ÜBUNGS-BLOCK 2
LÖSUNGEN

Aufgabe N-1:

Die sogenannte *main-Methode* wird für jedes Java-Programm benötigt - denn ihre Implementierung, d.h. ihre Logik, *ist* das Programm. Natürlich muss damit jeder Java-Programmierer diese Methode kennen. In dieser Aufgabe sollst du die Definition der main-Methode nun üben, bis du sie aus dem Kopf korrekt niederschreiben kannst. Da sie recht komplex ist, gebe ich dir als Hilfestellung alle Bestandteile der main-Methode. Versuche, diese Teile nun in die korrekte Anordnung zu bringen:

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

Lösung auf der nächsten Seite...

Lösung:

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

Anmerkung:
Die main-Methode sieht in Wahrheit nur *üblicherweise* so aus, wie in den Videos und hier in der Lösung gezeigt. Es sind aber einige Änderungen an dieser Definition möglich:

1) Die Keywords **`public`** und **`static`** können auch in verkehrter Reihenfolge definiert werden (dies ist übrigens nicht nur bei der main-Methode der Fall, sondern bei allen Methoden).

2) Der Name des Parameters "args" ist frei wählbar. Er wird nur üblicherweise als "args" bezeichnet, da es die Abkürzung für "arguments" ist. Dieser Parameter stellt sogenannte Programmstart-Argumente dar.

3) Die Definition des Parameters kann auch so aussehen:

```
String args[]
```

oder so:

```
String... args
```

4) Als letztes könnte man noch ein weiteres Keyword in die Definition einbauen, nämlich das Keyword **`final`**

...damit wäre also beispielsweise folgende Definition der main-Methode genauso korrekt:

```
static public final void main(String... meinNameIstEgal) { }
```

Was nun diese ganzen Dinge, in denen sich diese Version von der obigen Lösung unterscheidet, eigentlich darstellen, dazu wirst du in den verschiedenen späteren Videos noch mehr erfahren. Ich kann dir aber jetzt schon sagen: Diese ganzen Dinge haben in diesem Fall keinerlei echten Effekt, ändern also überhaupt nichts an deinem Programm. Semantisch sind die gerade gezeigte Definition und die obige Definition also identisch!

Merke: Du solltest die main-Methode immer so definieren, wie in den Videos und anfangs in der Lösung gezeigt, denn das ist die Form, in der die main-Methode üblicherweise definiert wird.

Aufgabe 1:

Benutze für diese Aufgabe die Obst-Klasse aus dem vorangegangenen Übungsblock. Du kannst deine eigene Lösung benutzen oder die Musterlösung:

```
class Obst {

	String bezeichnung;
	int einkaufspreis; // in EUR
	int verkaufspreis; // in EUR
	int naehrwert; // in kcal

	Obst(String dieBezeichnung, int derEinkaufspreis, int derNaehrwert) {
		bezeichnung = dieBezeichnung;
		einkaufspreis = derEinkaufspreis;
		verkaufspreis = einkaufspreis + einkaufspreis;
		naehrwert = derNaehrwert;
	}

	void aufpeppen() {
		verkaufspreis = verkaufspreis + verkaufspreis;
		naehrwert = naehrwert + 50;
	}
}
```

Wir wissen inzwischen, dass diese Klasse nicht ganz korrekt designed ist. Passe die Klasse so an, dass sie sich an das Prinzip der Daten-Kapselung hält. Überlege dir, welche Funktionalität du bereit stellen solltest, d.h. welche Setter und Getter – insbesondere welche Access Modifier dafür - Sinn machen. Halte dir dabei ganz bewusst die Aufgabenstellung aus dem ersten Übungsblock vor Augen: Was will der Obsthändler mit der Software machen? Stelle dir vor, was für Daten oder Werte er später in der Software verändern können will, und wie er das tun will.

Lösung auf der nächsten Seite...

```java
public class Obst {

	private String bezeichnung;
	private int einkaufspreis; // in EUR
	private int verkaufspreis; // in EUR
	private int naehrwert; // in kcal

	public Obst(String dieBezeichnung, int derEinkaufspreis,
			int derNaehrwert) {
		setBezeichnung(dieBezeichnung);
		setEinkaufspreis(derEinkaufspreis);
		setVerkaufspreis(einkaufspreis + einkaufspreis);
		setNaehrwert(derNaehrwert);
	}

	private void setBezeichnung(String dieBezeichnung) {
		bezeichnung = dieBezeichnung;
	}

	public String getBezeichnung() {
		return bezeichnung;
	}

	private void setEinkaufspreis(int derEinkaufspreis) {
		einkaufspreis = derEinkaufspreis;
	}

	public int getEinkaufspreis() {
		return einkaufspreis;
	}

	private void setVerkaufspreis(int derVerkaufspreis) {
		verkaufspreis = derVerkaufspreis;
	}

	public int getVerkaufspreis() {
		return verkaufspreis;
	}

	private void setNaehrwert(int derNaehrwert) {
		naehrwert = derNaehrwert;
	}

	public int getNaehrwert() {
		return naehrwert;
	}

	public void aufpeppen() {
		setVerkaufspreis(verkaufspreis + verkaufspreis);
		setNaehrwert(naehrwert + 50);
	}
}
```

Kommentar zur Lösung:

Der Kerngedanke der Daten-Kapselung liegt darin, eine direkte Manipulation von den Werten eines Objekts von „außerhalb“, das bedeutet aus anderen Klassen, zu unterbinden. Das erreicht man durch das Setzen des Access Modifiers private für die Instanz-Variablen. Das ist das A und O dieser Übung und muss auf jeden Fall so verinnerlicht werden!

Ansonsten besagt die Konvention für Klassen: Konstruktoren und Methoden, sowie die Klasse selber, werden pauschal per Access Modifier public nach außen hin verfügbar gemacht, es sei denn, man hat einen guten Grund dagegen.

Oft hat man auf Grund der Anforderungen tatsächlich einen Grund dagegen, meist im Bezug auf die Setter. So auch hier: Die Setter sind private! Es wäre nämlich nicht logisch, dass der Obsthändler in seiner Software einen „Apfel“ plötzlich zu einer „Banane“ machen kann, also einen Setter für das Attribut „bezeichnung“ zur Verfügung hat. Funktionalität, die keinen Sinn macht, sollte man auch nicht implementieren. Das gleiche gilt für alle anderen Werte.

Zwar will der Händler das Obst aufpeppen können, aber dafür haben wir ja eine eigene Methode. Diese muss dann aber natürlich public sein.

Auf jeden Fall sollte man Setter auch dann schreiben, wenn man sie private macht. Man möchte die Abfragen für einen Wert (die eventuell auch erst später implementiert werden, oder geändert werden)an *einer einzigen zentralen Stelle* – nämlich ebem dem Setter – haben. Alle anderen Stellen in der Klasse, die Werte manipulieren wollen, rufen dann diese Setter auf – siehe Konstruktor und die aufpeppen() Methode. Bei letzterer wäre es aber kein wirklicher Fehler, die Werte direkt zu verändern, da hier keine Eingabe von außen (also kein Parameter) vorliegt. Aber es ist keineswegs falsch und einfach „doppelt“ sicher.

Anmerkung: In den Videos wird innerhalb der tunen() Methode der Wert einer Instanz-Variablen geändert, trotzdem rufen wir dort nicht den passenden Setter auf, sondern ändern ihn direkt. Das liegt allerdings daran, dass diese Methode ihre Abfrage nicht einfach an den Setter übergeben kann: Der Setter würde wohl nur abfragen, ob die neue Leistung größer 0 ist. Das wäre sie auch wenn man von 100 auf 80 PS „runtertuned“. Der Setter der Leistung hat keinen Zugriff auf den Parameter, mit dem die tunen() Methode aufgerufen wird. Die tunen() Methode muss also selber eventuelle Überprüfungen vornehmen, und daher wird eine Überprüfung der neuen Leistung durch den Setter hinfällig. Man *kann* natürlich in diesem Beispiel *nach der Prüfung des Tunewertes* die setLeistung() Methode aufrufen. Aber es ist an dieser Stelle einfach nicht mehr nötig.

Aufgabe 2:

Schreibe ein Programm, um die neue Obst-Klasse ein wenig zu testen.

1. Erstelle ein Obst Objekt und speichere es dir in eine Variable mit geeingetem Namen
2. Peppe das Obst auf!
3. Lass dir im Anschluss den aktuellen Nährwert für dieses Obst auf der Konsole ausgeben
4. Wiederhole die Teilschritte 1 und 2, lass dir diesmal aber im Anschluss nicht den Nährwert, sondern die Bezeichnung des Obstes auf der Konsole ausgeben

Lösung:

```
public class ObstTest {

	public static void main(String[] args) {

		Obst apfel = new Obst("Apfel", 39, 60);
		apfel.aufpeppen();
		System.out.println(apfel.getNaehrwert());

		Obst banane = new Obst("Banane", 45, 75);
		banane.aufpeppen();
		String bananeBezeichnung = banane.getBezeichnung();
		System.out.println(bananeBezeichnung);
	}
}
```

Kommentar zur Lösung:

Zuerst einmal solltest du dir eine neue Klasse (mit sinnvollem Namen – CapitalCamelCase) für die main-Methode erstellt haben! Wir wollen die Kerndaten der Software immer von der Software-Logik trennen. Es kann ja auch mehrere verschiedene Programme geben, die auf den gleichen Daten beruhen. In der Praxis wird eine Videospiel-Reihe auch nicht jedes Jahr komplett neu programmiert, die Klassen bleiben zum größten Teil die selben.

Wichtig bei der main-Methode: sie ist statisch! Das darf man niemals vergessen, denn wenn man das tut lässt sich das Programm nicht starten (Man hat dann kein Programm, nur eine sehr unsinnige Klasse).

Die Werte, die hier für die Obst Objekte gewählt wurden sind natürlich beliebig. Auch ist es relativ egal, ob man sich den jeweiligen Wert, den man ausgibt, vorher in einer Variable abspeichert (im Falle der Banane oben) oder nicht (im Falle des Apfels). In diesem kleinen Beispiel ist das vorherige Zwischenspeichern eigentlich unnötig – es wird hier nur der Vollständigkeit halber gezeigt.

Aufgabe 3:

Erweitere das Programm aus Aufgabe 2, indem du ein drittes Obst Objekt erstellst.

Dieses sollst du aber **nicht** in einer Variablen abspeichern! Überlege dir nun, inwiefern die Teilschritte 2. und 3. aus der Aufgabe 2 für dieses Objekt noch anwendbar sind. Geht so etwas in diesem Fall?

Vorsicht*:*

Deine spontane Antwort darauf ist vermutlich falsch. Die folgenden Tipps können dir helfen, die korrekte Antwort zu finden:

1. Es reicht nicht aus, nur das Video 08 (Objekte) gesehen zu haben, da es sich dabei nur um ein Beispiel handelt. Bestimmte Aussagen über die Rückgabe (des Objektes aus dem Konstruktor) treffen, so wie sie in dem Video gesagt werden, nicht verallgemeinert zu.
2. Um das besser zu verstehen musst du auch das Video Video 11 (Getter & Setter) gesehen haben. Hier werden auch Methoden mit einem Rückgabewert vorgestellt und benutzt.
3. Achte in diesem Video einmal darauf, wie die Getter in einem System.out.println() aufgerufen werden. Überlege dir, wohin genau im Code die Rückgabe der Getter erfolgt.
4. Ein Konstruktor ist ist im Bezug auf die Rückgabe nichts anderes als eine Methode mit dem speziellen Rückgabewertes des Objektes, das er erzeugt.
5. Etwas Herumexperimentieren mit dem Code kann immer neue Kentnisse bringen!
   Tipp: Solange Eclipse keinen Fehler in deinem Code anzeigt (rote Unterringelung), ist die Syntax gültig

Lösung auf der nächsten Seite...

Lösung:

Vorbemerkung: Es ist absolut verständlich, wenn du nicht auf die richtige Lösung gekommen bist – davon wurde in dieser Übung ausgegangen! In der Tat fehlt es dir im Moment noch an ein paar Informationen, um sich die Antwort zusammenreimen zu können. Du solltest dir einfach ein paar Gedanken dazu machen, und versuchen zumindest teilweise hinter die Lösung kommen. Eigenständiges „Herumspielen“ am Code hilft, den Stoff besser zu verstehen.

Wichtig ist jetzt aber eigentlich nur, dass du die folgende Antwort verstehst. Sie lautet:

Jein...

Es ist nicht so, dass man mit einem Objekt gar nichts tun kann, wenn man keine Variable hat! Aber man kann in so einem Fall nur solange damit arbeiten, solange das Objekt „vor der Nase“ ist. Das bedeutet: Nur im Zuge des Erstellen des Objektes, d.h. solange man sich noch in der Anweisung befindet, in der der Konstruktor aufgerufen wurde.

Folgendes geht durchaus:

```
new Obst(...).aufpeppen();
```

Denn der Konstruktor liefert das Objekt, das dann zur Laufzeit anstelle von new Obst(...) im Code steht. Und man kann nicht nur mit einer Variablen per Punkt-Notation arbeiten, sondern auch direkt mit dem Objekt. Genau das passiert nämlich auch im Falle der Variablen: Es wird erstmal das Objekt aus der Variablen geholt. Wie man sich das genau vorstellen kann, sehen wir später im Kurs.

Deswegen funktioniert also der Aufruf der aufpeppen() Methode, auch ohne Variable, in die wir das Objekt speichern. Das gleiche gilt übrigens auch für die Ausgabe des Nährwerts. Auch das würde funktionieren:

```
System.out.println(new Obst(...).getBezeichnung());
```

Man kann also durchaus den Teilschritt 2 *oder* 3 auf dem zweiten Obst Objekt ausführen, aber *nicht beide*, denn: Im Gegensatz zum Konstruktor-Aufruf, der ein Objekt zurückliefert, auf dem wir noch aufpeppen() oder getBezeichnung() aufrufen können, liefern diese beide Methoden kein Obst-Objekt: aufpeppen() ist als void definiert, liefert also gar nichts. getBezeichnung() liefert zwar etwas, aber kein Obst, sondern einen String. Und einen String können wir nicht aufpeppen. Man kann also nur solange mit einem Objekt arbeiten, solange man im Kontext des Objekts steckt, d.h. solange der gesamte Term ein Obst-Objekt als Rückgabe liefert.

Aufgabe N-2:

Gegeben ist folgende Klasse, welche (unüblicherweise) sowohl ein Objekt designed, als auch ein Programm enthält:

```
public class Auto {

    public static void main(String[] args) {
       Auto volkswagen = new Auto(85, "Volkswagen");
    }

    private int leistung;
    private String hersteller;

    Auto(int dieLeistung, String derHersteller) {
       leistung = dieLeistung;
       hersteller = derHersteller;
       Auto bmw = new Auto(100, "BMW");
       Auto mercedes = new Auto(150, "Mercedes");
    }

}
```

Wie viele Auto-Objekte werden in diesem Programm erzeugt? Was sind jeweils die Werte für Leistung und Hersteller all dieser Auto-Objekte?

Lösung auf der nächsten Seite...

Lösung:

Diese Aufgabe war in gewisser Weise eine Fangfrage! Denn wenn man dieses Programm ausführt, stürzt es mit einer Fehlermeldung ab:

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

Dieses Verhalten hättest du natürlich nicht konkret vorhersagen können - es hätte dich aber durchaus stutzig machen sollen, dass innerhalb des Konstruktors, der ja für die Erzeugung eines Autos zuständig ist, wiederum der Konstruktor selbst aufgerufen wird. Was könnte das bedeuten?

Es bedeutet: Der Code resultiert in einer Endlosschleife! Das Programm schachtelt sich so tief in Konstruktor-Aufrufe hinein, dass es irgendwann nicht mehr weiß, wo der ursprüngliche Aufruf herkam (StackOverflowError bedeutet, dass er keinen Speicherplatz mehr für die Speicherung der Rücksprungadressen hat).

Für die Anzahl an Objekten, die erzeugt werden, könnte man demnach annehmen, dass überhaupt kein Objekt erzeugt wird, da schon der ursprüngliche Konstruktor-Aufruf aus der main-Methode niemals vollständig ausgeführt wird und daher niemals endet. Dies ist aber rein technisch gesehen nicht richtig, denn ein Objekt ist aus Sicht des Speichers nur eine Menge an Werten für Instanz-Variablen. Ob dieses Objekt jemals vom Konstruktor zurückgeliefert und abgespeichert werden kann, spielt dafür keine Rolle. Da die Deklaration der Variablen sowie die Zuweisung von Werten zu ihnen bereits vor dem erneuten Konstruktor-Aufruf im Konstruktor passiert, werden also folgende Objekte erzeugt:

Ein Auto[85, "Volkswagen"]
<x> mal ein Auto[100, "BMW"]

<x> kann dabei varriieren und hängt von der Größe und Belegung des Stacks ab. Irgendwann wird der Stack-Speicher verbraucht sein und das Programm stürzt ab.

Beachte, dass niemals ein Auto[150, "Mercedes"] erzeugt wird, denn an diese Zeile gelangt das Programm nie!

PS: Das Ganze hat übrigens nichts damit zu tun, dass die main-Methode nicht in einer separaten Klsase definiert ist. Das spielt für dieses Programm keine Rolle und wurde hier nur der Einfachheit halber so gemacht.

Aufgabe N-3:

Angenommen, du arbeitest in einem Team an einem größeren Projekt, und bist mit der Implementierung einer bestimmten Klasse beauftragt, die einen wichtigen Objekt-Typen innerhalb der gesamten Software darstellt.

Diese Klasse hast du bereits ganz zu Beginn des Projekts erstellt, damit der Datentyp der Klasse für andere Teile der Software bekannt ist. Deine Kollegen nutzen also deine Klasse, in dem sie beispielsweise Variablen und Parameter des Typs der Klasse in ihren Codeteilen verwenden.

Allerdings musst du die Logik innerhalb der Klasse erst noch ausimplementieren, z.B. was genau in den Rümpfen von Instanz-Methoden steht, oder mit welchen Werten gewisse Instanz-Variablen initialisiert werden sollen. Weil es eventuell zu schlimmen Fehlern führen kann, wenn man Objekte dieser Klasse in dem jetzigen Zustand erzeugt und damit arbeitet, möchtest du nicht, dass deine Kollegen sich Objekte der Klasse anlegen.

Problem: Deine Kollegen können sich das einfach nicht merken, oder erzeugen aus Versehen Objekte deiner Klasse. Und dann beschweren sie sich bei dir, dass das Programm mit irgendwelchen Fehlermeldungen abstürzt.

Wie könntest du dieses Problem beheben? Wie kannst du also eine Klasse bereit stellen, deren Datentyp zwar in der ganzen Software bekannt ist, von der deine Kollegen aber kein Objekt erzeugen können?

Lösung:

1) Implementiere einen eigenen Konstruktor, sodass der Default-Konstruktor nicht mehr existiert
2) Setze den Access Modifier des neuen Konstruktors auf **`private`**

```
public class MyClass {

    private MyClass(){
      // ...
    }

    // ...
}
```

Damit ist es nicht mehr möglich, aus anderen Klassen heraus ein Objekt deiner Klasse zu erzeugen, denn es existiert kein sichtbarer Konstruktor mehr. Für deine Klassen-internen Tests kannst du den Konstruktor aber nach wie vor verwenden, denn innerhalb der eigenen Klasse ist immer alles sichtbar, auch wenn es private ist.