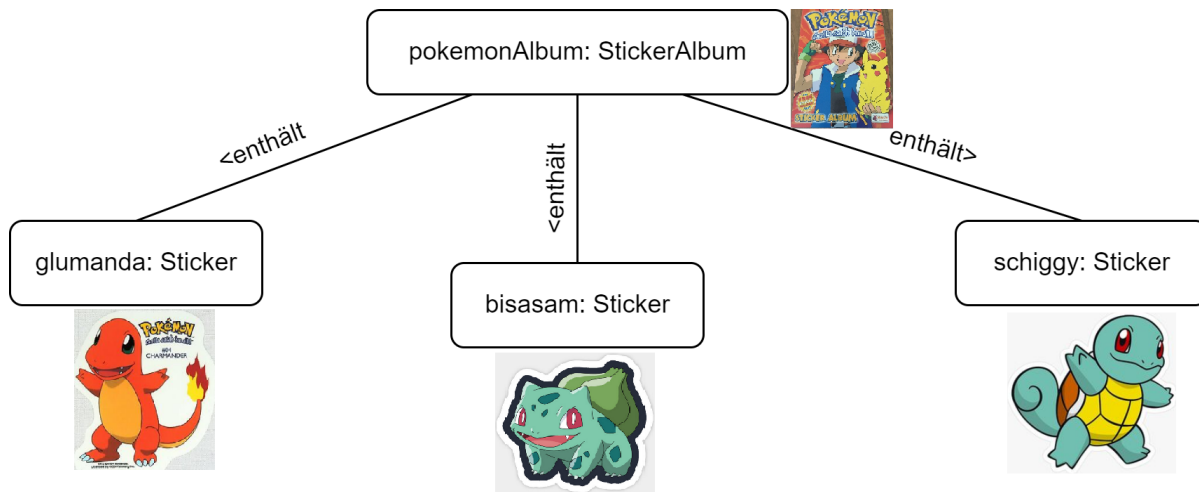


Neben Arrays ist es ein Wesentliches Ziel dieser Jahrgangsstufe, dass in Projekten mehrere Klassen verwendet werden (in kleinerer Form kam dies bereits in der neunten Klasse bei Klassendiagrammen vor!).

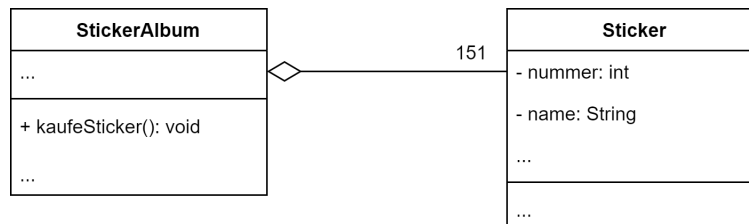
Damit Objekte verschiedener Klassen einfacher verknüpft werden können, werden sie häufig als Attribute angelegt - bisher waren Attribute hauptsächlich primitive Datentypen wie boolean, int oder double (String war schon die ganze Zeit eine Ausnahme!).

Zunächst ein anschaulicheres Beispiel: häufig „bestehen“ Objekte hauptsächlich aus anderen Objekten (zumindest in der verwendeten Modellierung). Wollen wir beispielsweise ein Stickeralbum modellieren, benötigen wir zwei Klassen, eine für das Album und eine für einen einzelnen Sticker, man kann sagen: das Stickeralbum **enthält** die Sticker (das entspricht auch der Realität relativ gut!).

Im Fachbegriff heißt so eine „enthält“-Beziehung **Aggregation**. Im Objektdiagramm sähe das z.B. so aus:



Im Klassendiagramm würde dies mit einem anderen Pfeil markiert werden:



Ein `StickerAlbum` besteht also aus 151 Objekten vom Typ `Sticker`. Die **Multiplizität** 1 beim `StickerAlbum` wird in der Regel weggelassen. Aggregationen werden - wie viele „Verbindungen“ zwischen Klassen - mit Attributen realisiert. Vereinfachen wir unser Beispiel weiter und nehmen an, dass es nur drei Sticker gibt, dann könnte eine grundlegende Implementierung des Stickeralbums so aussehen:

```

public class StickerAlbum {
    private Sticker glumanda;
    private Sticker bisasam;
    private Sticker schiggy;

    public StickerAlbum() {
        bisasam = new Sticker(1, "Bisasam");
        glumanda = new Sticker(4, "Glumanda");
        schiggy = new Sticker(7, "Schiggy");
    }
}

```

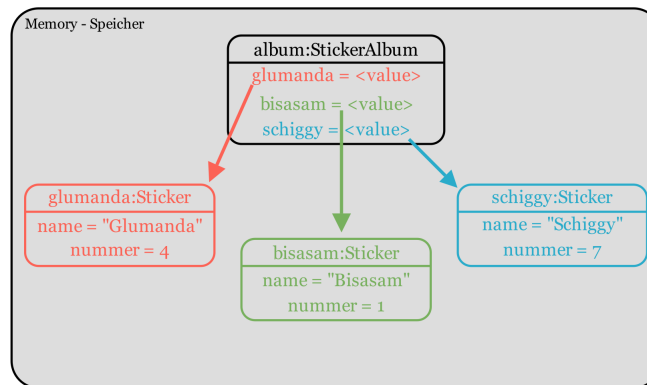
Dabei wird - wie schon beim Array - mit dem Schlüsselwort **new** neue Objekte der Klasse `Sticker` erzeugt. (Man spricht auch von **Instanziierung**).

```

public class Sticker() {
    int nummer;
    String name;
    public Sticker(int nummer, int name) {
        this.nummer = nummer;
        this.name = name;
    }
}

```

Wichtig: Die Sticker-Objekte sind nicht wirklich im StickerAlbum-Objekt „enthalten“ im Sinne von: im Speicher stehen sie „ineinander“. Vielmehr ist die deklarierte Variable ein Zeiger (Fachbegriff: **Referenz**), die auf den Ort im Speicher verweist, an dem die eigentlichen Daten abgelegt sind - noch genauer gesagt: auf die Adresse der ersten Speicherzelle, die für das erzeugte Objekt verwendet wird. (im Folgenden noch mit Objektkarten veranschaulicht):

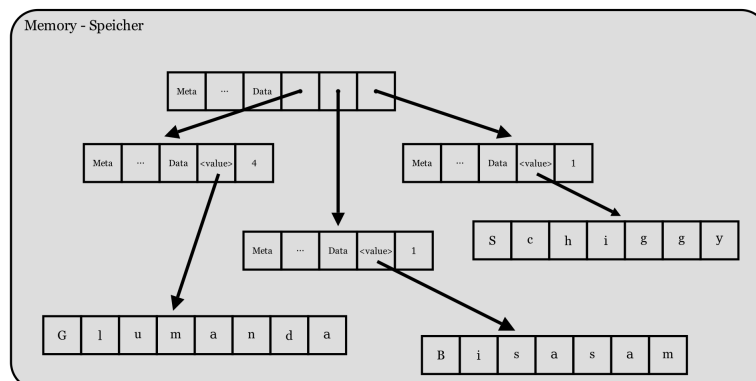


Ähnliches gilt auch für Arrays. Speichert man primitive Datentypen (also z.B. int, boolean, etc.) in einem Array, so ist die Variable eine Referenz in den Speicher, die an die erste Stelle mit Metadaten zeigt (z.B. liegt dort der Wert des length-Attributs, das wir schon verwendet haben). Anschließend liegen in den folgenden Speicherzellen die Werte explizit abgespeichert. Handelt es sich jedoch um ein Array von Objekten, so ist in jeder der folgenden Speicherzellen nur eine Referenz auf den Speicherort der jeweiligen Objekte vermerkt. Wären in unserem StickerAlbum von oben die Sticker in einem Array gespeichert, dann sähe Implementierung und Speicherveranschaulichung so aus:

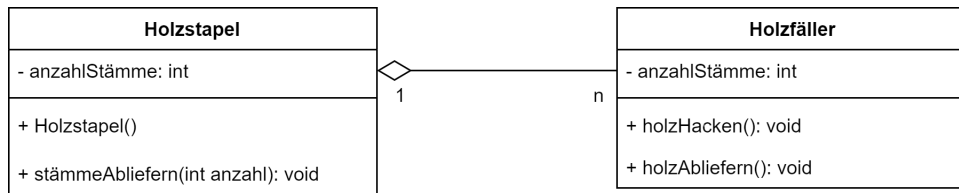
```

public class StickerAlbum {
    Sticker[] sticker;
    public StickerAlbum() {
        sticker = new Sticker[3];
        sticker[0] = new Sticker(1, "Bisasam");
        sticker[1] = new Sticker(4, "Glumanda");
        sticker[2] = new Sticker(7, "Schiggy");
    }
}

```



Noch einmal zurück zu **Referenzen**: immer wenn ein Objekt auf ein anderes Objekt „verweist“ (also nicht nur im Kontext von Arrays) spricht man von einer Referenz. Erst durch diese Referenzen können die Objekte überhaupt miteinander interagieren, da sie ansonsten nichts von der Existenz des jeweils anderen wissen würden. Dabei ergeben sich gewisse Fallstricke. Wir betrachten eine anschauliche Situation, hier als Klassendiagramm:



Wir modellieren Holzstapel und Holzfäller. Der Einfachheit halber liefern Holzfäller nur an einem einzigen Stapel ab (deswegen die 1), allerdings können beliebig viele (n) Holzfäller an diesem Stapel ablegen. Eine mögliche Implementierung der beiden Klassen sieht so aus:

```

public class Holzstapel {
    private int anzahlStämme;

    public Holzstapel() {
        anzahlStämme = 0;
    }

    public void stämmeAbliefern(int anzahl) {
        if(anzahl <= 0) {
            System.out.println("Das geht so nicht!");
            return;
        }
        anzahlStämme += anzahl;
    }
}
  
```

Im Holzstapel verwalten wir die Anzahl der bereits gefällten Stämme, es können dort nur Stämme abgeliefert werden.

```

public class Holzfäller {
    private Holzstapel stapel;
    private int anzahlStämme;

    public Holzfäller(Holzstapel stapel) {
        anzahlStämme = 0;
        this.stapel = stapel;
    }

    public void holzHacken() {
        anzahlStämme++;
    }

    public void holzAbliefern() {
        stapel.stämmeAbliefern(anzahlStämme);
        anzahlStämme = 0;
    }
}
  
```

In der Holzfällerklasse können wir entweder Holz hacken (das erhöht die Anzahl der Stämme) oder diese abgeben. Zwei entscheidende Stellen:

1. Im Konstruktor wird dieses Mal eine **Referenz** übergeben, wie oben schon erwähnt wird nicht das komplette Stapel Objekt innerhalb des Holzfällers gespeichert, der Holzfäller „kennt“ nur seinen Stapel, auf dem er etwas abliefern, d.h. insbesondere, dass **mehrere** Holzfäller eine Referenz zu **demselben** Stapel bekommen!

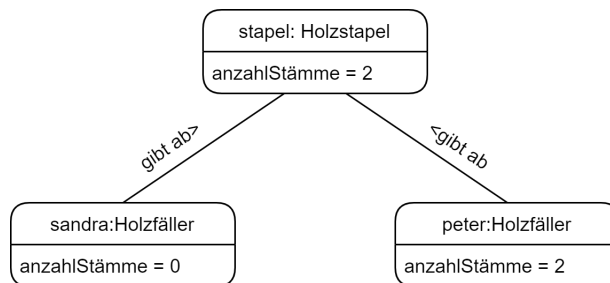
2. in `holzAbliefern()` nutzen wir die Punktnotation, um auf dem Holzstapel-Objekt die `stämmAbliefern()`-Methode auszuführen.

Im Fall eines Referenzattributes ist es also nicht unbedingt sinnvoll zu sagen: es handelt sich um eine Eigenschaft der Klasse, so wie wir das bei der Anzahl der Stämme sagen würden. Vielmehr handelt es sich um eine Kommunikationsmöglichkeit.

Hinweis: Streng genommen gilt das wie oben erwähnt auch für Strings, da diese selbst auch Objekte sind, hier ist die Bedeutung aber noch näher an der „Eigenschaft“ als im Fall unseres Holzstapels.

Als Veranschaulichung betrachten wir noch das Objektdiagramm, das den Zustand nach folgenden Methodenaufrufen beschreibt:

```
Holzstapel stapel = new Holzstapel();
Holzfäller peter = new Holzfäller();
Holzfäller sandra = new Holzfäller();
peter.holzHacken();
peter.holzHacken();
sandra.holzHacken();
sandra.holzHacken();
sandra.holzAbliefern();
```



Achtung: Auch Peter hält eine Referenz auf dasselbe Stapel-Objekt wie Sandra, d.h. auch wenn nur Sandra ihre Stämme abgeliefert hat, könnte Peter abrufen, ob bereits Stämme vorhanden sind (wenn es z.B. eine `gibAnzahl()`-Methode gäbe).

Gleichartig vs. Identisch:

In unserer Modellierung gibt es direkt nach der Erzeugung zwischen Peter und Sandra eigentlich keinen Unterschied, die Referenzen heißen zwar unterschiedlich, aber beide sind Objekte der Klasse `Holzfäller` mit einem Attribut „anzahlStämme“, das den Wert 0 hat. D.h. beide sind gleichartig (alle Attributwerte stimmen überein), sie sind aber nicht identisch, da dennoch auf zwei **verschiedene** Objekte verwiesen wird, deswegen würde der Code

```
System.out.println(peter == sandra);
```

auch den Wert „false“ auf die Konsole geben, obwohl beide die gleichen Attributwerte haben (die `anzahlStämme` mit 0 und die Referenz auf denselben Stapel!). Das `==` vergleicht in Java bei Objekten also nur Referenzen und prüft nicht auf Gleichheit von Attributwerten! Das ist besonders gemein bei Strings:

```
String a = "abc";
String b = a;
String c = "abc";
String d = new String("abc");
System.out.println(a == b);
System.out.println(a == c);
System.out.println(a == d);
```

Dieser Code liefert zweimal „true“ und einmal „false“. Bei `b` setzen wir einfach eine zweite Referenz, die auf **dasselbe** String-Objekt zeigt. Bei `c` dagegen „versteht“ Java von selbst, dass es sich um denselben String handeln soll und legt gar kein neues Objekt an! Wenn wir Java aber „zwingen“ (wie bei `d`), dass ein neues String-Objekt angelegt wird, dann ist der entsprechende Vergleich auch „falsch“, da die beiden Referenzen an unterschiedliche Stellen im Speicher zeigen.