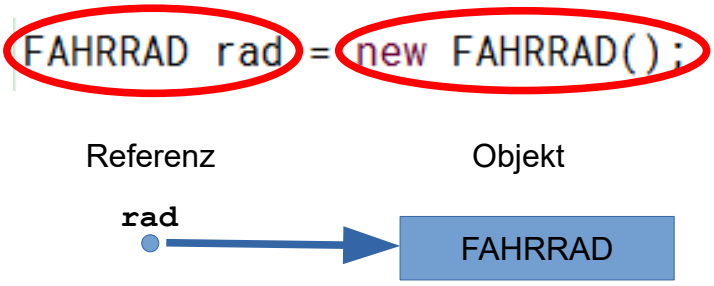
Referenzkonzept

Eine ***Referenz zeigt/verweist auf ein existierendes Objekt.***



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

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:

Durch das Konzept der Referenzen ist es möglich, dass:

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

Aufgabe 1:

Öffne das Projekt Fahrrad und erstelle eine neue Person. Beantworte die folgenden Fragen mithilfe des Codes und der Ausgabe des Programms:

Wie viele Referenzattribute vom Typ FAHRRAD existieren im Laufe des Programms?

Wie viele verschiedene Objekte vom Typ FAHRRAD existieren im Laufe des Programms?

Gibt das Programm bei deinem Sitznachbar dieselben Speicheradressen aus? Und warum ist das nicht so?

Was wird beim Vergleich von Referenzen eigentlich genau verglichen?

Was wird bei der Fahrradübergabe zum Händler eigentlich genau übergeben und zurückgegeben?

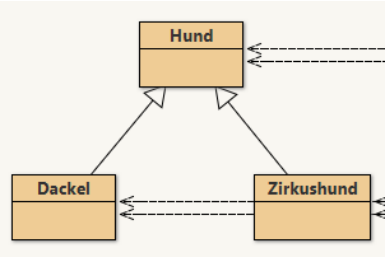
Wie viele Referenzen vom Typ Fahrrad existieren im Laufe des Programms? Tipp: Beachte auch den Fahrradhändler!

Aufgabe 2:

Öffne das Projekt Warteschlange und bearbeite die Aufgaben 1 und 2.

Referenzen in Vererbungshierarchien

- Eine Referenz der Oberklasse ***kann ein Objekt der Unterklasse referenzieren*** (Jeder Dackel ist ein Hund; der Dackel erbt ja auch alle Attribute/Eigenschaften sowie Fähigkeiten/Methoden von der Klasse Hund → somit ist er ja auch ein Hund).
- Aber eine Referenz der Unterklasse ***kann* NICHT *ein Objekt der Oberklasse referenzieren*** (Ein Hund muss kein Dackel sein, somit kann die speziellere Dackelreferenz keinen allgemeineres Objekt Hund referenzieren).




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

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

Aufgabe 3:

Bearbeite vom Projekt Schlange die Aufgabe 3.

Aufgabe 4:

Öffne das Projekt „Übung Polymorphie“ und verschaffe dir einen Überblick über das Klassendiagramm. Gehe anschließend in die Klasse POLYMORPHIE und beantworte die folgenden Fragen:

1. Teil: ausführbare Methoden

Im Konstruktor: Welche Methoden aus welchen Klassen in diesem Projekt (abgesehen von allgemeinen Objektmethoden wie wait() oder equals()) kannst du auf den jeweiligen Referenzen t, k, h, d, s aufrufen?

In der Methode unterschiedlicheDatenypen(): Welche Methoden aus welchen Klassen in diesem Projekt (abgesehen von allgemeinen Objektmethoden wie wait() oder equals()) kannst du auf den jeweiligen Referenzen t, k, h, t2, h2 aufrufen?

Welche Folgerung kannst du aus den letzten beiden Teilaufgaben ziehen? Bestimmt die Referenz oder das Objekt die **aufrufbaren** Methoden?

2. Teil: ausgeführte Methoden

Rufe die **Methode lauteAusgeben()** auf und vollziehe die Ausgabe anhand des Codes der Klassen nach. Warum gibt insbesondere der Dackel „Dackel bellt“ aber der Schäferhund „Hund bellt“ aus?

Rufe die **Methode unterschiedlicheDatenypenLauteAusgeben()** auf und vollziehe die Ausgabe anhand des Codes der Klassen nach.
Warum gibt insbesondere z. B. t2 (Tier 2) „Dackel bellt“ aus?

Welche Folgerung kannst du aus den letzten beiden Teilaufgaben ziehen? Bestimmt die Referenz oder das Objekt die **ausgeführten** Methoden?

Polymorphie und Casting

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***.

Unter ***implizitem Casting bei Referenzen*** versteht man das Zuweisen ***eines spezielleren Objekt einer allgemeineren Referenz***.
→ funktioniert ohne weiteres Zutun

```
TIER t = new KATZE();
HUND h2 = new SCHÄFERHUND();
```

Unter ***explizitem Casting bei Referenzen*** versteht man die Datentypänderung einer ***allgemeineren Referenz auf eine speziellere***, um auf alle Methoden des Objekts zuzugreifen zu können. Dies funktioniert nur, wenn das Objekt auch entsprechend des spezielleren Datentyp ist.

```
TIER t = new KATZE();
KATZE k = (KATZE) t;
k.schnurren();
```

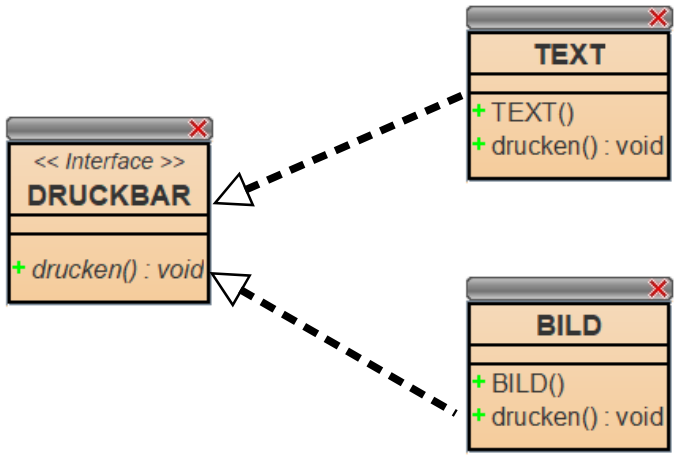
Wichtig: Objekte können ihren Datentyp nicht verändern! → Aus einem Hund kann man keine Katze machen!

Casting bei primitiven Datentypen (int, double, char, boolean) hängt an der Interpretation der Codierung des jeweiligen Datentyps. → **Stoff der 11. Jahrgangsstufe: Codierung und Verschlüsselung**

Interfaces und Referenzen

Ein Interface ist eine ***Schnittstelle***, die Klassen mit dem Interface immer vollständig implementieren müssen. Konkret beinhaltet ein Interface (z.B. DRUCKBAR, ANSTELLBAR, … usw.) ***Methoden(-deklarationen)***, die die Klassen mit diesem Interface für sich selbst entsprechend passend 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 {

    public void drucken(){
        //TODO
    }
}
```

Der Sinn davon ist , dass jede Klasse diese Methode drucken() 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();
```

Es gibt aber **keine Objekte vom Typ DRUCKBAR**, da DRUCKBAR ein Interface ist und keine Klasse. → nur von Klassen können Objekte erzeugt werden!

Aufgabe 5:

Bearbeite vom Projekt Schlange die Aufgabe 4.

Für ganz Schnelle: Bearbeite vom Projekt Schlange die Aufgabe 5.

Aufgabe 6:

Öffne und bearbeite das Projekt „Übung Interface“.

Zusammenfassung: wann nimmt man was?

Während eine klassische Vererbung (extends KLASSE) ein „ich bin ein“ zum Ausdruck bringt (ein DACKEL ist ein HUND), so verwendet man ein Interface (implements INTERFACE) anstatt eine Oberklasse immer dann, wenn man ein „ich kann ... tun“ zum Ausdruck bringen möchte.

Interfaces	***beschreiben also ein oder mehrere Methoden, die nötig sind, um eine Fähigkeit „nachzurüsten“.***
Oberklassen	***hingegen bringen zum Ausdruck, dass man eine Spezialisierung einer allgemeineren Spezies darstellt.***

Zusatz: 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 allgemeines undefiniertes Tier 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)

Unterschiede zwischen abstrakten Klassen und Interfaces

	Abstrakte Klasse	**Interface**
Attribute	Kann Attribute haben und vererben	Hat keine Attribute
Methoden	Kann normale Methoden mit Methodenrumpf bzw. Methodeninhalt haben, die vererbt werden. Kann auch abstrakte Methoden haben, die von den Unterklassen implementiert werden müssen.	Hat nur Methodensignaturen und Klassen mit diesem Interface müssen diese Methoden implementieren.
Logik	Siehe oben: Zusammenfassung	Siehe oben: Zusammenfassung