# ÜBUNGS-BLOCK 7
# LÖSUNGEN


**Aufgabe 1:**

Gegeben ist folgender Code:

```
Auto[] array = new Auto[3];
// Alle Autos im Array tunen:
for (int i = 1; i <= 3; i++) {
      array[i].tunen();
}
```

Was hälst du davon? Fällt dir irgendetwas auf?

*Lösung:*

Es gibt in diesem Code zwei Probleme:

Zuerst einmal versuchen wir hier nicht die Autos in den Fächern eins bis drei zu tunen, sondern die Autos in den Fächern zwei bis vier – denn der Index bei Arrays beginnt bei 0! Also ist das Fach mit Index drei das vierte Fach, das es gar nicht gibt. Die Folge wäre:

```
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 3
```

Dieser Fehler wird aber gar nicht auftreten, denn ein anderer Fehler kommt ihm – schon beim ersten Schleifendurchlauf – zuvor:

```
Exception in thread "main" java.lang.NullPointerException
```

**Merke**: Wenn man ein Array initialisiert, haben alle Fächer den Default-Wert des deklarierten Datentyps! Das heisst: Wir haben hier keine Autos. In allen Fächern steckt die null-Referenz! Ein Zugriff per Punkt-Notation, also eine De-Referenzierung, auf eine Variable, die gar keine gültige Referenz auf ein Objekt hat, führt immer zu einer NullPointerException (NPE).

*Anmerkung*:

Der Code ist absichtlich so geschrieben, dass er von der NPE ablenkt – es ist also in diesem Fall nicht so schlimm, wenn du nicht an *beide* Fehler gedacht hast.

Aber: Es handelt sich hier um **schwere** Fehler, und du solltest es dir zum Ziel machen, dass dir solche Fehler in Zukunft sofort ins Auge springen. Wenn ich dich um 3 Uhr nachts aus dem Schlaf wecke und „Array“ sage, dann solltest du ohne zu zögern sagen: „**Achtung** Index – **Achtung** NPE!“

Hintergrund: Beide Fehler sind keine Compile-Fehler, sondern sogenannte Runtime-Fehler (Laufzeit-Fehler). Wir werden solche Fehlermeldungen später im Kurs noch im Genauen behandeln, aber du solltest jetzt schon wissen: Runtime-Fehler bedeutet, sie treten erst bei der *Ausführung* des Codes auf. Eclipse würde hier beim Programmieren also noch keinen Fehler anzeigen. Deshalb sind solche Fehler besonders kritisch.

**Aufgabe 2:**

Gegeben ist der folgende Code:

```
1     int[] b = {5,10,20};
2     int a[] = new int[2]{5,10};
3     b = {10,15,20};
4     a = new int[3];
5     String d = new String[100];
6     double[][] e = new double[5][];
7     double[] f[] = {{10},{50, 20}};
```

Welche dieser Zeilen sind gültig, welche nicht?

*Lösung:*

1 Gültig – es wird ein int-Array mit drei Fächern angelegt und mit den angegebenen Werte befüllt.

2 Nicht gültig – die Klammern hinter dem Variablen-Namen sind noch okay, aber die Zuweisung nicht mehr. Man kann ein Array *entweder* im Zuge einer direkten Befüllung mittels {...} initialisieren, *oder* durch Angabe der Anzahl der Fächer. Beides zusammen geht nicht.

3 Nicht gültig – die direkte Befüllung mittels der {...} Schreibweise ist nur im Zuge der Variablen-Deklaration möglich, so wie es in Zeile 1 der Fall ist. Bei einer einfachen Zuweisung zu einer Variablen, die schon vorher deklariert wurde, geht diese Schreibweise nicht.

4 Gültig – diese Schreibweise ist immer erlaubt, egal ob es sich bei der Zuweisung gleichzeitig um die Variablen-Deklaration handelt oder nicht.

5 Nicht gültig – die deklarierte Variable besitzt den Datentyp String, nicht den Datentyp String-Array. Deshalb können wir ihr auch kein Array zuweisen.

6 Gültig – Es wird ein zwei-dimensionales Array erzeugt, das Platz für 5 weitere Arrays hat. Die Inhalte dieser 5 Fächer sind hier allerdings noch nicht initialisiert! D.h. an den Stellen e[0], e[1] usw. Befindet sich die null-Referenz!

7 Gültig – die Position der eckigen Klammern bei der Variablen-Deklaration – also ob sie hinter den Datentyp oder hinter den Variablen-Namen geschrieben werden - ist nicht nur bei eindimensionalen Arrays frei wählbar. *Jedes* Paar an eckigen Klammern lässt sich an eine dieser Stellen schreiben. Bei einem zwei-dimensionalen Array gibt es also schon 3 verschiedene Schreibweisen für die Variablen-Deklaration: Beide Klammern hinter den Datentypen, beide Klammern hinter den Namen, oder wie es in diesem Beispiel der Fall ist. Dieses zwei-dimensionale Array hat 2 Fächer. Jedes Fach ist ein Array. Das Array in Fach eins hat wiederum ein Fach. Das Array in Fach zwei hat zwei Fächer.

**Aufgabe 3:**

In Video 39 haben wir die Summenfunktion durch Rekursion gelöst. Hier nochmal der Code:

```
static long berechneSumme(int zahl){
	if(zahl == 1){
		return 1;
	}
	return zahl + berechneSumme(zahl-1);
}
```

Zu diesem Zeitpunkt konnten wir diese Aufgabe nicht anders lösen. Später in Video-Block 7 haben wir allerdings die Schleifen kennengelernt. Ein alternativer Ansatz zur rekursiven Methode ist die sogenannte iterative Methode. Dabei ruft sich die Methode nicht selber auf - sie wird nur einmal aufgerufen und setzt wiederkehrende Logik in einer Schleife um.

Implementiere die Summenfunktion durch Iteration.

*Lösung:*

```
static long berechneSumme(int zahl) {
	long ergebnis = 0;
	while (zahl > 0) {
		ergebnis += zahl;
		zahl--;
	}
	return ergebnis;
}
```

Es gibt hier viele verschiedene Lösungen, der obige Code ist nur *ein* Beispiel.

Obwohl while-Schleifen üblicherweise suggerieren, dass zum Zeitpunkt des Beginns der Schleife die Anzahl der Durchgänge noch nicht feststeht (was hier ja nicht der Fall ist), habe ich mich gegen eine for-Schleife entschieden. Denn in diesem Fall ist die Logik aus der while-Schleife vermutlich intuitiver herauszulesen als es in einer for-Schleife der Fall ist:

```
static long berechneSumme(int zahl) {
	long ergebnis = 0;
	for (int i = zahl; i > 0; i--) {
		ergebnis += i;
	}
	return ergebnis;
}
```

Dies ist allerdings nur meine persönliche Empfindung. Das bedeutet, dass es keinesfalls falsch ist, eine for-Schleife zu verwenden, und es ist auch nicht pauschal „schlechter“. Wenn du findest, eine Lösung mit for-Schleife ist verständlicher, dann benutze auch eine for-Schleife.

*Anmerkung*: Generell gelten rekursive Varianten als eleganter, was vorallem durch die Kompaktheit des Codes kommt. In Fällen, bei denen aber der „Overhead“ durch eine rekursive Variante, d.h. die Menge an Rücksprungadressen, die während der Rekursion gespeichert werden müssen, kritisch werden kann, ist eine iterative Variante vorzuziehen. So ein Fall liegt hier bei einem großen Input-Wert vor! Bei for-Schleifen ist die Zählvariable (hier „i“) eine lokale Variable die bei jedem Schleifendurchlauf entfernt und neu angelegt wird. Es sammelt sich also keine große Menge Overhead im Speicher an. Analog bei der while-Schleife.

**Aufgabe 4:**

In einem der vorherigen Übungs-Blöcke gab es eine Aufgabe zur Quersummenberechnung. Die Funktionalität war zu diesem Zeitpunkt allerdings auf ausschließlich zweistellige Zahlen beschränkt.

Mit dem Wissen, dass du inzwischen hast, kannst du eine Quersummenberechnung für eine beliebig große (positive) Eingabe-Zahl vom Datentyp int implementieren.

Schreibe eine neue Methode *berechneQuersumme(int)* unter der Bedingung, dass – außer für die Aufsummierung der einzelnen Ziffern – **keinerlei weitere** mathematischen Operationen verwendet werden! Außer dem "+"-Operator ist also kein weiterer mathematischer Operator erlaubt, auch nicht "- " als Vorzeichen-Operator.

*Tipps:*

- Denke daran, dass der "+"-Operator neben der Addition noch eine weitere Bedeutung hat
- Siehe API-Doc Klassen „String" und „Integer" – vielleicht findest du etwas nützliches

*Lösung:*

```
static int berechneQuersumme(int i) {
	String s = i + ""; // oder: String s = String.valueOf(i);
	int result = 0;
	for (char c : s.toCharArray()) {
		int z = Integer.parseInt(c + "");
		result += z;
	}
	return result;
}
```

Aus mathematischer Sicht befinden wir uns unter den Bedingungen in der Aufgabenstellung in einer auswegslosen Situation. Wir brauchen also einen anderen Ansatz. Wie kann man eine Zahl noch interpretieren, wenn nicht als eben solche? Als Text!

Der “+“-Operator lässt uns eine Zahl in einen String konvertieren. Wir nutzen einfach den Trick, „nichts“ (einen leeren String) an die Zahl zu hängen – und zwar als Text. Damit erhalten wir die Zahl als String, da im Zuge einer Konkatenation alle Teilterme als Strings interpretiert werden, und als Ergebnis auch ein solcher zurückgegeben wird.

Ein Blick in die API für die Klasse String lässt uns die nützliche Methode String#toCharArray() finden. Diese zerlegt den String in seine einzelnen Zeichen.

**Vorsicht**: Auch wenn man mit diesen char-Werten schon rechnen könnte, da sie intern nichts anderes sind als int-Werte, so entpsrechen sie nicht den eigentlichen Zahlen! Der char-Wert entspricht nämlich dem Dezimal-Wert des *Zeichens* der Zahl in der ASCII Tabelle. Wir benötigen also das Zeichen als Zahl.

Da es keine direkte Methode gibt, die einen char in die entsprechende eigentliche int-Zahl umwandelt, müssen wir hier abermals den Wert erst in einen String umwandeln, und anschließend mittels Integer.parseInt(String) – oder auch Integer.valueOf(String) –den eigentlichen int-Wert herausziehen.