Prinzipien der Programmierung

Daniel Merkle

Wintersemester 2024

(Teil 02)

Wiederkehrende Probleme und Lösungsansätze

Lernziele

- Sie wissen, dass bestimmte Teilprobleme, wie das Einlesen von Eingaben oder Berechnungen, in Programmen immer wieder auftreten.
- Sie sind sich bewusst, dass es für bestimmte Teilprobleme Lösungsmuster gibt.
- Sie üben, Lösungsansätze für Teilprobleme zu kombinieren, um umfassendere Probleme zu lösen.

Einführung

- **Teilprobleme** treten in Programmen häufig auf:
  - Lesen von Benutzereingaben
  - Durchführung von Berechnungen
  - Anwendung von bedingter Logik
- **Lösungspatterns** helfen, diese Teilprobleme systematisch zu lösen.

Benutzereingaben lesen

Verwendung des `Scanner`

- **Importieren** der Klasse:

```
import java.util.Scanner;
```

- **Erstellen** eines `Scanner`-Objekts:

```
Scanner reader = new Scanner(System.in);
```

- **Lesen** verschiedener Datentypen:

```
String text = reader.nextLine();
int number = Integer.valueOf(reader.nextLine());
double decimal = Double.valueOf(reader.nextLine());
boolean bool = Boolean.valueOf(reader.nextLine());
```

Quiz User Input

Berechnungen durchführen

Schritte zur Durchführung von Berechnungen

1. **Identifizieren** der Eingaben und Deklarieren der Variablen.
2. **Identifizieren** der Operation und Deklarieren einer Variablen für das Ergebnis.
3. **Verwendung** des Ergebnisses (z.B. Ausgabe, weitere Berechnungen).

Beispiel: Summe zweier Zahlen

```java
// Eingaben deklarieren
int first = 1;
int second = 2;

// Berechnung durchführen
int sum = first + second;

// Ergebnis ausgeben
System.out.println("Die Summe von " + first + " und " + second + " ist " + sum);
```

Kombinierte Muster: Eingabe und Berechnung

Beispiel: Produkt von Benutzereingaben

```java
import java.util.Scanner;

public class Programm {
    public static void main(String[] args) {
        Scanner reader = new Scanner(System.in);

        // Eingaben lesen
        int first = Integer.valueOf(reader.nextLine());
        int second = Integer.valueOf(reader.nextLine());

        // Berechnung durchführen
        int product = first * second;

        // Ergebnis ausgeben
        System.out.println("Das Produkt von " + first + " und " + second + " ist
    }
}
```

Was ist eine Deklaration?

Die **Deklaration** einer Variable gibt ihren Namen und ihren Typ an, aber sie weist ihr keinen Wert zu. Die Deklaration reserviert Speicherplatz für die Variable.

```
int number;  // Deklaration einer Variablen vom Typ int
```

Was ist eine Definition?

Die **Definition** einer Variable weist ihr einen Wert zu. Dies geschieht normalerweise nach der Deklaration oder kombiniert mit der Deklaration.

```
number = 5;  // Definition, Zuweisung eines Wertes
```

Deklaration und Definition kombinieren

Es ist auch möglich, die Deklaration und Definition einer Variable in einer einzigen Anweisung zu kombinieren:

```
int number = 5;  // Deklaration und Definition in einer Zeile
```

Hier wird sowohl der Name und Typ der Variablen festgelegt als auch ein Wert zugewiesen.

Unterschied: Deklaration vs. Definition

- **Deklaration**: Nur der Name und Typ der Variable wird festgelegt. Kein Speicher wird zugewiesen, wenn kein Wert angegeben wird.
- **Definition**: Der Variable wird ein Wert zugewiesen, was den Speicherbedarf festlegt.

Beispiel:

```
int number;       // Deklaration
number = 10;      // Definition (Wertzuweisung)
```

Was ist ein Lösungsmuster?

Ein **Lösungsmuster** ist ein allgemeiner Ansatz zur Lösung eines wiederkehrenden Problems in der Programmierung. Beispiele sind das Einlesen von Benutzereingaben oder das Berechnen von Summen.

Bedingte Logik

Verwendung von `if`-Anweisungen

- **Syntax einer einfachen Bedingung**:

```
if (bedingung) {
    // Codeblock, wenn Bedingung wahr ist
}
```

- **Mit `else`**:

```
if (bedingung) {
    // Wenn wahr
} else {
    // Wenn falsch
}
```

- **Mit `else if`**:

```
if (bedingung1) {
    // Wenn Bedingung1 wahr
} else if (bedingung2) {
    // Wenn Bedingung2 wahr
} else {
    // Andernfalls
}
```

Beispiel: Zahlenvergleich

```
int value = 15;
if (value > 42) {
    System.out.println("ok");
} else {
    System.out.println("nicht ok");
}
```

Beispiel: Verschachtelte Bedingungen

Ein Beispiel für eine verschachtelte Bedingung:

```
if (value > 5) {
    System.out.println("Greater than 5");
} else if (value < 0) {
    System.out.println("Less than 0");
} else {
    System.out.println("Between 0 and 5");
}
```

Bedingte Logik und Berechnungen kombinieren

Ein Programm, das die Summe von zwei Zahlen berechnet und abhängig vom Ergebnis eine Nachricht ausgibt:

```java
import java.util.Scanner;

public class Program {
    public static void main(String[] args) {
        Scanner reader = new Scanner(System.in);

        int first = Integer.valueOf(reader.nextLine());
        int second = Integer.valueOf(reader.nextLine());

        int sum = first + second;

        if (sum > 100) {
            System.out.println("too much");
        } else if (sum < 0) {
            System.out.println("too little");
        } else {
            System.out.println("ok");
        }
    }
}
```

Quiz bedingte Logik, `print` vs `println`

Teaser: Einführung in Design Patterns

- **Design Patterns** sind Muster, die häufige Lösungen für bestimmte Probleme darstellen.
- Sie fördern **Wiederverwendbarkeit** und **Best Practices** in der Softwareentwicklung.
- Erlauben eine **effiziente Kommunikation** zwischen Entwicklern.

Kategorien von Design Patterns

1. **Erzeugungsmuster (Creational Patterns)**: Sie helfen bei der Erstellung von Objekten, z.B. Singleton oder Factory.
2. **Strukturmuster (Structural Patterns)**: Sie beschreiben, wie Klassen und Objekte zu größeren Strukturen kombiniert werden können, z.B. Adapter oder Composite.
3. **Verhaltensmuster (Behavioral Patterns)**: Sie regeln die Kommunikation zwischen Objekten und die Zuweisung von Verantwortlichkeiten, z.B. Observer oder Strategy.

Jedes Muster stellt eine Lösung für ein spezifisches Problem dar und fördert gut strukturierte, wartbare Software.

Beispiel 1: Singleton Pattern (Erzeugungsmuster)

```
public class Singleton {
    private static Singleton instance;

    private Singleton() {} // Verhindert Instanzierung

    public static Singleton getInstance() {
        if (instance == null) {
            instance = new Singleton();
        }
        return instance;
    }
}
```

Erklärung:

Das Singleton-Muster stellt sicher, dass nur eine Instanz der Klasse `Singleton` erstellt wird. Die Methode `getInstance` gibt immer dieselbe Instanz zurück.

Beispiel 2: Strategy Pattern (Verhaltensmuster)

```
// Strategy Interface
interface PaymentStrategy {
    void pay(int amount);
}

// Implementations
class CreditCardStrategy implements PaymentStrategy {
    public void pay(int amount) {
        System.out.println("Paid " + amount + " using Credit Card.");
    }
}

class PayPalStrategy implements PaymentStrategy {
    public void pay(int amount) {
        System.out.println("Paid " + amount + " using PayPal.");
    }
}
```

```
// Context
class ShoppingCart {
    private PaymentStrategy strategy;

    public ShoppingCart(PaymentStrategy strategy) {
        this.strategy = strategy;
    }

    public void checkout(int amount) {
        strategy.pay(amount);
    }
}
```

Erklärung:

Das Strategy-Muster erlaubt es, das Zahlungsverhalten dynamisch auszuwählen. `CreditCardStrategy` und `PayPalStrategy` implementieren die gleiche Schnittstelle `PaymentStrategy`.

Beispiel 3: Strukturmuster: Adapter

Erläuterung:
Das Adapter-Muster verbindet inkompatible Schnittstellen, indem es eine Klasse erstellt, die eine andere Klasse "anpasst", damit sie eine gewünschte Schnittstelle implementiert. So können zwei Systeme, die nicht direkt kompatibel sind, miteinander arbeiten.

Warum ein Strukturmuster?
Das Adapter-Muster beschreibt, wie zwei Klassen zusammenarbeiten können, indem die Struktur der Schnittstellen angepasst wird, was es zu einem Strukturmuster macht.

Adapter Design Pattern

- **Problem**: Inkompatible Schnittstellen miteinander verbinden.
- **Lösung**: Eine Klasse, die die Schnittstelle eines anderen Objekts anpasst.

```
class Adapter implements Target {
    private Adaptee adaptee;
    public Adapter(Adaptee adaptee) {
        this.adaptee = adaptee;
    }
    public void request() {
        adaptee.specificRequest();
    }
}
```

Adapter Pattern: Cartesian to Polar Coordinates

Problem

- Wir haben ein System, das mit kartesischen Koordinaten (x, y) arbeitet.
- Unser neues Modul oder externe Bibliothek erwartet Polar-Koordinaten (r, θ).
- Ziel: Verbindung beider Systeme ohne Änderung des vorhandenen Codes.

Lösung: Adapter Pattern

- **Adapter Pattern** ermöglicht die Zusammenarbeit inkompatibler Schnittstellen.
- Der Adapter konvertiert kartesische Koordinaten in Polar-Koordinaten.

```
class CartesianToPolarAdapter implements PolarCoordinate {
    private CartesianCoordinate cartesian;

    public CartesianToPolarAdapter(CartesianCoordinate cartesian) {
        this.cartesian = cartesian;
    }

    public double getRadius() {
        return Math.sqrt(cartesian.getX() * cartesian.getX() + cartesian.getY()
    }

    public double getAngle() {
        return Math.atan2(cartesian.getY(), cartesian.getX());
    }
}
```

Beispiel-Nutzung

```
CartesianCoordinate cartesian = new CartesianCoordinate(3, 4);
PolarCoordinate polar = new CartesianToPolarAdapter(cartesian);

System.out.println("Radius (r): " + polar.getRadius());  // 5.0
System.out.println("Winkel (θ): " + polar.getAngle());   // 0.93 rad
```

Warum Adapter?

- **Zweck des Adapters**: Verbindung von inkompatiblen Schnittstellen.
- **Vorteil**: Bestehender Code muss nicht geändert werden.
- **Strukturmuster**: Anpassung der Schnittstelle, nicht der Implementierung.
- Ermöglicht Wiederverwendung von vorhandenen Klassen in neuen Kontexten.

Vorteile von Design Patterns

- **Wiederverwendbarkeit:** Bewährte Lösungen können in verschiedenen Projekten genutzt werden.
- **Lesbarkeit und Wartbarkeit:** Der Code wird einfacher zu verstehen und zu pflegen.
- **Flexibilität:** Erlauben einfachere Änderungen oder Erweiterungen der Software.

Zusammenfassung

- Sie haben gelernt, wie man Muster für Benutzereingaben und Berechnungen erkennt.
- Sie wissen, wie man bedingte Logik verwendet.
- Sie können diese Muster kombinieren, um komplexere Probleme zu lösen.

Wiederholungen in Java

Lernziele

- Schleifen in Java verstehen und nutzen.
- Den `break`-Befehl zum Beenden einer Schleife verwenden.
- Den `continue`-Befehl nutzen, um zum Anfang der Schleife zurückzukehren.
- Ein Programm schreiben, das Eingaben wiederholt liest, bis eine bestimmte Bedingung erfüllt ist.

Warum Schleifen verwenden?

Programme müssen oft wiederholte Aufgaben erledigen, z.B. das Lesen von Eingaben oder das Berechnen einer Summe. Ohne Schleifen müssten wir den gleichen Code mehrfach schreiben. Beispiel:

```
Scanner scanner = new Scanner(System.in);
int sum = 0;

System.out.println("Input a number: ");
sum += Integer.valueOf(scanner.nextLine());

System.out.println("Input a number: ");
sum += Integer.valueOf(scanner.nextLine());

System.out.println("The sum of the numbers is " + sum);
```

Eine Schleife zur Berechnung verwenden

Anstatt denselben Code mehrfach zu wiederholen, verwenden wir eine Schleife, um die Berechnungen effizienter zu gestalten:

```
Scanner scanner = new Scanner(System.in);
int numbersRead = 0;
int sum = 0;

while (numbersRead < 5) {
    System.out.println("Input number: ");
    sum += Integer.valueOf(scanner.nextLine());
    numbersRead += 1;
}

System.out.println("The sum of the numbers is " + sum);
```

Endlosschleifen

Eine Schleife kann auch "unendlich" laufen, wenn sie keine Bedingung zum Beenden hat.

Das folgende Programm druckt "unendlich oft" "I can program":

```
while (true) {
    System.out.println("I can program!");
}
```

Beende die Schleife mit `break`, um den unendlichen Ablauf zu stoppen.

Schleife beenden mit `break`

Mit dem Befehl `break` kann eine Schleife beendet werden, wenn eine bestimmte Bedingung erfüllt ist.

```
int number = 1;

while (true) {
    System.out.println(number);
    if (number >= 5) {
        break;
    }
    number++;
}

System.out.println("Ready!");
```

Verschachtelte Schleifen

```java
for (int i = 1; i <= 3; i++) {
    for (int j = 1; j <= 3; j++) {
        if (j == 2) {
            break; // Beendet die innere Schleife (die 'j'-Schleife)
        }
        System.out.println("i = " + i + ", j = " + j);
    }
}
```

Ausgabe

```
i = 1, j = 1
i = 2, j = 1
i = 3, j = 1
```

`break` beendet die innerste Schleife.

Eingaben in Schleifen verarbeiten

Eine Schleife kann verwendet werden, um Eingaben vom Nutzenden zu verarbeiten. Hier ein Beispiel, das wiederholt nach einer Eingabe fragt, bis der Nutzende "y" eingibt:

```
Scanner scanner = new Scanner(System.in);

while (true) {
    System.out.println("Exit? (y exits)");
    String input = scanner.nextLine();
    if (input.equals("y")) {
        break;
    }
    System.out.println("Let's continue!");
}
```

Eingabe von Zahlen bis 0

Das folgende Programm fragt den Nutzenden nach einer Zahl, bis der Nutzende 0 eingibt:

```
Scanner scanner = new Scanner(System.in);

while (true) {
    System.out.println("Input a number (0 to quit)");
    int command = Integer.valueOf(scanner.nextLine());
    if (command == 0) {
        break;
    }
    System.out.println("You entered " + command);
}
```

Was sind Schleifen?

Eine **Schleife** in Java wiederholt einen Codeblock, solange eine bestimmte Bedingung erfüllt ist. Schleifen ermöglichen die Ausführung wiederholter Aufgaben mit weniger Code.

Was macht `break`?

Der Befehl `break` beendet eine Schleife vorzeitig, auch wenn die Schleifenbedingung weiterhin wahr ist. Es wird oft verwendet, um Schleifen gezielt zu verlassen, wenn eine spezielle Bedingung erfüllt ist.

Zurück zum Anfang der Schleife mit `continue`

Manchmal muss eine Schleife nicht beendet, sondern direkt zum Anfang zurückgesetzt werden. Der `continue`-Befehl überspringt den restlichen Schleifenblock und setzt die Schleife am Anfang fort:

```
Scanner scanner = new Scanner(System.in);

while (true) {
    System.out.println("Insert positive integers: ");
    int number = Integer.valueOf(scanner.nextLine());

    if (number <= 0) {
        System.out.println("Unfit number! Try again.");
        continue;
    }

    System.out.println("Your input was " + number);
}
```

Verschachtelung und `continue`

```
for (int i = 1; i <= 3; i++) {
    for (int j = 1; j <= 3; j++) {
        if (j == 2) {
            continue; // Überspringt den Rest des inneren Schleifendurchlaufs, w
        }
        System.out.println("i = " + i + ", j = " + j);
    }
}
```

Ausgabe

```
i = 1, j = 1
i = 1, j = 3
i = 2, j = 1
i = 2, j = 3
i = 3, j = 1
i = 3, j = 3
```

Was macht dieses Program?

```java
public class UnklarerCode {
    public static void main(String[] args) {
        int grenze = 50; // Obergrenze festlegen

        for (int a = 2; a <= grenze; a++) {
            boolean status = true;
            for (int b = 2; b * b <= a; b++) {
                if (a % b == 0) {
                    status = false;
                    break;
                }
            }
            if (status) {
                System.out.println(a); // Ausgabe der Zahl, wenn status true ble
            }
        }
    }
}
```

Quiz `while` und `continue`

Zusammenfassung

- Sie haben gelernt, wie Schleifen, `break` und `continue` verwendet werden.
- Schleifen ermöglichen es, wiederholte Aufgaben effizient zu lösen.
- Der Befehl `break` beendet Schleifen, während `continue` zum Anfang der Schleife zurückkehrt.

Noch mehr Schleifen in Java

Lernziele

- Bedingte `while`-Schleifen verwenden.
- Die Struktur und Anwendung von `for`-Schleifen verstehen.
- Die Wahl zwischen `while`- und `for`-Schleifen treffen.

Bedingte Schleifen

Im Gegensatz zu `while (true)` -Schleifen, die "unendlich" laufen, bis sie durch `break` beendet werden, können Schleifen auch Bedingungen enthalten, die während der Ausführung überprüft werden:

```
int number = 1;

while (number < 6) {
    System.out.println(number);
    number++;
}
```

Die Schleife wird beendet, wenn die Bedingung `number < 6` falsch wird.

Code Visualization, `while` , 3-more-loops, 2 Beispiele

Die `for`-Schleife

`for`-Schleifen bieten eine kompakte Möglichkeit, eine Schleife zu schreiben, die eine bestimmte Anzahl von Durchläufen macht:

```
for (int i = 0; i < 10; i++) {
    System.out.println(i);
}
```

Eine `for`-Schleife hat vier Komponenten:

1. Initialisierung: `int i = 0;`
2. Bedingung: `i < 10;`
3. Inkrementierung: `i++`
4. Der Schleifenblock

Code Visualization, `for`, 3-more-loops

Wann `while` und wann `for`?

- Verwenden Sie eine **`while`-Schleife**, wenn die Anzahl der Durchläufe nicht im Voraus bekannt ist, z.B. wenn Sie auf eine Nutzereingabe warten.
- Verwenden Sie eine **`for`-Schleife**, wenn die Anzahl der Schleifendurchläufe im Voraus bekannt ist, z.B. beim Durchlaufen von Zahlenbereichen.

Schleifenstrukturen vergleichen

Vergleichen wir eine `while` - und eine `for` -Schleife, die dasselbe tun:

```
int i = 0;
while (i < 10) {
    System.out.println(i);
    i++;
}
```

```
for (int i = 0; i < 10; i++) {
    System.out.println(i);
}
```

Beide Schleifen geben die Zahlen 0 bis 9 aus, aber die `for` -Schleife ist kürzer und übersichtlicher.

Erweiterte Schleifen

Manchmal möchten Sie, dass eine Schleife nach einer bestimmten Anzahl von Wiederholungen endet. Dies kann durch eine einfache Bedingung in der Schleife oder durch die Verwendung von `break` erreicht werden.

```
int result = 0;

for (int i = 0; i < 4; i++) {
    result += 3;
}

System.out.println(result);
```

Code Visualization(s), genau das, aber mit `while`

While-Programme sind berechnungsuniversell

- **Berechnungsuniversialität** bedeutet, dass ein System jede berechenbare Funktion ausführen kann.
- **While-Programme** können jede Berechnung durchführen, die auch eine Turing-Maschine durchführen kann.
- **Schlussfolgerung:** While-Programme sind **Turing-vollständig** und somit berechnungsuniversell.

Berechnungsuniversialität

- **Kontrollstrukturen:**
  - Verwendung von Schleifen (`while`) und Bedingungen (`if`).
- **Unbegrenzter Speicher:**
  - Theoretisch unbegrenzte Variablen und Speicherplatz.
- **Simulation von Turing-Maschinen:**
  - **Band** als Datenstruktur (z.B. Arrays oder Listen).
  - **Zustände** durch Programmfluss und Variablen repräsentiert.
- **Fazit:**
  - Jede berechenbare Funktion kann mit While-Programmen implementiert werden.
  - Daher sind sie berechnungsuniversell.

Kombination von Bedingungen

Schleifen können auch mehrere Bedingungen enthalten. Wenn eine Schleife zum Beispiel nur dann weiterlaufen soll, wenn zwei Bedingungen erfüllt sind, kann man dies einfach durch logische Operatoren erreichen:

```
while (a < b && c > d) {
    // Code
}
```

Zusammenfassung

- Bedingte Schleifen ermöglichen flexible Wiederholungen.
- `for`-Schleifen sind ideal für Schleifen mit bekannter Anzahl an Durchläufen.
- Wählen Sie den passenden Schleifentyp basierend auf der Situation.

Eine Kuriosität?

Ein Ausflug nach `C`:

```c
#include <stdio.h>

int main() {
    int i = 5;
    int result = i++ * i++ * i++;  // Ausdruck mit Post-Inkrement
    printf("Wert von i nach Berechnung: %d\n", i); // i ist jetzt 8
    printf("Ergebnis von i++ * i++ * i++: %d\n", result);
    return 0;
}
```

```
210
```

Und in Java:

```
public class PostIncrementTest {
    public static void main(String[] args) {
        int i = 5;
        int result = i++ * i++ * i++;  // Ausdruck mit Post-Inkrement
        System.out.println("Wert von i nach Berechnung: " + i);     // i ist jet
        System.out.println("Ergebnis von i++ * i++ * i++: " + result); // Ergebn
    }
}
```

```
210
```

So what?

Nochmal: Eine Kuriosität?

Erstmal Java:

```
public class PostIncrementTest {
    public static void main(String[] args) {
        int i = 5;
        int result = ++i * ++i * ++i;  // Ausdruck mit Prä-Inkrement
        System.out.println("Wert von i nach Berechnung: " + i);     // i ist jet
        System.out.println("Ergebnis von ++i * ++i * ++i: " + result); // Ergebn
    }
}
```

```
336
```

Looks good (6 * 7 * 8)

Noch ein Ausflug nach `C`:

```c
#include <stdio.h>

int main() {
    int i = 5;
    int result = ++i * ++i * ++i;  // Ausdruck mit Prä-Inkrement
    printf("Wert von i nach Berechnung: %d\n", i); // i ist jetzt 8
    printf("Ergebnis von ++i * ++i * ++i: %d\n", result);
    return 0;
}
```

`336` (Apple Silicon M2, Ventura)

`392` (AMD)

Was ist da passiert? `C` ist unterspezifiziert!

Spezifikation einer Programmiersprache

- Eine **Spezifikation** ist ein formales, oft schriftliches Dokument, das die **Regeln**, **Syntax**, und **Semantik** einer Programmiersprache beschreibt.
- Sie definiert **wie** die Sprache funktionieren **soll** – unabhängig von einer konkreten Implementierung.
- Beispiele: die Spezifikationen für **C**, **Java**, **Python** sind in offiziellen Dokumenten festgehalten.

Compiler ist eine Implementierung

- Ein **Compiler** ist eine **Implementierung** der Spezifikation.
- Er wandelt Code, der gemäß der Spezifikation geschrieben ist, in Maschinencode um.
- Es kann **mehrere Compiler** für dieselbe Sprache geben (z.B. **GCC** und **Clang** für C).

Methoden und das Aufteilen von Programmen in kleinere Teile

Lernziele

- Verstehen, was Methodenparameter und Rückgabewerte sind.
- Erstellen und Aufrufen von Methoden.
- Arbeiten mit Methoden, die Parameter verwenden oder Werte zurückgeben.

Was ist eine Methode?

Eine **Methode** ist ein benannter Satz von Anweisungen. Methoden sind eine Möglichkeit, den Code in wiederverwendbare Teile zu zerlegen.

```
public static void greet() {
    System.out.println("Greetings from the method world!");
}
```

Die Methode `greet` kann überall im Code aufgerufen werden. Das spart Zeit und vermeidet doppelten Code.

Methodenaufruf

Um eine Methode aufzurufen, geben wir ihren Namen gefolgt von Klammern ein:

```
greet();
```

Im Beispiel unten rufen wir die Methode `greet` viermal auf:

```
public static void main(String[] args) {
    greet();
    greet();
    greet();
    greet();
}
```

Code Visualization, greet and greet (Methodenaufruf)

Benennung von Methoden

Richtlinien zur Benennung von Methoden

- Methoden sollten in **camelCase** benannt werden.
- Der erste Buchstabe des Methodennamens ist klein, alle weiteren Wörter beginnen mit einem Großbuchstaben.
- Beispiel: `thisIsAnExampleOfMethodName()`

Schlechte Benennung von Methoden

Hier ist ein Beispiel für eine schlecht benannte Methode:

```
public static void This_method_says_woof ( ) {
System.out.println("woof");
}
```

Probleme:

- Großbuchstaben am Anfang
- Wörter durch Unterstriche getrennt
- Falsche Einrückung
- Klammern sind nicht korrekt platziert

Korrekte Benennung von Methoden

Dies ist die korrekte Art, eine Methode zu benennen:

```
public static void thisMethodSaysWoof() {
    System.out.println("woof");
}
```

Korrekt:

- camelCase
- Keine Leerzeichen vor den Klammern
- Richtig eingerückt (hier: vier Zeichen Einrückung)

Einrückung in Methoden

- Der Code innerhalb von Methoden wird **um vier Zeichen eingerückt**.
- Dies macht den Code lesbarer und strukturiert.

Schlecht eingerückt:

```
public static void thisMethod() {
System.out.println("woof");
}
```

Korrekt eingerückt:

```
public static void thisMethod() {
    System.out.println("woof");
}
```

Quiz User Methoden Namen

Methoden mit Parametern

Eine Methode kann Parameter erhalten, die bei ihrem Aufruf an sie übergeben werden:

```
public static void printMessage(String message) {
    System.out.println(message);
}
```

Der Aufruf könnte dann so aussehen:

```
printMessage("Hello from a parameter!");
```

Code Visualization, Methoden mit mehreren Parametern, 4-methods

Methoden mit Rückgabewerten

Methoden können Werte zurückgeben, die dann vom aufrufenden Code verwendet werden:

```
public static int add(int a, int b) {
    return a + b;
}
```

```
int result = add(5, 3);
System.out.println(result);  // Ausgabe: 8
```

Rückgabewert vs. `void`

- `void` bedeutet, dass eine Methode nichts zurückgibt.
- Ein Rückgabewert gibt dem aufrufenden Programm Daten zurück:

```
public static int multiply(int a, int b) {
    return a * b;
}
```

Parameterwerte kopieren

Parameter in Methoden werden **kopiert**, sodass Änderungen innerhalb der Methode keine Auswirkungen auf die originalen Variablen haben.

```
public static void changeNumber(int number) {
    number = 10;
}
```

Code Visualization, scope, Veränderung von `min` in `printNumbers` ändert nicht `min` in `main`, 4-methods

Code Visualization, scope, Veränderung von `number`, 4-methods

Quiz scope

Quiz noch mehr scope, Variablen in Methoden

Methoden mit Rückgabewerten

Die Definition einer Methode, die einen Wert zurückgibt, sieht so aus:

```
public static int alwaysReturnsTen() {
    return 10;
}
```

Solche Methoden können in Ausdrücken verwendet werden:

```
int result = alwaysReturnsTen() * 2;
```

Quiz Rückgagbewerte

Was ist der Aufrufstack?

Der **Aufrufstack** ist ein spezieller Speicherbereich, den Java verwendet, um den Status von Methodenaufrufen zu speichern.

- **Jede Methode**, die aufgerufen wird, erzeugt einen **Frame** auf dem Stack.
- Der Frame enthält Informationen über die **lokalen Variablen** der Methode und den **Rücksprungpunkt**.

Der Aufrufstack in Aktion

Angenommen, wir haben folgendes Programm:

```
public static void main(String[] args) {
    start();
}

public static void start() {
    int a = 5;
    int b = 6;
    int result = sum(a, b);
    System.out.println("Sum: " + result);
}

public static int sum(int x, int y) {
    return x + y;
}
```

Aufrufstack am Anfang

1. Die **`main`-Methode** wird aufgerufen.
2. Ein **Frame** für `main` wird auf dem Aufrufstack erstellt.

```
Stack:
main
```

Der `start`-Methodenaufruf

Wenn die Methode `start()` aufgerufen wird, wird ein neuer Frame auf den Stack gelegt.

- Der Frame enthält die lokalen Variablen `a` und `b`.

```
Stack:
start
  a = 5
  b = 6
main
```

Aufruf der `sum`-Methode

Beim Aufruf von `sum(a, b)` passiert Folgendes:

1. Die Werte von `a` und `b` werden an die Methode `sum` übergeben.
2. Ein neuer Frame für `sum` wird erstellt.

```
Stack:
sum
  x = 5
  y = 6
start
  a = 5
  b = 6
main
```

Rückgabe des Ergebnisses

Die Methode `sum` berechnet das Ergebnis `x + y` und gibt es zurück.

1. Der Frame für `sum` wird vom Stack entfernt.
2. Der Wert 11 wird im `start`-Frame gespeichert.

```
Stack:
start
  a = 5
  b = 6
  result = 11
main
```

Stack nach Abschluss der Methode

Nachdem `start()` beendet ist:

1. Der Frame von `start` wird entfernt.
2. Die Ausführung kehrt zu `main` zurück.

```
Stack:
main
```

Wichtige Konzepte

- Der **Aufrufstack** verwaltet Methodenaufrufe und Rücksprünge.
- Jede Methode erhält einen eigenen **Frame**.
- Frames werden hinzugefügt, wenn Methoden aufgerufen werden, und entfernt, wenn die Methode beendet ist.

Code Visualization, Stack (Methodenaufruf), gleiche Visualisierung, Fokus auf Stack

Für Anfänger ausgezeichnet: Pythontutor (für Javacode Visualisierung)

Code Visualization, Stack mit Variablen, `printStars`

Code Visualization, Stack mit Variablen, `sum(first, second)`

Code Visualization, Methodenaufruf innerhalb von Methoden, `multiplicationTable(3)`

Scope und Stack

Konzept des Scope (Gültigkeitsbereichs)

- Der **Gültigkeitsbereich (Scope)** beschreibt den Bereich des Programms, in dem eine Variable oder Funktion gültig und zugänglich ist.
- In modernen Programmiersprachen können Variablen lokal oder global definiert werden. Der Gültigkeitsbereich beeinflusst:
  - **Sichtbarkeit**: Welche Teile des Programms können auf die Variable zugreifen?
  - **Lebensdauer**: Wie lange existiert die Variable im Speicher?

Arten von Scopes

1. **Globaler Scope**:
    - Variablen oder Methoden, die global definiert sind, sind im gesamten Programm sichtbar.
    - Typisches Beispiel: `static` Variablen in Java.
2. **Lokaler Scope**:
    - Variablen sind nur innerhalb eines bestimmten Blocks, wie einer Methode oder Schleife, gültig.
    - Nach Verlassen des Blocks wird die Variable „vergessen“.

Beispiel:

```
public void example() {
    int x = 10;  // x existiert nur in diesem Block
}
```

- **Block Scope**: Jede `{}` -Blockstruktur (wie Schleifen oder if-Statements) definiert einen eigenen Gültigkeitsbereich.

Stack und Scope: Verbindung zwischen Variablen und Speicher

- **Stack** und **Scope** sind eng miteinander verknüpft:
  - Der **Stack** speichert lokale Variablen und verwaltet den Ausführungszustand eines Programms.
  - **Jede Methode** hat ihren eigenen Speicherbereich auf dem Stack, den sogenannten **Frame**, der u.a. die lokalen Variablen der Methode enthält.

Prinzipien:

- **LIFO-Prinzip**: Der Stack arbeitet nach dem „Last In, First Out"-Prinzip. Der letzte Aufruf einer Methode ist der erste, der abgeschlossen wird.
- **Dynamische Speicherverwaltung**: Der Stack ist ein dynamisch wachsender und schrumpfender Speicherbereich, der die Ausführungsmethoden rekursiv verwaltet.

Scope und Speicher (Theoretischer Hintergrund)

- **Lexikalischer Scope**: In vielen Programmiersprachen wird der Gültigkeitsbereich zur **Kompilierungszeit** festgelegt. Das bedeutet, dass der Compiler bestimmt, welche Variablen in welchem Block sichtbar sind.
- **Stack Frames**: Ein Stack Frame wird für jede Methode erstellt, wenn sie aufgerufen wird, und enthält:
    - Lokale Variablen der Methode
    - Rücksprungadresse zum Aufrufer
- **Stack Overflow**: Zu viele rekursive Aufrufe oder extrem tiefe Funktionsaufrufe führen zum **Stack Overflow**, da der Stack nur begrenzt Speicher hat.

Dynamischer vs. Lexikalischer Scope

In Programmiersprachen gibt es zwei Haupttypen von Scoping: dynamischen und lexikalischen (statischen) Scope.

- Dynamischer Scope: Der Gültigkeitsbereich wird zur Laufzeit bestimmt, basierend auf der Aufrufkette.
- Lexikalischer Scope (von den meisten modernen Programmiersprachen genutzt): Der Gültigkeitsbereich wird zur Kompilierzeit bestimmt, basierend auf der Struktur des Codes.

Dynamischer Scope in Bash

```bash
#!/bin/bash

my_function() {
  echo "Inner function: $var"
}

outer_function() {
  local var="Dynamischer Scope in Bash"
  my_function
}

var="Global Variable"
outer_function
```

- In Bash ist der Scope dynamisch.
- `my_function` greift auf `var` zu, das von `outer_function` gesetzt wurde.

Erwartete Ausgabe: `Dynamischer Scope in Bash`

In dynamischen Scopes wie in Bash wird die Variable `var` aus der Aufrufkette von `outer_function` verwendet.

Lexikalischer Scope in Java

```java
public class ScopeExample {

    static void myFunction() {
        // var is not visible here
    }

    static void outerFunction() {
        String var = "Statischer Scope in Java";
        myFunction();
    }

    public static void main(String[] args) {
        String var = "Globale Variable";
        outerFunction();
    }
}
```

- In Java ist der Scope lexikalisch.
- `myFunction` kann nicht auf die `var` -Variable in `outerFunction` zugreifen.

Erwartete "Ausgabe", falls auf `var` zugegriffen würde: **Fehler**: `var cannot be resolved to a variable`

Lexikalischer Scope bedeutet, dass Variablen nur innerhalb ihres definierten Bereichs sichtbar sind.

Unterschied: Dynamischer vs. Lexikalischer Scope

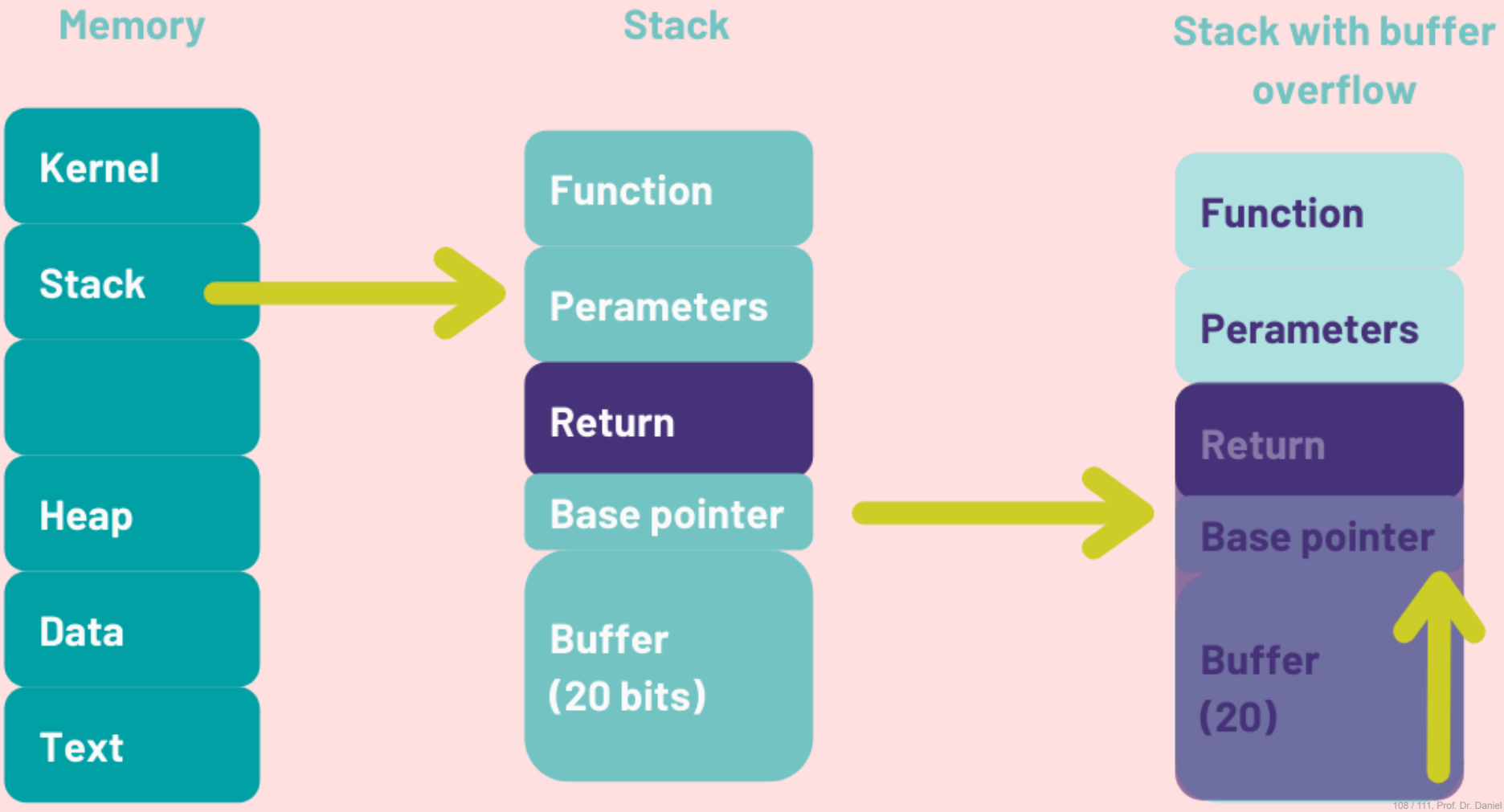
- **Dynamischer Scope**: Variablen werden anhand der Aufrufkette zur Laufzeit aufgelöst.
    - Beispiel: Bash.
    - Funktionen haben Zugriff auf Variablen der aufrufenden Funktion.
- **Lexikalischer Scope**: Variablen werden anhand des Codes zur Kompilierzeit aufgelöst.
    - Beispiel: Java.
    - Funktionen haben keinen Zugriff auf Variablen außerhalb ihres Bereichs.

- Dynamischer Scope ist flexibel, kann aber unerwartete Fehler verursachen, da Variablen über die Aufrufkette verteilt werden.
- Lexikalischer Scope ist kontrollierter und führt zu weniger unerwarteten Fehlern, da der Gültigkeitsbereich strikt an die Code-Struktur gebunden ist.

Dieser Unterschied beeinflusst, wie Variablen innerhalb eines Programms aufgelöst werden und kann unterschiedliche Ergebnisse hervorrufen.

Zurück zu Stack und Frames

Buffer Overflow Attack



Scope und Laufzeitverhalten

- Der **Scope** beeinflusst das **Laufzeitverhalten** eines Programms direkt:
  - **Laufzeitfehler**: Variablen außerhalb ihres Scopes aufrufen führt zu Laufzeitfehlern.
  - **Speicherfreigabe**: Wenn der Stack Frame einer Methode freigegeben wird, werden alle lokalen Variablen „vergessen“ und der Speicher freigegeben.

Visualisierung des Speichers bei Method A, die Method B aufruft:

```
main
  – stack frame (main)
  – stack frame (method A)
  – stack frame (method B)
```

Zusammenfassung

- Methoden helfen, Programme zu strukturieren und Code wiederverwendbar zu machen.
- Methoden können Parameter haben und Werte zurückgeben.
- Parameterwerte werden bei Methodenaufrufen kopiert.
- Stack, Frame und S cope