Elektronikschule Tettnang	Softwareentwicklung	Datum:	Klasse:
	Polymorphie / instanceof Upcasting und Downcasting	Name:	

Ausgangslage:

```
public class Tier {
    public void speak() {
        System.out.println("schnauf schnauf");
    }
}

public class Vogel extends Tier {
    public void speak() {
        System.out.println("zwitscher zwitscher");
    }
    public void fly() {
        System.out.println("fliegen");
    }
}

public class Katze extends Tier {
    public void speak() {
        System.out.println("miau miau");
    }
    public void jump() {
        System.out.println("springen");
    }
}
```

`speak()` ist überschrieben aus der Elternklasse.
`fly()` und `jump()` sind individuelle Methoden der Kindklassen.

Ziel:
Wir stellen uns nun vor wir erzeugen in der Testklasse viele Vögel und viele Katzen und vielleicht noch weitere Tiere wie Hunde oder Elefanten. Diese Tiere sollen dann in einer Schleife durchlaufen werden und für jedes Tier die `speak()`-Methode aufgerufen werden.

Bisheriger Ansatz:
Vögel und Katzen jeweils in separaten Arrays oder Listen speichern und anschließend separat durchlaufen.

```
public class Test {
    public static void main(String[] args) {
        Vogel v1 = new Vogel();
        …
        ArrayList<Vogel> vogelListe = new ArrayList<Vogel>();
        vogelListe.add(v1);
        …
        Katze k1 = new Katze();
        …
        ArrayList<Katze> katzenListe = new ArrayList<Katze>();
        katzenListe.add(k1);
        …
        for(Vogel v : vogelListe) {
            v.speak();
        }
        for(Katze k : katzenListe) {
            k.speak();
        }
    }
}
```

Problem: _____

Lösung mit Upcasting:

```
public class Test {
    public static void main(String[] args) {
        Tier t1 = new Vogel();        // Upcasting Vogel zu Tier
        Tier t2 = new Katze();        // Upcasting Katze zu Tier
        …
        ArrayList<Tier> tierListe = new ArrayList<Tier>();
        tierListe.add(t1);
        tierListe.add(t2);
        …
        for(Tier t : tierListe) {
            t.speak(); // Namensraum von Tier, aber Methoden der Kindklassen
        }              // Ausgabe „zwitscher zwitscher“ oder „miau miau“
    }
}
```

Die Variablen `t1` und `t2` sind vom Typ `Tier` und haben somit den Namensraum der Klasse `Tier`. `t1` enthält aber ein Objekt vom Typ `Vogel` und `t2` ein Objekt vom Typ `Katze`. Der Compiler kennt die `speak()`-Methode, da diese im Namensraum von `Tier` ist, aufgerufen wird aber die Methode der Kindobjekte. Dieses Verhalten nennt man Polymorphie.

Weiteres Ziel:
Aufrufen der individuellen Methoden `fly()` und `jump()` für jedes Tier in der Schleife.

Problem: _____

Lösung mit dem instanceof-Operator und Downcasting:

```
public class Test {
    public static void main(String[] args) {
        Tier t1 = new Vogel();        // Upcasting Vogel zu Tier
        Tier t2 = new Katze();        // Upcasting Katze zu Tier
        …
        ArrayList<Tier> tierListe = new ArrayList<Tier>();
        tierListe.add(t1);
        tierListe.add(t2);
        …
        for(Tier t : tierListe) {
            t.speak();                // Egal ob Vogel oder Katze

            if(t instanceof Vogel) {  // Ist dieses Tier ein Vogel?
                Vogel v = (Vogel)t;   // Downcasting zu Vogel
                v.fly();
            }
            if(t instanceof Katze) {  // Ist dieses Tier eine Katze?
                Katze k = (Katze)t;   // Downcasting zu Katze
                k.jump();
            }
        }
    }
}
```

Mit dem `instanceof`-Operator lässt sich überprüfen, ob eine Variable ein Objekt von einem bestimmten Typ enthält. Falls ja, kann man die Variable „downcasten“ und bekommt somit Zugriff auf den Namensraum vom Kindobjekt.