Bedingung erfüllt ist!

Erstellen Sie dazu eine Klasse `BreakDemo` mit der Methode `main`. Schreiben Sie darin eine Schleife, die ein Array von Zahlen durchsucht und anhält, wenn eine bestimmte Bedingung erfüllt ist (z. B. wenn ein Wert gefunden wird, der größer als 50 ist). Verwenden Sie dazu die **break**-Anweisung.

Testen Sie die Schleife mit den folgenden Beispieldaten für die Bedingung “Wert > 50”:

1. Ein Array mit den Werten 10, 20, 30, 40, 50, 60 und 70.
2. Ein Array mit den Werten 5, 15, 25, 35, und 45.
3. Ein Array mit den Werten 60, 70, 80 und 90.
4. Ein leeres Array.

Geben Sie für jedes Array die durchsuchten Werte und den gefundenen Wert (falls vorhanden) auf der Konsole aus. Wenn kein Wert gefunden wird, der die Bedingung erfüllt, geben Sie eine Meldung aus.

Sie können sich an den folgenden Beispielausgaben orientieren.

Array : [10, 20, 30, 40, 50, 60, 70]

Kriterium: 50

Überprüfte Werte:

10

20

30

40

50

60

Gefundener Wert: 60

Array : [5, 15, 25, 35, 45]

Kriterium: 50

Überprüfte Werte:

5

15

25

35

45

Kein Wert erfüllt die Bedingung.

Array : [60, 70, 80, 90]

Kriterium: 50

Überprüfte Werte:

60

Gefundener Wert: 60

Array : []

Kriterium: 50

Array ist leer. Kein Wert überprüfbar.

4.2 Reflexion im Coding-Team

Diskutieren Sie im Coding-Team, um Ihr Verständnis zu vertiefen!

4.2.1 Vorteile von **break**

Welche Vorteile bietet die Verwendung von **break in dieser Aufgabe?**

Diskutieren Sie, warum **break** in einem Szenario wie der Suche nach einem bestimmten Wert in einem Array nützlich ist.

Überlegen Sie, wie die Schleife ohne **break** gestaltet werden müsste. Ist diese Alternative immer effizienter oder einfacher?

4.2.2 Bedingung in der Schleifenkonstruktion

In welchen Situationen wäre es sinnvoller, die Bedingung direkt in der Schleifenkonstruktion zu verarbeiten, anstatt **break zu verwenden?**

Diskutieren Sie, ob und warum es besser sein könnte, die Bedingung direkt in der Schleifenkonstruktion zu verarbeiten, so zum Beispiel mit **if** (`array[i] < 50`) statt mit **break**.

4.2.3 Missverständnisse durch **break**

Wie wirkt sich die Verwendung von **break auf die Wartbarkeit und Lesbarkeit des Codes aus?**

Überlegen Sie, wie eine **break**-Anweisung den Lesefluss in einer Schleife beeinflusst, insbesondere in einer verschachtelten Schleife.

Diskutieren Sie ein Szenario, in dem eine unerwartete **break**-Anweisung zu Missverständnissen führen könnte, z. B. wenn die Bedingung komplex ist oder Kommentare fehlen.

4.2.4 Alternativen zu `break`

Welche Alternativen zur Verwendung von `break` gibt es und wann könnten diese vorteilhafter sein?

Diskutieren Sie Alternativen wie:

- Verwendung einer zusätzlichen booleschen Variablen (`found`), um den Abbruch der Schleife zu steuern.
- Verwendung von `return` in einer Methode, um die Schleife zu verlassen. Überlegen Sie, in welchen Kontexten diese Alternativen klarer oder flexibler sein könnten, z. B. bei der Verarbeitung von Daten in mehreren Schleifen oder Methoden.
- Probieren Sie es aus: Nehmen Sie den Code Ihrer Schleife und erstellen Sie daraus eine eigene, neue Methode.¹¹ Wie können Sie die Schleife und die Methode nun am elegantesten mit `return` verlassen, sobald der Wert gefunden wurde? Vergleichen Sie die Lesbarkeit dieser Lösung mit Ihrer `break`-Implementierung.

¹¹Z. B. `int findFirstValue(int[] numbers, int limit)`