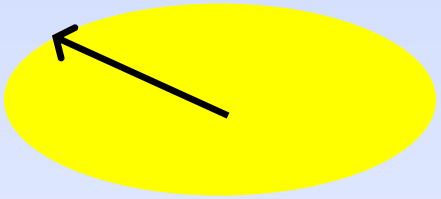
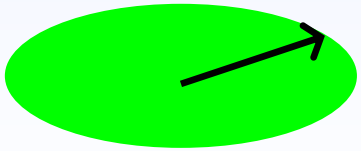
Kreise

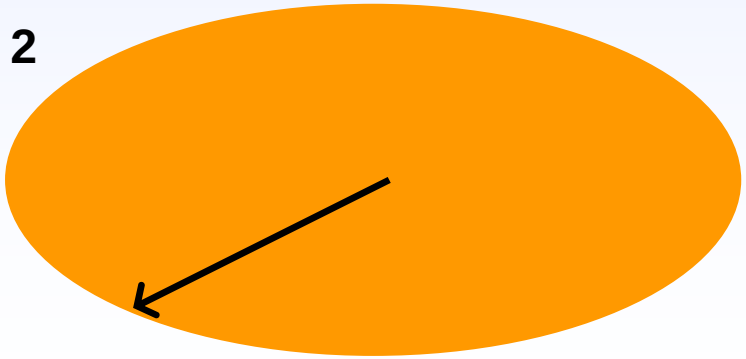


Radius r = 1,5

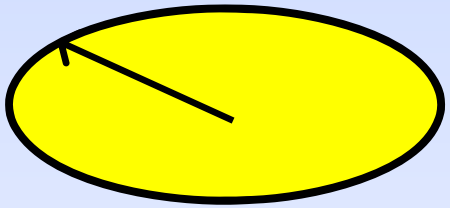
Radius r = 1



Radius r = 2

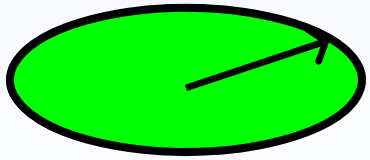


Kreise: Umfang U=2*pi* r

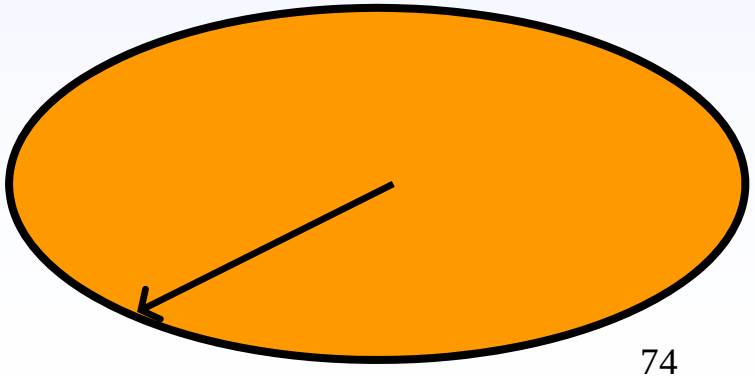


Radius r = 1,5
Umfang U = 9,42

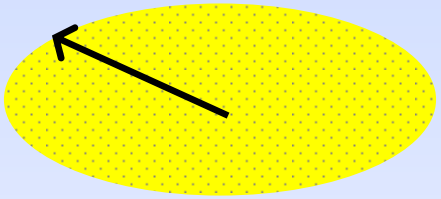
Radius r = 1
Umfang U = 6,28



Radius r = 2
Umfang U = 12,56



Kreise: Fläche $F=r^2\,pi$

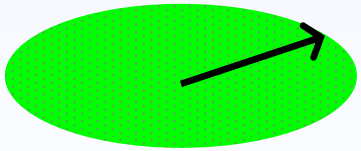


Radius r = 1,5

Fläche F = 7,07

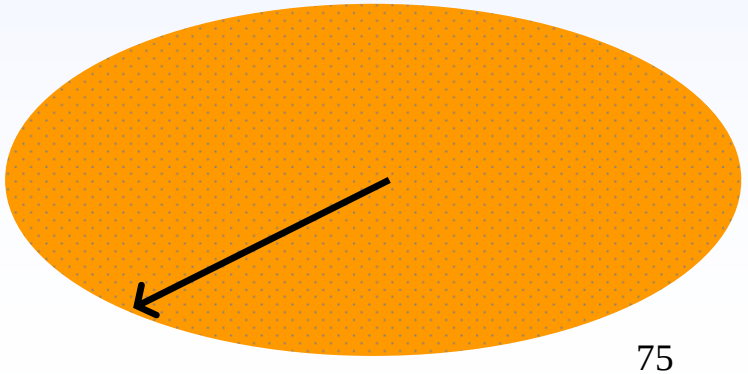
Radius r = 1

Fläche F = 3,14

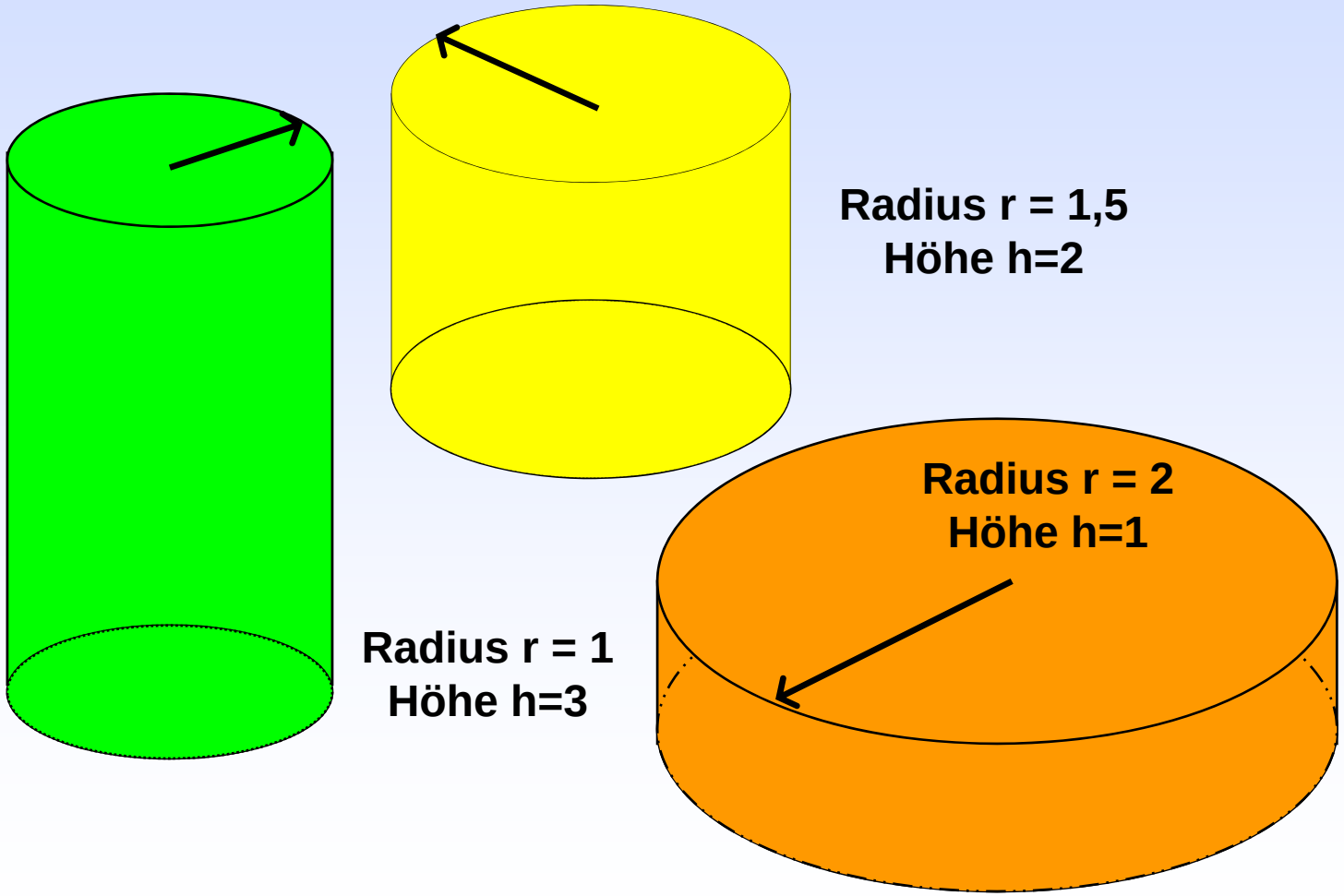


Radius r = 2

Fläche F = 12,56



Zylinder: Oberfläche ist Uh+2F



Programm Zylinder (1)

```
class Kreis {
      double r;

      public Kreis(){}
      public double flaeche()
            {return 3.14*r*r}
      public double umfang() {…}
      public setRadius() {…}

}
```

Programm Zylinder (2)

```
class Zylinder extends Kreis {
      private double hoehe;

      public Zylinder(){}

      public setHoehe() {…}
      public double flaeche() {
            double fl , umf;
            umf = super.umfang();
            fl = super.flaeche();
            //super.Kreis() //Konstruktor
            return 2*fl + hoehe*umf;
      }
}
```

Dynamische Bindung zur Programmlaufzeit (Polymorphismus) 1/2

```
public class ZylinderTest {
     public static void main(String args[]) {
          Kreis k[] = new Kreis[10];

          for(int i=0 ; i<=9 ; i++) {
           if (Math.random() <= 0.5) {
                k[i] = new Kreis(i+0.5);
           }  else {
            k[i] = new Zylinder();
            k[i].setRadius(i+1);
              ( (Zylinder) k[i]).setHoehe(i+1.5);
          }
          ...
```

Dynamische Bindung zur Programmlaufzeit 2/2

```
            ...
            System.out.println(
                k[i].getClass().getName() +
          "  k[" + i + "] " +
                " mit Radius  " + k[i].r +
                " und Flaeche " +
                k[i].flaeche()
            );
            }
    }
}
```

Polymorphismus

Das Polymorphismus-Prinzip besagt, dass eine Objekt-Variable der Superklasse sowohl auf Objekte der Superklasse als auch auf Objekte abgeleiteter Klassen referenzieren kann

```
k[i] = new Kreis(i+0.5); //Basisklasse
k[i] = new Zylinder(); //Abgeleitete Klasse
```

Static-1

Static-Variablen sind sogenannte Klassenvariablen, d. h. sie existieren nur einmal pro Klasse. Jede Instanz der Klasse kann darauf zugreifen

z.B

```
class Auto{
        public
                static int anzahlDerInstanzen;
        }
```

Statische Variablen – Klassenvariablen

```
class Kreis3{
      public static int anzahl_kreise = 0 ;
      final static double MAXRADIUS=5.0;
      private double r;
      ...
      public Kreis3(double r) {
            this.r=r;
            anzahl_kreise++;
      }
...
}
```

Static -2

Verwendung als globale Variable über eigene Klasse

```
public class GlobalerZaehler {
                public static int wert = 0;}
```

- **Zugriff aus jeder anderen Instanz**
- **es muss keine Instanz für die globale Variable angelegt werden (Klassenname wird beim Zugriff vorangestellt)**

```
GlobalerZaehler.wert=GlobalerZaehler.wert+1;
```

Alternative: static-Variable in gemeinsamer Basisklasse anlegen und vererben

Static -3

Verwendung zum Anlegen einer Funktion (z. B. einer mathematischen Funktion),

so dass keine Klasseninstanz notwendig wird, um diese Funktion nutzen zu können.

Aufruf durch „Klassenname.funktionsname()“

```
public class Klassenname
{
	public static void funktionsname()
	{
	}
}
```

Aufgabe

Binomialkoeffizient mit static-Methode

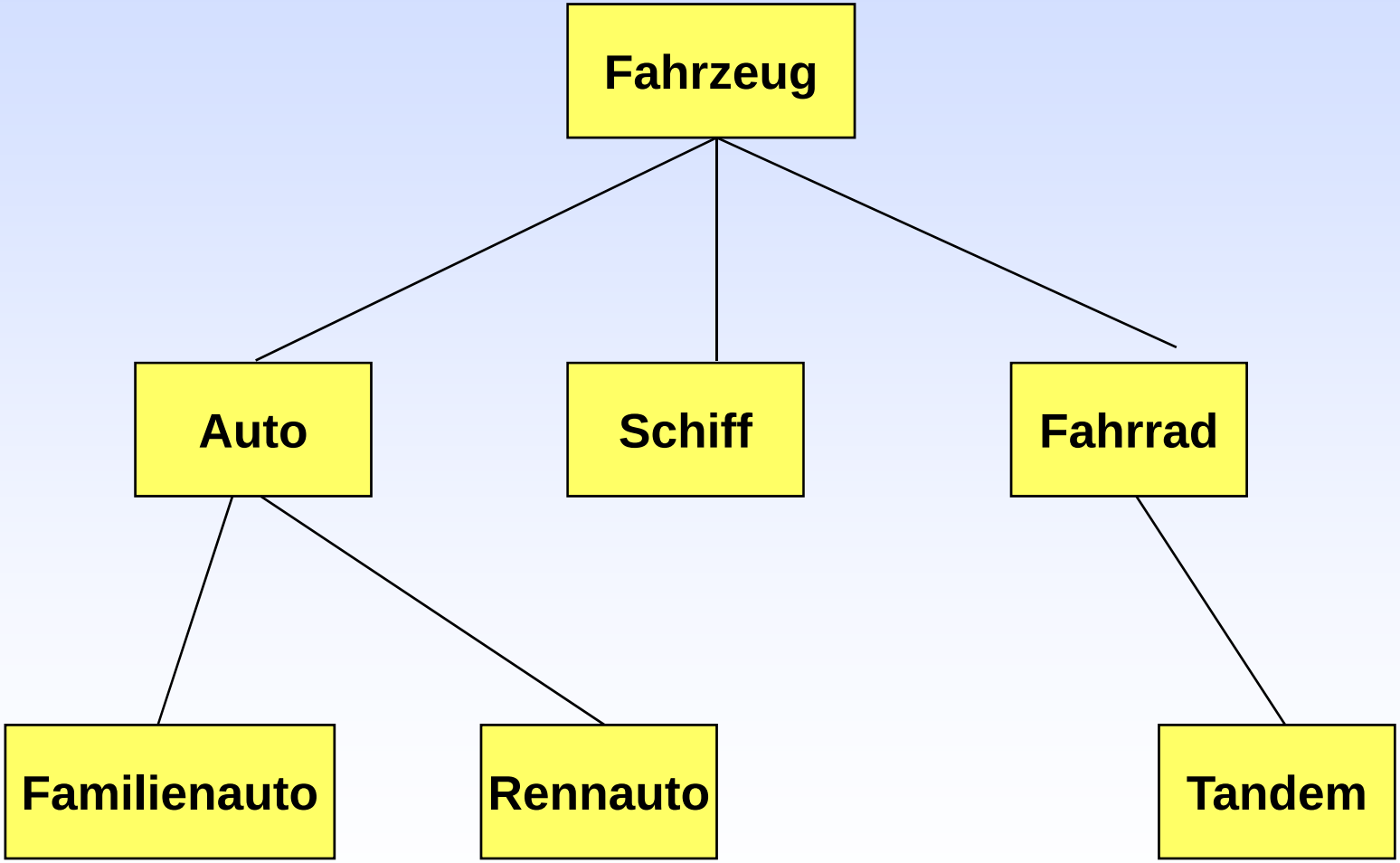
Modifikatoren: Zugriff

	public	**protected**	default	**private**
Klasse	ja	ja	ja	ja
Paket	ja	ja	ja	nein
Abgeleitete Klasse	ja	ja	nein	nein
überall	ja	nein	nein	nein

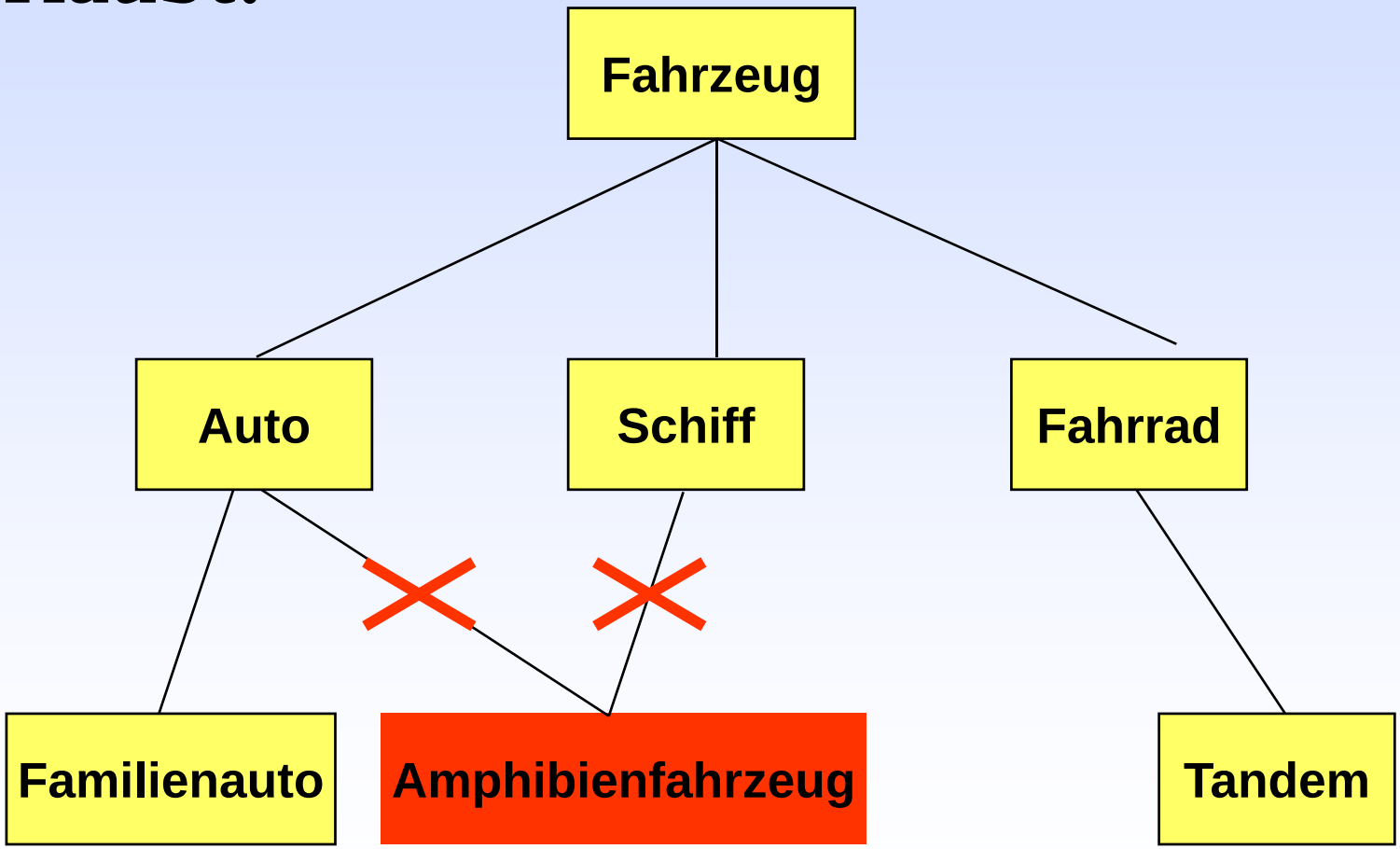
Vorteile eines konsequenten Information Hiding

- Es werden einfache und klare Schnittstellen für die Verwendung einer Klasse geschaffen.
- Es wird die Sicherheit erhöht, dass Daten stets gültige Werte haben.
- Eine weitgehende Unabhängigkeit für die internen Programmierung ist gewährleistet.
- Programmierfehler beim Zusammenspiel der Klassen werden weitgehend verhindert.

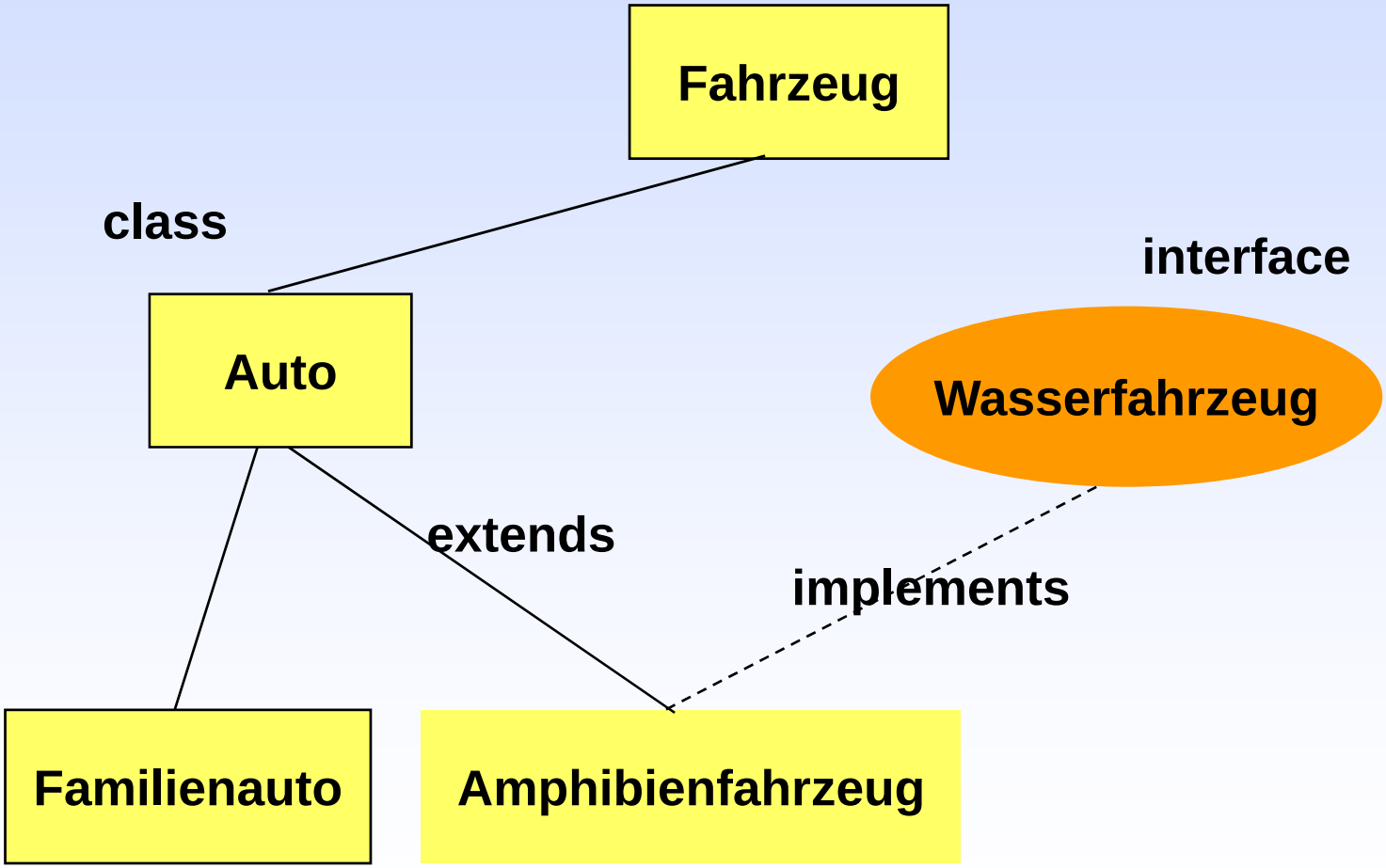
Eine Hierarchie von Fahrzeugen



Mehrfache Vererbung ist nicht erlaubt!



Simulation von Vererbung



Interface

```
public interface InterfaceName {

      public type name1 (parameterliste) ;

      public void name2 (parameterliste) ;
      ...
}
```

Anmerkung: Eine Mehrfachvererbung kann u. U. auch durch eine Kaskade von Einfachvererbungen nachgebildet werden

Interface: Implementierung

```
public class ClassName implements
             InterfaceName {
      ...
      public type name1 (parameterliste) {
            ... ; // Statements
      }

      public void name2 (parameterliste) {
      }
}
```

Mehrere Interfaces

```
public class SubClass
             extends SuperClass
             implements Interface1,
                                Interface2
// Zuerst muss extends genutzt werden
{
      ...
}
```

Interface – zu beachten

- Keine Variablen als Attribute nur Konstanten (implizit: public static final)
- Nur abstrakte Methoden

Aufgabe

Klassen Mehrfachvererbung