Anatomie einer Java-Funktion

Eine Funktion in Java ist ein Codeblock, der eine spezifische Aufgabe ausführt.

Wie man eine Funktion in Java schreibt

Funktionen in Java können, je nach Zweck, entweder einen Wert zurückgeben oder keinen Wert zurückgeben.

Funktion mit Rückgabewert

Eine Funktion, die einen Wert zurückgibt, hat folgendes Format:

```
RückgabewertTyp funktionsName() {
    // Code zur Durchführung der Aufgabe
    return wert;
}
```

- **RückgabewertTyp**: Gibt den Typ des Wertes an, den die Funktion zurückgeben wird (z.B. `int`, `double`, `String`, `Date`, etc.).
- **funktionsName**: Der Name, den Sie der Funktion zuweisen.

Funktion ohne Rückgabewert

In Java wird eine Funktion, die keinen Wert zurückgibt, mit dem Schlüsselwort `void` deklariert:

```
void funktionsName() {
    // Code zur Durchführung der Aufgabe
}
```

- **void** ist ein reserviertes Schlüsselwort in Java und zeigt an, dass die Funktion keinen Wert zurückgibt.

Wichtige Schritte zur Definition einer Funktion

1. **Name**: Wählen Sie einen Namen für die Funktion, der ihren Zweck beschreibt.
2. **Parameter**: Verwenden Sie runde Klammern `()`, um Parameter zu übergeben (Parameter sind optional).
3. **Klammern**: Nutzen Sie geschweifte Klammern `{}`, um den Codeblock zu definieren. Die öffnende Klammer `{` sollte in der gleichen Zeile wie der Funktionsname stehen.

Beispiel:

```
void sendeEmail() {
    // Code zum Versenden einer E-Mail
}
```

Die `main()`-Funktion

Jedes Java-Programm muss mindestens eine Funktion enthalten, und diese Funktion heißt in der Regel `main()`. Die `main()`-Funktion ist der Einstiegspunkt des Programms:

- Wenn ein Java-Programm ausgeführt wird, wird die `main()`-Funktion aufgerufen, und der darin enthaltene Code wird ausgeführt.
- Die `main()`-Funktion existiert nicht unabhängig; sie gehört immer zu einer **Klasse**.

Beispiel einer `main()`-Funktion:

```
public static void main(String[] args) {
    // Code, der beim Start des Programms ausgeführt wird
}
```

Die `class` als Container für verwandte Funktionen

Eine Klasse in Java dient dazu, verwandte Funktionen zu organisieren, ähnlich wie in einem Supermarkt, wo verwandte Produkte gruppiert sind. Jedes Java-Programm sollte mindestens eine Klasse enthalten, die die `main()`-Funktion enthält. Funktionen innerhalb von Klassen werden als Methoden bezeichnet.

Um eine Klasse zu erstellen, verwenden Sie das Schlüsselwort `class`, gefolgt von einem aussagekräftigen Namen. Definieren Sie die Methoden innerhalb der Klasse.

Beispiel:

```
class Main {
    public static void main(String[] args) {
        // Code, der beim Start des Programms ausgeführt wird
    }
}
```

Namenskonventionen in Java

In Java:

- **Klassen** folgen der PascalNamenskonvention, bei der der erste Buchstabe jedes Wortes großgeschrieben wird.
- **Methoden** nutzen die camelNamenskonvention, bei der der erste Buchstabe jedes Wortes großgeschrieben wird, außer beim ersten Wort.

Variablen in Java

1. Einführung in Variablen in Java

In Java sind Variablen grundlegende Bausteine, die zur Speicherung von Daten dienen, die im gesamten Programm abgerufen und manipuliert werden können. Variablen in Java sind Container, die Datenwerte enthalten. Jede Variable hat einen spezifischen Datentyp, der definiert, welche Art von Wert sie speichern kann. Diese Datentypen reichen von primitiven Typen wie Ganzzahlen, Gleitkommazahlen, Zeichen und Booleans bis hin zu komplexen Objekten, die aus Klassen erstellt werden. Java erzwingt eine strikte Typprüfung, was bedeutet, dass jede Variable vor der Verwendung mit einem spezifischen Datentyp deklariert werden muss. Dies stellt die Datenintegrität sicher und reduziert Laufzeitfehler.

Wichtige Punkte:

- **Deklaration**: Jede Variable muss vor der Verwendung mit einem Datentyp deklariert werden.
- **Initialisierung**: Variablen können bei der Deklaration initialisiert werden.
- **Gültigkeitsbereich (Scope)**: Variablen in Java haben einen Gültigkeitsbereich, der bestimmt, wo im Code auf sie zugegriffen werden kann.
- **Lebensdauer**: Die Lebensdauer einer Variable wird durch ihre Deklaration bestimmt und besteht nur innerhalb dieses Bereichs.

2. Beispielcode für Variablen in Java

Hier ist ein Beispiel in Java, das zeigt, wie verschiedene Variablentypen deklariert und initialisiert werden.

```
public class VariableExamples {
    public static void main(String[] args) {
        // Numerische Variablen
        int age = 25; // Integer-Typ
        double salary = 45000.75; // Double-Typ

        // Textvariable
        String name = "Alice"; // String-Typ für Text

        // Boolean-Variable
        boolean isStudent = true; // Boolean-Typ

        // Variablen ausgeben
        System.out.println("Name: " + name);
        System.out.println("Alter: " + age);
        System.out.println("Gehalt: " + salary);
        System.out.println("Ist Student: " + isStudent);
    }
}
```

3. Datentypen in Java (JDK 23)

Die folgende Tabelle gibt einen umfassenden Überblick über die in JDK 23 verfügbaren Datentypen, kategorisiert nach Art. Jeder Datentyp hat spezifische Grenzen und beansprucht eine bestimmte Menge an Speicher, wie in der Tabelle dargestellt.

Kategorie	Datentyp	Gültige Grenzen	Speichergröße	Beispieldeklaration in Java
Numerisch	`byte`	-128 bis 127	1 Byte	`byte b = 100;`
	`short`	-32,768 bis 32,767	2 Bytes	`short s = 20000;`
	`int`	-2,147,483,648 bis 2,147,483,647	4 Bytes	`int i = 500000;`
	`long`	-9,223,372,036,854,775,808 bis 9,223,372,036,854,775,807	8 Bytes	`long l = 5000000000L;`
	`float`	Ungefähr ±3.4e−038 bis ±3.4e+038	4 Bytes	`float f = 3.14f;`
	`double`	Ungefähr ±1.7e−308 bis ±1.7e+308	8 Bytes	`double d = 3.14159;`
Text	`char`	Einzelnes Unicode-Zeichen	2 Bytes	`char c = 'A';`
	`String`	Zeichenfolge	Variiert (Objekt)	`String s = "Hello World";`
Boolean	`boolean`	`true` oder `false`	1 Bit	`boolean flag = true;`

4. Ungültige Variablenzuweisungen

In Java hat jeder Datentyp spezifische Anforderungen und Einschränkungen. Hier sind einige häufige ungültige Variablenzuweisungen, die in Java Fehler verursachen würden:

- **Ungültige `float`-Zuweisung (fehlendes 'f' Suffix)**:

  ```
  float f = -35.0; // Fehler: Float-Literal erfordert ein 'f'-Suffix für den float-Typ
  ```

  Korrektur:

  ```
  float f = -35.0f;
  ```

- **Außerhalb des Bereichs `byte`-Zuweisung**:

  ```
  byte b = 150; // Fehler: Wert überschreitet den Bereich von byte (-128 bis 127)
  ```

Korrektur:

```
byte b = 127;
```

- **Außerhalb des Bereichs `short`-Zuweisung**:

```
short s = 40000; // Fehler: Wert überschreitet den Bereich von short (-32,768 bis 32,767)
```

Korrektur:

```
short s = 32767;
```

- **Ungültige `char`-Zuweisung (mehrere Zeichen)**:

```
char c = 'AB'; // Fehler: char kann nur ein einzelnes Zeichen speichern
```

Korrektur:

```
char c = 'A';
```

- **Ungültige `boolean`-Zuweisung (numerischer Wert)**:

```
boolean flag = 1; // Fehler: boolean kann nur true oder false sein
```

Korrektur:

```
boolean flag = true;
```

5. Übungen mit verschiedenen Variablentypen

1. **Übung 1**: Deklariere eine `int`-Variable namens `year` und setze sie auf das aktuelle Jahr. Gib den Wert der Variable aus.
2. **Übung 2**: Deklariere eine `double`-Variable namens `temperature` und setze sie auf einen Wert mit einer Dezimalstelle. Gib die Temperatur aus.

3. **Übung 3**: Erstelle eine `String`-Variable namens `city` und setze sie auf den Namen deiner Heimatstadt. Zeige den Stadtnamen an.

4. **Übung 4**: Definiere eine `boolean`-Variable namens `isAvailable` und setze sie auf `false`. Gib den Verfügbarkeitsstatus aus.

5. **Übung 5**: Deklariere eine `char`-Variable namens `grade` und setze sie auf einen Buchstaben zwischen 'A' und 'F'. Gib die Note aus.

String-Methoden in Java

In Java bieten Strings eine Vielzahl an Methoden, die es Entwicklern ermöglichen, Textdaten flexibel und effizient zu verarbeiten und zu manipulieren. Die folgenden Methoden erlauben das Abrufen von Informationen über den String, das Durchführen von Transformationen, das Suchen von Zeichen oder Teilstrings und viele andere nützliche Operationen. Jede Methode ist auf einen spezifischen Anwendungsfall zugeschnitten und hilft dabei, Strings auf unterschiedliche Weise zu handhaben.

Die Tabelle unten beschreibt die gängigsten Methoden für Strings in Java, inklusive einer Erklärung, der Syntax und einem Beispiel zur Veranschaulichung.

Methode	Erklärung	Syntax	Beispiel
`length()`	Gibt die Anzahl der Zeichen im String zurück.	`str.length()`	`"Hallo".length()` → 5
`toUpperCase()` / `toLowerCase()`	Konvertiert den gesamten String in Groß- oder Kleinbuchstaben.	`str.toUpperCase()`, `str.toLowerCase()`	`"Hallo".toUpperCase()` → `"HALLO"`
`substring(start, end)`	Gibt einen Teil des Strings zurück, von `start` bis `end` (Endindex exklusiv).	`str.substring(start, end)`	`"Hallo".substring(1, 4)` → `"all"`
`charAt(index)`	Liefert das Zeichen an der angegebenen Position.	`str.charAt(index)`	`"Hallo".charAt(1)` → `"a"`
`equals(other)` / `equalsIgnoreCase(other)`	Vergleicht zwei Strings; `equalsIgnoreCase` ignoriert die Groß-/Kleinschreibung.	`str.equals(other)`, `str.equalsIgnoreCase(other)`	`"Hallo".equals("hallo")` → `false`, `"Hallo".equalsIgnoreCase("hallo")` → `true`
`contains(substring)`	Überprüft, ob der String die angegebene Zeichenkette enthält.	`str.contains(substring)`	`"Hallo".contains("all")` → `true`
`startsWith(prefix)` / `endsWith(suffix)`	Überprüft, ob der String mit einer bestimmten Zeichenkette beginnt oder endet.	`str.startsWith(prefix)`, `str.endsWith(suffix)`	`"Hallo".startsWith("Ha")` → `true`, `"Hallo".endsWith("lo")` → `true`
`trim()`	Entfernt führende und nachfolgende Leerzeichen.	`str.trim()`	`" Hallo ".trim()` → `"Hallo"`
`replace(old, new)`	Ersetzt alle Vorkommen eines bestimmten Zeichens oder einer Zeichenkette im String.	`str.replace(old, new)`	`"Hallo".replace("l", "x")` → `"Haxxo"`
`indexOf(char)` / `lastIndexOf(char)`	Liefert die erste bzw. letzte Position des angegebenen Zeichens im String.	`str.indexOf(char)`, `str.lastIndexOf(char)`	`"Hallo".indexOf("l")` → 2, `"Hallo".lastIndexOf("l")` → 3

Methode	Erklärung	Syntax	Beispiel
`String.format()`	Erstellt formatierte Strings ähnlich wie `printf`.	`String.format(format, args...)`	`String.format("Name: %s", "Anna")` → `"Name: Anna"`
`repeat(count)`	Wiederholt den String die angegebene Anzahl von Malen.	`str.repeat(count)`	`"Ha".repeat(3)` → `"HaHaHa"`
`isEmpty()` / `isBlank()`	Überprüft, ob der String leer ist bzw. nur aus Leerzeichen besteht.	`str.isEmpty()`, `str.isBlank()`	`"".isEmpty()` → `true`, `" ".isBlank()` → `true`
`compareTo(other)`	Vergleicht zwei Strings lexikographisch (alphabetisch).	`str.compareTo(other)`	`"Hallo".compareTo("Hallo")` → `0`

Diese Methoden ermöglichen vielseitige Operationen und Manipulationen mit Strings in Java.

String vs. StringBuilder vs. StringBuffer

Java bietet drei Hauptoptionen zur Verarbeitung von Zeichenketten (`String`, `StringBuilder`, und `StringBuffer`), wobei jede Klasse ihre eigenen Vor- und Nachteile hat.

Unterschiede zwischen `String`, `StringBuilder` und `StringBuffer`

Typ	Veränderlichkeit	Threadsicherheit	Verwendung
`String`	Unveränderlich	Ja	Für unveränderbare Texte. Jede Änderung erzeugt eine neue Instanz, was bei häufigen Änderungen ineffizient sein kann.
`StringBuilder`	Veränderlich	Nein	Schneller als `StringBuffer` und ideal für die Verwendung in Single-Thread-Umgebungen, wenn viele Änderungen erforderlich sind.
`StringBuffer`	Veränderlich	Ja	Threadsicher, wird daher bei Multi-Thread-Umgebungen verwendet, in denen Strings häufig geändert werden müssen.

Vorteile und Methoden von `StringBuffer`

`StringBuffer` ist besonders nützlich, wenn in Multi-Thread-Umgebungen viele Änderungen am Text erforderlich sind, da die Klasse threadsicher ist. Das bedeutet, dass mehrere Threads gleichzeitig sicher auf einen `StringBuffer` zugreifen und ihn ändern können. Allerdings ist `StringBuffer` dadurch etwas langsamer als `StringBuilder`.

Wichtige Methoden von `StringBuffer`

Methode	Erklärung	Beispiel
`append(str)`	Fügt eine Zeichenkette am Ende des `StringBuffer`-Objekts an.	`sb.append(" Welt!")`
`insert(index, str)`	Fügt eine Zeichenkette an einer bestimmten Position ein.	`sb.insert(5, "Java ")`
`replace(start, end, str)`	Ersetzt den Teilstring zwischen `start` und `end` mit `str`.	`sb.replace(0, 4, "Hallo")`
`delete(start, end)`	Löscht die Zeichen von `start` bis `end` im `StringBuffer`.	`sb.delete(5, 10)`
`reverse()`	Kehrt die Zeichenkette im `StringBuffer` um.	`sb.reverse()`
`toString()`	Konvertiert den Inhalt des `StringBuffer` in einen `String`.	`sb.toString()`

Beispiel für `StringBuffer`

```java
public class StringBufferExample {
    public static void main(String[] args) {
        StringBuffer sb = new StringBuffer("Hallo");

        // Anhängen von Text
        sb.append(" Welt!");
        System.out.println("Nach append: " + sb); // "Hallo Welt!"

        // Einfügen von Text
        sb.insert(6, "Java ");
        System.out.println("Nach insert: " + sb); // "Hallo Java Welt!"

        // Ersetzen eines Teilstrings
        sb.replace(6, 10, "Programmierer");
        System.out.println("Nach replace: " + sb); // "Hallo Programmierer Welt!"

        // Löschen eines Teilstrings
        sb.delete(5, 18);
        System.out.println("Nach delete: " + sb); // "Hallo Welt!"

        // Umkehren der Zeichenkette
        sb.reverse();
        System.out.println("Nach reverse: " + sb); // "!tleW ollaH"

        // Zurück in die Originalreihenfolge
        sb.reverse();
        System.out.println("Zurück zur Originalreihenfolge: " + sb); // "Hallo Welt!"
    }
}
```

Zusammenfassung der Vorteile von `StringBuffer`

- **Threadsicher**: `StringBuffer` kann sicher von mehreren Threads gleichzeitig geändert werden, was ihn zur besten Wahl für Multi-Thread-Umgebungen macht.
- **Effizient für häufige Änderungen**: Im Gegensatz zu `String`, das bei jeder Änderung ein neues Objekt erstellt, bietet `StringBuffer` die Möglichkeit, denselben Speicherplatz für Änderungen zu verwenden.
- **Umfangreiche Methoden**: Ähnlich wie `StringBuilder` bietet `StringBuffer` Methoden wie `append`, `insert`, `replace`, `delete`, und `reverse`, was ihn sehr flexibel macht.

Fazit

`StringBuffer` ist die beste Wahl, wenn Sie in einer Multi-Thread-Umgebung arbeiten und Zeichenketten häufig ändern müssen. Andernfalls ist `StringBuilder` die effizientere Wahl für Single-Thread-Umgebungen, da er schneller ist und weniger Overhead verursacht. Beide Klassen bieten leistungsstarke Methoden zur String-Manipulation und sind ideal für Szenarien, in denen die Unveränderlichkeit von `String` zu Performanceproblemen führen könnte.

Java Namenskonventionen

Java folgt bestimmten Konventionen, um den Code einheitlich und gut lesbar zu gestalten. Hier sind die wichtigsten Konventionen für Klassen, Methoden und Variablen.

Klassenkonventionen

Klassennamen sollten im **PascalCase**-Stil geschrieben werden. Das bedeutet, dass jedes Wort mit einem Großbuchstaben beginnt und es keine Unterstriche gibt. Der Name der Klasse sollte beschreibend für die Funktion oder den Zweck der Klasse sein.

Beispiele:

```
class Employee { }
class CustomerAccount { }
class ProductCatalog { }
```

Methoden-Konventionen

Methodennamen sollten im **camelCase**-Stil geschrieben werden, ähnlich wie Variablennamen, jedoch soll der Name eine Aktion oder Aufgabe beschreiben, da Methoden in der Regel etwas "tun". Jedes Wort nach dem ersten beginnt mit einem Großbuchstaben.

Beispiele:

```
void calculateSalary() { }
String getCustomerName() { }
boolean isAvailable() { }
```

Variablenkonventionen

Variablennamen sollten ebenfalls im **camelCase**-Stil geschrieben werden. Das erste Wort beginnt mit einem Kleinbuchstaben, und jedes darauffolgende Wort beginnt mit einem Großbuchstaben. Variablennamen sollten beschreibend für den Inhalt oder Zweck der Variable sein.

Beispiele:

```
int employeeAge;
String firstName;
double accountBalance;
```

Konstante Variablen

Für Variablen, die **konstant** sind (also ihren Wert nach der Initialisierung nicht ändern), verwendet man in Java das Schlüsselwort `final`. Der Variablenname wird vollständig in Großbuchstaben geschrieben, und einzelne Wörter werden durch Unterstriche getrennt.

Beispiel:

```
final int MAX_USERS = 100;
final double PI = 3.14159;
final String DEFAULT_COUNTRY = "Germany";
```

Allgemeine Namenskonventionen

- **Klassen** folgen der **PascalCase**-Konvention.
- **Methoden** und **Variablen** folgen der **camelCase**-Konvention.
- **Konstanten** (mit `final` deklariert) werden in **GROßBUCHSTABEN** geschrieben.
- Vermeiden Sie Sonderzeichen und Abkürzungen, die schwer verständlich sind.
- Verwenden Sie keine reservierten Schlüsselwörter als Namen.
- Halten Sie Namen **kurz und prägnant**, ohne die Bedeutung zu verlieren.

Diese Konventionen helfen dabei, Java-Code lesbar, wartbar und konsistent zu halten.

Kontrollstrukturen in Java

In Java werden Kontrollstrukturen verwendet, um den Fluss eines Programms basierend auf bestimmten Bedingungen oder Wiederholungen zu steuern. Zu den wichtigsten Kontrollstrukturen zählen die bedingten Anweisungen (`if`, `else`, `switch`) und Schleifen (`for`, `while`, `do-while`). Diese ermöglichen die Ausführung von Code abhängig von Bedingungen oder wiederholt bis eine Bedingung erfüllt ist.

Bedingte Anweisungen

1. `if` und `else`-Anweisung

Die `if`-Anweisung führt Code nur aus, wenn eine Bedingung wahr (`true`) ist. Mit `else` kann man Alternativen angeben, falls die Bedingung falsch (`false`) ist.

```
int age = 20;

if (age >= 18) {
    System.out.println("Du bist volljährig.");
} else {
    System.out.println("Du bist minderjährig.");
}
```

2. `else if`-Anweisung

Die `else if`-Anweisung erlaubt es, mehrere Bedingungen nacheinander zu prüfen, bevor ein `else`-Block als Fallback ausgeführt wird.

```
int score = 85;

if (score >= 90) {
    System.out.println("Note: A");
} else if (score >= 80) {
    System.out.println("Note: B");
} else if (score >= 70) {
    System.out.println("Note: C");
} else {
    System.out.println("Nicht bestanden");
}
```

3. `switch`-Anweisung

Die `switch`-Anweisung ist nützlich, wenn man den Wert einer Variablen gegen mehrere mögliche Fälle (`case`) prüfen möchte.

```java
int day = 3;
String dayName;

switch (day) {
    case 1:
        dayName = "Montag";
        break;
    case 2:
        dayName = "Dienstag";
        break;
    case 3:
        dayName = "Mittwoch";
        break;
    case 4:
        dayName = "Donnerstag";
        break;
    case 5:
        dayName = "Freitag";
        break;
    case 6:
        dayName = "Samstag";
        break;
    case 7:
        dayName = "Sonntag";
        break;
    default:
        dayName = "Ungültiger Tag";
        break;
}

System.out.println("Heute ist " + dayName);
```

Schleifen

Schleifen sind Strukturen, die es ermöglichen, Anweisungen wiederholt auszuführen, solange eine bestimmte Bedingung erfüllt ist.

1. `for`-Schleife

Eine `for`-Schleife ist ideal für bekannte Wiederholungen, bei denen die Anzahl der Iterationen im Voraus festgelegt ist.

```java
for (int i = 0; i < 5; i++) {
    System.out.println("Durchlauf Nummer: " + i);
}
```

2. `while`-Schleife

Die `while`-Schleife wiederholt eine Anweisung, solange eine Bedingung wahr (`true`) ist. Diese Schleife eignet sich, wenn die Anzahl der Durchläufe nicht im Voraus bekannt ist.

```
int count = 0;

while (count < 5) {
    System.out.println("Durchlauf Nummer: " + count);
    count++;
}
```

3. `do-while`-Schleife

Die `do-while`-Schleife ähnelt der `while`-Schleife, jedoch wird der Codeblock mindestens einmal ausgeführt, da die Bedingung erst am Ende geprüft wird.

```
int count = 0;

do {
    System.out.println("Durchlauf Nummer: " + count);
    count++;
} while (count < 5);
```

Erweiterte Kontrollstrukturen

1. `break`-Anweisung

`break` wird verwendet, um eine Schleife oder einen `switch`-Block vorzeitig zu beenden.

```
for (int i = 0; i < 10; i++) {
    if (i == 5) {
        break; // Schleife wird abgebrochen, wenn i 5 erreicht
    }
    System.out.println("i: " + i);
}
```

2. `continue`-Anweisung

`continue` überspringt die aktuelle Iteration und fährt mit der nächsten Schleifeniteration fort.

```
for (int i = 0; i < 10; i++) {
    if (i % 2 == 0) {
        continue; // Gerade Zahlen werden übersprungen
    }
```

```
        System.out.println("Ungerade Zahl: " + i);
    }
```

3. Verschachtelte Schleifen

Schleifen können auch ineinander verschachtelt werden, um mehrdimensionale Strukturen (wie Tabellen) zu durchlaufen.

```
for (int i = 1; i <= 3; i++) {
    for (int j = 1; j <= 3; j++) {
        System.out.print(i * j + " ");
    }
    System.out.println(); // Zeilenumbruch nach jeder inneren Schleife
}
```

Kontrollstrukturen Zusammenfassung

Kontrollstruktur	Erklärung	Beispiel
`if`	Führt einen Codeblock aus, wenn eine Bedingung wahr ist.	`if (age >= 18) { System.out.println("Volljährig"); }`
`else if`	Prüft eine zusätzliche Bedingung, wenn die vorherige `if`-Bedingung falsch ist.	`else if (score >= 80) { System.out.println("Note: B"); }`
`else`	Fallback-Codeblock, der ausgeführt wird, wenn keine vorherige Bedingung erfüllt ist.	`else { System.out.println("Nicht bestanden"); }`
`switch`	Überprüft eine Variable gegen mehrere mögliche Werte (`case`-Blöcke).	`switch (day) { case 1: ... break; }`
`for`	Führt eine festgelegte Anzahl von Wiederholungen aus, typischerweise bei bekannten Durchläufen.	`for (int i = 0; i < 5; i++) { ... }`
`while`	Wiederholt eine Anweisung, solange eine Bedingung wahr ist.	`while (count < 5) { ... }`
`do-while`	Ähnlich wie `while`, führt aber den Codeblock mindestens einmal aus.	`do { ... } while (count < 5);`
`break`	Beendet eine Schleife oder einen `switch`-Block vorzeitig.	`if (i == 5) { break; }`
`continue`	Überspringt die aktuelle Schleifeniteration und fährt mit der nächsten fort.	`if (i % 2 == 0) { continue; }`

Kontrollstruktur	Erklärung	Beispiel
Verschachtelte Schleifen	Führt Schleifen in einer weiteren Schleife aus, z. B. für tabellenartige Strukturen.	`for (int i = 1; i <= 3; i++) { for (int j = 1; j <= 3; j++) { ... } }`

Diese Kontrollstrukturen sind die Basis für die Steuerung des Programmflusses in Java und ermöglichen das Erstellen flexibler und leistungsfähiger Anwendungen.

Eingabemöglichkeiten und Datentypenkonvertierungen in Java

In Java bietet die `Scanner`-Klasse eine einfache Möglichkeit, Benutzereingaben von der Konsole zu lesen. Unterschiedliche Methoden der `Scanner`-Klasse erlauben das Lesen von verschiedenen Datentypen wie `int`, `double`, `String`, `boolean`, etc. Außerdem gibt es Methoden zur Konvertierung von Eingaben in unterschiedliche Datentypen, die die `Wrapper`-Klassen (wie `Integer`, `Double`) bereitstellen.

---

1. Eingabe verschiedener Datentypen

Ganzzahl (`int`)

Für die Eingabe einer ganzen Zahl verwenden wir die Methode `nextInt()`.

```
import java.util.Scanner;

Scanner scanner = new Scanner(System.in);
System.out.print("Geben Sie eine ganze Zahl ein: ");
int intWert = scanner.nextInt();
System.out.println("Eingegebener Integer: " + intWert);
```

Kommazahl (`double`)

Für die Eingabe einer Kommazahl verwenden wir `nextDouble()`. In Java wird eine Komma(``,`) für Dezimalwerte verwendet.

```
System.out.print("Geben Sie eine Kommazahl ein: ");
double doubleWert = scanner.nextDouble();
System.out.println("Eingegebener Double: " + doubleWert);
```

Einzelnes Zeichen (`char`)

Um ein einzelnes Zeichen einzulesen, lesen wir zuerst einen `String` mit `next()` und greifen dann auf das erste Zeichen mit `charAt(0)` zu.

```
System.out.print("Geben Sie einen Buchstaben ein: ");
char charWert = scanner.next().charAt(0);
System.out.println("Eingegebener Char: " + charWert);
```

Wahrheitswert (`boolean`)

Für die Eingabe eines `boolean`-Wertes verwenden wir die Methode `nextBoolean()`. Erwartet wird `true` oder `false`.

```
System.out.print("Geben Sie true oder false ein: ");
boolean boolWert = scanner.nextBoolean();
System.out.println("Eingegebener Boolean: " + boolWert);
```

Text (`String`)

Für die Eingabe eines Textes verwenden wir `nextLine()`.

```
System.out.print("Geben Sie einen Text ein: ");
String text = scanner.nextLine();
System.out.println("Eingegebener Text: " + text);
```

---

2. Konvertierung zwischen Datentypen

Konvertierung von `String` zu `int`

Um eine Zahl im `String`-Format in einen `int` zu konvertieren, verwenden wir `Integer.parseInt()`.

```
System.out.print("Geben Sie eine Zahl als Text ein: ");
String intAlsText = scanner.nextLine();
int konvertierterInt = Integer.parseInt(intAlsText);
System.out.println("Konvertierter Integer: " + konvertierterInt);
```

Konvertierung von `String` zu `double`

Für die Konvertierung eines `String` in `double` wird `Double.parseDouble()` verwendet.

```
System.out.print("Geben Sie eine Kommazahl als Text ein: ");
String doubleAlsText = scanner.nextLine();
double konvertierterDouble = Double.parseDouble(doubleAlsText);
System.out.println("Konvertierter Double: " + konvertierterDouble);
```

Konvertierung von `int` zu `String`

Um eine ganze Zahl (`int`) in einen `String` zu konvertieren, kann man `Integer.toString()` verwenden.

```
System.out.print("Geben Sie eine Zahl ein: ");
int zahl = scanner.nextInt();
```

```
String zahlAlsText = Integer.toString(zahl);
System.out.println("Konvertierte Zahl als Text: " + zahlAlsText);
```

Konvertierung von `double` zu `String`

Um eine Kommazahl (`double`) in einen `String` zu konvertieren, kann man `Double.toString()` verwenden.

```
System.out.print("Geben Sie eine Kommazahl ein: ");
double kommazahl = scanner.nextDouble();
String kommazahlAlsText = Double.toString(kommazahl);
System.out.println("Konvertierte Kommazahl als Text: " + kommazahlAlsText);
```

Konvertierung von `String` zu `boolean`

Für die Konvertierung eines `String` zu `boolean` wird `Boolean.parseBoolean()` verwendet.

```
System.out.print("Geben Sie true oder false als Text ein: ");
String boolAlsText = scanner.next();
boolean konvertierterBoolean = Boolean.parseBoolean(boolAlsText);
System.out.println("Konvertierter Boolean: " + konvertierterBoolean);
```

---

3. Zusammenfassende Tabelle der Eingaben und Konvertierungen

Aktion	**Datentyp**	**Beispielcode**
Eingabe einer ganzen Zahl	`int`	`int intWert = scanner.nextInt();`
Eingabe einer Kommazahl	`double`	`double doubleWert = scanner.nextDouble();`
Eingabe eines einzelnen Zeichens	`char`	`char charWert = scanner.next().charAt(0);`
Eingabe eines Wahrheitswerts	`boolean`	`boolean boolWert = scanner.nextBoolean();`
Eingabe eines Textes	`String`	`String text = scanner.nextLine();`
Konvertierung `String` zu `int`	`int`	`int konvertierterInt = Integer.parseInt(scanner.nextLine());`
Konvertierung `String` zu `double`	`double`	`double konvertierterDouble = Double.parseDouble(scanner.nextLine());`
Konvertierung `int` zu `String`	`String`	`String zahlAlsText = Integer.toString(scanner.nextInt());`
Konvertierung `double` zu `String`	`String`	`String kommazahlAlsText = Double.toString(scanner.nextDouble());`

Aktion	Datentyp	Beispielcode
Konvertierung `String` zu `boolean`	`boolean`	`boolean konvertierterBoolean = Boolean.parseBoolean(scanner.next());`

Mit dieser Übersicht und den Beispielcodes wird der Umgang mit den verschiedenen Eingabetypen und die Konvertierung von `String` zu anderen Datentypen sowie die Rückkonvertierung von Zahlen und `boolean` zu `String` verdeutlicht. Diese Kenntnisse sind besonders wichtig, um Eingaben des Benutzers korrekt zu verarbeiten und weiterzuverarbeiten.

Arrays in Java

Einführung

Ein **Array** ist eine Datenstruktur in Java, die es ermöglicht, eine Sammlung von Elementen desselben Datentyps an einer einzigen Stelle im Speicher zu speichern. Arrays sind besonders nützlich, wenn Sie mehrere Werte speichern und verwalten müssen, ohne dafür viele einzelne Variablen deklarieren zu müssen. Ein Array hat eine feste Größe, die bei der Initialisierung festgelegt wird und nicht mehr verändert werden kann.

---

1. Deklaration und Initialisierung von Arrays

Arrays müssen zuerst deklariert und initialisiert werden, bevor sie verwendet werden können. In Java gibt es verschiedene Möglichkeiten, Arrays zu erstellen und mit Werten zu füllen.

Deklaration und Initialisierung

Hier einige Möglichkeiten, Arrays zu deklarieren und zu initialisieren:

Aktion	Syntax	Beispiel
Deklaration	`dataType[] arrayName;`	`int[] numbers;`
Initialisierung	`arrayName = new dataType[size];`	`numbers = new int[5];`
Deklaration und Initialisierung	`dataType[] arrayName = new dataType[size];`	`int[] numbers = new int[5];`
Sofortige Initialisierung mit Werten	`dataType[] arrayName = {value1, value2, ...};`	`int[] numbers = {1, 2, 3};`

Beispiel: Deklaration und Initialisierung eines Arrays

```
int[] intArray = new int[10]; // Deklariert ein int-Array mit 10 Elementen
int[] intArray2 = {1, 2, 3, 4, 5}; // Deklariert und initialisiert ein Array mit Werten
```

---

2. Zugriff auf Array-Elemente

Arrays sind null-basiert, das heißt, das erste Element befindet sich an Index `0`. Wir können auf jedes Element eines Arrays zugreifen, indem wir seinen Index verwenden.

Beispiel: Zugriff und Modifikation von Array-Elementen

```
intArray[0] = 22; // Setzt das erste Element auf 22
System.out.println(intArray[0]); // Gibt das erste Element aus: 22
```

3. Array-Befüllung mit Schleifen

Häufig müssen Arrays in Schleifen gefüllt oder bearbeitet werden. Dies kann mit einer `for`- oder `foreach`-Schleife erfolgen.

Beispiel: Array-Befüllung mit einer Schleife

```
for (int i = 0; i < intArray.length; i++) {
    intArray[i] = i + 1; // Füllt das Array mit Werten von 1 bis 10
}

System.out.println(Arrays.toString(intArray)); // Gibt das gesamte Array aus
```

Verwendung von `Arrays.fill()`

Alternativ kann das Array auch komplett mit einem bestimmten Wert gefüllt werden.

```
Arrays.fill(intArray, 0, 5, 42); // Füllt die ersten fünf Elemente mit dem Wert 42
```

4. Mehrdimensionale Arrays

Mehrdimensionale Arrays, wie zweidimensionale Arrays, können verwendet werden, um Matrizen oder Tabellen darzustellen.

Beispiel: Mehrdimensionales Array

```
int[][] matrix = {
    {1, 2, 3},
    {4, 5, 6},
    {7, 8, 9}
};

System.out.println("Element in Zeile 1, Spalte 2: " + matrix[1][2]); // Gibt 6 aus
```

5. Ausgabe von Arrays

Zur Ausgabe des gesamten Inhalts eines Arrays verwenden wir oft die `Arrays.toString()` Methode. Mehrdimensionale Arrays können mit `Arrays.deepToString()` ausgegeben werden.

Beispiel: Array-Ausgabe

```
System.out.println(Arrays.toString(intArray)); // Gibt alle Elemente des Arrays
aus
System.out.println(Arrays.deepToString(matrix)); // Gibt alle Elemente des
zweidimensionalen Arrays aus
```

---

6. Sortieren von Arrays

Arrays lassen sich einfach mit `Arrays.sort()` sortieren, was besonders nützlich ist, wenn Sie Werte in aufsteigender Reihenfolge organisieren möchten.

Beispiel: Sortieren eines Arrays

```
double[] dArray = {4.4, 2.2, 3.3, 1.1};
System.out.println("Original: " + Arrays.toString(dArray));
Arrays.sort(dArray);
System.out.println("Sortiert: " + Arrays.toString(dArray));
```

---

7. Vergleich von Arrays

Java bietet verschiedene Möglichkeiten, Arrays zu vergleichen. Der Vergleich der Inhalte zweier Arrays kann mit `Arrays.equals()` erfolgen.

Beispiel: Vergleich von Arrays

```
int[] intArray3 = {1, 2, 3};
int[] intArray4 = {1, 2, 3};
System.out.println(Arrays.equals(intArray3, intArray4)); // Gibt true aus
```

---

8. Suchen in Arrays

Die Suche nach einem bestimmten Element in einem Array kann linear oder mit `Arrays.binarySearch()` (nur für sortierte Arrays) durchgeführt werden.

Beispiel: Binäre Suche

```
int position = Arrays.binarySearch(dArray, 2.2);
System.out.println("Position von 2.2: " + position);
```

Zusammenfassung: Array-Operationen in Java

Operation	Beispielcode	Beschreibung
Deklaration und Initialisierung	`int[] arr = new int[5];`	Erstellt ein Array mit fester Größe
Zugriff auf Elemente	`arr[0] = 42;`	Setzt den Wert an Index 0 auf 42
Länge des Arrays	`arr.length`	Gibt die Anzahl der Elemente zurück
Array-Ausgabe	`System.out.println(Arrays.toString(arr));`	Gibt alle Elemente des Arrays aus
Sortieren	`Arrays.sort(arr);`	Sortiert das Array
Vergleichen	`Arrays.equals(arr1, arr2);`	Vergleicht zwei Arrays auf Inhalt
Suchen	`Arrays.binarySearch(arr, value);`	Sucht ein Element im Array
Mehrdimensionales Array	`int[][] matrix = new int[3][3];`	Erstellt ein 2D-Array

Arrays sind eine grundlegende und wichtige Struktur in Java. Mit diesem Wissen können Sie Daten effizient speichern und verarbeiten, indem Sie die zahlreichen Methoden zur Bearbeitung von Arrays anwenden.

Java Collections: Ein Überblick mit Lösungen

Die **Java Collections** bieten flexible Datenstrukturen zum Speichern und Verwalten von Objekten. In Java gibt es verschiedene Collection-Typen, die jeweils unterschiedliche Eigenschaften und Einsatzmöglichkeiten bieten. Hier sind die wichtigsten Collections-Typen:

1. **Listen** (`List`)
2. **Mengen** (`Set`)
3. **Verzeichnisse** (`Map`)

Hinweis: Zum Verwenden dieser Klassen sollten Sie `import java.util.*;` hinzufügen.

---

1. Listen

Listen sind geordnete Datenstrukturen, die aufeinanderfolgende Elemente speichern. Sie erlauben den Zugriff über einen Index und das Einfügen an beliebigen Stellen. Die wichtigsten Implementierungen sind `ArrayList`, `LinkedList`, `Vector`, und `Stack`.

1.1 `ArrayList`

Die `ArrayList` speichert Elemente in einem dynamischen Array. Sie eignet sich gut für überwiegend lesenden Zugriff.

```
import java.util.ArrayList;

public class ListBeispiel {
    public static void main(String[] args) {
        ArrayList<String> autoListe = new ArrayList<>();
        autoListe.add("BMW");
        autoListe.add("Audi");
        autoListe.add("Ford");

        System.out.println("Inhalt der ArrayList: " + autoListe); // Inhalt der
ArrayList: [BMW, Audi, Ford]
        System.out.println("Erstes Element: " + autoListe.get(0)); // Erstes
Element: BMW
        System.out.println("Größe der Liste: " + autoListe.size()); // Größe der
Liste: 3
    }
}
```

Methoden von `ArrayList`

Methode	Beschreibung
`add(element)`	Fügt ein Element hinzu.

Methode	Beschreibung
`get(index)`	Gibt das Element am angegebenen Index zurück.
`size()`	Gibt die Anzahl der Elemente in der Liste zurück.
`remove(index)`	Entfernt das Element am angegebenen Index.
`contains(element)`	Prüft, ob ein Element in der Liste vorhanden ist.

2. Mengen (`Set`)

Mengen sind Datenstrukturen, die keine Duplikate zulassen. Die Reihenfolge der Elemente ist nicht festgelegt. Die gängigsten Implementierungen sind `HashSet` und `TreeSet`.

2.1 `HashSet`

Ein `HashSet` ist eine ungeordnete Menge, in der jedes Element nur einmal vorkommt.

```
import java.util.HashSet;

public class SetBeispiel {
    public static void main(String[] args) {
        HashSet<String> autoSet = new HashSet<>();
        autoSet.add("BMW");
        autoSet.add("Audi");
        autoSet.add("Ford");
        autoSet.add("BMW"); // Duplikat, wird nicht hinzugefügt.

        System.out.println("Inhalt des HashSet: " + autoSet); // Inhalt des
HashSet: [BMW, Audi, Ford]
        System.out.println("Ist Audi im Set? " + autoSet.contains("Audi")); // Ist
Audi im Set? true
    }
}
```

Methoden von `HashSet`

Methode	Beschreibung
`add(element)`	Fügt ein Element hinzu, wenn es nicht bereits existiert.
`size()`	Gibt die Anzahl der Elemente in der Menge zurück.
`contains(element)`	Prüft, ob ein Element in der Menge vorhanden ist.
`remove(element)`	Entfernt das Element aus der Menge.

3. Verzeichnisse (`Map`)

Verzeichnisse (Maps) speichern Daten als Schlüssel-Wert-Paare, wobei jeder Schlüssel eindeutig ist. Die gängigsten Implementierungen sind `HashMap` und `TreeMap`.

3.1 `HashMap`

Eine `HashMap` ordnet jedem Schlüssel einen Wert zu. Dies ist besonders nützlich für schnelle Zuordnungstabellen.

```
import java.util.HashMap;

public class MapBeispiel {
    public static void main(String[] args) {
        HashMap<String, Integer> zahlenMap = new HashMap<>();
        zahlenMap.put("Eins", 1);
        zahlenMap.put("Zwei", 2);
        zahlenMap.put("Drei", 3);

        System.out.println("Inhalt der HashMap: " + zahlenMap); // Inhalt der
HashMap: {Eins=1, Zwei=2, Drei=3}
        System.out.println("Wert für 'Drei': " + zahlenMap.get("Drei")); // Wert
für 'Drei': 3
        System.out.println("Enthält Schlüssel 'Eins'? " +
zahlenMap.containsKey("Eins")); // Enthält Schlüssel 'Eins'? true
    }
}
```

Methoden von `HashMap`

Methode	Beschreibung
`put(key, value)`	Fügt ein Schlüssel-Wert-Paar hinzu oder aktualisiert den Wert.
`get(key)`	Gibt den Wert für den angegebenen Schlüssel zurück.
`size()`	Gibt die Anzahl der Schlüssel-Wert-Paare zurück.
`containsKey(key)`	Prüft, ob ein Schlüssel vorhanden ist.
`containsValue(value)`	Prüft, ob ein Wert vorhanden ist.

Zusammenfassung

Collection-Typ	Beschreibung	Beispiele
List	Geordnete Liste von Elementen mit Zugriff per Index	`ArrayList`, `LinkedList`
Set	Menge von einzigartigen Elementen, keine Duplikate	`HashSet`, `TreeSet`

Collection-Typ	Beschreibung	Beispiele
Map	Schlüssel-Wert-Paare für schnelles Suchen nach einem Schlüssel	`HashMap`, `TreeMap`

Java Collections bieten eine Vielzahl an Methoden zur Verwaltung von Daten. Jede Collection hat ihre spezifischen Eigenschaften und eignet sich für bestimmte Anwendungen.

Methoden in Java: Grundlagen, Aufbau, Aufruf, Parameter, Argumente und Wertübergabe

In Java sind **Methoden** wiederverwendbare Codeblöcke, die eine bestimmte Aufgabe oder Berechnung ausführen. Sie bestehen aus einem definierten Namen, einem Rückgabetyp, optionalen Parametern und einem Methodenrumpf, der die Anweisungen enthält.

Grundlagen der Methoden

Eine **Methode** in Java besteht aus vier Hauptbestandteilen:

1. **Rückgabetyp**: Der Datentyp, den die Methode zurückgibt, oder `void`, wenn keine Rückgabe erfolgt.
2. **Methodenname**: Der Name der Methode, der ihren Zweck beschreibt und durch den sie aufgerufen wird.
3. **Parameter**: Optional; Eingabewerte, die der Methode übergeben werden.
4. **Methodenrumpf**: Der Codeblock, der die Anweisungen der Methode enthält.

Syntax für eine Methode

```
Rückgabetyp methodenName(Parameterliste) {
    // Methodenrumpf
}
```

Einfache Methoden

Beispiel: Methode ohne Rückgabewert (`void`)

```
public class BeispielMethoden {
    // Methode ohne Rückgabewert, die einen Text ausgibt
    public void begruessung() {
        System.out.println("Hallo, willkommen zur Java-Programmierung!");
    }
}
```

Erklärung:

- `void` zeigt an, dass die Methode nichts zurückgibt.
- `begruessung` ist der Methodenname.
- Der Methodenrumpf enthält eine einfache Anweisung, die eine Begrüßung ausgibt.

Aufruf der Methode

```
```

```
public class Main {
    public static void main(String[] args) {
        BeispielMethoden obj = new BeispielMethoden(); // Erstellen eines Objekts
der Klasse
        obj.begruessung(); // Aufruf der Methode
    }
}
```

Methoden mit Rückgabewert

Beispiel: Methode mit Rückgabewert (`int`)

```
public class Rechner {
    // Methode mit Rückgabewert vom Typ int
    public int addiere(int zahl1, int zahl2) {
        int summe = zahl1 + zahl2;
        return summe; // Rückgabe der Summe
    }
}
```

Erklärung:

- `int` ist der Rückgabetyp, daher erwartet der Methodenaufruf eine `int`-Rückgabe.
- `addiere` ist der Methodenname.
- `int zahl1, int zahl2` sind Parameter, die bei jedem Methodenaufruf übergeben werden müssen.
- `return` gibt die Berechnungsergebnisse zurück.

Aufruf der Methode

```
public class Main {
    public static void main(String[] args) {
        Rechner rechner = new Rechner(); // Erstellen eines Objekts
        int ergebnis = rechner.addiere(5, 3); // Aufruf der Methode mit Argumenten
        System.out.println("Die Summe ist: " + ergebnis); // Ausgabe der Summe
    }
}
```

Überladung von Methoden (Method Overloading)

Java unterstützt **Methodenüberladung** (Overloading), d.h., es können mehrere Methoden mit demselben Namen, jedoch unterschiedlichen Parametertypen oder -anzahlen, erstellt werden.

Beispiel: Überladene Methoden

```java
public class Ueberladen {
    // Methode für zwei Parameter
    public int addiere(int zahl1, int zahl2) {
        return zahl1 + zahl2;
    }

    // Methode für drei Parameter
    public int addiere(int zahl1, int zahl2, int zahl3) {
        return zahl1 + zahl2 + zahl3;
    }
}
```

Erklärung:

- Beide Methoden heißen `addiere`, unterscheiden sich jedoch durch die Anzahl der Parameter.
- Der Compiler entscheidet, welche Methode aufzurufen ist, basierend auf den übergebenen Argumenten.

Aufruf der Methode

```java
public class Main {
    public static void main(String[] args) {
        Ueberladen rechner = new Ueberladen();
        int ergebnisZwei = rechner.addiere(5, 10); // Aufruf der Methode mit zwei Parametern
        int ergebnisDrei = rechner.addiere(5, 10, 15); // Aufruf der Methode mit drei Parametern

        System.out.println("Summe von zwei Zahlen: " + ergebnisZwei);
        System.out.println("Summe von drei Zahlen: " + ergebnisDrei);
    }
}
```

Parameter, Argumente und Wertübergabe

Parameter und Argumente

Parameter sind Platzhalter in der Methodendeklaration, die durch **Argumente** beim Aufruf der Methode ersetzt werden. Beispiel:

```java
public int multipliziere(int zahl1, int zahl2) { // zahl1 und zahl2 sind Parameter
    return zahl1 * zahl2;
}

// Aufruf der Methode mit Argumenten
int ergebnis = multipliziere(5, 3); // 5 und 3 sind Argumente
```

Wertübergabe

1. **Call by Value**: Für primitive Datentypen – es wird nur der Wert übergeben, Änderungen in der Methode wirken sich nicht auf die ursprüngliche Variable aus.

```
public void aendereWert(int zahl) {
    zahl = zahl + 10;
    System.out.println("Innerhalb der Methode: " + zahl); // Innerhalb der
Methode: 20
}

int original = 10;
aendereWert(original);
System.out.println("Außerhalb der Methode: " + original); // Außerhalb der
Methode: 10
```

Statische Methoden (`static`)

Definition: Statische Methoden gehören zur Klasse selbst und können ohne Erstellen einer Instanz (eines Objekts) der Klasse aufgerufen werden. Sie werden mit dem `static`-Schlüsselwort definiert.

Eigenschaften von Statischen Methoden

- **Zugehörigkeit zur Klasse**: Eine statische Methode gehört zur Klasse und nicht zu einer spezifischen Instanz (Objekt) der Klasse.
- **Zugriff über Klassennamen**: Statische Methoden werden häufig über den Klassennamen aufgerufen (`Klassenname.methodenName()`).
- **Zugriffsbeschränkungen**: Statische Methoden können **nur auf andere statische Variablen und Methoden** innerhalb derselben Klasse zugreifen.
- **Speicher**: Statische Methoden werden einmal im Speicher abgelegt und bleiben dort für die Laufzeit des Programms.

Beispiel für eine Statische Methode

```
public class MatheUtils {
    // Statische Methode, um das Quadrat einer Zahl zu berechnen
    public static int berechneQuadrat(int zahl) {
        return zahl * zahl;
    }
}

public class Main {
    public static void main(String[] args) {
        // Aufruf der statischen Methode ohne Erstellen einer Instanz von
MatheUtils
        int ergebnis = MatheUtils.berechneQuadrat(5);
```

```
        System.out.println("Das Quadrat ist: " + ergebnis); // Das Quadrat ist: 25
    }
}
```

Nicht-Statische Methoden (Instanzmethoden)

Definition: Nicht-statische Methoden (auch Instanzmethoden genannt) gehören zu einer Instanz (einem Objekt) der Klasse. Sie benötigen eine Instanz, um aufgerufen zu werden.

Eigenschaften von Nicht-Statischen Methoden

- **Zugehörigkeit zur Instanz**: Sie sind an eine spezifische Instanz der Klasse gebunden.
- **Zugriff auf Instanzvariablen und Methoden**: Nicht-statische Methoden können auf **alle Instanzvariablen und Instanzmethoden** der Klasse zugreifen.
- **Aufruf durch Instanziierung**: Eine Instanz der Klasse muss zuerst erstellt werden, bevor die Methode aufgerufen werden kann.

Beispiel für eine Nicht-Statische Methode

```
public class Kreis {
    // Instanzvariable für den Radius
    public double radius;

    // Nicht-statische Methode zur Berechnung des Kreisumfangs
    public double berechneUmfang() {
        return 2 * Math.PI * radius;
    }
}

public class Main {
    public static void main(String[] args) {
        // Erstellen einer Instanz der Klasse Kreis
        Kreis kreis1 = new Kreis();
        kreis1.radius = 5.0;  // Direkte Zuweisung des Radius

        // Aufruf der nicht-statischen Methode über die Instanz
        double umfang = kreis1.berechneUmfang();
        System.out.println("Der Umfang des Kreises ist: " + umfang); // Ausgabe
des Umfangs
    }
}
```

Zusammenfassung der Methoden in Java

Merkmal	Statische Methode	Nicht-Statische Methode
Zugehörigkeit	Zur Klasse	Zur Instanz (dem Objekt) der Klasse

Merkmal	Statische Methode	Nicht-Statische Methode
Aufruf	Über den Klassennamen	Über die Instanz der Klasse
Zugriff auf Instanzvariablen	Nein	Ja
Anwendung	Für allgemeine, unveränderliche Funktionen, Utility-Methoden	Methoden, die auf Objektdaten zugreifen und sie verändern

Java-Methoden bieten eine strukturierte und wiederverwendbare Möglichkeit, Code zu organisieren und zu modularisieren. Statische Methoden eignen sich für allgemeine, unveränderliche Aufgaben und nicht-statische Methoden sind nützlich, wenn sie auf Instanzdaten zugreifen und diese verändern sollen.

Enums in Java

Einführung

Enums in Java sind eine spezielle Art von Klasse, die eine Sammlung von benannten Konstanten definiert. Sie werden häufig verwendet, wenn eine feste Menge von Werten benötigt wird, z. B. Wochentage, Farben oder Zustände.

Vorteile von Enums

- **Typensicherheit**: Enums verhindern, dass ungültige Werte verwendet werden.
- **Klarheit**: Der Code wird durch die Verwendung von Konstanten besser lesbar.
- **Zusätzliche Funktionalität**: Enums können Methoden, Felder und Konstruktoren enthalten.

Grundlagen: Einfache Enum-Deklaration

```
public enum Day {
    MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY, SUNDAY
}
```

Dieses Enum repräsentiert die sieben Wochentage.

Verwendung von Enums

Zuweisung eines Enum-Wertes

```
Day today = Day.MONDAY;
```

Vergleich eines Enum-Wertes

```
if (today == Day.MONDAY) {
    System.out.println("Back to work!");
}
```

Iteration durch alle Werte eines Enums

```
for (Day day : Day.values()) {
    System.out.println(day);
}
```

Erweiterte Enums mit Feldern und Methoden

Enums können zusätzliche Felder, Konstruktoren und Methoden enthalten.

```
public enum Planet {
    MERCURY(3.30e+23, 2.4397e6),
    VENUS(4.87e+24, 6.0518e6),
    EARTH(5.97e+24, 6.3781e6),
    MARS(6.42e+23, 3.3972e6);

    private final double mass;   // Masse in Kilogramm
    private final double radius; // Radius in Metern

    // Konstruktor
    Planet(double mass, double radius) {
        this.mass = mass;
        this.radius = radius;
    }

    public double getMass() {
        return mass;
    }

    public double getRadius() {
        return radius;
    }

    public double calculateSurfaceGravity() {
        double G = 6.67430e-11; // Gravitationskonstante
        return G * mass / (radius * radius);
    }
}
```

Handout: Zeiger und Referenzen in Java

Einführung

In Java werden Zeiger und Referenzen nicht explizit vom Programmierer verwaltet, wie es in Sprachen wie C oder C++ der Fall ist. Java nutzt Referenzen, um auf Objekte im Speicher zu zeigen, und der **Garbage Collector** übernimmt die Speicherverwaltung automatisch.

Zeiger und Referenzen in Java

Unterschiede zwischen Zeigern und Referenzen

Eigenschaft	Zeiger (C/C++)	Referenzen (Java)
Direkter Zugriff	Zeigt direkt auf Speicheradressen	Verweist auf ein Objekt, keine Adressmanipulation
Speicherverwaltung	Manuell, durch den Entwickler	Automatisch, durch den Garbage Collector
Null-Sicherheit	Kann auf null oder ungültige Adressen zeigen	Kann `null` sein, aber ungültige Adressen sind nicht möglich
Gefährlichkeit	Kann Speicherprobleme oder Sicherheitslücken verursachen	Sicherer, da keine Adressarithmetik möglich ist

Referenzen in Java

Referenzen in Java sind "Zeiger-ähnlich", erlauben aber keine direkten Speicheroperationen. Ein Referenztyp speichert die Adresse eines Objekts im Speicher, auf das verwiesen wird.

Beispiel: Referenzen

```
public class ReferenzBeispiel {
    public static void main(String[] args) {
        String original = "Hallo, Welt!";
        String kopie = original;

        System.out.println("Original: " + original);
        System.out.println("Kopie: " + kopie);

        // Ändere Referenz
        kopie = "Neue Nachricht!";
        System.out.println("Nach der Änderung:");
        System.out.println("Original: " + original);
        System.out.println("Kopie: " + kopie);
    }
}
```

Erklärung:

- Beide Variablen `original` und `kopie` zeigen anfangs auf dasselbe Objekt.
- Nach der Änderung verweist `kopie` auf ein neues Objekt, während `original` unverändert bleibt.

Garbage Collector (GC)

Der Garbage Collector (GC) ist eine automatische Speicherverwaltung in Java, die ungenutzte Objekte identifiziert und deren Speicher freigibt.

Merkmale des Garbage Collectors

1. **Automatische Speicherfreigabe**: Der GC verfolgt, welche Objekte nicht mehr erreichbar sind, und gibt deren Speicher zurück.
2. **Keine manuelle Kontrolle**: Der Entwickler braucht sich nicht um Speicherfreigabe zu kümmern, was Fehler wie Speicherlecks minimiert.
3. **Phasen der Speicherbereinigung**:
    - **Markieren**: Findet alle erreichbaren Objekte.
    - **Bereinigen**: Entfernt alle nicht markierten (unerreichbaren) Objekte.
    - **Kompaktieren**: Organisiert den Speicher, um Fragmentierung zu vermeiden.

Beispiel: Garbage Collection

```
public class GarbageCollectorBeispiel {
    public static void main(String[] args) {
        // Erstelle ein Objekt und weise es einer Referenz zu
        String text = new String("Hallo, Welt!");

        // Entferne die Referenz auf das Objekt
        text = null;

        // Fordere den Garbage Collector auf, zu arbeiten
        System.gc();

        System.out.println("Garbage Collector wurde angefordert.");
    }
}
```

Hinweise:

- `System.gc()` fordert den Garbage Collector auf, aber es gibt keine Garantie, dass er sofort ausgeführt wird.
- Objekte, die keine Referenzen mehr haben, gelten als "garbage" (Müll) und werden entfernt.

Liste aller Java Exceptions

1. Checked Exceptions

Checked Exceptions werden vom Compiler überprüft und müssen behandelt oder weitergegeben (`throws`) werden. Sie treten bei vorhersehbaren Problemen auf, wie z. B. beim Arbeiten mit Dateien oder Datenbanken.

Exception	Beschreibung
IOException	Fehler bei der Ein-/Ausgabe (z. B. Datei nicht gefunden).
FileNotFoundException	Eine Datei wurde nicht gefunden.
ClassNotFoundException	Eine Klasse konnte nicht geladen werden.
SQLException	Fehler bei der Arbeit mit einer Datenbank.
InterruptedException	Ein Thread wurde während des Wartens unterbrochen.
MalformedURLException	Eine fehlerhafte URL wurde verwendet.
ParseException	Fehler beim Parsen von Daten (z. B. String zu Datum).
InstantiationException	Ein Objekt einer abstrakten Klasse oder Schnittstelle wurde instanziiert.
NoSuchFieldException	Ein angeforderter Klassen- oder Objekt-Field existiert nicht.
NoSuchMethodException	Eine Methode, die aufgerufen wird, existiert nicht.

2. Unchecked Exceptions (Runtime Exceptions)

Unchecked Exceptions treten zur Laufzeit auf und sind typischerweise das Ergebnis von Programmierfehlern. Sie werden nicht vom Compiler überprüft.

Exception	Beschreibung
ArithmeticException	Fehler bei mathematischen Operationen (z. B. Division durch null).
NullPointerException	Zugriff auf ein `null`-Objekt oder Methode eines `null`.
IndexOutOfBoundsException	Zugriff auf einen Index außerhalb der Array-/Listen-Grenzen.
ArrayIndexOutOfBoundsException	Zugriff auf ein Array mit ungültigem Index.
StringIndexOutOfBoundsException	Zugriff auf eine ungültige Zeichenposition in einem String.
ClassCastException	Ungültiger Typ-Cast (Typumwandlung).
IllegalArgumentException	Ungültiges Argument an eine Methode übergeben.
NumberFormatException	Fehler bei der Konvertierung von String zu Zahl.
UnsupportedOperationException	Eine Methode wird nicht unterstützt.

Exception	Beschreibung
ConcurrentModificationException	Gleichzeitige Änderungen an einer Collection, die nicht unterstützt werden.
IllegalStateException	Der Zustand eines Objekts ist für den aktuellen Aufruf ungültig.

3. Errors

Errors sind schwerwiegende Fehler, die meist vom System verursacht werden und nicht durch den Code behoben werden können. Sie sollten in der Regel nicht abgefangen werden.

Error	Beschreibung
OutOfMemoryError	Kein Speicherplatz im Heap-Speicher verfügbar.
StackOverflowError	Der Stack-Speicher ist erschöpft (z. B. durch unendliche Rekursion).
VirtualMachineError	Schwerwiegender Fehler in der JVM (z. B. JVM-Absturz).
AssertionError	Fehler durch eine fehlgeschlagene `assert`-Anweisung.
NoClassDefFoundError	Eine Klasse, die zur Laufzeit benötigt wird, konnte nicht gefunden werden.
LinkageError	Probleme beim Verknüpfen einer Klasse oder eines Programms.
ThreadDeath	Ein Thread wurde abrupt beendet.

Zusammenfassung: Fehlertypen und Behandlung

Typ	Beispiele	Behandlung
Checked Exceptions	`IOException`, `SQLException`	Mit `try-catch` blockieren oder mit `throws` deklarieren.
Unchecked Exceptions	`NullPointerException`, `ArithmeticException`	Optional mit `try-catch` blockieren, durch präventive Logik vermeiden.
Errors	`OutOfMemoryError`, `StackOverflowError`	Nicht abfangen. Fehlerprotokollierung und ggf. Neustart des Systems.

Handout: Ausnahmebehandlung in Java

Einführung

Ausnahmebehandlung (Exception Handling) in Java ist ein Mechanismus, um auf Fehler und unerwartete Zustände während der Programmausführung zu reagieren. Java bietet eine klare Struktur, um Laufzeitfehler abzufangen, zu behandeln und ggf. weiterzugeben.

Fehlertypen in Java

Fehlertyp	Beschreibung	Beispiele	Empfohlene Handlung
Checked Exceptions	Fehler, die vom Compiler überprüft werden. Müssen im Code behandelt oder deklariert werden.	`IOException`, `SQLException`	Verwende `try-catch` oder `throws`, um den Fehler abzufangen.
Unchecked Exceptions	Fehler, die zur Laufzeit auftreten. Werden nicht vom Compiler überprüft.	`NullPointerException`, `ArrayIndexOutOfBoundsException`	Vermeide fehlerhafte Logik oder behandle sie optional mit `try-catch`.
Errors	Schwere Fehler, die auf Systemebene auftreten. Können meist nicht durch den Code behoben werden.	`OutOfMemoryError`, `StackOverflowError`	Fehlerprotokollierung und Neustart des Systems.

Grundkonzepte der Ausnahmebehandlung

1. **Ausnahmen abfangen und behandeln**: Mit `try-catch` Blöcken können Fehler kontrolliert abgefangen und verarbeitet werden.
2. **Ausnahmen weitergeben**: Mit `throws` können Methoden deklarieren, dass sie bestimmte Fehler nicht selbst behandeln.
3. **Ausnahmen auslösen**: Mit `throw` kann eine Ausnahme manuell erzeugt werden.

Ausnahmebehandlung in Java: Beispiele

1. Exceptions abfangen und behandeln

Der `try-catch`-Block ermöglicht es, bestimmte Fehler zu erkennen und darauf zu reagieren.

```
```

```
public class AusnahmeBeispiel {
    public static void main(String[] args) {
        try {
            int result = 10 / 0; // Division durch null
        } catch (ArithmeticException e) {
            System.out.println("Fehler: Division durch null ist nicht erlaubt!");
        }
    }
}
```

Erklärung:

- Die Division durch null löst eine `ArithmeticException` aus.
- Der `catch`-Block fängt diese Ausnahme ab und verhindert einen Absturz des Programms.

---

2. Ausnahmen weitergeben

Eine Methode kann deklarieren, dass sie bestimmte Fehler nicht selbst behandelt, sondern an den Aufrufer weitergibt.

```
import java.io.File;
import java.io.FileNotFoundException;
import java.util.Scanner;

public class AusnahmeWeitergeben {
    public static void main(String[] args) throws FileNotFoundException {
        leseDatei("nicht_existierende_datei.txt");
    }

    public static void leseDatei(String dateiname) throws FileNotFoundException {
        Scanner scanner = new Scanner(new File(dateiname));
        System.out.println(scanner.nextLine());
    }
}
```

Erklärung:

- Die Methode `leseDatei` verwendet `throws`, um eine `FileNotFoundException` an den Aufrufer weiterzugeben.
- Der Aufrufer (`main`) muss diese Ausnahme entweder abfangen oder selbst deklarieren.

---

3. Ausnahmen auslösen

Mit `throw` kann eine Ausnahme manuell erzeugt werden.

```
```

```
public class AusnahmeAuslösen {
    public static void main(String[] args) {
        try {
            überprüfeAlter(-5); // Ungültiges Alter
        } catch (IllegalArgumentException e) {
            System.out.println("Fehler: " + e.getMessage());
        }
    }

    public static void überprüfeAlter(int alter) {
        if (alter < 0) {
            throw new IllegalArgumentException("Alter darf nicht negativ sein.");
        }
        System.out.println("Alter ist gültig: " + alter);
    }
}
```

Erklärung:

- Wenn das Alter negativ ist, wird eine `IllegalArgumentException` ausgelöst.
- Der Fehler wird im `catch`-Block behandelt.

---

4. Eigene Exceptions erstellen

Benutzerdefinierte Exceptions können durch die Erweiterung der Klasse `Exception` erstellt werden.

```
public class EigeneException {
    public static void main(String[] args) {
        try {
            prüfePasswort("123"); // Unsicheres Passwort
        } catch (UngültigesPasswortException e) {
            System.out.println("Fehler: " + e.getMessage());
        }
    }

    public static void prüfePasswort(String passwort) throws
UngültigesPasswortException {
        if (passwort.length() < 6) {
            throw new UngültigesPasswortException("Das Passwort ist zu kurz.");
        }
        System.out.println("Passwort ist sicher.");
    }
}

// Benutzerdefinierte Exception
class UngültigesPasswortException extends Exception {
    public UngültigesPasswortException(String nachricht) {
        super(nachricht);
    }
}
```

Erklärung:

- Die Klasse `UngültigesPasswortException` erweitert `Exception` und definiert eine benutzerdefinierte Fehlermeldung.
- Die Methode `prüfePasswort` wirft diese Ausnahme, wenn das Passwort unsicher ist.

Laufzeitfehler (Unchecked Exceptions)

Fehlertyp	Beschreibung
NullPointerException	Aufruf einer Methode oder eines Attributs auf einem `null`-Objekt.
ArrayIndexOutOfBoundsException	Zugriff auf einen Index außerhalb der Array-Grenzen.
ArithmeticException	Fehler bei arithmetischen Operationen (z. B. Division durch null).

Beispiel: NullPointerException abfangen

```
public class NullPointerBeispiel {
    public static void main(String[] args) {
        try {
            String text = null;
            System.out.println(text.length()); // Führt zu NullPointerException
        } catch (NullPointerException e) {
            System.out.println("Fehler: Ein null-Objekt hat keine
Eigenschaften!");
        }
    }
}
```

Zusammenfassende Tabelle

Konstrukt	Verwendung
`try-catch`	Fängt Exceptions ab und verarbeitet sie.
`finally`	Führt einen Block immer aus, unabhängig davon, ob eine Ausnahme auftritt oder nicht.
`throws`	Deklariert, dass eine Methode eine Ausnahme an den Aufrufer weitergeben kann.
`throw`	Löst eine Ausnahme manuell aus.
Checked Exceptions	Müssen behandelt oder deklariert werden.
Unchecked Exceptions	Können optional behandelt werden. Treten typischerweise aufgrund von Programmierfehlern auf.

Konstrukt	Verwendung
Errors	Systemfehler, die nicht durch den Code behandelt werden können.