ÜBUNGS-BLOCK 5
LÖSUNGEN

Aufgabe 1:

In dem Video zu den komplexen Datentypen aus dem Video-Block 5 haben wir die Klasse „Color“ gesehen. Du weisst jetzt, wo du dir weiterhelfen kannst, wenn du zu dem Umgang mit Ressourcen aus der Java-Bibliothek Fragen hast.

Schreibe ein Programm, dass:

1. In einer Varaiblen auf ein Color-Objekt der Farbe gelb referenziert
2. Die Farbkanäle dieses Objekts auf der Konsole ausgibt, im Format:
   <R> - <G> - <B>
3. Anschließend den Farbton etwas dunkler macht
4. Am Ende nochmals die Farbkanäle auf der Konsole ausgibt

Lösung:

```
public class ColorDemo {

	public static void main(String[] args) {

		Color gelb = new Color(255,255,0);

		System.out.println(gelb.getRed()+" - "+gelb.getGreen()
		                                   +" - "+gelb.getBlue());

		gelb = gelb.darker();

		System.out.println(gelb.getRed()+" - "+gelb.getGreen()
		                                   +" - "+gelb.getBlue());
	}
}
```

Anmerkung zur Lösung:

Natürlich war hier die Übung, sich mit der API-Doc zu beschäftigen. Niemand kennt alle Klassen auswendig, und man muss nicht alles wissen, man muss nur wissen, wo es steht. Der Umgang mit der API-Doc und das Herausfinden von Informationen, nach denen man sucht, ist daher sehr wichtig für jeden Java-Programmierer.

Nun konkret zu dem Code aus der Lösung:

Bei der Erzeugung und Initialisierung der Variablen gibt es neben dem Weg des Konstruktor-Aufrufs auch die Möglichkeit, eine Konstante aus der Color-Klasse dafür zu nutzen. Wenn man sich die Dokumentation der Klasse ansieht, wird man eine solche nämlich für die gängigsten Farben finden:

```
Color gelb = Color.YELLOW;
```

Wie du vielleicht auch gesehen hast, gibt es für all diese Konstanten noch eine zweite Variante, deren Namen sich nicht an die Namenskonvention für Konstanten hält:

```
Color gelb = Color.yellow;
```

Das kommt daher, dass es zu der Zeit, als diese Klasse geschrieben wurde, diese Konvention noch gar nicht gab. Später wurden die weiteren Variablen mit korrekter Namenskonvention hinzugefügt, die alten Versionen blieben aber erhalten damit Programme, die diese nutzen, nicht mit dem nächsten Java-Release plötzlich nicht mehr kompilieren.

Rein logisch gesehen ist es egal, welche Variante man nutzt – es stecken in beiden Fällen Objekte mit identischen Werten dahinter. Allerdings sollte man die neueren Versionen nutzen, wenn sie schon angeboten werden...

Die Aufrufe der Getter sollten sich verstehen, das erste, was man tut wenn man etwas von einem Objekt abfragen will: Nach Gettern suchen!

Der Kern dieser Aufgabe besteht nun in der Teilaufgabe, die Farbe zu verdunkeln. Wenn man die Klassen-Dokumentation nach geeigneten Methoden durchsucht, stösst man auf die darker() Methode.

ABER: Wenn man genau hinsieht, sieht man, dass sie etwas returned: Ein Color-Objekt. Das sollte einen schon stutzig machen. In der Beschreibung der Methode steht dann noch einmal ganz deutlich:

> **Returns:**
> a new `Color` object that is a darker version of this `Color`.

Das “new” ist hier ausschlaggebend: Das eigentliche Color-Objekt wird also gar nicht verändert. Daher müssen wir uns das neue Objekt wieder in unsere Variable abspeichern, um deren Inhalt verändert zu haben.

Ansonsten hat man den Effekt des Call By Value, nämlich dass eben das eigentliche Objekt gar nicht verändert wird, sondern eine neue Referenz zurückgegeben wird. Wenn man einfach nur so etwas schreibt:

```
gelb.darker();
```

würde die Ausgabe beim zweiten System.out.println() nach wie vor die selbe sein wie vor dem Methodenaufruf.

Aufgabe N-1:

Sieh dir mal folgenden Code an:

```
public static void main(String[] args) {
    int x = 0;
    int y = 0;
    initializeXY(x, y);
    System.out.println("x=" + x + ", y=" + y);
}

private static void initializeXY(int x, int y) {
    x = 5;
    y = 8;
}
```

Wenn du in den Videos gut aufgepasst hast, dann solltest du wissen, dass die Ausgabe hier x=0, y=0 lautet, und **nicht** x=5, y=8, wie es der Programmierer hier wohl erwartet hatte.

Der Grund: Java übergibt nicht die Variablen selbst in eine Methode, sondern *kopiert* lediglich deren *Werte* in neue Variablen - nämlich die, die in der Parameterliste der Methode deklariert sind. Wie du weißt nennt sich dieser Übergabemechanismus Call By Value.

Das Gegenstück dazu nennt sich *Call By Reference* - dabei werden tatsächlich die Referenzen auf die ursprünglichen Variablen übergeben, und neue Zuweisungen zu diesen Variablen haben einen nachhaltigen Effekt, auch außerhalb der Methode. Der Programmierer, der diesen Code geschrieben hat, dachte wohl, dass Java mit Call By Reference arbeiten kann, so wie es in anderen Programmiersprachen (z.B. C) der Fall ist - stimmt aber nicht, es gibt nur Call By Value in Java!

Deine Aufgabe:

Finde einen Weg, um nun *trotzdem* den Effekt des Call By Reference in Java zu "simulieren". Es geht also darum, dass in obigem Beispiel dann tatsächlich die Werte 5 und 8 ausgegeben werden, und das unter der Bedingung, dass:

1) weiterhin die initializeXY-Methode genutzt wird, um die Werte zu setzen
2) Die initializeXY Methode nach wie vor void bleibt, also *nichts zurückgibt*!

Das einzige, was du an der Methode verändern darfst - und das ist nun der **Tipp** den ich dir gebe: Du darfst die Parameterliste der Methode verändern! Entsprechend darfst/musst du dann natürlich auch in der main-Methode Änderungen vornehmen.

Diese Aufgabe ist ganz schön schwierig, aber ich bitte dich darum, dir Zeit zu nehmen und darüber nachzudenken, wie man den gewünschten Effekt irgendwie realisieren könnte. Denke "out of the box" - verbeiße dich nicht in den gegebenen Code. Es gibt auch soetwas wie "außerhalb dieses Codes". Werde kreativ! Viel Spaß beim Grübeln :)

PS: Okay, *einen* Tipp gibt's doch noch: Was war nochmal der Unterschied zwischen einer Zuweisung und einer De-Referenzierung...

Lösung:

Wir wissen, dass eine Neuzuweisung zu einem übergebenen Wert keine nachhaltige Änderung hat. Aber: Wir wissen, dass Werteänderungen über eine *De-Referenzierung* durchaus nachhaltig sind (siehe Video "Referenzen").

Der Grundgedanke der Lösung lautet also: Packe die Werte für x und y in ein *Objekt*:

```
public class XY {
    public int x;
    public int y;
}
```

Das Programm können wir damit wie folgt abändern:

```
    public static void main(String[] args) {
       XY xy = new XY();
       initializeXY(xy);
       System.out.println("x=" + xy.x + ", y=" + xy.y);
    }

    private static void initializeXY(XY xy) {
       xy.x = 5;
       xy.y = 8;
    }
```

Und das kommt dem Effekt des Call By Reference doch durchaus recht nahe. Es ist also durchaus möglich, eine Art Quasi-Call By Reference in Java zu erreichen - und zwar indem man die Daten in Objekte kapselt, und dann die Zuweisungen über De-Referenzierung macht.

...Wenn du nicht auf diese Lösung gekommen bist: Absolut verständlich! Es ging primär um den *Versuch* zur Lösung. Wenn du aber an soetwas wie ein "Objekt" gedacht hast, dann war das schon alles was wirklich erwartet wurde! Und wenn du die Lösung tatsächlich so umsetzen konntest, dann kannst du mächtig stolz auf dich sein!

PS: Noch eine Anmerkung zu der Lösung. Dass die Instanz-Variablen x und y in der XY-Kalsse als public deklariert sind, ist wie wir wissen kein gutes Design. Besser wäre, sie zu verstecken und entsprechende Getter und Setter zu nutzen. Der direkte Zugriff wurde hier nur erlaubt, da damit die Ähnlichkeit zu dem ursprünglichen Code höher ist, und es somit etwas deutlicher wird, dass man das als eine Art Call By Reference verstehen kann.

Aufgabe 2:

Gegeben sind folgende Klassen:

```
public class KlasseA {

	private String sa;

	public KlasseA(String sa) {
		setSa(sa);
	}

	public String getSa() {
		return sa;
	}

	public void setSa(String sa) {
		this.sa = sa;
	}
}

public class KlasseB {

	private KlasseA a;
	private String sb;

	public KlasseB(KlasseA a) {
		setA(a);
		setSb(a.getSa());
	}

	public String getSb() {
		return sb;
	}

	public void setSb(String sb) {
		this.sb = sb;
	}

	public void setA(KlasseA a) {
		this.a = a;
	}
}
```

Sowie das folgende Programm:

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

		KlasseA a = new KlasseA(new String("A1"));
		// 1
		KlasseB b = new KlasseB(new KlasseA(new String("A2")));
		// 2
		a = null;
		// 3
		b.setSb("B");
		// 4
		b.setA(null);
		// 5
	}
}
```

Im Programm ist jede zweite Zeile mit einem Kommentar versehen. Für jeden Zeitpunkt, an dem die Ausführung des Programms an einer kommentierten Zeile angelangen würde, sollst du eine Skizze des Speicher- und Referenzmodells zu diesem Zeitpunkt malen.

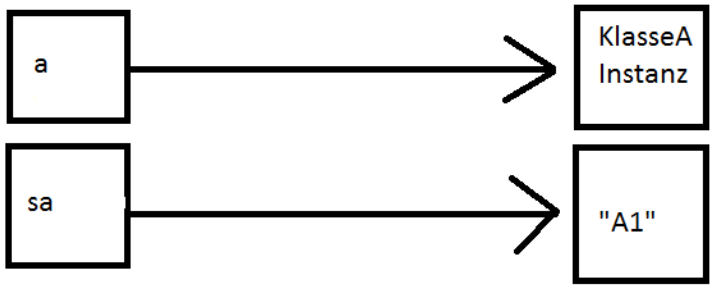
Orientiere dich an den Skizzen die ich dazu in den Videos gezeichnet habe. Die Skizzen soll für diesen Zeitpunkt also jeweils aufzeigen:

1) Welche Objekte es im Hauptspeicher gibt
2) Welche Variablen es gibt
3) Welche Referenzen (Zeiger von Variablen auf Objekte) es gibt

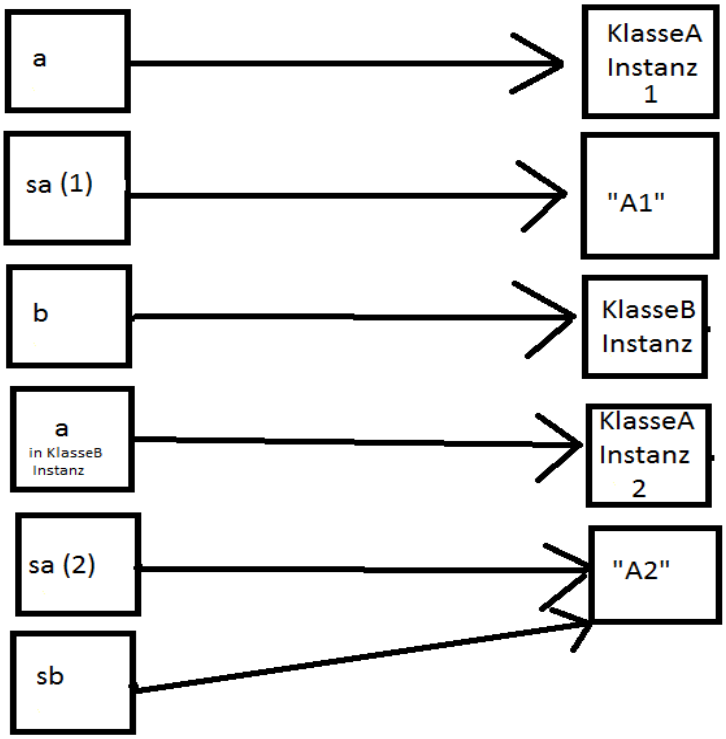
Anmkerung: Es soll für diese Aufgabe davon ausgegangen werden, dass der Garbage Collector auf Hochtouren arbeitet und ein Objekt, sobald es zur Räumung freigegeben ist, *sofort* aus dem Speicher löscht.

Lösung auf der nächsten Seite...

Bei Kommentar 1: Es wurde eine Instanz der KlasseA angelegt sowie eine lokale Variable „a“, die darauf referenziert. Im Zuge des Konstruktoraufrufes wurde zudem eine String-Instanz mit Inhalt „A1“ erzeugt, deren Referenz im Konstruktor-Rumpf der KlasseA der Instanz-Variablen „sa“ zugeordnet wurde. Demnach sieht das Speichermodell zu diesem Zeitpunkt wie folgt aus:
(Links die Variablen, rechts die Objekte auf dem Heap)

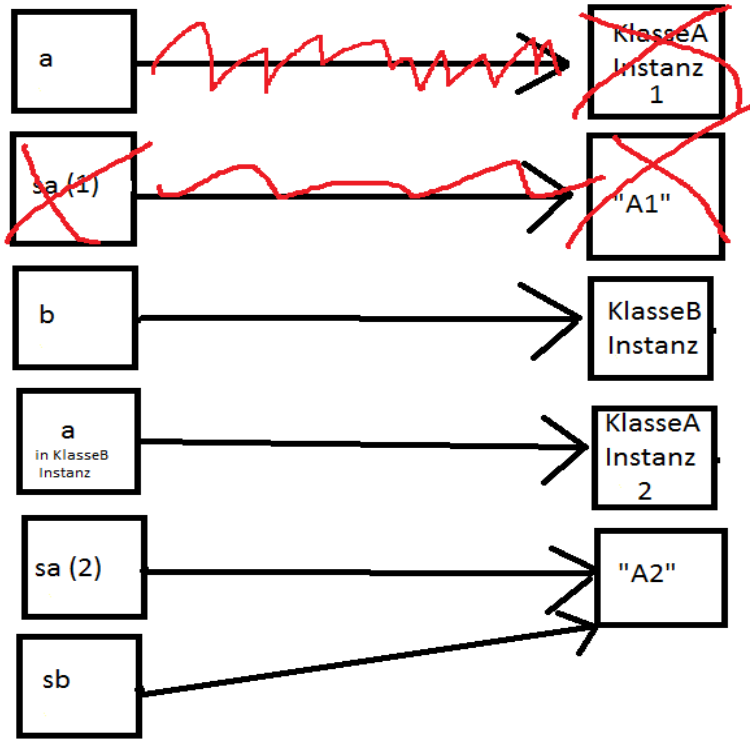


Bei Kommentar 2: Es wird eine Instanz der KlasseB angelegt und deren Referenz in die Variable „b“ gespeichert. Während diesem Vorgang wird eine zweite KlasseA-Instanz angelegt, sowie eine zweite String-Instanz. Die gerade angelegte KlasseA-Instanz wird innerhalb der KlasseB-Instanz in der Instanzvariablen „a“ abgespeichert. Der neu erstellte String wird sowohl von der Instanz-Variablen „sa“ der neuen KlasseA-Instanz referenziert, als auch von der Instanz-Variablen „sb“ der KlasseB-Instanz. Das Speichermodell zu diesem Zeitpunkt:

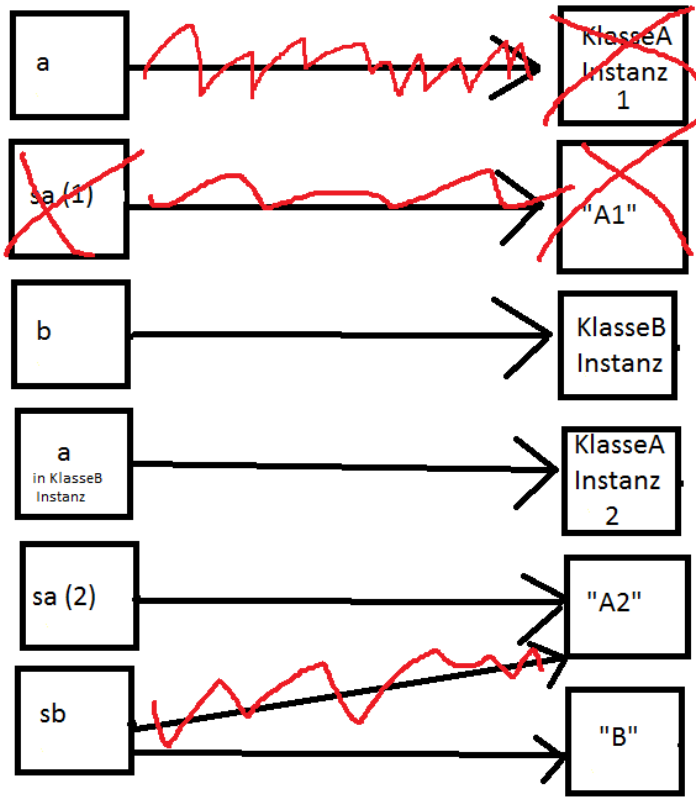


Bei Kommentar 3: Die Referenz in der lokalen Variablen „a“ der main-Methode zu der ersten KlasseA-Instanz wird gelöscht. Damit referenziert nun keine Variable mehr diese KlasseA-Instanz, womit sie vom Garbage Collector entfernt wird. Da sie eine Instanz-Variable hat, die allerdings nur eine interne Referenzierung enthält, wird auch diese zusammen mit dem Objekt gelöscht. Das führt dazu, dass der String „A1“ nicht mehr referenziert, und somit entfernt wird. In so einem Fall wird sogar die komplette Instanz-Variable aufgelöst – denn ohne die Instanz, zu der eine Instanz-Variable gehört, ist die Variable nicht mehr greifbar.

Das Speichermodell also:



Bei Kommentar 4: Eine neue String-Instanz „B“ wird erstellt, und mit der Instanz-Variablen „sb“ der KlasseB-Instanz referenziert. Diese Instanz-Variable hatte zuvor bereits eine Referenz auf den String „A2“, die nun durch die neue Referenz ersetzt wird. Da „A2“ noch immer durch die Instanz-Variable der noch existierenden KlasseA-Instanz referenziert wird, wird diese auch nicht vom Garbage Collector weggeräumt:



Bei Kommentar 5: Die Referenz der Instanz-Variablen „a“ der KlasseB-Instanz wird aufgelöst. Damit wird die noch bestehende KlasseA-Instanz nicht mehr referenziert. Sie wird gelöscht. Es werden alle Instanz-Variablen dieser KlasseA-Instanz entfernt, also die Variable „sa“. Letztere referenzierte auf den String „A2“. Dieser ist nun unreferenziert und wird auch gelöscht.

Das Speichermodell in seinem letzten Zustand vor Programm-Terminierung: