ÜBUNGS-BLOCK 4
LÖSUNGEN

Aufgabe 1:

Welche Bedeutung haben Packages in Java, und worauf muss man bei ihrer Verwendung achten?

Lösung:

Packages können drei Zwecken dienen:

1. Die Strukturierung / Ordnung von Quellcode-Dateien. Um effektiver arbeiten zu können, und damit man in komplexen Projekten die Übersicht behält, macht es Sinn, inhaltlich aufeinander bezogene Klassen in verschiedene Packages zu legen.
2. Die Verwendung von Packages kann auch im Hinblick auf das Design eine Rolle spielen: Dadurch, dass manche Access Modifier sich in verschiedenen Konstellationen von Klassen in Packages anders verhalten, kann der Zugriff auf Daten für verschiedene Klassen unterschiedlich ausfallen.
3. Als letztes dienen Packages dazu, den Dateien einen eindeutigeren Namen zu geben, was Namenskonflikten bei gleichnamigen Dateien vorbeugt. So gibt es beispielsweise zwei Datenstrukturen mit dem Namen "List" in der Java Bibliothek. Dadurch, dass sie aber in unterschiedlichen Packages enthalten sind (java.awt, und java.util), können wir diese beiden bei der Nutzung (z.B. beim import-Statement) voneinander unterscheiden.

Beachten muss man bei der Verwendung von Packages:

1. Jede Klasse innerhalb eines packages muss die package-Anweisung enthalten. Dies muss die erste Zeile im Code sein (ausgenommen Kommentare)
2. Packages sind in Java nicht nur eine logische Struktur, sondern sie spiegeln sich auch physikalisch in der Ordnerstruktur wider. D.h. eine Klasse in einem package "mypackage" muss sich innerhalb des src-Ordners in einem Unterordner namens "mypackage" befinden
3. Wann immer eine Klasse eine andere Klasse verwenden möchte, die sich in einem anderen Package befindet, muss diese Klasse über eine entsprechende import-Anweisung bekannt gemacht werden.

Aufgabe 2:

Gegeben sind die folgenden Zeilen Code:

```
1      int a = 5;
2      double b = (double)a;
3      float c = b;
4      long d = a;
5      byte e = (short) 10;
6      long f = 2999999999999;
7      double g = c;
8      byte h = e + e;
```

a) Welche dieser sind Anweisungen gültig, welche nicht? Wo findet implizites Casting statt? Im Falle der expliziten Castings sollte zudem entschieden werden, inwiefern sie überhaupt nötig sind.

b) Worauf muss man bei explizitem Casting achten?

Lösung:

a)

Zeile 1: Nichts besonderes, natürlich gültig ;)

Zeile 2: Gültig, aber explizites Casting von int nach double nicht nötig, würde auch implizit geschehen

Zeile 3: Nicht gültig, explizites Casting nach float nötig, da Wertebereich float < Wertebereich double

Zeile 4: gültig, implizites Casting von int nach long

Zeile 5: gültig, aber es wird durch das explizite Casting unnötigerweise erst von int nach short, und dann erst nach byte gecastet. Ohne das explizite Cast würde direkt von int nach byte gecastet werden (implizit)

Zeile 6: Ungültig, der Wert wird als int interpretiert und übersteigt den Wertebereich von int. Ein Casting in dem Sinne ist hier nicht möglich, da der Fehler schon darin besteht, dass ein ungültiges int-Literal definiert wurde. Man muss hier ein „l“ oder „L“ an die Zahl hängen, um das Literal zu einem long zu machen. Damit wäre kein weiteres Casting mehr nötig.

Zeile 7: Gültig, implizites Cast von float nach double

Zeile 8: **Ungültig**! Achtung dieses Beispiel ist besonders trickreich: Die Variable „e“ ist zwar vom Typ byte, aber: Genauso wie ganzzahlige Literale im Code als int interpretiert werden, so ist auch das Ergebnis einer Berechnung zweier ganzen Zahlen immer ein int! Daher wäre hier ein explizites Casting (mit Klammerung!) nötig:

```
byte h = (byte)(e + e);
```

b)

Bei explizitem Casting kann es zu einem Überlauf/Unterlauf des Wertebereiches kommen. Wenn der zu castende Wert zu groß oder zu klein ist, wird die Differenz dadurch verarbeitet, dass einfach wieder beim Minimum/Maximum angefangen wird.Eventuell kann es mehrere komplette Durchläufe durch den Wertebereich geben. So kann es zum Beispiel bei dem Casting von einem sehr großen long-Wert zu einem int-Wert kommen, der nicht mehr dem ursprünglichen long-Wert entspricht.

Aufgabe N-1:

Sieh dir mal folgendes Programm an:

```
public static void main(String[] args) {
   int i = 10;
   int j = 3;
   int erg1 = i / j;
   double erg2 = i / j;
   double erg3 = (double) i / j;
   System.out.println("erg1 = " + erg1);
   System.out.println("erg2 = " + erg2);
   System.out.println("erg3 = " + erg3);
}
```

Was meinst du wird das Programm bei der Ausführung für *erg1* - *erg3* ausgeben?

Lösung:

```
erg1 = 3
erg2 = 3.0
erg3 = 3.3333333333333335
```

Die erste Berechnung ist eine Division auf zwei ganzzahligen Zahlentypen (int), und soetwas liefert immer eine ganze Zahl als Ergebnis; wenn die Rechnung rein mathamtisch nicht aufgeht, werden jegliche Nachkommastellen einfach abgeschnitten (Achtung: Es wird NICHT ab-/aufgerundet!)

Die zweite Berechnung liefert zunächst das selbe Ergebnisse wie die erste Variante, aus dem selben Grund. *Nach* dieser Berechnung wird der Wert zu einer Fließkommazahl umgewandelt. Aus der ganzen Zahl 3 wird somit die Fließkommazahl 3.0 - die Berechnung war aber rein mathematisch noch immer nicht korrekt.

Die dritte Berechnung ist nun tatsächlich korrekt. Dadurch, dass wir einen der Terme, nämlich die Variable i, zu double casten, *bevor* wir die Division durchführen, wird auch auf Fließkommazahlen gerechnet, d.h. das Ergebnis ist auch eine Fließkommazahl mit dem korrekten mathematischen Wert (welcher allerdings nur angenähert ist).

Anmerkung: Es spielt hier aber keine Rolle, welcher Wert genau zu double gecastet wird. Man könnte genauso gut auch j casten: `i / (double) j;`

Aufgabe N-2:

Erstelle eine Klasse namens **ProzentRechnung** und implementiere sie so, dass gilt:

• Beim Erzeugen eines ProzentRechnung-Objekts muss ein beliebiger ganzzahliger Wert (Typ int) übergeben werden (wir gehen davon aus, dass nur Werte > 0 übergeben werden)

• Anschließend kann man zwei Methoden auf dem Objekt aufrufen: **wertZuProzent**, und **prozentZuWert**

• Beide Methoden nehmen einen int-Wert als Eingabe

• **wertZuProzent** liefert den prozentuellen Anteil des übergebenen Wertes an der Richtgröße, mit der das Objekt initialisiert wurde

• **prozentZuWert** interpretiert den übergebenen Wert als prozentuellen Wert an der Richtgröße, mit der das Objekt initialisiert wurde, und errechnet den entsprechenden absoluten Wert

Ein Beispiel-Programm, in dem diese Klasse verwendet wird, könnte dann also so aussehen:

```
public static void main(String[] args) {
   ProzentRechnung pr = new ProzentRechnung(500);
   System.out.println(pr.wertZuProzent(1000));
   System.out.println(pr.wertZuProzent(23));
   System.out.println(pr.prozentZuWert(50));
   System.out.println(pr.prozentZuWert(1));
}
```

Die Ausgabe würde lauten:

```
200.0 // 1000 entspricht 200% von 500
4.6 // 23 entspricht 4.6% von 500
250.0 // 50% von 500 sind 250
5.0 // 1% von 500 sind 5
```

Lösung auf der nächsten Seite...

Lösung:

```
public class ProzentRechnung {

    private double richtGroesse;

    public ProzentRechnung(int richtGroesse) {
      this.richtGroesse = richtGroesse;
    }

    public double prozentZuWert(int prozent) {
      return (richtGroesse * prozent) / 100.;
    }

    public double wertZuProzent(int wert) {
      return (wert / richtGroesse) * 100.;
    }
}
```

Anmerkung:

Die im Konstruktor übergebene Richtgröße müssen wir uns in einer Instanz-Variablen abspeichern, da sie in den Methoden bekannt sein muss, dort aber nicht als Parameter übergeben wird.

Obwohl der übergebene Wert vom Typ int ist, können wir die Instanz-Variable als double deklarieren, d.h. bei der Zuweisung im Konstruktor wird der übergebene Wert bereits implizit zu einem double gecastet. Dadurch müssen wir in den Berechnungen der Methoden nicht mehr explizit nach double casten, um korrekte Berechnungen zu erhalten.

Wenn die Instanz-Variable *richtGroesse* auch mit int typisiert wäre, müsste man die Berechnungen natürlich mit einem expliziten Cast durchführen, um korrekte Ergebnisse zu erhalten:

```
    public double prozentZuWert(int prozent) {
      return ((double)richtGroesse * prozent) / 100.;
    }

    public double wertZuProzent(int wert) {
      return ((double)wert / richtGroesse) * 100.;
    }
```

Beachte, dass in der Lösung auch das Literal 100 explizit über den nachfolgenden Punkt zu einem double gecastet wird. Dies ist zwar eigentlich nicht nötig, da der linke Teilterm jeweils bereits ein double ist, und dieser Cast sowieso implizit durchgeführt werden würde. Allerdings verdeutlicht es noch einmal, dass wir hier mit Fließkommazahlen rechnen möchten.

Aufgabe 3:

Gegeben sei folgendes (unglaublich unsymphatisches) Programm:

```
public class OhMyGod {

	public static void main(String[] args) {

		int a = 1;
		int b = a + ++a;
		int c = b++ - a + a--;

		/* BRAIN COOL-DOWN HERE */

		System.out.println(c++ + a++);
		System.out.println(--b - (-++c));
		System.out.println(a - c);
	}
}
```

Frage - Du ahnst es schon: Was gibt das Programm aus?

Wichtig: Du sollst versuchen, möglichst viel im Kopf zu berechnen. Nach den ersten drei Zeilen darfst du dir die Werte, die du dir bis dahin für a,b und c im Kopf ausgerechnet hast, auf einen Zettel schreiben. **Aber nicht davor!** Die letzten drei Zeilen sollst du dann ausgehend von den notierten Zwischenwerten wieder im Kopf berechnen, ohne weitere Zwischennotizen.

Anmerkung: Auch wenn diese Aufgabe wie pure Schikane erscheint und absolut künstlich konstruiert ist: Du trainierst damit deine Konzentrationsfähigkeit. Konzentration beim Programmieren ist unglaublich wichtig – du sollst merken, wie schnell sich ein Fehler einschleichen kann, wenn man nicht voll bei der Sache ist! Außerdem trainiert so etwas deine Debugging Skills. Das bedeutet, du kannst Fehler im Code (und die wird es immer wieder geben) schneller auffinden und beseitigen, wenn du es gewohnt bist viele Daten gleichzeitig im Kopf zu halten. Sieh es als Herausforderung! Rechne zwei oder drei mal nach, bevor du dich für eine entgültige Lösung entschließt. Wenn du die Aufgabe unter den gestellten Bedingungen lösen kannst, hast du wirklich eine großartige Leistung vollbracht!

Viel ~~Spaß~~ Erfolg ;)

Lösung:

Die Ausgabe lautet:

```
4
8
-3
```

Lass dich nicht demotivieren falls du nicht auf die richtige Lösung gekommen bist! Es geht auch nicht darum, dass man als guter Programmierer soetwas fehlerfrei im Kopf lösen können muss. Wer sich für die Aufgabe nicht gute 10 Minuten Zeit nimmt wird wahrscheinlich einen Fehler machen, egal wie gut er programmieren kann. Es war einfach eine kleine Übung für den Kopf.

Wichtig ist nun aber, dass du genau verstehst, wie diese Werte zustande kommen:

			a	b	c
`int a = 1;`	a =	1	1	-	-
`int b = a + ++a;`	b = 1 + 2 =	3	2	3	-
`int c = b++ - a + a--;`	c = 3 - 2 + 2 =	3	1	4	3
`System.out.println(c++ + a++);`	(3 + 1) =	**4**	2	4	4
`System.out.println(--b - (-++c));`	(3 - (-5)) =	**8**	2	3	5
`System.out.println(a - c);`	(2 – 5) =	**-3**	2	3	5

Aufgabe 4:

Im letzten Übungsblock gab es eine Aufgabe, bei der einem Kunden im Zuge der Instanz-Initialisierung eine ID zugewiesen werden sollte. Zur Erinnerung hier nochmal die Musterlösung:

```
public class Kunde {

	private static int idCounter = 1;

	private final int id;
	{
		id = idCounter;
		idCounter = idCounter + 1;
	}

	public int getId() {
		return id;
	}
}
```

Mit dem im Video-Block 4 neu erworbenen Wissen kannst du diese Lösung optimieren, indem du den Code etwas kürzt, die eigentlich Logik aber noch in der selben Form beibehälst. Siehst du, wie?

Lösung:

Gemeint war hier die Verwendung des Inkrement-Operators. Dieser gestattet es, eine Zuweisung im Zuge der Erhöhung der Variable zu machen. Man kann sich somit den gesamten Initialisierungsblock sparen und statt dessen einfach folgendes schreiben:

```
private final int id = idCounter++;
```

Aufgabe 5:

Du kennst sicherlich den Begriff der *Quersumme* aus der Mathematik: Die Quersumme einer Zahl ist die Summe aller Ziffern dieser Zahl. Beispiel:

5 Quersumme = 5
11 Quersumme = 1 + 1 = 2
923 Quersumme = 9 + 2 + 3 = 14

Implementiere folgende Methode:

```
public static int berechneQuersumme(int zahl){ /* ... */ }
```

in einer geeigneten Klasse, sodass sie das tut, wonach sie klingt.

Du kannst davon ausgehen, dass die Methode immer mit einer **positiven, zweistelligen** Zahl (10 <= zahl <= 99) aufgerufen wird. Für alle anderen Zahlen muss diese Methode **nicht** ein korrektes Ergebnis liefern!

Implementiere in deiner Klasse zudem die main-Methode, um deine berechneQuersumme Methode mit ein paar Zahlen zwischen 10 und 99 zu testen.

Tipp: Das Problem lässt sich durch geschicktes „Herumspielen“ an der übergebenen Zahl lösen. Es ist ein wenig mathematische Kreativität gefragt! Überlege dir, was Java so alles an Möglichkeiten für die Manipulation von Zahlen zu bieten hat.

Lösung:

```
public static int berechneQuersumme(int zahl) {
      double shift = zahl / 10.0;
      int ersteZiffer = (int) shift;
      int zweiteZiffer = zahl - 10 * ersteZiffer;
      return ersteZiffer + zweiteZiffer;
}
```

Zeile 1: Wenn man eine zweistellige Zahl durch 10 dividiert, wird diese Zahl in eine Fließkommazahl geteilt, bei der die erste Ziffer vor dem Komma, und die zweite Ziffer hinter dem Komma steht.

Zeile 2: In dieser Zeile liegt der Knackpunkt zur Lösung der Aufgabe: Bei einem Cast von double nach int in Java werden alle Nachkommastellen einfach verworfen. Da wir die zweite Ziffer der Zahl in den Nachkommastellen-Bereich geschoben haben, erhalten wir so nur die erste Ziffer der ursprünglichen Zahl.

Zeile 3: Sobald wir die erste Ziffer extrahieren konnten ist es nicht mehr so schwer, die zweite Ziffer herauszufinden. Wir ziehen einfach den „Zehner“-Wert von der Zahl ab, und erhalten als Rest nur noch einen einstelligen Wert, nämlich die zweite Ziffer.

Zeile 4: Ziffer 1 + Ziffer 2 = Quersumme