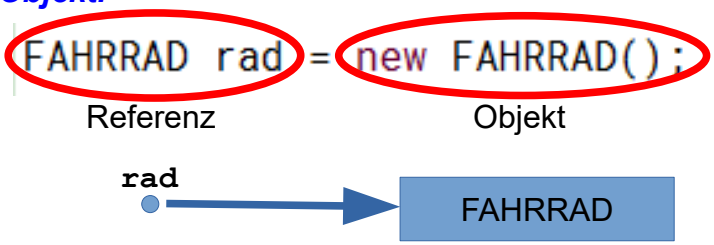
Vorbereitung

Nachtrag zu Referenzen/Referenzattributen

Die Referenzattribute bekommen keine „Namen“, sondern eine ***Referenz zeigt/verweist auf ein existierendes Objekt.***



Die Referenzen vom Typ FAHRRAD können (erstmal) nur auf Objekte vom Typ FAHRRAD zeigen. Die Referenz zeigt dabei auf die Speicheradresse, an der das Objekt im Arbeitsspeicher (RAM) liegt. Die Speicheradresse wird im Hexadezimalzahl angegeben:

→ `System.out.println(rad);` → `FAHRRAD@1db4d56c`

Durch das Konzept der Referenzen ist es möglich, dass:

- eine Referenz nacheinander ***auf unterschiedliche Objekte*** zeigt, oder
- mehrere Referenzen ***auf dasselbe Objekt*** zeigen.
- eine Referenz auf ***kein*** Objekt zeigt. Der Referenzwert ist dann ***null***.

Aufgabe 1:

a) *Öffne das Projekt Fahrrad, vollziehe die Schritte nach und verinnerliche das Konzept der Referenz.*

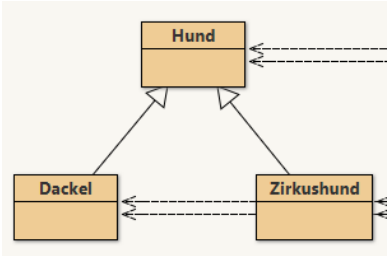
b) *Zeichne eine kurze Objektdiagrammreihe in der du den Code des Konstruktors der Klasse PERSON graphisch darstellst.*

Referenzen und Vererbung

Wiederholung: Die Vererbung ist eine besondere Beziehung zwischen Klassen. Man nennt Vererbung auch eine „ist ein“-Beziehung. Das bedeutet, dass die Unterklasse eine ***Spezialisierung*** der Oberklasse ist und **alle Attribute und Methoden der Oberklasse erbt**. Des Weiteren kann die Unterklasse weitere eigene Attribute und Methoden implementieren und so die Oberklasse ***erweitern***.

Das bedeutet für das Referenzkonzept:

- Eine Referenz der Oberklasse ***kann ein Objekt der Unterklasse referenzieren*** (Ein Dackel ist ein Hund).
- Aber eine Referenz der Unterklasse ***kann NICHT ein Objekt der Oberklasse referenzieren*** (Ein Hund muss kein Dackel sein, somit kann die Dackelreferenz keinen allgemeinen Hund referenzieren).




```
Hund h = new Dackel(1,"Bello");
Dackel d = new Hund(2,"Kylo");
```

```
incompatible types: Hund cannot be converted to Dackel
```

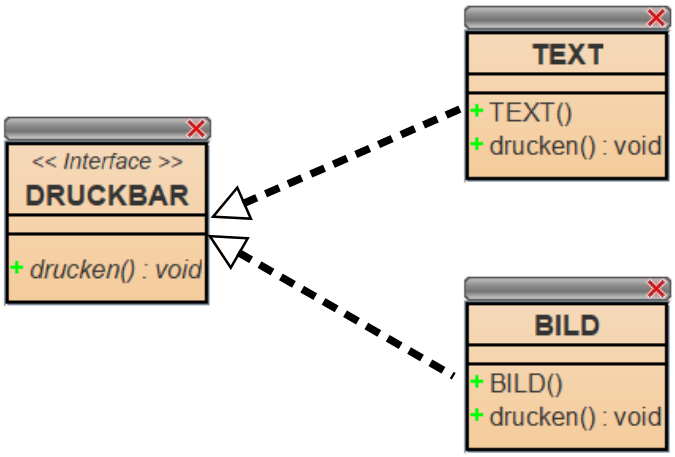
Aufgabe 2:

Öffne das Projekt Hund und sieh dir die Klasse TestklasseReferenzen an.

Interface

Ein ***Interface*** ist eine Schnittstelle, die Klassen mit dem Interface immer vollständig implementieren müssen. Konkret beinhaltet ein Interface (z.B. DRUCKBAR, AUFLISTBAR, … usw.) ***Methoden(-deklarationen)***, die die Klassen mit diesem Interface für sich selbst entsprechend implementieren müssen. So ist sichergestellt, dass jede Klasse mit diesem Interface diese Methoden (z.B. drucken(), … usw.) enthält und diese damit auch aufgerufen werden können.

Im Klassendiagramm wird dies folgendermaßen dargestellt:




```
public interface DRUCKBAR{
    public void drucken();
}
```

```
public class BILD implements DRUCKBAR {
```

Die jeweiligen Methoden drucken() bei BILD und TEXT werden so implementiert, dass der jeweilige Text oder das jeweilige Bild gedruckt werden kann. Der Sinn davon ist , dass jede ***Klasse diese Methoden entsprechend passend für sich implementiert*** und diese Implementierungen sich unterscheiden können.

Zudem definiert das Interface (hier: DRUCKBAR) ***einen neuen Datentyp***. Referenzen dieses Typs können auf **alle Objekte referenzieren**, deren Klassen dieses Interface implementieren.

```
DRUCKBAR d1 = new BILD();
DRUCKBAR d2 = new TEXT();
```

Datentypumwandlung in Java (Cast)

Implizite Datentypumwandlung / impliziter Cast

Wie bereits in den beiden vorherigen Abschnitten erwähnt kann ein spezielleres Objekt (hier: Dackel, Bild, Text) kann immer einem generelleren Typ (hier: Hund, Druckbar) zugewiesen werden. Hierfür ist kein Cast notwendig, bzw. es wird ein impliziter Cast durchgeführt. **Beispiel**:

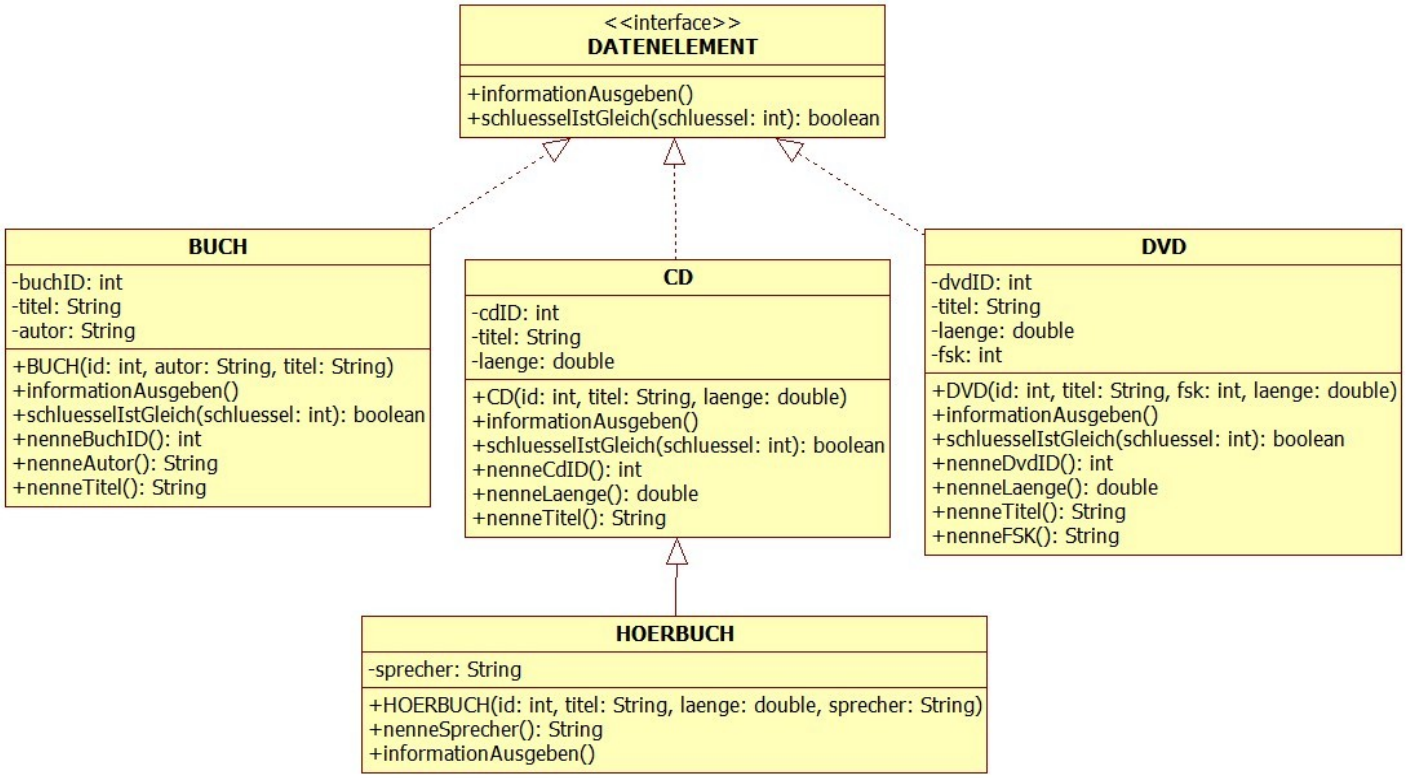
```
BUCH buch1 = new BUCH(1,"BUCH_autor","BUCH_titel");
DATENELEMENT d1  = buch1;  //impliziter cast
d1.
```

```
boolean     equals(Object)
Class<?>    getClass()
int         hashCode()
void        informationAusgeben()
```

DATENELEMENT

`void informationAusgeben()`

→ Mit dem Bezeichner d1 kann nunmehr nicht auf alle BUCH-Methoden (nenneTitel(), nenneAutor(),...) zugegriffen werden, sondern lediglich auf die Interface-Methoden.



Explizite Datentypumwandlung / expliziter Cast

Sobald die andere Richtung beschritten werden soll (allgemein -> speziell), wird ein ***expliziter Cast*** notwendig. Dieser ist allerdings nur erfolgreich, wenn das Objekt, das umgewandelt werden soll, mindestens so speziell wie der gewünschte Datentyp ist.

Ein ***expliziter Cast*** wird erzeugt über ***(NEUERDATENTYP)*** Objektbezeichner/Referenz

Vorsicht: Diese Datentypumwandlung funktioniert nur bei Referenzen, um Zugriff auf die zusätzlichen Attribute und Methoden der Unterklasse zu erhalten.

Objekte können nicht umgewandelt werden: Stell dir vor du würdest ein CD-Objekt in ein HOERBUCH-Objekt umwandeln wollen. → Problem: Das Attribut sprecher würde gar nicht existieren und entsprechend die Methode nenneSprecher() nicht funktionieren.

Beispiele:

Das allgemeinere DATENELEMENT soll zur spezielleren CD werden.

→ funktioniert nur, falls das Objekt ursprünglich als CD oder HOERBUCH instanziiert wurde. Sonst käme es zu einer ClassCastException nicht beim Kompilieren sondern während der Laufzeit des Programms, eben dann, wenn der Fehler auftritt.

```
DATENELEMENT d1 = new BUCH(1,"BUCH_autor","BUCH_titel");
DATENELEMENT d2 = new DVD(2,"DVD_titel",12, 120.2);
DATENELEMENT d3 = new CD(3,"CD_titel",80.3);
DATENELEMENT d4 = new HOERBUCH(4,"HOERBUCH_titel",3.2,"HOERBUCH_sprecher");

BUCH buch1 = (BUCH)d1;
DVD dvd1 = (DVD)d2;
CD cd1 = (CD)d3;
HOERBUCH hb1 = (HOERBUCH)d4;

System.out.println(buch1.nenneTitel());
System.out.println(dvd1.nenneFSK());
System.out.println(cd1.nenneLaenge());
System.out.println(hb1.nenneSprecher());
```

CD cd = new CD(4,“CD_Titel“,80.3);

HOERBUCH hoerbuch = (HOERBUCH)cd; → ***ClassCastException***

Ausnahme bei Zahlen/primitiven Datentypen:

double zahl = 5.1234;

int d = (int)zahl;
(Falls eine Fließkommazahl zu einer Ganzzahl gecastet wird, gehen die Nachkommastellen verloren. Diese Information kann eine Ganzzahl nicht speichern. → keine ClassCastException, denn Zahlen sind primitive Datentypen und keine Klasse)

Aufgabe 3:

Öffne das Projekt Hund und vollziehe den Code der Klasse TestklasseCasten nach.

Polymorphie in Java

Die Polymorphie der Objektorientierten Programmierung ist eine Eigenschaft, die immer im Zusammenhang mit Vererbung und Schnittstellen (Interfaces) auftritt. Eine Methode ist ***polymorph*** („***vielgestaltig***“), wenn sie ***in verschiedenen Klassen die gleiche Signatur hat***, jedoch erneut implementiert ist.

Gibt es in einem Vererbungszweig einer Klassenhierarchie mehrere Methoden auf unterschiedlicher Hierarchieebene, jedoch mit gleicher Signatur, wird ***erst zur Laufzeit*** bestimmt, welche der Methoden für ein gegebenes Objekt verwendet wird. Bei einer mehrstufigen Vererbung wird jene Methode verwendet, die direkt in der Objektklasse definiert ist, oder jene, die im Vererbungszweig am weitesten "unten" liegt.

Aufgabe 4:

Öffne das Projekt Hund und vollziehe den Code der Klasse TestklassePolymorphie nach.

Abstrakte Klassen und Methoden

Klassen und Methoden können in Java den Modifikator ***abstract*** bekommen. Bei Klassen bedeutet das abstract, dass man ***keine Objekte von dieser Klasse erzeugen kann***. Dies kann bei Oberklassen Sinn ergeben, wenn man z.B. nicht möchte, dass ein allgemeiner undefinierter Hund erzeugt werden soll. Die Attribute und Methoden werden weiterhin normal vererbt.

In abstrakten Klassen können es ***abstrakte Methoden*** definiert werden. Diese werden genauso wie die Methode beim Interface angegeben (ohne Methodenrumpf). Abstrakte Methoden einer Oberklassen ***müssen von den Unterklassen implementiert werden*** (→ wie beim Interface)

Aufgabe 5:

a) *Öffne das Projekt AbstrakterHund und definiere den Hund als abstrakte Klasse und teste, ob du nun kein Objekt von Hund mehr erzeugen kannst.*

b) *Deklariere die Methode bellen() als abstrakte Methode. Stelle anschließend wieder übersetzbare Code her, indem du die auftretenden Fehlermeldungen behebst.*