Prinzipien der Programmierung

Daniel Merkle

Wintersemester 2024

(Teil 03)

Ein sehr bekanntes Experiment

Quiz Fehler entdecken

Quiz: **Zählen Sie, wie oft von Personen mit weißem Oberteil der Ball gepasst wird.**

Count

Wahrnehmungsbedingte Blindheit

- **Definition**: Wenn unser Gehirn auf eine Aufgabe fokussiert, blendet es oft andere Informationen aus, was zu Fehlern führen kann.
- **Beispiel in der Programmierung**: Entwickler:innen übersehen manchmal offensichtliche Fehler, weil sie sich zu sehr auf einen bestimmten Teil des Codes konzentrieren.
- **Lösung**: Übung und regelmäßige Pausen helfen, die Fähigkeit zu verbessern, wesentliche Informationen zu erkennen und Fehler besser zu finden.

Kommentare und Lesbarkeit des Codes

- **Kommentare im Code**: Dienen dazu, den Zweck und die Funktionsweise von Codeabschnitten zu erklären, besonders beim Debugging.
- **Selbstdokumentierender Code**: Gut gewählte Variablen- und Methodennamen machen den Code verständlicher, ohne übermäßig viele Kommentare.

Beispiel für selbstdokumentierenden Code:

```
public static void printValuesFromTenToOne() {
    int value = 10;
    while (value > 0) {
        System.out.println(value);
        value = value - 1;
    }
}
```

Fehler finden mit Print-Debugging

- **Print-Debugging**: Ein einfacher Ansatz, um Fehler zu finden, indem man **Print-Anweisungen** hinzufügt, die den Programmablauf und den Zustand von Variablen während der Ausführung anzeigen.
- **Vorteil**: Hilft, den Programmfluss nachzuvollziehen und kritische Stellen im Code zu identifizieren.
- **Eckfälle (Corner Cases)**: Tests in extremen Situationen (z.B. keine Eingaben, sehr große Werte), um seltene Fehler zu erkennen.

Beispiel für Print-Debugging:

```
System.out.println("-- values: " + values + ", sum: " + sum);
```

Quiz Fehler entdecken

Presentation:IntelliJ IDEA Debugging

Einführung in Listen

- **Was sind Listen?**
  - Ein Werkzeug, um viele Werte desselben Typs zu speichern.
  - Java verwendet dafür die **ArrayList**.
  - Man kann Elemente hinzufügen, entfernen und nach bestimmten Werten durchsuchen.
- **Warum Listen verwenden?**
  - Spart das Erstellen vieler einzelner Variablen.
  - Flexibel und dynamisch in der Größe.

Erstellen und Verwenden von Listen

- **Importieren**: `import java.util.ArrayList;`
- **Liste erstellen**:

```
ArrayList<String> liste = new ArrayList<>();
```

- **Hinzufügen von Elementen**:

```
liste.add("Wert");
```

- **Abrufen eines Wertes**:

```
System.out.println(liste.get(0));
```

Typen in Listen

- Listen in Java speichern nur Referenztypen.
  - Für primitive Datentypen gibt es Wrapper-Klassen:
    - `int` → `Integer`
    - `double` → `Double`
- Beispiel:

```
ArrayList<Integer> zahlen = new ArrayList<>();
zahlen.add(10);
```

Werttypen und Referenztypen

Java Datentypen:

- **Werttypen (Primitive Types):**
  - Speichern den **tatsächlichen Wert**
  - Beispiele: `int`, `double`, `boolean`, `char`
- **Referenztypen (Reference Types):**
  - Speichern eine **Referenz (Verweis)** auf ein Objekt im Speicher
  - Beispiele: `String`, `Array`, `ArrayList`, Objekte von Klassen

Werttypen (Primitive Types)

Werttypen:

- Speichern **direkt den Wert** in der Variable.
- Primitive Typen:
  - `int` (Ganzzahlen)
  - `double` (Fließkommazahlen)
  - `boolean` (Wahrheitswerte)
  - `char` (Zeichen)
  - Weitere: `byte`, `short`, `long`, `float`

Wichtige Eigenschaft:

- Bei der Übergabe an eine Methode wird **eine Kopie des Wertes** (sprich: per Wert, per value) übergeben.

Referenztypen (Reference Types)

Referenztypen:

- Speichern eine **Referenz** (Speicheradresse) auf ein Objekt.
- Beispiele: `String`, `Array`, `ArrayList`, Objekte von Klassen.

Wichtige Eigenschaft:

- Bei der Übergabe an eine Methode wird **die Referenz** (sprich: per reference) übergeben.
- Änderungen innerhalb der Methode wirken sich auf das **ursprüngliche Objekt** aus.

Unterschiede zwischen Wert- und Referenztypen

Eigenschaft	Werttypen	Referenztypen
Speicherort	Speichert den tatsächlichen Wert	Speichert eine Referenz (Speicheradresse)
Datentypen	Primitive Typen (`int`, `double`)	Objekte, Arrays, Strings, etc.
Übergabe an Methoden	Wert wird kopiert	Referenz wird kopiert, Objekt bleibt gleich
Speicherbedarf	Fest definiert	Dynamisch, abhängig vom Objekt
Änderung in Methoden	Keine Auswirkungen auf Originalwert	Änderungen betreffen das ursprüngliche Objekt

Beispiel für Referenztypen

Beispiel:

```
ArrayList<String> list1 = new ArrayList<>();
list1.add("Hello");
list1.add("World");

ArrayList<String> list2 = list1;  // list2 verweist auf dasselbe Objekt

System.out.println(list2);  // Ausgabe: [Hello, World]
```

Erklärung:

- **list1** und **list2** verweisen auf dasselbe Objekt.
- Änderungen über eine Referenz beeinflussen das gemeinsame Objekt.

Füge ein drittes Element hinzu

```
list2.add("Elephant");  // Füge "Elephant" zu list2 hinzu
System.out.println(list1);  // Ausgabe: [Hello, World, Elephant]
```

- Da `list2` auf dasselbe Objekt wie `list1` zeigt, wird `"Elephant"` auch in `list1` sichtbar.

Ergebnis nach der Änderung

Die Ausgabe von `list1` nach Hinzufügen von `"Elephant"` zu `list2`:

```
[Hello, World, Elephant]
```

Beispiel mit primitiven Typen

```java
public class Beispiel {
    public static void main(String[] args) {
        int zahl = 10;
        manipuliereZahl(zahl);
        System.out.println(zahl);  // Ausgabe: 10
    }

    public static void manipuliereZahl(int z) {
        z = 20;  // Wert von z wird geändert, aber nicht das Original
    }
}
```

- **Erklärung**: Der Wert von `zahl` bleibt nach dem Funktionsaufruf unverändert, weil nur eine Kopie übergeben wurde.

Beispiel mit Referenztyp (Objekt)

```
public class Beispiel {
    public static void main(String[] args) {
        IntegerWrapper zahl = new IntegerWrapper(10);
        manipuliereIntegerWrapper(zahl);
        System.out.println(zahl.wert);  // Ausgabe: 15
    }

    public static void manipuliereIntegerWrapper(IntegerWrapper z) {
        z.add(5);  // Hier wird der Wert um 5 erhöht
    }
}

class IntegerWrapper {
    public int wert;

    public IntegerWrapper(int wert) {
        this.wert = wert;
    }

    // Methode zum "Überladen" von +
    public void add(int x) {
        this.wert += x;  // Wert des Objekts wird direkt verändert
    }
}
```

- **Erklärung**: Hier wird der Inhalt des Objekts `obj` verändert, weil das Objekt über die Referenz modifiziert wird.

Beispiel: Wrapper "Integer" (Immutable)

```
public class Beispiel {
    public static void main(String[] args) {
        Integer zahl = 10;
        manipuliereInteger(zahl);
        System.out.println(zahl);  // Ausgabe: 10
    }

    public static void manipuliereInteger(Integer z) {
        z = z + 5;  // Das neue Integer-Objekt wird erstellt, aber das Original
    }
}
```

- **Erklärung**: Da `Integer` **immutable** (unveränderlich) ist, wird bei der Zuweisung `z = 20` nur ein neues `Integer`-Objekt erstellt. Das ursprüngliche Objekt bleibt unverändert.
- **Fazit**: Wrapper-Klassen wie `Integer` sind immutable, daher wird das ursprüngliche

Arbeiten mit Listen

- **Indexing**: Listen beginnen bei 0 zu zählen.
  - `liste.get(0)` gibt den ersten Wert zurück.
- **Fehlerbehandlung**: Zugriff auf nicht existierende Indizes führt zu `IndexOutOfBoundsException`.

Iteration über Listen

- **Schleifen**:
  - `for` -Schleife:

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

  - **while-Schleife**:

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

Schleifen über Listen (Fortsetzung)

- **Verwendung der `.size()`-Methode**:
  - Gibt die Anzahl der Elemente in der Liste zurück.
  - Beispiel:

```
System.out.println("Liste hat " + liste.size() + " Elemente.");
```

- **Umgekehrte Iteration**:

```
for (int i = liste.size() - 1; i >= 0; i--) {
    System.out.println(liste.get(i));
}
```

For-Each-Schleife ohne Notwendigkeit des Index

For-Each-Schleife Syntax:

```
for (TypeOfVariable nameOfVariable: nameOfList) {
    // Schleifeninhalt
}
```

Erklärung:

- **TypeOfVariable**: Typ der Elemente in der Liste (z. B. `String`, `Integer`)
- **nameOfVariable**: Name der temporären Variablen, die jedes Element speichert
- **nameOfList**: Name der Liste, über die iteriert wird

Beispiel – For-Each über Liste von Lehrern

```
import java.util.ArrayList;

public class RepeatStatement {
    public static void main(String[] args) {
        ArrayList<String> teachers = new ArrayList<>();

        teachers.add("Simon");
        teachers.add("Samuel");
        teachers.add("Ann");
        teachers.add("Anna");

        for (String teacher: teachers) {
            System.out.println(teacher);
        }
    }
}
```

Vorteile dieser Art `for` Schleife

- **Einfachheit**: Keine manuelle Verwaltung des Indexes erforderlich.
- **Lesbarkeit**: Klarer und verständlicher Code.
- **Sicherheit**: Kein Risiko, den Index zu überschreiten.

Vergleich mit der traditionellen For-Schleife

For-Schleife (traditionell):

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

- **for-each-Schleife** eliminiert die Notwendigkeit, den Index explizit zu verwalten.

Geordnete und ungeordnete Datenstrukturen

- **Geordnete Datenstrukturen** bewahren die Reihenfolge der eingefügten Elemente oder sortieren sie.
    - Beispiel: Listen (z.B. `ArrayList`), `LinkedHashSet`, `TreeSet`.
- **Ungeordnete Datenstrukturen** garantieren keine Reihenfolge der Elemente.
    - Beispiel: `HashSet`.
- Diese Strukturen werden verwendet, um Duplikate zu vermeiden oder schnellen Zugriff auf Elemente zu ermöglichen.

Beispiel: `HashSet` (Ungeordnete Struktur)

```java
import java.util.HashSet;
import java.util.Set;

Set<String> fruits = new HashSet<>();
fruits.add("Apple");
fruits.add("Banana");
fruits.add("Orange");

for (String fruit : fruits) {
    System.out.println(fruit); // Keine garantierte Reihenfolge der Elemente
}
```

- In einem `HashSet` gibt es **keine garantierte Reihenfolge** der Elemente.
- Die Ausgabe der Elemente kann sich bei jedem Durchlauf unterscheiden.

Werte aus einer Liste entfernen

- **Elemente entfernen**:
  - Nach Index: `liste.remove(1);`
  - Nach Wert: `liste.remove("Wert");`
- **Wichtig bei Integer-Listen**:
  - Zum Entfernen eines Wertes in einer Integer-Liste:

```
liste.remove(Integer.valueOf(15));
```

Entfernen eines Elements aus einer ArrayList

```
ArrayList<String> list = new ArrayList<>();
list.add("A");
list.add("B");
list.add("C");
list.add("D");
list.add("E");

System.out.println(list);
list.remove(3);
System.out.println(list);
```

- Originale Liste: `[A, B, C, D, E]`
- Entferne das Element an Index 3 (`D`).

Ergebnis nach dem Entfernen

Nach dem Entfernen von `"D"`:

- Elemente mit höheren Indizes (wie `"E"`) rutschen nach vorne.

```
// Ergebnis:
[A, B, C, E]
```

- Vorher:
  - `"D"` bei Index 3, `"E"` bei Index 4
- Nachher:
  - `"E"` verschiebt sich auf Index 3

Entfernen eines Werts mit `Integer.valueOf(15)`

```
ArrayList<Integer> list = new ArrayList<>();
list.add(15);
list.add(10);
list.add(15);
list.add(20);

System.out.println(list);
list.remove(Integer.valueOf(15));
System.out.println(list);
```

- Originale Liste: `[15, 10, 15, 20]`
- Entferne das erste Vorkommen von `15`.

Ergebnis nach dem Entfernen

Nach dem Entfernen des ersten `15`:

- Nur das **erste** Vorkommen von `15` wird entfernt.

```
// Ergebnis:
[10, 15, 20]
```

- Vorher:
  - Zwei Instanzen von `15`.
- Nachher:
  - Nur die **erste** Instanz von `15` wurde entfernt.

Überprüfen, ob ein Wert in einer Liste ist

- **Methode `.contains()`**:
  - Überprüft, ob ein Wert in der Liste vorhanden ist.
  - Gibt `true` oder `false` zurück:

```
if (liste.contains("Wert")) {
    System.out.println("Wert ist in der Liste.");
}
```

Methodenparameter mit Listen

- Listen können als Parameter an Methoden übergeben werden.
  - Beispiel:

```
public static void printListe(ArrayList<String> liste) {
    for (String s : liste) {
        System.out.println(s);
    }
}
```

- Methoden können auch mehrere Parameter haben:

```
public static void printSmallerThan(ArrayList<Integer> numbers, int threshol
    for (int number: numbers) {
        if (number < threshold) {
            System.out.println(number);
        }
    }
}
```

Referenztypen in Listen

- Listen sind Referenztypen.
    - Methoden, die Listen als Parameter verwenden, ändern das originale Listenobjekt.
- Beispiel:

```
public static void removeFirst(ArrayList<Integer> numbers) {
    numbers.remove(0);
}
```

Quiz `remove`

Quiz nochmal `remove`

Zusammenfassung der wichtigsten ArrayList-Methoden

- **.add()**: Fügt ein Element zur Liste hinzu.
- **.size()**: Gibt die Anzahl der Elemente in der Liste zurück.
- **.get()**: Holt das Element an einem bestimmten Index.
- **.remove()**: Entfernt ein Element (nach Index oder Wert).
- **.contains()**: Überprüft, ob ein Element in der Liste ist.

Code `RepeatStatement` [keine Darstellung der Liste selbst]

Presentation: Das gleiche in IntelliJ IDEA! [Darstellung der Liste]

Quiz Index in Listen

Code `RepeatStatement` mit `for (TypeOfVariable nameOfVariable: nameOfList)`

Beispiel: ArrayList

Java's **ArrayList** kann nur **Referenztypen** speichern, keine primitiven Typen wie `int`, `char` oder `double`.

- Beispiele: `ArrayList<String>`, `ArrayList<Integer>` (Wrapper für primitive Typen)

Das Verhalten der Liste hängt vom Typ der enthaltenen Objekte ab – ob sie **immutable** oder **mutable** sind.

Beispiel 1: ArrayList mit `String`

```
ArrayList<String> list = new ArrayList<>();
String a = "Hello", b = "World";
list.add(a); list.add(b); list.add(a); list.add(b);

list.set(0, list.get(0) + "li");

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

Erwartete Ausgabe:

```
[Helloli, World, Hello, World]
```

Beispiel 2: ArrayList mit `StringBuilder`

```
ArrayList<StringBuilder> list = new ArrayList<>();
StringBuilder a = new StringBuilder("Hello"), b = new StringBuilder("World");
list.add(a); list.add(b); list.add(a); list.add(b);

list.get(0).append("li");

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

Erwartete Ausgabe:

```
[Helloli, World, Helloli, World]
```

Mutable vs. Immutable

- **Immutable Objekte**: Unveränderlich nach ihrer Erstellung. Änderungen erzeugen ein **neues Objekt**.
  - Beispiele: `String`, Wrapper-Klassen (`Integer`, `Double`)
- **Mutable Objekte**: Änderbar nach ihrer Erstellung.
  - Beispiel: `StringBuilder`, Arrays, Listen

Zusammenfassung

- In einer `ArrayList` können nur **Referenztypen** gespeichert werden, keine primitiven Typen.
- Bei **immutable** Objekten wie `String` führen Änderungen zur Erzeugung neuer Objekte.
- Bei **mutable** Objekten wie `StringBuilder` beeinflussen Änderungen alle Referenzen auf dasselbe Objekt.

Einführung in Arrays

- **Array**: Ein Container-Objekt, das eine **fixe Anzahl** von Werten des gleichen Typs speichern kann.
- Elemente eines Arrays sind **nummerierte Positionen**, beginnend bei 0.
- Arrays sind ein grundlegender Datentyp in Java und bieten eine effiziente Möglichkeit, viele Werte zu speichern.

Erstellen eines Arrays

1. **Deklaration eines Arrays**:

```
int[] numbers = new int[3];
```

   - Erstellt ein Array `numbers` mit Platz für **3 ganze Zahlen**.

2. **Array von Strings**:

```
String[] strings = new String[5];
```

   - Erstellt ein Array `strings`, das **5 Strings** speichern kann.

Elemente zuweisen und darauf zugreifen

1. **Zuweisung eines Wertes**:

```
numbers[0] = 2;
numbers[2] = 5;
```

   - Der Wert 2 wird an **Index 0** und der Wert 5 an **Index 2** gespeichert.

2. **Zugriff auf Elemente**:

```
System.out.println(numbers[0]);
System.out.println(numbers[2]);
```

3. **Wichtiger Hinweis**:

   - Die **Indizes eines Arrays** beginnen bei **0** und enden bei **Länge des Arrays - 1**.

Beispiel – Zugriff auf Array-Werte

1. **Array-Zugriff mit Benutzereingabe**:

```
Scanner reader = new Scanner(System.in);
int[] numbers = {42, 13, 12, 7, 1};

System.out.println("Welchen Index sollen wir abrufen?");
int index = Integer.valueOf(reader.nextLine());
System.out.println(numbers[index]);
```

   - Der Benutzer gibt einen Index ein, und der Wert an diesem Index wird ausgegeben.

Tauschen von Werten in Arrays

1. **Beispiel für das Tauschen zweier Werte**:
   - Werte an zwei vom Benutzer eingegebenen Indizes tauschen:

```
int temp = numbers[index1];
numbers[index1] = numbers[index2];
numbers[index2] = temp;
```

2. **Beispielausgabe**:

```
1
3
5
7
9
Give two indices to swap: 2, 4
1
3
9
7
5
```

Quiz `swap`

Iteration über ein Array

1. **Schleife zum Durchlaufen eines Arrays**:

```
int[] numbers = {42, 13, 12, 7};
int index = 0;
while (index < numbers.length) {
    System.out.println(numbers[index]);
    index++;
}
```

   - Iteriert über die **Länge des Arrays** mit der Schleifenbedingung `index < numbers.length`.

2. **Beispielausgabe**:

```
42
13
12
7
```

Code Visualization, Iteration mit Bedingung, 3-arrays

Wann exakt wird die Schleife abgebrochen?

Wichtige Eigenschaften von Arrays

- Die **Größe** eines Arrays kann mit `array.length` abgefragt werden (kein Methodenaufruf!).
- Arrays sind **Referenztypen**, was bedeutet, dass sie einen Verweis auf den Speicherort der Daten enthalten.
- Arrays haben eine **feste Größe**, die bei der Erstellung festgelegt wird und nicht mehr verändert werden kann.

Java: Zugriff auf einen ungültigen Array-Index

- In Java wird beim Zugriff auf einen ungültigen Index eine `ArrayIndexOutOfBoundsException` geworfen.
- Java prüft automatisch die Array-Grenzen zur Laufzeit und verhindert unzulässige Zugriffe.

```
public class Beispiel {
    public static void main(String[] args) {
        int[] array = {1, 2, 3};
        System.out.println(array[5]);  // Exception wird geworfen
    }
}
```

- **Ausgabe**:

```
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: Index 5
```

C: Zugriff auf einen ungültigen Array-Index

- In C gibt es keine automatische Array-Grenzprüfung.
- Der Zugriff auf einen ungültigen Index kann zu **undefiniertem Verhalten** führen, was bedeutet, dass möglicherweise Speicher manipuliert wird, der nicht zum Array gehört.

```c
#include <stdio.h>

int main() {
    int array[3] = {1, 2, 3};
    array[5] = 10;  // Unerlaubter Zugriff
    printf("array[5]: %d\n", array[5]);
}
```

- **Mögliches Ergebnis**:
    - Das Programm könnte fehlerfrei laufen und den Wert anzeigen, der an einer unerwarteten Speicherposition steht.

- **Java**:
  - Array-Grenzprüfungen sind zur Laufzeit integriert.
  - Ungültiger Zugriff auf einen Array-Index wirft eine `ArrayIndexOutOfBoundsException`.
- **C**:
  - Keine automatische Array-Grenzprüfung.
  - Ungültiger Zugriff führt zu **undefiniertem Verhalten**, was zu (Speicher)problemen führen kann.

Wiederholung: Arbeiten mit Strings

- **Erstellen und Ausgeben von Strings**:

```
String magicWord = "abracadabra";
System.out.println(magicWord);
```

- **Lesen eines Strings mit Scanner**:

```
Scanner reader = new Scanner(System.in);
String name = reader.nextLine();
System.out.println(name);
```

- **Verkettung von Strings**:

```
String phrase = "Hi " + name + " und bis später!";
System.out.println(phrase);
```

Vergleich von Strings

- **Strings dürfen nicht mit `==` verglichen werden.**

  Verwenden Sie stattdessen:

```
if (text.equals("marzipan")) {
    System.out.println("Die Texte sind gleich!");
}
```

- **Negation von String-Vergleichen**:

```
if (!text.equals("cake")) {
    System.out.println("Text ist nicht 'cake'!");
}
```

Aufteilen eines Strings mit `split()`

- **Aufteilen eines Strings anhand eines Trennzeichens**:

```
String text = "first second third";
String[] pieces = text.split(" ");
for (int i = 0; i < pieces.length; i++) {
    System.out.println(pieces[i]);
}
```

- **Beispielausgabe**:

```
first
second
third
```

String-Verarbeitung und Methoden

- **Strings nach Leerzeichen aufteilen und bestimmte Teile finden**:

```
String text = "Do you have a favorite flavor";
String[] parts = text.split(" ");
for (String part : parts) {
    if (part.contains("av")) {
        System.out.println(part);
    }
}
```

- **Beispielausgabe**:

```
have
favorite
flavor
```

Daten im festen Format verarbeiten

- **Verarbeitung von CSV-Daten**:

```
String input = "sebastian,2";
String[] parts = input.split(",");
System.out.println("Name: " + parts[0] + ", age: " + parts[1]);
```

- **Berechnung der Summe der Altersangaben**:

```
String[] inputs = { "John,25,Engineer", "Alice,30,Doctor", "Bob,22,Student"}

int sum = 0;

for (String input : inputs) {
    String[] parts = input.split(",");
    int age = Integer.valueOf(parts[1]);  // parts[1] is the second element,
    sum += age;
}
```

Nützliche String-Methoden

1. **Länge eines Strings**:

```
int length = word.length();
System.out.println("Length: " + length);
```

2. **Zeichen an einer bestimmten Position abrufen**:

```
char character = text.charAt(0);
System.out.println(character);
```

3. **Beispiel: Längster Name und Durchschnitt des Geburtsjahrs**:

```
if (name.length() > longestName.length()) {
    longestName = name;
}
```

String-Vergleich in Java

Vergleich mit `==` (Referenzvergleich):

- Mit `==` wird **nur** geprüft, ob beide Variablen auf dasselbe String-Objekt verweisen.
- **Beispiel**:

```
String a = "Hallo";
String b = "Hallo";
if (a == b) {
    System.out.println("Gleich!");
} else {
    System.out.println("Nicht gleich!");
}
```

- Ausgabe: `"Gleich!"` (hier, weil Java String-Literal-Pooling verwendet).

Vergleich mit `equals()` (Inhaltsvergleich):

- Mit `equals()` wird geprüft, ob die Inhalte der Strings gleich sind.
- **Beispiel**:

```
String a = new String("Hallo");
String b = new String("Hallo");
if (a.equals(b)) {
    System.out.println("Gleich!");
} else {
    System.out.println("Nicht gleich!");
}
```

- Ausgabe: `"Gleich!"` (auch bei unterschiedlichen Objekten).

Fazit:

- Verwende immer **`equals()`**, um den Inhalt von Strings zu vergleichen.
- **`==` vergleicht nur die Referenzen**, nicht die Inhalte.

Unterschiedliche Längen- und Größenabfragen in Java

Arrays: `.length`

- **Beschreibung**: Arrays in Java haben eine feste Größe, die mit dem Attribut `length` abgefragt wird.
- **Beispiel**:

```
int[] zahlen = {1, 2, 3};
int laenge = zahlen.length;
```

Strings: `.length()`

- **Beschreibung**: Strings sind Objekte in Java und ihre Länge wird durch die Methode `length()` abgefragt.
- **Beispiel**:

```
String text = "Hallo";
int laenge = text.length();
```

Container: `.size()`

- **Beschreibung**: Container wie `ArrayList` oder `HashSet` haben eine dynamische Größe, die durch die Methode `size()` abgefragt wird.
- **Beispiel**:

```
ArrayList<String> liste = new ArrayList<>();
int groesse = liste.size();
```

Zusammenfassung `size/length`

- **Arrays**:
  - Feste Größe, mit `length` als Attribut.
  - **Grund**: Arrays sind einfache Datenstrukturen, deren Länge sich nicht ändert, daher reicht ein Attribut.
- **Strings**:
  - Länge wird über `length()` als Methode abgefragt.
  - **Grund**: Strings sind Objekte. Die Methode `length()` bietet Flexibilität durch Kapselung, Erweiterbarkeit und Optimierung.
- **Container** (z.B. `ArrayList`):
  - Dynamische Größe, abgefragt mit `size()`.
  - **Grund**:
    - **`size()`** wird verwendet, weil Container (wie `ArrayList`, `HashSet` etc.) ihre Größe dynamisch ändern. `length()` würde andeuten, dass die Größe statisch ist, wie bei Arrays.
    - **Dynamische Datenstrukturen**: Container können Elemente hinzufügen und entfernen, sodass ihre Größe nicht fest ist. Daher wird `size()` als Methode genutzt, um die aktuelle Anzahl der Elemente zu berechnen oder abzurufen.
    - **Flexibilität**: Die Methode ermöglicht es, zusätzliche Logik für das Zählen der Elemente oder die Verwaltung interner Datenstrukturen zu kapseln.

Prinzipien bzgl. Fehler entdecken

- **Abstrakte Sicht: Wahrnehmungsbedingte Blindheit**
  - Programmierer tendieren dazu, Fehler in ihrem eigenen Code zu übersehen
  - In der Wissenschaft der Programmiersprachen wird dies als kognitives Problem betrachtet, das durch Fokussierung auf Details verstärkt wird
  - **Prinzip**: *Separation of Concerns* – Fokus auf modulare Strukturierung und Vermeidung von Überkomplexität
- **Prinzipien der Programmierung: Debugging**
  - Debugging ist ein essenzieller Teil des Softwareentwicklungsprozesses
  - Methoden wie *Print-Debugging* implementieren das Prinzip der **Transparenz** im Code
  - Erzeugung von *Nachvollziehbarkeit* durch das Einfügen von Konsolenausgaben zur Laufzeit

Prinzipien bzgl. Listen

- **Abstrakte Sicht: Datenstrukturen und Listen**
  - Listen gehören zu den grundlegenden linearen Datenstrukturen
  - Unterschied zwischen *dynamischen* (ArrayLists) und *statischen* (Arrays) Listenstrukturen
  - **Prinzip**: *Abstraktion* – Die konkrete Implementierung der (z.B. dynamischen) Liste wird hinter Methoden versteckt (z.B. `add`, `get`)
- **Prinzipien der Programmierung: Modularität (bei Listen)**
  - *Iteration* über Listen ist eine fundamentale Technik in der Programmierung
  - Verwendung von for-Schleifen zeigt das Prinzip der **Effizienz**: Statt manueller Wiederholung, wird die Iteration automatisiert
  - **Modularität**: Trennung der Datenspeicherung von der Verarbeitung der Daten

Prinzipien bzgl. Arrays

- **Abstrakte Sicht: Arrays als Referenztypen**
  - Arrays sind statische, referenzartige Datentypen, die effizient im Speicher allokiert werden
  - Arrays bieten den Vorteil von **Indexbasiertem Zugriff**: O(1) Komplexität für Lese-/Schreibzugriffe
  - **Prinzip**: *Effizienz* (computational) – Arrays bieten direkten Zugriff auf Speicherbereiche und minimieren Overhead im Vergleich zu dynamischen Strukturen
- **Prinzipien der Programmierung: Speicherverwaltung**
  - In Arrays wird **Speicherkontrolle** direkt von der Programmiererin/Programmierer übernommen (Größe muss vorab festgelegt werden)
  - **Determinismus**: Bei Arrays ist die Struktur und Kapazität zur Laufzeit festgelegt, was die Performance vorhersagbar macht

Nicht-Determinismus durch Multithreading

- Multithreading kann zu **nicht-deterministischem Verhalten** führen, wenn mehrere Threads gleichzeitig auf dieselben Daten zugreifen.
- Ohne **Synchronisation** kann das Ergebnis des Programms je nach Ausführungsreihenfolge der Threads variieren.
- Dies führt zu sogenannten **Race Conditions**, bei denen das Ergebnis nicht vorhersehbar ist.

Beispiel: Race Condition in Java

```java
public class Beispiel {
    private static int counter = 0;

    public static void main(String[] args) throws InterruptedException {
        Thread thread1 = new Thread(Beispiel::increment);
        Thread thread2 = new Thread(Beispiel::increment);

        thread1.start();
        thread2.start();

        thread1.join();
        thread2.join();

        System.out.println("Counter: " + counter);  // Ergebnis unvorhersehbar
    }

    private static void increment() {
        for (int i = 0; i < 10000; i++) {
            counter++;  // Nicht synchronisierter Zugriff
        }
```

Lösung: Synchronisierung des Zugriffs

- Um das Problem zu beheben, kann der Zugriff auf `counter` synchronisiert werden:

```
private static synchronized void increment() {
    for (int i = 0; i < 10000; i++) {
        counter++;  // Synchronisierter Zugriff
    }
}
```

- **Ergebnis**: Mit Synchronisation wird der Zugriff auf die Variable kontrolliert, und die Threads arbeiten geordnet.
- Das Verhalten wird nun **deterministisch**, da die Zugriffsreihenfolge festgelegt ist.

Vergleich: `Array` vs `ArrayList`

Eigenschaft	`Array`	`ArrayList`
Größe	Fixe Länge	Dynamische Größe
Zugriff auf Elemente	Direkt über `[]` (O(1))	Über `get()` (O(1))
Einfügen von Elementen	Nicht möglich	Über `add()` (dynamisch, O(1) am Ende)
Löschen von Elementen	Nicht möglich	Über `remove()` (O(n))
Primitive Datentypen	Ja	Nein (nur mit Wrapper-Klassen)
Speicherplatz	Fix, effizient	Flexibel, benötigt manchmal mehr Speicher für interne Arrays
Verwendung	Wenn fixe Größe ausreicht	Wenn dynamische Anpassung der Größe notwendig ist

Vergleich zwischen `ArrayList` und `LinkedList`

- Beide bieten dieselben Zugriffsmethoden:
  - `add()`, `get()`, `remove()`, `size()`
  - Unterschied liegt in der internen Implementierung

Eigenschaft	`ArrayList` (dynamisches Array)	`LinkedList` (verkettete Liste)
Zugriff über Index	Sehr schnell (O(1))	Langsam (O(n))
Einfügen/Entfernen am Ende	Schnell (O(1))	Schnell (O(1))
Einfügen/Entfernen in der Mitte	Langsam (O(n))	Schneller als `ArrayList` (O(1))
Speicherbedarf	Geringer, da kompakt im Array	Höher, da zusätzliche Speicherplätze für die Verweise benötigt werden
Verwendung	Wenn häufiger auf Elemente zugegriffen wird	Wenn häufiger Elemente eingefügt oder entfernt werden

Prinzipien bzgl.: Strings verwenden

- **Abstrakte Sicht: Strings und Textverarbeitung**
  - Strings sind in den meisten Sprachen als unveränderliche (immutable) Datentypen implementiert
  - Die Verarbeitung von Strings und die Definition von Musterb verwendet **Regular Expressions** (zukünftige Vorlesung), ein mächtiges Werkzeug der Textanalyse
  - **Prinzip**: *Kapselung* – String-Methoden wie `split`, `equals`, und `contains` verstecken die komplexe Stringmanipulation hinter einfachen Schnittstellen
- **Prinzipien der Programmierung: Modulare Verarbeitung**
  - Die Möglichkeit, Strings in kleinere Einheiten zu zerlegen, folgt dem Prinzip der **Modularität**
  - Vermeidung von *Nebenwirkungen* bei unveränderlichen Typen (z.B. Strings) trägt zur Stabilität und Vorhersagbarkeit des Codes bei

Prinzipien der Programmierung: Abstraktion und Modularität

- **Abstraktion**:
  - Datenstrukturen (Listen, Arrays) und Datentypen (Strings) werden abstrahiert, sodass die Interaktion mit ihnen vereinfacht wird
  - Wissenschaftliche Konzepte wie *ADTs* (Abstract Data Types) bilden die Grundlage für diese Abstraktion
- **Modularität**:
  - Programme sollten in kleine, wiederverwendbare Module unterteilt werden (Methoden, Datenstrukturen, Bibliotheken)
  - Fehlerentdeckung und Debugging profitieren von klarer Modularisierung: Bessere Lesbarkeit und Wartbarkeit
- **Effizienz und Sicherheit**:
  - Speicherverwaltung (bei Arrays) und fehlerfreies Handling von Referenzen (bei Listen) sind zentral für effiziente Programme
  - Prinzipien wie **Speicherkontrolle** und **Datensicherheit** (z.B. durch die Verwendung unveränderlicher Datentypen) helfen, Performance zu optimieren und Fehler zu vermeiden

Erweiterte Fehlererkennung und Debugging-Techniken

- **Exception Handling**:
  - Vermeidung unerwarteter Programmabbrüche durch **gezielte Fehlerbehandlung**.
  - Prinzip der **Fehler-Isolierung**: Unerwartete Fehler werden durch `try-catch` Blöcke isoliert.
  - Benutzerdefinierte Exceptions fördern **klare Fehlerkommunikation** im Code.
- **Unit Testing**:
  - Prinzip der **Modularität**: Jede Funktion wird isoliert getestet.
  - Effiziente **Automatisierung** der Fehlererkennung durch Unit-Test-Frameworks wie JUnit.

Erweiterte Datenstrukturen

- **HashMaps und Sets**:
  - **HashMap**: Ermöglicht schnelles **Suchen und Zuordnen** von Schlüssel-Wert-Paaren (Effizienz durch O(1) Zugriffszeit).
  - **Set**: Datenstruktur ohne Duplikate, ideal für **Mengenoperationen**.
  - Prinzip der **Effizienz**: Datenstrukturwahl basierend auf Zugriffs- und Suchzeiten.
- **Multidimensionale Arrays**:
  - Ermöglichen das Speichern und Verarbeiten von **mehrdimensionalen Daten** (z.B. Matrizen).
  - Prinzip der **Datenorganisation**: Komplexe Strukturen werden übersichtlich in Arrays abgebildet.

Algorithmen zur Datenverarbeitung

- **Such- und Sortieralgorithmen**:
  - **Binäre Suche**: Logarithmische Laufzeit O(log n), effizient für große sortierte Listen.
  - **Quicksort**: Divide-and-Conquer-Ansatz, mit einer durchschnittlichen Komplexität von O(n log n).
  - Prinzip der **Effizienzsteigerung**: Optimierung von Algorithmen für datenintensive Operationen.
- **Divide-and-Conquer-Prinzip**:
  - Rekursive Aufteilung eines Problems in kleinere Unterprobleme zur leichteren Lösung.
  - Quicksort als Beispiel für den Einsatz dieses Prinzips.

Fortgeschrittene String-Verarbeitung

- **Reguläre Ausdrücke**:
  - Effizientes Werkzeug zur **Mustererkennung** in Texten.
  - Prinzip der **Datenmanipulation**: Durch `Pattern` und `Matcher` wird Text schnell durchsucht oder modifiziert.
- **StringBuilder**:
  - Vermeidet das ineffiziente Erstellen neuer String-Objekte bei vielen **String-Operationen**.
  - Prinzip der **Speichereffizienz**: Reduzierung des Speicherverbrauchs bei intensiver String-Bearbeitung.

Zusammenfassende Prinzipien der Programmierung

- **Abstraktion**:
  - Datenstrukturen und Algorithmen bieten eine höhere Abstraktionsebene, auf der Komplexität reduziert wird.
- **Modularität**:
  - Unit Tests und exception handling trennen **Fehlerbehandlung** und **Datenmanipulation**, was zu einem strukturierten Code führt.
- **Effizienz**:
  - Optimierte Datenstrukturen (HashMap, Set) und Algorithmen (Binäre Suche, Quicksort) zielen auf **Performanz** bei großen Datenmengen ab.