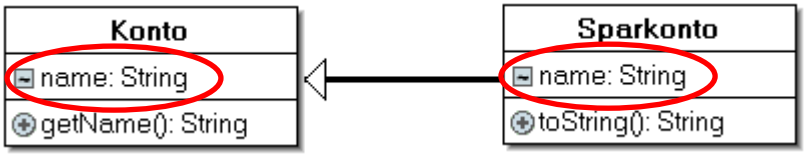
Verdecken und Überschreiben

Im letzten Kapitel haben Sie gelernt, dass man durch die Vererbung Attribute und Methoden der Basisklasse erbt. Zudem können in der Subklasse Elemente definiert werden, die die Elemente der Basisklasse erweitern spezialisieren, aber auch anpassen.

Möchte man vordefinierte Attribute und Methoden in der Subklasse anders einsetzen (Beispielsweise möchte man die Ausgabe von abgeleiteten Objekten durch die Methode **toString()** anders gestalten, sie wurde aber bereits in der Basisklasse definiert), so können diese verdeckt bzw. überschrieben werden.

Verdecken

Wenn Sie in der Subklasse ein **Attribut** definieren, das den gleichen Namen hat wie das geerbte Attribut, so verdeckt das neue Attribut in der abgeleiteten Klasse das geerbte Attribut.




```
// Klasse Konto
public class Konto {
  String name = "Konto";

  /** Getter-Methode liefert Namen zurück */
  public String getName(){
    return name;
  }

}

// abgeleitete Klasse Sparkonto
public class Sparkonto extends Konto{
  String name="Sparkonto";

  /** erzeugt eine String-Ausgabe*/
  public String toString(){
    return "Ich bin ein " + name +
    "\n aber auch ein " + super.name;
  }
}

// Testklasse zur Verdeutlichung
public class Verdeckung {

  public static void main(String[] args) {
    Sparkonto meins = new Sparkonto();
    System.out.println(meins);
    // Methode der Oberklasse greift auf Attribut der Oberklasse zu
    System.out.println(meins.getName());
    // Hier wird auf das Attribut der Unterklasse zugegriffen
    System.out.println(meins.name);
    // Hier erfolgt eine Umwandlung in ein Objekt der Basisklasse (Konto)
    System.out.println(((Konto)meins).name);
  }
}
```

Ein Objekt der abgeleiteten Klasse ist gleichzeitig ein (IS-A) Objekt der Superklasse. Dies ist beim Zugriff auf verdeckte Attribute zu bedenken!

Während eine in der Subklasse definierte Methode stets auf das neu definierte, verdeckende Feld zugreift, benutzen Methoden der Superklasse das verdeckte Attribut, das in der Superklasse definiert wurde.

Beispiel: `System.out.println(meins.getName());` →"Konto"

Die beiden Attribute leben nebeneinander für die Laufzeit des Objekts. Möchte man nun von der Unterklasse aus das verdeckte Attribut der Oberklasse zugreifen, so geschieht dies mit dem Schlüsselwort **super**.

Beispiel:

```
public String toString(){
  return "Ich bin ein " + name +
  "\n aber auch ein " + super.name;
}
```

Von außen können Sie ebenfalls über ein Objekt der Subklasse auf das Attribut der Superklasse zugreifen, jedoch nicht mittels des Schlüsselwortes super. In diesem Fall müssen Sie das Objekt in ein Objekt der Superklasse mittels eines **Cast** umwandeln.

Beispiel: `System.out.println(((Konto)meins).name);`

Prinzipiell ist ein verdecken von Variablen zu vermeiden!!!

Überschreiben

Definiert man **Methoden** neu, die bereits in der Superklasse definiert wurden, so spricht man hingegen vom Überschreiben.

Im Gegensatz zur Verdeckung ist das Überschreiben von Methoden für die Vererbung von großer Bedeutung. Sie erlaubt dem Autor der abgeleiteten Klasse die Anpassung der Methode auf seine Bedürfnisse. Ein Beispiel hierfür ist die Methode **toString()** der Superklasse Object. Für jede Instanz einer Klasse ist es sinnvoll diese Methode der Klasse Object anzupassen und eine spezielle Ausgabe zu erzeugen.

Zu beachten ist hierbei, dass die überschrieben Methode neben dem gleichen Namen auch die gleiche Signatur und den gleichen Rückgabetyp wie die Methode der Superklasse haben muss.

Aus der Subklasse heraus kann die Methode der Oberklasse wieder über **super** erfolgen. Aber Objekte vom Typ der erweiterten Klasse können **nicht** mehr die ursprüngliche Implementierung einer Methode benutzen (weder mit **sub.M** noch mit **((Superklasse) sub).M**.

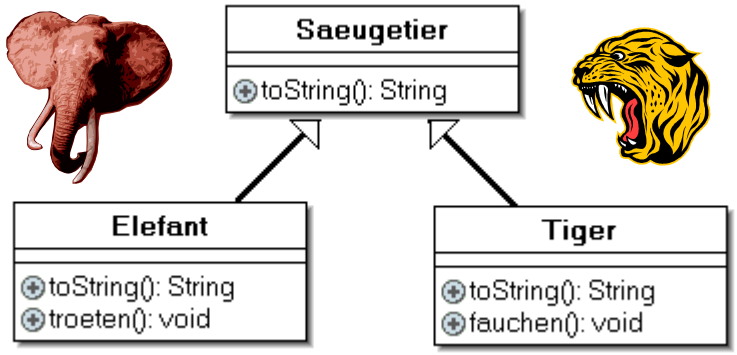
Die Spezielle Bedeutung der Überschreibung von Methoden erfolgt beim Studium des dritten Kriteriums der objektorientierten Programmierung: der Polymorphie.

3.Prinzip der OOP: Polymorphie

Der Begriff der Polymorphie stammt aus dem Griechischen und bedeutet Vielgestaltigkeit.
Die Grundtechnik sind die Definition von polymorphen Methoden und die Arbeit mit polymorphen Variablen. Die Kombination erschließt dem Programmierer ein neues mächtiges Konzept.

Polymorphe Variablen

Ein Objekt einer Klasse **Tiger** kann auch in die Rolle eines Objekts einer (direkten oder indirekten) Oberklasse von **Saeugetier** schlüpfen, denn jedes Objekt der abgeleiteten Klasse enthält auch ein Objekt der Basisklasse.
Dieser spezielle Aufbau gestattet es, Objekte der abgeleiteten Klasse auch als Objekte der Basisklasse zu behandeln.




```
public class Saeugetier {...}
public class Tiger extends Saeugetier {...}

Saeugetier tier = new Saeugetier();
Tiger tiger = new Tiger();

tier = tiger; // polymorphe Variable, da tier im Laufe des Programms auf
              // Objekte unterschiedlichen Typs verweist.
```

Die Variable **tier** ist eine **polymorphe Variable**, da sie im Laufe des Programms auf Objekte unterschiedlichen Typs verweist. Sie haben zuvor das Überschreiben von Methoden kennengelernt.

Was passiert nun bei einem Aufruf der überschriebenen Methode?

```
System.out.println(saeuger);
```

Wird nun die toString-Methode aufgerufen, die zum Typ der Referenz passt, oder diejenige, die zum Typ des Objekts passt?

- Grundsätzlich ruft der Compiler die Methode auf, die zum **Typ des Objekts** gehört.
- Lediglich bei Methoden, die als **private** oder **static** deklariert wurden, ruft der Compiler die Methodenversion auf, die zum Typ der Referenz gehört.