ÜBUNGS-BLOCK 8
LÖSUNGEN

Aufgabe 1:

Gegeben sind drei Klassen (linke Spalte) sowie ein Programm (rechts):

```
public class A {

	int var;

	public void setVar(int var) {
		this.var = var;
	}

	public int getVar() {
		return var;
	}

	public void printVar() {
		System.out.println(var);
	}
}
```

```
public class B extends A {

	int var;

	public void setVar(int var) {
		this.var = var;
		super.setVar(2 * var);
	}

	public void printVar() {
		System.out.println(getVar());
	}
}
```

```
public class C extends B {

	int var;

	public int getVar() {
		return var;
	}

	public void setVar(int var) {
		super.setVar(getVar());
	}
}
```

```
public class Programm {

	public static void main(String[] args){

		A a = new A();
		A b = new B();
		A c = new C();

		a.setVar(10);
		b.setVar(3);
		c.setVar(b.getVar());

		a.printVar();
		b.printVar();
		c.printVar();
	}
}
```

Frage: Wie lautet die Ausgabe des Programms?

Anmerkung:
Natürlich soll die Aufgabe durch Nachdenken gelöst werden, nicht durch Starten in Eclipse ;)

Warnung:
Diese Aufgabe ist ein echter Brainkiller! Lass dir Zeit mit dieser Aufgabe, nimm dir wenn nötig Blatt und Stift zur Hand, und überprüfe dein Ergebnis. 30 Minuten solltest du mindestens einrechnen.
Wer diese Aufgabe korrekt löst, kann sich nicht nur darüber freuen, dass er die Scopes bei Vererbung verstanden hat, sondern auch darüber, dass er ein enormes Konzentrationsvermögen besitzt!

Lösung:

Die Ausgabe lautet:

```
10
6
0
```

Wie es dazu kommt:

```
A a = new A();
```

Ein Objekt der Klasse A wird erzeugt. Es enthält eine Instanz-Variable „var“, der der Default-Wert 0 zugewiesen wird.

```
A b = new B();
```

Ein Objekt der Klasse B wird erzeugt. Es enthält *zwei* Instanz-Variablen, beide mit dem Namen „var“. Im Zuge der super-Initialisierung wird der Variablen aus der Vaterklasse der Default -Wert 0 zugewiesen, im Zuge der darauf folgenden Initialisierung aus der Klasse B wird der Variablen aus der Klasse B auch der Default -Wert 0 zugewiesen.

Beachte, dass es für die Instanz-Erzeugung keine Rolle spielt, welchen Datentypen wir für die Variable, in die wir das Objekt abspeichern wählen. (Ob „A“ oder „B“ spielt für das Objekt keine Rolle)

```
A c = new C();
```

Ein Objekt der Klasse C wird erzeugt. Es enthält *drei*(!) Instanz-Variablen des Namen „var“. Allen wird der Default-Wert zugewiesen. Auch hier gilt: Nicht vom deklarierten Typ „A“ verwirren lasen.

```
a.setVar(10);
```

Der Setter aus der Klasse A wird aufgerufen. Er weist der (für dieses Objekt einzigen!) Instanz-Variable aus Klasse A den Wert 10 zu.

```
b.setVar(3);
```

Der Setter aus der Klasse B(!) wird aufgerufen. Beachte, dass der deklarierte Datentyp für die Variable „b“ vom Typ „A“ ist. Allerdings wird bei Aufrufen von Instanz-Methoden immer die Version aus derjenigen Klasse aufgerufen, von der das Objekt tatsächlich abstammt – nicht von derjenigen Klasse, deren Datentyp deklariert ist! Das referenzierte Objekt ist vom Typ B, **nicht** A.

In der Klasse B ist der Setter aus Klasse A überschrieben. Zwar fehlt die @Override Annotation, und so etwas sollte man vermeiden. Aber sie ist nicht Pflicht – Overriding ist es auch ohne!

Zunächst weist dieser Setter der Variablen aus Klasse B den Wert 3 zu, anschließend wird der Setter aus Klasse A aufgerufen, mit dem Wert 2*3 = 6.

Das Objekt der Klasse B hält jetzt also die Werte 3 (var aus B) und 6 (var aus A).

Anmerkung: Natürlich verstößt dieser Setter gegen die Konvention, da er zwei verschiedene Instanz-Variablen beeinflusst! Aber es war ja auch Sinn dieser Aufgabe, Verwirrung zu stiften...

```
c.setVar(b.getVar());
```

Zunächst wird der Getter des durch die Variable „b“ referenzierten Objekts aufgerufen. Das wäre also demnach der Getter aus der Klasse B. In B wird so eine Methode nicht definiert, aber sie wurde (unüberschrieben) vererbt. Wir landen also im Rumpf des Getters der Klasse A. Dieser liefert den Wert von „var“ aus A. Und der wurde in der vorigen Anweisung auf 6 gesetzt. Also haben wir:

```
c.setVar(6);
```

„c“ referenziert ein Objekt des Typs C, daher wird der Setter aus Klasse C ausgeführt. (Überschrieben aus Klasse B). Frecherweise ignoriert dieser Setter den ihm übergebenen Wert gänzlich. Er tut etwas ganz anderes, nämlich: `super.setVar(getVar());`

Zunächst wird also der Getter aufgerufen. Wir wissen: `getVar() = this.getVar(),` und da auch der von B geerbte Getter überschrieben wurde, landen wir im Getter der Klasse C. Dieser returned den Wert von „var“ in Klasse C! (Denn sowohl „var“ aus A als auch „var“ aus B werden von dieser Variablen in C überschattet). Der Wert davon ist noch immer 0. Also: `super.setVar(0);`

Durch das „super“ rufen wir nun nicht den Setter aus Klasse C, sondern explizit den Setter der Vaterklasse (B) auf. Dieser weist zuerst var(B) den übergebenen Wert 0 zu, dann var(A) den Wert 0. (2*0 = 0). Var(C) war ja sowieso schon immer 0. Im Falle des Objektes der Klasse C haben also alle Instanz-Variablen haben Wert 0.

Damit sind nun alle Zuweisungen gemacht, Zusammenfassung:

Objekt (Referenz)	var(A)	var(B)	var(C)
a	10	-	-
b	6	3	-
c	0	0	0

```
a.printVar();
```

druckt den Wert von var aus, gemeint ist natürlich var(A) – sonst gibt es ja kein anderes. Wert = 10.

```
b.printVar();
```

Die Methode wurde in B überschrieben. Allerdings ruft sie den Getter der Vaterklasse (A) auf, was zum selben Ergebnis führt wie wenn B diese Methode aus A niemals überschrieben hätte...
Sie liefert also var(A) = 6.

ACHTUNG: Wer hier jetzt deshalb auch 10 erwartet hätte, ist mit den Objekten durcheinander gekommen! Merke: Wir haben hier im Falle von Instanzen der Klasse B bzw C zwar zwei bzw drei Variablen, die aus unterschiedlichen Klassen kommen – aber trotzdem sind es ja *Instanz-Variablen*, das bedeutet: Jedes Objekt hat seine *eigenen*! D.h. folgenden Dinge haben **nichts** miteinander zu tun:

```
var(A) für das durch „b“ referenzierte Objekt <-> var(A) für das von „a“ referenzierte Objekt
```

Wer hier einen Fehler gemacht hat: Nicht traurig sein – es war eine Falle, und der Sinn dieser Übung.

```
c.printVar();
```

printVar() ist in der Klasse C nicht definiert, wurde aber von B geerbt. Wir haben ja gerade im letzten Schritt festgestellt, dass die printVar() Methode der Klasse B den Wert von var (A) ausgibt. Dieser ist für das durch c referenzierte Objekt – sowie übrigens auch var(B) und var(C): 0!

Aufgabe 2:

Wirf mal einen Blick auf die folgende Klasse:

```
public class StringArrayHelp {

	private final String[] a;

	public StringArrayHelp(String[] a) {
		this.a = a;
	}

	/* Liefert die Anzahl an "echten" Elementen im Array */
	public int getElementCount() {
		int count = 0;
		for (String s : a) {
			if (s != null) {
				count++;
			}
		}
		return count;
	}

	/* Liefert die Anzahl der "freien" Fächer im Array */
	public int getFreeSlotCount() {
		int count = 0;
		for (String s : a) {
			if (s == null) {
				count++;
			}
		}
		return count;
	}

	/*
	 * Liefert den Index des ersten "freien" Fachs im Array,
	 * -1 falls kein freies Fach mehr vorhanden.
	 */
	public int getNextFreeSlot() {
		int freeSlot = -1;
		boolean continueSearch = true;
		for (int i = 0; i < a.length; i++) {
			if (continueSearch) {
				if (a[i] == null) {
					freeSlot = i;
					continueSearch = false;
				}
			}
		}
		return freeSlot;
	}
}
```

Diese Klasse stammt von einem Freund von dir, der wie du gerade Java erlernt. Er hat sie sich neulich geschrieben für ein Programm, in welchem er recht viel mit String-Arrays gearbeitet hat. Du musst wohl zugeben, dass die Idee für diese Klasse recht gut war: Dein Freund hat damit eine in seinem Programm immer wiederkehrende Logik in eine eigene Strukturen ausgelagert, um redundanten Code zu vermeiden. Es besteht wohl kein Zweifel darin, dass ihm diese Hilfsklasse die Arbeit mit seinen String-Arrays im Programm erleichtert hat.

Jetzt arbeitet er gerade an einem neuen Programm. In diesem arbeitet er allerdings weniger mit Strings, sondern hauptsächlich mit Instanzen der (von ihm geschriebenen) Klassen „Konto“ und „Bank“ – und demnach mit enstprechenden Arrays. Da er seine StringArrayHelp-Klasse im alten Programm recht nützlich fand, möchte er das selbe Prinzip nun auch auf sein neues Programm anwenden.

Er möchte dafür seine StringArrayHelp-Klasse (angepasst) übernehmen für zwei neue Klassen „KontoArrayHelp“ und „BankArrayHelp“.

Bevor er das tut, sollst du das aber absegnen, denn dein Freund hält dich für einen sehr guten Programmierer und demnach viel von deiner Meinung.

Also: Was hälst du von seiner ursprünglichen StringArrayHelp-Klasse? Und was von der Idee der Klassen „KontoArrayHelp“ und „BankArrayHelp“ für sein neues Programm? Kann man es besser machen? Was für Ratschläge würdest du deinem Freund geben?

Schreibe dir alles, was dir dazu einfällt oder was du verbessern würdest auf. Begründe deine Entscheidungen und Vorschläge! Lass dir mit dieser Aufgabe Zeit. Es ist auch keinesfalls gesagt, dass du diese Aufgabe durch bloßes Nachdenken lösen sollst. Starte deine Entwicklungsumgebung und mach dich an eine optimale Lösung!

Tipp: Diese Aufgabe hat *auch* etwas mit dem Stoff zu tun, den du in diesem Video-Block gelernt hast. Aber nicht *nur*. Betrachte das ganze möglichst distanziert – versuche an *alles* zu denken, was du bisher gelernt hast, und denke über alle möglichen eventuellen Verbesserungsmöglichkeiten nach!

Achtung: Die Aufgabe besteht aber **nicht** darin, komplett neue Features zu programmieren! Die drei Methoden aus der Klasse deines Freundes sollen – von der Idee her und dem, was sie tun – übernommen werden!

Deine Aufgabe ist es also, das, was dein Freund programmiert hat, zu *verbessern* – auch im Hinblick auf das Vorhaben für sein neues Programm.

Diese Verbesserungen können auf zwei Ebenen stattfinden:

1) im Design
2) in den Implementierungs-Details

Am besten, du denkst zuerst über das Design nach. Dazu zählen die Methoden-Signaturen (Access Modifier, Rückgabewert, Parameter) – zum Beispiel. Vielleicht siehst du ja noch einen anderen Aspekt des Designs? Danach kannst du dir überlegen, was genau du in die Methodenrümpfe hineinschreibst.

Viel Spaß!

Lösung:

```
public class ArrayHelp {

	/* Liefert die Anzahl an "echten" Elementen im Array */
	public static int getElementCount(Object[] a) {
		int count = 0;
		for (Object o : a) {
			if (o != null) {
				count++;
			}
		}
		return count;
	}

	/* Liefert die Anzahl der "freien" Fächer im Array */
	public static int getFreeSlotCount(Object[] a) {
		return a.length - getElementCount(a);
	}

	/*
	 * Liefert den Index des ersten "freien" Fachs im Array,
	 * -1 falls kein freies Fach mehr vorhanden.
	 */
	public static int getNextFreeSlot(Object[] a) {
		for (int i = 0; i < a.length; i++) {
			if (a[i] == null) {
				return i;
			}
		}
		return -1;
	}
}
```

Kommentar:

Wir widmen uns zuerst dem Design für diese Aufgabe: Zunächst einmal ist es immer besser, solche Hilfsklassen **statisch** zu halten. In der vorgegebenen StringArrayHelp-Klasse war das nicht der Fall: Man musste eine Instanz davon erzeugen und ein Array daran koppeln, um die Methoden nutzen zu können. Unter einem „StringArrayHelpObjekt“ kann man sich aber nichts vorstellen, und es macht die Sache nur umständlicher wenn man parallel zu den Arrays, mit denen man arbeitet, noch solche StringArrayHelp-Objekte durch den Code „mitschleifen“ muss (oder ständig per Getter das Array aus diesem Konstrukt herauszieht). Zudem birgt es immer die Gefahr von Flüchtigkeitsfehlern, wenn wir verschiedene Variablen für ein und die selbe Referenz im Programm halten. Es ist also empfehlenswert, die Instanz-Variable loszuwerden und alle Methoden statisch zu machen. Natürlich muss man ihnen dann beim Aufruf noch das Array mitgeben, auf dem sie arbeiten sollen.

Und hier kommen wir zum zweiten Aspekt im Bezug auf das Design: Der Freund wollte zwei Klassen „KontoArrayHelp“ und „BankArrayHelp“ erstellen – er hatte sich das wohl so gedacht dass er jeweils den Datentyp des Arrays anpasst. Aber wozu? Wir haben gelernt, dass wir immer auf möglichst abstraktem Level programmieren sollen. Wir wollen keine Informationen, die wir nicht brauchen, das macht den Code flexibler.

Wenn man kurz überlegt, kommt man zu folgendem Schluss: Was die Methoden tun, hat nichts mit dem Datentyp der im Array liegenden Elemente zu tun. Die Methoden liefern das selbe Ergebnis, egal ob in dem Array nun Strings, Kontos oder Banken stecken. Um diese Klasse für beliebige Arrays verwenden zu können nutzen wir also einen möglichst abstrakten Datentyp. Der abstrakteste Typ, den es gibt, ist der Typ Object. Unser Freund braucht also nicht zwei verschiedene Klassen für seine Konto- und Bank-Arrays. Er kann diese eine Klasse für alle Arrays verwenden, auch wenn später noch weitere (komplexe) Datentypen in seinen Programmen mitmischen. Stichwort Polymorphie und Subsitutionsprinzip.

Nun zu den Details der Implementierung: Die getElementCount()-Methode aus der StringArrayHelp-Klasse war in Ordnung. Bei der Methode getFreeSlotCount() hätte man sich allerdings einiges an Code sparen können, wie die vorliegende Lösung zeigt. Hier eine kleine Anmerkung: Wenn man es kritisch sieht muss man sagen, dass unsere Lösung jetzt langsamer als die Lösung aus der StringArrayHelp-Klasse ist. Ob man nun die Elemente != null zählt, oder die Elemente == null spielt im Durschnitt zwar keine Rolle. Wir haben aber zusätzlich noch einen Zugriff auf das length-Attribut und eine Rechenoperation (Subtraktion). Aber: Trotzdem ist diese Lösung eleganter, denn sie macht den Code schlanker und leichter zu lesen. Was die Performanz angeht, so reden wir hier von ein paar wenigen Nano(!)-Sekunden für diesen zusätzlichen Zugriff.

Lesbarkeit des Codes > Performanz des Codes!

Auch die Methode getNextFreeSlot() kann man besser lösen als es in der ursprünglichen Klasse der Fall war. Dort wurde mit einem booleschen Flag gearbeitet, das für jeden Schleifendurchlauf festlegt, ob die darin enthaltene Logik überhaupt noch ausgeführt werden soll.

In jedem Fall wurde die Schleife in dieser Version aber immer voll durchlaufen, d.h. selbst wenn schon das erste Fach frei war, würde das gesamte Array durchlaufen werden. Besser ist es, die Schleife – in diesem Fall auch gleich die gesamte Methode – zu verlassen, sobald wir das Ergebnis haben. Der Rest interessiert uns ja soweiso nicht mehr.

Das kann sich dann bei einem Array von ein paar hunderttausend Fächern durchaus merklich auf die Performanz der Software auswirken.

Aufgabe 3:

Ein Online-Entertainment-Shop verkauft Bücher, Filme und Videospiele. Die Software dafür soll die folgenden Informationen über die jeweiligen Artikel pflegen:

Für Bücher:	***Für Filme:***	***Für Videospiele:***
Bezeichnung (Buchtitel)	Bezeichnung (Filmtitel)	Bezeichnung (Spieltitel)
Artikelnummer	Artikelnummer	Artikelnummer
Preis	Preis	Preis
Autor	Regisseur	Plattform (zB PC, Wii, Xbox, ...)
Anzahl der Seiten	Spieldauer (in Minuten)	Genre (zB Shooter, Strategie, ...)

Außerdem gibt es zu jedem dieser Artikel spezielle Zusatz-Informationen oder -Inhalte, auf die man allerdings nur als Besitzer eines Premium Accounts zugreifen kann. Dafür kann ein Nutzer auf der Webseite bei einem Artikel den Button „Premium Request“ klicken. Im Falle von Büchern wird ihm dann – falls er einen gültigen Premium Account hat - ein Ausschnitt (zB die ersten 30 Seiten) des Buches präsentiert. Bei Filmen werden Trailer oder Teile des Bonus-Materials in einem Videoplayer abgespielt. Und für Videospiele lässt sich kostenlos eine Demo-Version des Spiels herunterladen.

Es kann für solch einen Premium Request die folgende Klasse genutzt werden:

```
public class WebServerFeatures {

	public static void videoAbspielen(String videoLink) {
		/*
		 * Öffnet einen Videoplayer auf der Website und spielt ein Video ab,
		 * welches sich auf dem Server unter der angegebenen Adresse (Link)
		 * befindet.
		 */
	}

	public static void pdfAnzeigen(String pdfLink) {
		/*
		 * Öffnet ein Textbetrachtungsprogramm auf der Website und lädt ein
		 * Snippet aus einem Buch.
		 */
	}

	public static void dateiHerunterladen(String dateiLink) {
		/*
		 * Öffnet einen Download-Manager, mit deren Hilfe eine Datei
		 * heruntergeladen werden kann.
		 */
	}
}
```

Der Einfachheit halber sind die Methoden hier nur pseudo-implementiert. Wir wollen einfach mal so tun, als würden sie auch wirklich das machen, was in den Kommentaren steht. (Es ist also nicht nötig, daran etwas zu ändern!)

Die Aufgabenstellung befindet sich auf der nächsten Seite...

Aufgabenstellung:

Implementiere die nötigen Datenstrukturen und Logik für diese Software. Klassen, die du schreibst, sollten alle üblichen Bestandteile aufweisen, und sich an die Konventionen halten (zB soll es keine Objekte geben, die im Zuge der Erzeugung nicht explizit mit einem Wert initialisiert werden).

Versuche ein Design zu schaffen, welches einen möglichst hohen Abstraktionsgrad in der Software erlaubt und den Implementierungsaufwand (Menge an Code) möglichst gering hält, auch in Bezug auf eventuelle spätere Erweiterungen in der Software (zB neue Arten von Artikeln). Überlege dir, welche Features dir Java für das Design von Objekten/Klassen und deren Zusammenhänge bietet.

Schreibe außerdem eine Testklasse mit einer main-Methode, in welcher du ein paar Bücher, Filme und Videospiele erzeugst und in einer passenden Datenstruktur abspeicherst (Eine lokale Speicherung in der main-Methode reicht). Dann sollen alle Artikel auf der Konsole ausgegeben werden. Dabei sollen –soweit möglich- alle relevanten Informationen über die jeweiligen Artikel angezeigt werden.

Überlege dir, wie du eine solche Ausgabe am besten/einfachsten realisieren kannst!

Lösung auf der nächsten Seite...

Lösung:

```
public abstract class ShopArtikel {

	private String artikelNr;
	private String bezeichnung;
	private double preis;
	private String premiumRequestLink;

	public ShopArtikel(String artikelNr, String bezeichnung, double preis,
			String premiumRequestLink) {
		setArtikelNr(artikelNr);
		setBezeichnung(bezeichnung);
		setPreis(preis);
		setPremiumRequestLink(premiumRequestLink);
	}

	public String getArtikelNr() {
		return artikelNr;
	}

	public void setArtikelNr(String artikelNr) {
		this.artikelNr = artikelNr;
	}

	public String getBezeichnung() {
		return bezeichnung;
	}

	public void setBezeichnung(String bezeichnung) {
		this.bezeichnung = bezeichnung;
	}

	public double getPreis() {
		return preis;
	}

	public void setPreis(double preis) {
		this.preis = preis;
	}

	public String getPremiumRequestLink() {
		return premiumRequestLink;
	}

	public void setPremiumRequestLink(String premiumRequestLink) {
		this.premiumRequestLink = premiumRequestLink;
	}

	public abstract void premiumRequest();

	@Override
	public String toString() {
		return bezeichnung + " (" + artikelNr + ", " + preis + " €)";
	}

}
```

Die Gemeinsamkeiten von Büchern, Filmen und Videospielen halten wir in einer abstrakten(!) Oberklasse fest. Abstrakt aus zwei Gründen: Erstens gibt es so etwas wie einen „Artikel“ im Shop nicht zu kaufen. Es gibt nur konkret Bücher, Filme, Spiele. Zweitens enthält die Klasse eine abstrakte Methode...

...premiumRequest() muss abstrakt sein, denn an dieser Stelle fehlt uns die Information darüber, was für eine Datei unser Link im Genauen enthält. Wir wissen also nicht, welche Methode aus der WebServerFeatures-Klasse wir aufrufen sollen. Trotzdem wollen wir die Methode in dieser Oberklasse definieren – immerhin *gibt* es ja für jeden Artikel einen solchen Request. D.h. wir möchten dass diese Methode verfügbar ist, wenn wir mit dem Datentypen „ShopArtikel“ arbeiten.

Wir überschreiben in dieser Klasse auch die von Object geerbte Methode toString(), denn wir möchten die Artikel später einfach aus der Konsole ausgeben können.

Nun erstellen wir eine Klasse für jeden Artikel-Typ, zB für Bücher:

```
public class Buch extends ShopArtikel {

	private String autor;
	private int anzahlSeiten;

	public Buch(String artikelNr, String bezeichnung, double preis,
			String autor, int anzahlSeiten, String snippetLink) {
		super(artikelNr, bezeichnung, preis, snippetLink);
		setAutor(autor);
		setAnzahlSeiten(anzahlSeiten);
	}

	public String getAutor() {
		return autor;
	}

	public void setAutor(String autor) {
		this.autor = autor;
	}

	public int getAnzahlSeiten() {
		return anzahlSeiten;
	}

	public void setAnzahlSeiten(int anzahlSeiten) {
		this.anzahlSeiten = anzahlSeiten;
	}

	@Override
	public void premiumRequest() {
		WebServerFeatures.pdfAnzeigen(getPremiumRequestLink());
	}

	@Override
	public String toString() {
		return "[BUCH] " + autor + ", " + anzahlSeiten + " Seiten: "
				+ super.toString();
	}

}
```

Wir erben von der „ShopArtikel“-Klasse, und erweitern die Subklasse um typspezifische Eigenschaften. Hier implementieren wir nun auch die Logik der premiumRequest()-Methode, denn hier wissen wir, welche Art von Datei sich hinter dem Link befinden muss und demnach welches Feature des Servers wir ansprechen müssen.

Außerdem überschreiben wir – nochmals - die toString()-Methode. Wir reichern die geerbte Implementierung (super.toString()) mit typspezifischen Informationen an.

Analog zur Buch-Klasse implementieren wir die Klasse „Film“ und „Videospiel“:

```java
public class Film extends ShopArtikel {

	private String regisseur;
	private int spieldauer;

	public Film(String artikelNr, String bezeichnung, double preis,
			String regisseur, int spieldauer, String videoLink) {
		super(artikelNr, bezeichnung, preis, videoLink);
		setRegisseur(regisseur);
		setSpieldauer(spieldauer);
	}

	public String getRegisseur() {
		return regisseur;
	}

	public void setRegisseur(String regisseur) {
		this.regisseur = regisseur;
	}

	public int getSpieldauer() {
		return spieldauer;
	}

	public void setSpieldauer(int spieldauer) {
		this.spieldauer = spieldauer;
	}

	@Override
	public void premiumRequest() {
		WebServerFeatures.videoAbspielen(getPremiumRequestLink());
	}

	@Override
	public String toString() {
		return "[FILM] " + regisseur + ", " + spieldauer + " Min.: "
				+ super.toString();
	}

}
```

```
public class Videospiel extends ShopArtikel {

	private String plattform;
	private String genre;

	public Videospiel(String artikelNr, String bezeichnung, double preis,
			String plattform, String genre, String demoLink) {
		super(artikelNr, bezeichnung, preis, demoLink);
		setPlattform(plattform);
		setGenre(genre);
	}

	public String getPlattform() {
		return plattform;
	}

	public void setPlattform(String plattform) {
		this.plattform = plattform;
	}

	public String getGenre() {
		return genre;
	}

	public void setGenre(String genre) {
		this.genre = genre;
	}

	@Override
	public void premiumRequest() {
		WebServerFeatures.dateiHerunterladen(getPremiumRequestLink());
	}

	@Override
	public String toString() {
		return "[SPIEL] " + plattform + ", " + genre + " : " +
		super.toString();
	}

}
```

Für die Testklasse erzeugen wir ein paar Artikel, die wir in einem Array abspeichern. Da wir lediglich alle Artikel auf der Konsole anzeigen lassen sollen, und die einzelnen Artikel durch das Overriding der toString()-Methode selbstständig eine geeignete Ausgabe erzeugen, können wir als Datentypen den abstrakten Typ „ShopArtikel“ wählen, und die Objekte einfach in einen System.out.println()-Aufruf hineinstecken:

```
public class Test {

	public static void main(String[] args) {

		ShopArtikel[] artikel = {
			new Buch("7F23084", "Java 101", 39.90, "Thomas Klutsch",
				612, "/bücher/9823023.pdf"),
			new Film("A982100", "Die Java-Hacker", 12.90, "Thomas Klutsch",
				92, "/filme/2938423.mov"),
			new Videospiel("X823712", "Java Spielesammlung", 19.90, "PC",
				"Action", "/games/237893742.zip")
		 };

		for (ShopArtikel a : artikel) {
			System.out.println(a);
		}
	}
}
```