Lückentext

Übung1)

public class \_\_\_\_\_\_\_ { // Tipp: Hier steht der Name der Klasse

public static void \_\_\_\_\_\_\_(String[] \_\_\_\_\_\_\_) { // Tipp: Dies ist die main-Methode

System.\_\_\_\_\_\_.println("Hello, World!"); // Tipp: So gibt man Text aus

}

}

Um dieses Programm auszuführen, muss es zunächst \_\_\_\_\_\_\_\_\_ werden. Dies geschieht mit dem Java \_\_\_\_\_\_\_\_\_. Dabei wird der Quellcode in \_\_\_\_\_\_\_\_\_ umgewandelt, welcher von der Java \_\_\_\_\_\_\_\_\_ ausgeführt werden kann.

Im obigen Code ist \_\_\_\_\_\_\_ der Name der Klasse. In Java ist jedes Programm in einer oder mehreren Klassen organisiert. Die \_\_\_\_\_\_\_-Methode ist der \_\_\_\_\_\_\_\_\_ des Programms. Sie wird automatisch ausgeführt, wenn das Programm gestartet wird. System.out.println() ist eine Anweisung, die den Text "Hello, World!" auf der \_\_\_\_\_\_\_\_\_ ausgibt. String[] args repräsentiert die \_\_\_\_\_\_\_\_\_, die dem Programm beim Start übergeben werden können.

**Lösungen:**

1. HelloWorld // Tipp: Klassenname beginnt üblicherweise mit einem Großbuchstaben
2. main // Tipp: Wichtige Methode für den Programmstart
3. args // Tipp: Kurz für "arguments"
4. out // Tipp: Teil von System.out
5. kompiliert // Tipp: Vor der Ausführung notwendig
6. Compiler // Tipp: Übersetzt den Code
7. Bytecode // Tipp: Nicht direkt vom Computer ausführbar
8. Virtual Machine // Tipp: Führt den Bytecode aus
9. HelloWorld // Tipp: Muss mit dem Dateinamen übereinstimmen
10. Einstiegspunkt // Tipp: Wo die Ausführung beginnt
11. Konsole // Tipp: Wo die Ausgabe erscheint
12. Befehlszeilenargumente // Tipp: Zusätzliche Infos für das Programm

Übung2)

**Lückentext: Variablen in Java (mit Lösungshilfe)**

In Java sind \_\_\_\_\_\_\_\_\_ Behälter für Daten. (Tipp: Was speichern wir, um Werte zu halten?) Jede Variable hat einen \_\_\_\_\_\_\_\_\_, der bestimmt, welche Art von Daten sie speichern kann. (Tipp: z.B. Zahl, Text, etc.) Um eine Variable zu deklarieren, gibst du zuerst den \_\_\_\_\_\_\_\_\_ der Variablen an, gefolgt vom \_\_\_\_\_\_\_\_\_ der Variablen. (Tipp: Was kommt zuerst: "int x" oder "x int"?) Zum Beispiel deklariert int alter; eine Variable namens alter, die \_\_\_\_\_\_\_\_\_ speichern kann. (Tipp: Was speichert "int"?)

Du kannst einer Variablen einen Wert zuweisen, indem du den \_\_\_\_\_\_\_\_\_ verwendest. (Tipp: Welches Zeichen benutzt man für "ist gleich"?) Zum Beispiel weist alter = 20; der Variablen alter den Wert 20 zu. Du kannst eine Variable auch bei der Deklaration \_\_\_\_\_\_\_\_\_. (Tipp: Das Gegenteil von "nur deklarieren") Zum Beispiel deklariert und initialisiert String name = "Max"; eine Variable namens name mit dem Wert "Max".

Java bietet verschiedene Datentypen, darunter:

* int: für \_\_\_\_\_\_\_\_\_ (Tipp: ganze Zahlen)
* double: für \_\_\_\_\_\_\_\_\_ (Tipp: Kommazahlen)
* boolean: für \_\_\_\_\_\_\_\_\_ (Tipp: nur zwei Werte möglich)
* String: für \_\_\_\_\_\_\_\_\_ (Tipp: Mehrere Zeichen)

Variablennamen in Java sind \_\_\_\_\_\_\_\_\_ und dürfen nicht mit einer Ziffer beginnen. (Tipp: Macht es einen Unterschied, ob man Groß- oder Kleinschreibung nutzt?)

**Lösungen:**

1. Variablen
2. Datentyp
3. Datentyp, Namen
4. ganze Zahlen
5. Zuweisungsoperator
6. initialisieren
7. ganze Zahlen
8. Fließkommazahlen
9. Wahrheitswerte
10. Zeichenketten
11. case-sensitive

Übung3)

**Lückentext: Kontrollfluss in Java**

Der \_\_\_\_\_\_\_\_\_ in einem Java-Programm bestimmt die Reihenfolge, in der die Anweisungen ausgeführt werden. Normalerweise werden Anweisungen von oben nach unten ausgeführt, aber mit Kontrollflussstrukturen können wir diese Reihenfolge ändern.

Die einfachste Kontrollflussstruktur ist die \_\_\_\_\_\_\_\_\_. Sie ermöglicht es, einen Codeblock nur dann auszuführen, wenn eine bestimmte \_\_\_\_\_\_\_\_\_ wahr ist. Das Schlüsselwort für diese Struktur ist \_\_\_\_\_\_\_\_\_.

Java

if (\_\_\_\_\_\_\_\_\_) {

// Code, der ausgeführt wird, wenn die Bedingung wahr ist

}

Manchmal möchten wir auch Code ausführen, wenn die Bedingung falsch ist. Hierfür verwenden wir das Schlüsselwort \_\_\_\_\_\_\_\_\_.

Java

if (\_\_\_\_\_\_\_\_\_) {

// Code für den wahren Fall

} else {

// Code für den falschen Fall

}

Für Mehrfachauswahlen gibt es die \_\_\_\_\_\_\_\_\_-Struktur.

Java

if (bedingung1) {

// Code für den ersten Fall

} else if (bedingung2) {

// Code für den zweiten Fall

} else {

// Code, der ausgeführt wird, wenn keine der Bedingungen wahr ist

}

Neben bedingten Anweisungen gibt es auch \_\_\_\_\_\_\_\_\_, mit denen wir Code wiederholt ausführen können. Die am häufigsten verwendete Schleife ist die \_\_\_\_\_\_\_\_\_-Schleife.

Java

for (initialisierung; \_\_\_\_\_\_\_\_\_; aktualisierung) {

// Code, der wiederholt ausgeführt wird

}

Eine andere Schleifenart ist die \_\_\_\_\_\_\_\_\_-Schleife, die einen Codeblock so lange ausführt, wie eine Bedingung wahr ist.

Java

while (\_\_\_\_\_\_\_\_\_) {

// Code, der wiederholt ausgeführt wird

}

**Lösungen mit Lösungstipps:**

1. **Kontrollfluss** (Tipp: Wie nennt man den "Fluss" der Ausführung?)
2. **if-Anweisung** (Tipp: Grundlegende "Wenn-dann"-Logik)
3. **Bedingung** (Tipp: Was muss erfüllt sein, damit der Code ausgeführt wird?)
4. **if** (Tipp: Das Java-Schlüsselwort für die Bedingung)
5. **Bedingung** (Tipp: Ein Ausdruck, der "wahr" oder "falsch" ergibt)
6. **else** (Tipp: Das Gegenstück zu "if")
7. **Bedingung** (Tipp: Wie bei der ersten if-Anweisung)
8. **if-else-if** (Tipp: Für mehrere aufeinanderfolgende Bedingungen)
9. **Schleifen** (Tipp: Code wiederholt ausführen)
10. **for** (Tipp: Schleife mit Zähler)
11. **Bedingung** (Tipp: Solange diese gilt, wird die Schleife ausgeführt)
12. **while** (Tipp: Schleife, die auf einer Bedingung basiert)
13. **Bedingung** (Tipp: Wie bei der for-Schleife)

Übung4)

**Lückentext: Schleifen in Java**

\_\_\_\_\_\_\_\_\_ sind Kontrollflussstrukturen, die es ermöglichen, einen Codeblock wiederholt auszuführen. Es gibt verschiedene Arten von Schleifen in Java, jede mit ihren eigenen Anwendungsfällen.

Die \_\_\_\_\_\_\_\_\_-Schleife ist besonders nützlich, wenn du genau weißt, wie oft ein Codeblock ausgeführt werden soll. Sie besteht aus drei Teilen:

* \_\_\_\_\_\_\_\_\_ : Hier wird eine Variable initialisiert, die als Zähler dient.
* \_\_\_\_\_\_\_\_\_ : Diese Bedingung wird vor jeder Ausführung des Codeblocks geprüft. Solange sie wahr ist, wird der Codeblock ausgeführt.
* \_\_\_\_\_\_\_\_\_ : Hier wird der Zähler nach jeder Ausführung des Codeblocks aktualisiert (z.B. erhöht).

Java

for (\_\_\_\_\_\_\_\_\_; \_\_\_\_\_\_\_\_\_; \_\_\_\_\_\_\_\_\_) {

// Code, der wiederholt ausgeführt wird

}

Ein Beispiel für eine for-Schleife:

Java

for (int i = 0; i < 10; i++) {

System.out.println(i);

}

In diesem Beispiel wird die Variable i von 0 bis \_\_\_\_\_\_\_\_\_ gezählt.

Die \_\_\_\_\_\_\_\_\_-Schleife ist eine weitere Art von Schleife, die einen Codeblock so lange ausführt, wie eine bestimmte \_\_\_\_\_\_\_\_\_ wahr ist. Im Gegensatz zur for-Schleife ist die Anzahl der Wiederholungen hier nicht im Voraus bekannt.

Java

while (\_\_\_\_\_\_\_\_\_) {

// Code, der wiederholt ausgeführt wird

}

Ein Beispiel für eine while-Schleife:

Java

int zahl = 5;

while (zahl > 0) {

System.out.println(zahl);

zahl--;

}

In diesem Beispiel wird die Variable zahl so lange ausgegeben und dekrementiert, bis sie den Wert \_\_\_\_\_\_\_\_\_ erreicht.

Es gibt auch die \_\_\_\_\_\_\_\_\_-Schleife, die der while-Schleife sehr ähnlich ist. Der Hauptunterschied besteht darin, dass die Bedingung hier am \_\_\_\_\_\_\_\_\_ der Schleife geprüft wird. Das bedeutet, dass der Codeblock in einer do-while-Schleife immer \_\_\_\_\_\_\_\_\_ ausgeführt wird, auch wenn die Bedingung von Anfang an falsch ist.

Java

do {

// Code, der wiederholt ausgeführt wird

} while (\_\_\_\_\_\_\_\_\_);

**Lösungen mit Lösungstipps:**

1. **Schleifen** (Tipp: Mehrzahl von "Schleife")
2. **for** (Tipp: Schleife mit fester Anzahl von Wiederholungen)
3. **Initialisierung** (Tipp: Was passiert am Anfang der Schleife mit dem Zähler?)
4. **Bedingung** (Tipp: Was muss gelten, damit die Schleife weiterläuft?)
5. **Aktualisierung** (Tipp: Wie verändert sich der Zähler nach jeder Runde?)
6. **Initialisierung; Bedingung; Aktualisierung** (Tipp: Die drei Teile der for-Schleife)
7. **9** (Tipp: Die Schleife läuft, solange i *kleiner* als 10 ist)
8. **while** (Tipp: Schleife, die auf einer Bedingung basiert)
9. **Bedingung** (Tipp: Wie bei der while-Schleife)
10. **Bedingung** (Tipp: Was muss wahr sein, damit die Schleife weiterläuft?)
11. **0** (Tipp: Die Schleife läuft, solange zahl *größer* als 0 ist)
12. **do-while** (Tipp: Eine Variante der while-Schleife)
13. **Ende** (Tipp: Wo wird die Bedingung geprüft?)
14. **mindestens einmal** (Tipp: Wann wird der Codeblock *sicher* ausgeführt?)
15. **Bedingung** (Tipp: Wie bei der while-Schleife)

Übung6)

**Lückentext: Arrays in Java**

\_\_\_\_\_\_\_\_\_ in Java sind Objekte, die mehrere Werte desselben \_\_\_\_\_\_\_\_\_ speichern können. Du kannst dir ein Array wie eine Liste von Variablen vorstellen.

Um ein Array zu deklarieren, gibst du den Datentyp der Elemente an, gefolgt von \_\_\_\_\_\_\_\_\_ und dem Namen des Arrays. Zum Beispiel deklariert int[] zahlen; ein Array namens zahlen, das \_\_\_\_\_\_\_\_\_ speichern kann.

Um ein Array zu erstellen, verwendest du das Schlüsselwort \_\_\_\_\_\_\_\_\_ und gibst die \_\_\_\_\_\_\_\_\_ des Arrays in eckigen Klammern an. Zum Beispiel erstellt zahlen = new int[10]; ein Array, das 10 Integer-Werte speichern kann.

Du kannst auch ein Array deklarieren und erstellen und es gleichzeitig \_\_\_\_\_\_\_\_\_, indem du die Werte in geschweiften Klammern angibst. Zum Beispiel erstellt String[] namen = {"Alice", "Bob", "Charlie"}; ein Array namens namen mit den Werten "Alice", "Bob" und "Charlie". Die Länge dieses Arrays ist \_\_\_\_\_\_\_\_\_.

Auf die Elemente eines Arrays greifst du über ihren \_\_\_\_\_\_\_\_\_ zu. Der Index des ersten Elements ist \_\_\_\_\_\_\_\_\_, der des zweiten Elements ist 1 usw. Zum Beispiel greift zahlen[0] auf das erste Element des Arrays zahlen zu.

Du kannst die \_\_\_\_\_\_\_\_\_ eines Arrays mit der Eigenschaft \_\_\_\_\_\_\_\_\_ abrufen. Zum Beispiel gibt zahlen.length die Länge des Arrays zahlen zurück.

Arrays sind in Java \_\_\_\_\_\_\_\_\_, was bedeutet, dass ihre Länge nach der Erstellung nicht mehr geändert werden kann.

**Lösungen mit Lösungshilfen:**

1. **Arrays** (Tipp: Mehrzahl von "Array")
2. **Datentyps** (Tipp: Was für eine Art von Daten speichert das Array?)
3. **eckigen Klammern** (Tipp: [ ])
4. **ganze Zahlen** (Tipp: Was speichert ein int-Array?)
5. **new** (Tipp: Schlüsselwort zum Erstellen von Objekten)
6. **Länge** (Tipp: Wie viele Elemente soll das Array haben?)
7. **initialisieren** (Tipp: Werte direkt zuweisen)
8. **3** (Tipp: Zähle die Elemente in den geschweiften Klammern)
9. **Index** (Tipp: Position eines Elements im Array)
10. **0** (Tipp: Der erste Platz in einer Liste beginnt normalerweise nicht bei 1)
11. **Länge** (Tipp: Anzahl der Elemente)
12. **.length** (Tipp: So fragt man die Größe eines Arrays ab)
13. **statisch** (Tipp: Unveränderlich in der Größe)

**Übung7)**

\_\_\_\_\_\_\_\_\_ in Java sind wie kleine "Helfer", die eine bestimmte Aufgabe erledigen. Stell dir vor, du hast ein Rezept für Kuchen backen. Die Methode wäre dann ein einzelner Schritt in diesem Rezept, z.B. "Eier aufschlagen" oder "Mehl hinzufügen". Methoden helfen uns, unseren Code zu \_\_\_\_\_\_\_\_\_ und zu vermeiden, dass wir denselben Code immer wieder schreiben müssen.

Jede Methode hat einen \_\_\_\_\_\_\_\_\_, der angibt, welche Art von Ergebnis die Methode zurückgibt, wenn sie fertig ist. Wenn die Methode keine Ergebnis zurückgibt, verwenden wir das Schlüsselwort \_\_\_\_\_\_\_\_\_. Das bedeutet, die Methode macht etwas, gibt uns aber kein "Produkt" zurück.

Jede Methode hat auch einen \_\_\_\_\_\_\_\_\_, damit wir sie später aufrufen können. Nach dem Namen kommen \_\_\_\_\_\_\_\_\_ (runde Klammern). Manchmal können wir der Methode in den Klammern Informationen geben, die sie für ihre Aufgabe benötigt. Diese Informationen nennen wir \_\_\_\_\_\_\_\_\_. Stell dir vor, du sagst dem "Eier aufschlagen"-Helfer, *wie viele* Eier er aufschlagen soll.

Der Code, den die Methode ausführt, steht in geschweiften \_\_\_\_\_\_\_\_\_. Das ist der eigentliche "Arbeitsbereich" des Helfers.

Um eine Methode zu benutzen, schreiben wir ihren \_\_\_\_\_\_\_\_\_ und die Klammern. Wenn die Methode Informationen (Parameter) erwartet, müssen wir diese Informationen (jetzt \_\_\_\_\_\_\_\_\_ genannt) in den Klammern angeben.

Java

public static \_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_ ( \_\_\_\_\_\_\_\_\_ ) {

// Code der Methode (der "Arbeitsbereich" des Helfers)

\_\_\_\_\_\_\_\_\_ ergebnis; // Wenn die Methode ein Ergebnis zurückgibt, nennen wir es hier "ergebnis"

return ergebnis;

}

// Beispielaufruf:

\_\_\_\_\_\_\_\_\_ ( \_\_\_\_\_\_\_\_\_ );

Wichtige Begriffe:

* **Parameter:** Das sind die "Informationen", die wir der Methode *geben*, wenn wir sie definieren. Wie viele Eier? Welches Mehl?
* **Argumente:** Das sind die tatsächlichen *Werte* für diese Informationen, wenn wir die Methode *benutzen*. Also z.B. 2 Eier, 500g Mehl.
* \_\_\_\_\_\_\_\_\_: Das ist wie das "Zurückgeben" des Helfers. Er gibt uns z.B. den aufgeschlagenen Teig zurück.

**Lösungen mit Lösungshilfen:**

1. **Methoden** (Tipp: Wie nennt man die "Helfer" im Code?)
2. **Organisieren** (Tipp: Was hilft uns, den Code übersichtlicher zu machen?)
3. **Rückgabetyp** (Tipp: Was für eine Art von "Produkt" liefert der Helfer?)
4. \*\*void\*\* (Tipp: Was bedeutet "kein Produkt"?)
5. **Name**, **Klammern** (Tipp: Wie rufen wir den Helfer?)
6. **Parameter** (Tipp: Welche "Informationen" braucht der Helfer?)
7. **Klammern** (Tipp: Wo steht der Code des Helfers?)
8. **Name** (Tipp: Wie heißt der Helfer?)
9. **Argumente** (Tipp: Die tatsächlichen Werte für die Informationen)
10. **Rückgabetyp, Name, Parameter** (Tipp: Die "Beschreibung" des Helfers)
11. **return** (Tipp: Wie gibt der Helfer das "Produkt" zurück?)
12. **Name, Argumente** (Tipp: Wie "benutzen" wir den Helfer?)
13. **Parameter** (Tipp: Die "Platzhalter" für die Informationen)
14. **Werte** (Tipp: Was setzen wir für die Platzhalter ein?)
15. \*\*return\*\* (Tipp: Der Helfer gibt uns etwas zurück)

Übung8)

In Java sind \_\_\_\_\_\_\_\_\_ wie Baupläne für Dinge. Stell dir vor, du hast einen Bauplan für ein Auto. Der Bauplan sagt, welche Teile das Auto hat (z.B. Farbe, Anzahl der Räder) und was es kann (z.B. fahren, hupen).

Ein \_\_\_\_\_\_\_\_\_ ist ein echtes Ding, das nach dem Bauplan gebaut wurde. Also ein echtes Auto, das die Farbe hat, die im Bauplan steht, und fahren kann.

Die Teile, die das Auto hat (Farbe, Anzahl der Räder), nennen wir in Java \_\_\_\_\_\_\_\_\_ oder auch \_\_\_\_\_\_\_\_\_.

Die Dinge, die das Auto kann (fahren, hupen), nennen wir in Java \_\_\_\_\_\_\_\_\_.

Um einen Bauplan für ein Auto in Java zu machen, verwenden wir das Schlüsselwort \_\_\_\_\_\_\_\_\_.

Java

public class Auto {

// Teile des Autos (Attribute)

String farbe;

int anzahlRaeder;

// Was das Auto kann (Methoden)

void fahren() {

System.out.println("Das Auto fährt los!");

}

void hupen() {

System.out.println("Tuut-Tuut!");

}

}

Um ein echtes Auto nach dem Bauplan zu bauen, verwenden wir das Schlüsselwort \_\_\_\_\_\_\_\_\_.

Java

Auto meinAuto = new Auto();

Jetzt ist meinAuto ein \_\_\_\_\_\_\_\_\_ vom Typ Auto.

Um auf die Teile (Attribute) oder das, was das Auto kann (Methoden) zuzugreifen, verwenden wir einen \_\_\_\_\_\_\_\_\_.

Java

meinAuto.farbe = "Blau"; // Wir geben dem Auto die Farbe Blau

meinAuto.fahren(); // Wir lassen das Auto fahren

**Lösungen mit Lösungshilfen:**

1. **Klassen** (Tipp: Wie nennt man die Baupläne?)
2. **Objekt** (Tipp: Wie nennt man das echte Ding?)
3. **Attribute** oder **Variablen** (Tipp: Wie nennt man die Teile?)
4. **Methoden** (Tipp: Wie nennt man das, was das Auto kann?)
5. \*\*class\*\* (Tipp: So beginnt man einen Bauplan in Java)
6. \*\*new\*\* (Tipp: So baut man ein echtes Ding nach dem Bauplan)
7. **Objekt** (Tipp: Ein echtes Auto)
8. **Punkt** (.) (Tipp: So greift man auf die Teile und das, was es kann, zu)

 Quellen