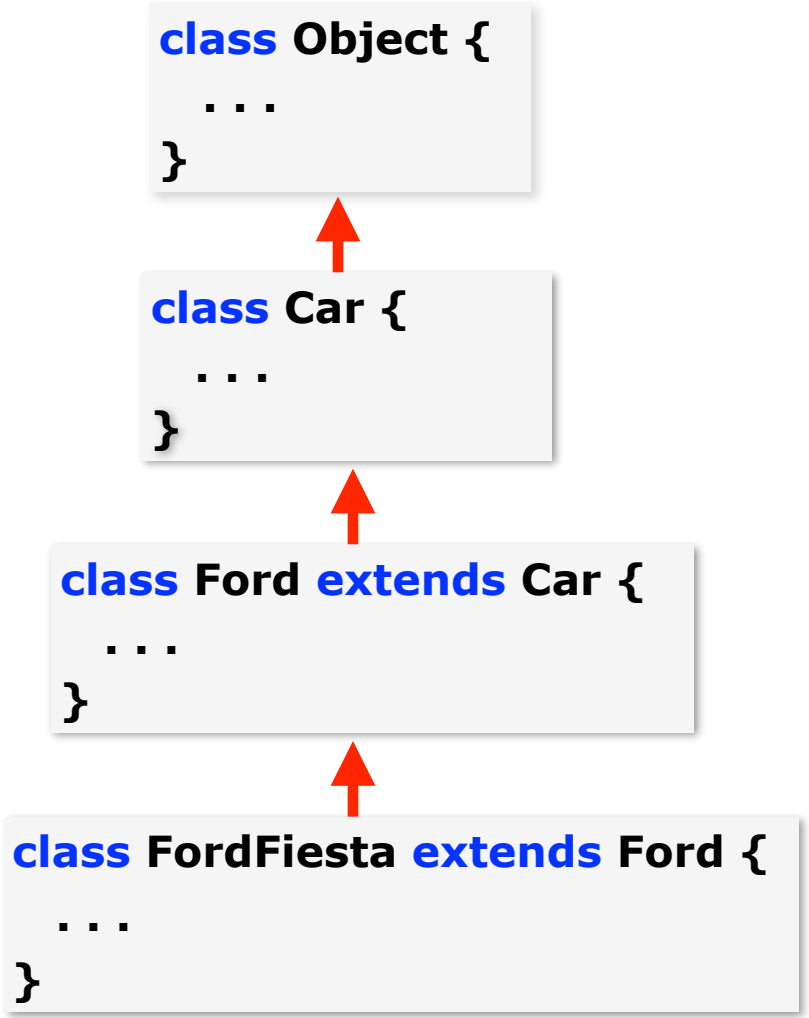
Vererbung

Ein wesentliches Merkmal objektorientierter Sprachen ist die Möglichkeit, Eigenschaften vorhandener Klassen auf neue Klassen zu übertragen. (**Wiederverwendbarkeit**)

Durch Hinzufügen neuer Elemente oder Überschreiben der vorhandenen kann die Funktionalität der abgeleiteten Klasse erweitert werden.

Vererbung



Legale Zuweisungen sind:

```
...
Object obj = new Car();
Car car = new Ford();
Car car = new FordFiesta();
Object obj = new FordFiesta();
...
```

Illegale Zuweisungen sind z.B.:

```
...
Car car = new Object();
FordFiesta ford = new Ford();
...
```

Abstrakte Methoden

- Abstrakte Methoden enthalten **nur die Deklaration des Methodenkopfes**, aber **keine Implementierung** des Methodenrumpfes.

- Abstrakte Methoden haben anstelle der geschweiften Klammern mit den auszuführenden Anweisungen ein Semikolon am Ende. Zusätzlich wird die Definition mit dem Attribut **abstract** versehen.

- Abstrakte Methoden definieren nur eine Schnittstelle, die durch Überschreiben innerhalb einer abgeleiteten Klasse implementiert werden kann.

Beispiel:

```
public abstract void  paint();
```

Abstrakte Klassen

- Eine Klasse, die mindestens eine abstrakte Methode enthält, muss als abstrakte Klasse deklariert werden.
- Die bereits implementierten Methoden in einer abstrakten Klasse können von anderen Klassen geerbt werden, welche die abstrakten Methoden zusätzlich implementieren können.
- **Eine abstrakte Klasse wird in einer Unterklasse konkretisiert, wenn dort alle ihre abstrakten Methoden implementiert sind.**
- Abstrakte Klassen werden mit dem Schlüsselwort **abstract** markiert.

Abstrakte Klassen

- Abstrakte Klassen sind künstliche Oberklassen, die geschaffen werden, um Gemeinsamkeiten mehrerer Klassen zusammenzufassen.
- Abstrakte Klassen dienen nur zur besseren Strukturierung der Software.
- Objekte können nicht aus einer abstrakten Klasse erzeugt werden.
- Die fehlende Implementierung wird in den Unterklassen "nachgeliefert", sonst sind diese auch abstrakt.

Abstrakte Klassen

Freie Universität Berlin

Beispiel

```
public abstract class Figur {
    protected double x, y;
    public double getX() { return x; }
    public double getY() { return y; }
    public void setX( double x ) { this.x = x; }
    public void sety( double y ) { this.y = y; }
    public abstract double area();
}
```

Abstrakte Methode

Implementierung

```
public class Circle extends Figur {
    private double radio;
    static final private double PI = 3.1415926535897;
    public double area() {
        return PI*radio*radio;
    }
}
```

Eine abstrakte Klasse wird in einer Unterklasse konkretisiert, wenn alle ihre abstrakten Methoden implementiert werden.

Kapselung und Abstrakte Datentypen (ADT)

Klassen definieren neue Datentypen und die Operationen, die auf diesen Datentypen erlaubt sind.

Ein **abstrakter Datentyp** ist eine Typdefinition oder Spezifikation, die **unabhängig von einer konkreten Implementierung** ist.

In Java versucht man mit Hilfe von **Interfaces** konkrete Implementierungen von Datentypspezifikation zu trennen.

Interfaces (Motivation)

Beispiel:

Kunde

Schnittstelle
Interface

- Lautstärke
- Senderwechsel
- Farbjustierung

Implementierung

Die Kunden brauchen nur zu wissen, wie das Interface benutzt werden kann.

Die Implementierung muss nur wissen, welches Interface implementiert werden soll.

Die Implementierung kann vertauscht werden ohne die Kunden zu tauschen.

Interfaces

In Java sind Interfaces sowohl ein Abstraktionsmittel (zum Verbergen von Details einer Implementierung) als auch ein Strukturierungsmittel zur Organisation von Klassenhierarchien!

Eine Schnittstelle (interface) in Java legt eine **minimale Funktionalität** (Methoden) fest, die in einer implementierenden Klasse vorhanden sein soll.

Interfaces (Schnittstellen)

- Interfaces sind vollständig abstrakte Klassen.
- keine Methode ist implementiert.
- keine Instanzvariable ist deklariert.
- nur statische Variablen können deklariert werden.

```
public interface Collection {
        public void add(Object o);
        public void remove(Object o);
        public boolean contains(Object o);
}
```

Alle Methoden sind implizit **abstract** und **public**.

Interfaces

Verschiedene Implementierungen desselben Typs sind möglich, und die Implementierungen können geändert werden, ohne dass der Benutzer es merkt!

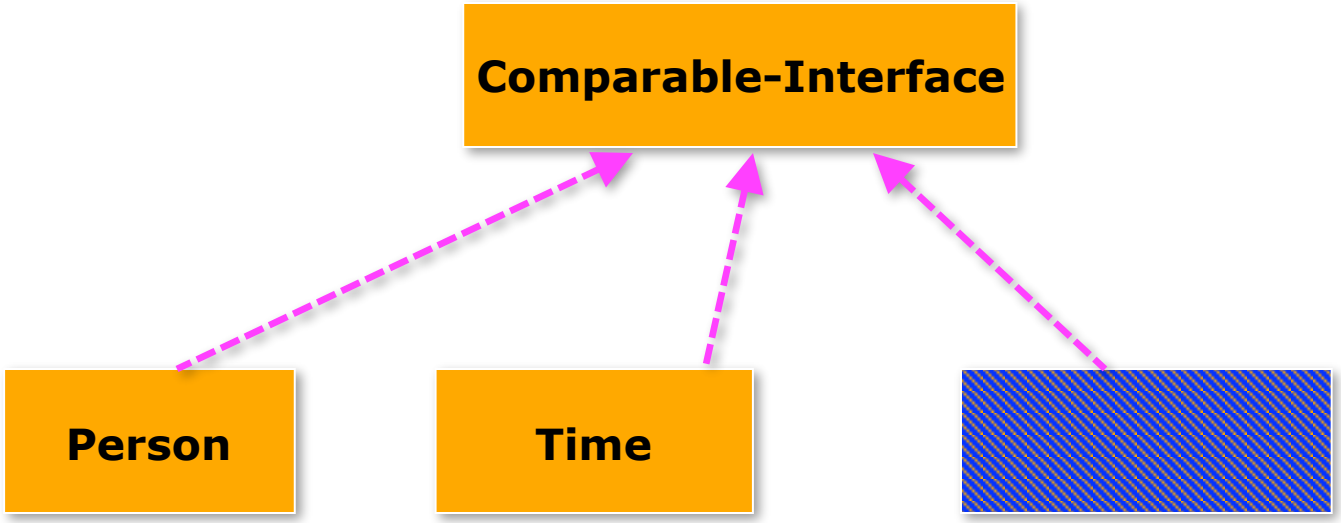
```
public class Set implements Collection {
    public void add( Object o ) { ... }
    public void remove( Object o ) { ... }
    public boolean contains( Object o ) {... }
}
```

Alle Methoden des Interface müssen implementiert werden.

Implementierung von Interfaces

Beispiel:

Comparable-Interface



Interfaces (Schnittstellen)

Freie Universität Berlin

```
interface Comparable {
    int compareTo( Object other );
}
```

```
public class Time implements Comparable {
        private int seconds;
        private int minutes;
        private int hours;
    public int compareTo( Object obj ) {
        Time time = (Time) obj;
        if ( this.toSeconds()<time.toSeconds() )
                return -1;
        else if ( this.toSeconds()>time.toSeconds() )
                return 1;
        else
                return 0;
    }
    ...
```

Interfaces

(Schnittstellen)

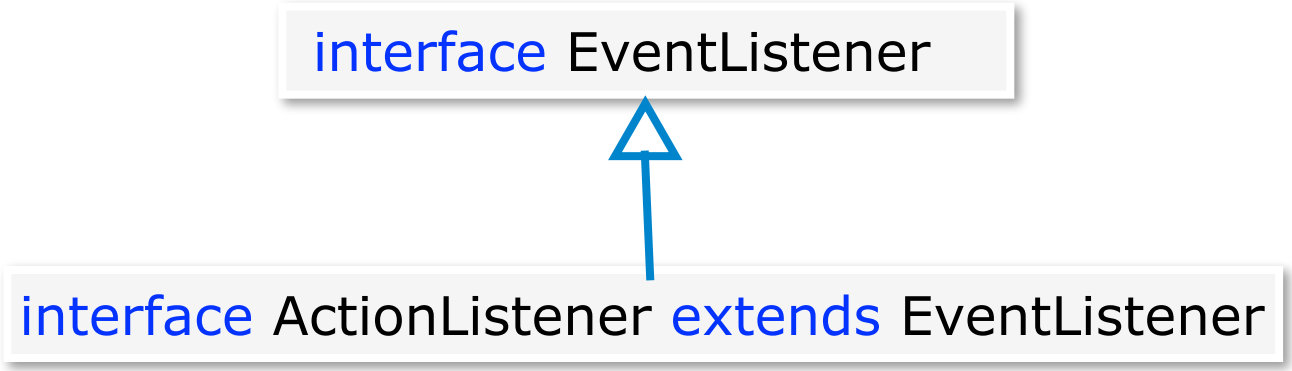
```
public class Sortierer {

    Comparable [] list;

    Sortierer( Comparable[] list ){ this.list = list; }

    public void bubbleSort(){
            boolean swap = true;
            Comparable temp;
            while ( swap ) {
                    swap = false;
                    for ( int i=0; i<list.length-1; i++ ) {
                            if ( list[i].compareTo( list[i+1] ) == 1 ) {
                                    temp = list[i];
                                    list[i] = list[i+1];
                                    list[i+1] = temp;
                                    swap = true;
                            }
                    }
            }
    } // end of class Sortierer
```

```
...
public static void main( String[] args ){
        Time[] zeit = new Time[6];

        zeit[0] = new Time(); zeit[0].setTime(3,10,20);
        zeit[1] = new Time(); zeit[1].setTime(7,10,20);
        zeit[2] = new Time(); zeit[2].setTime(5,10,20);
        zeit[3] = new Time(); zeit[3].setTime(4,10,20);
        zeit[4] = new Time(); zeit[4].setTime(2,10,20);
        zeit[5] = new Time(); zeit[5].setTime(0,10,20);

        Sortierer s = new Sortierer(zeit);

        s.bubbleSort();

        for (int i=0; i<zeit.length; i++){
                System.out.println("zeit["+i+"]="+zeit[i]);
        }
}
```

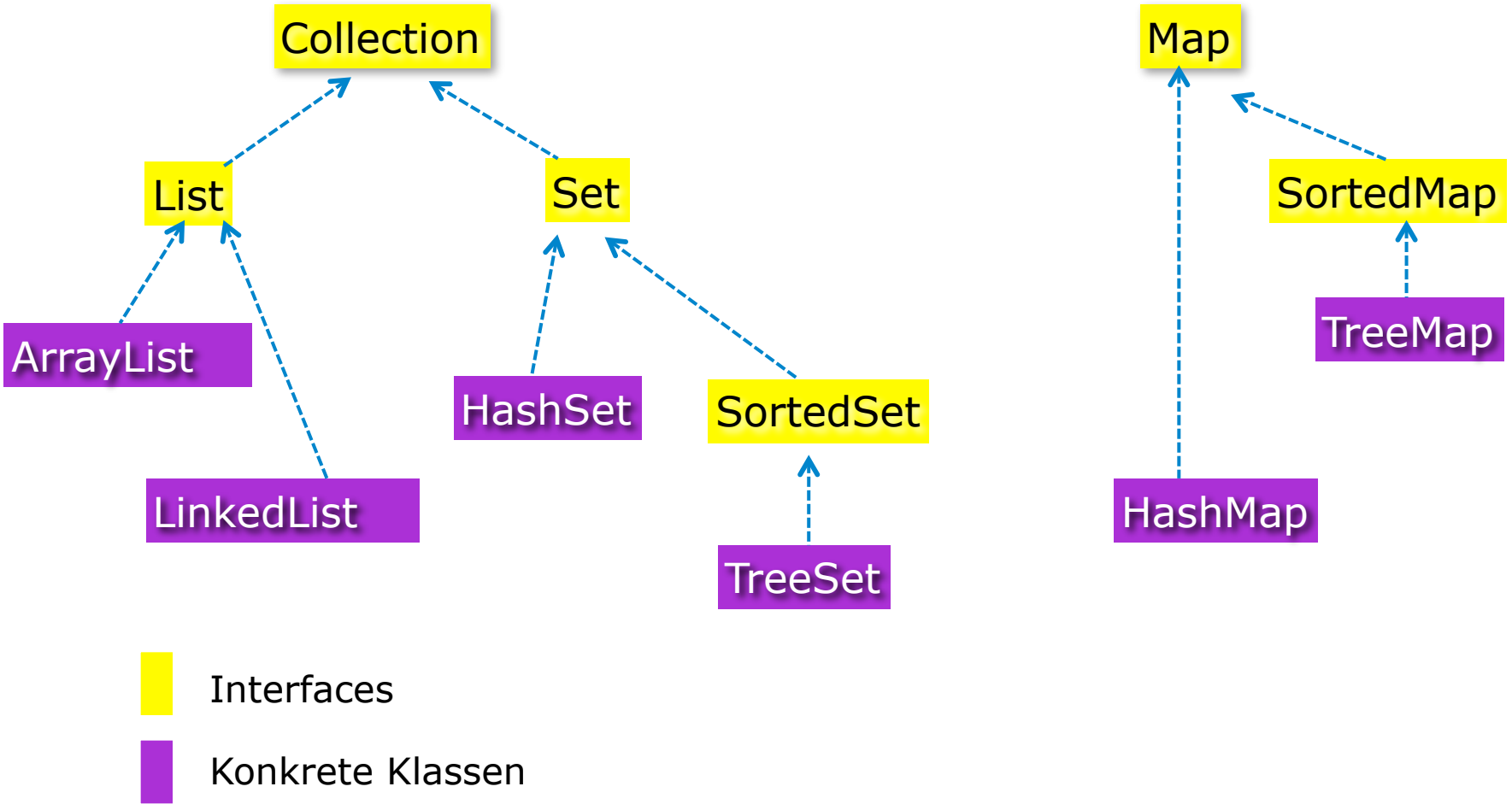
```
zeit[0]=00:10:20
zeit[1]=02:10:20
zeit[2]=03:10:20
zeit[3]=04:10:20
zeit[4]=05:10:20
zeit[5]=07:10:20
```

Interfaces als Strukturierungsmittel

Interfaces in Java können als Unterinterfaces von anderen Interfaces definiert werden.

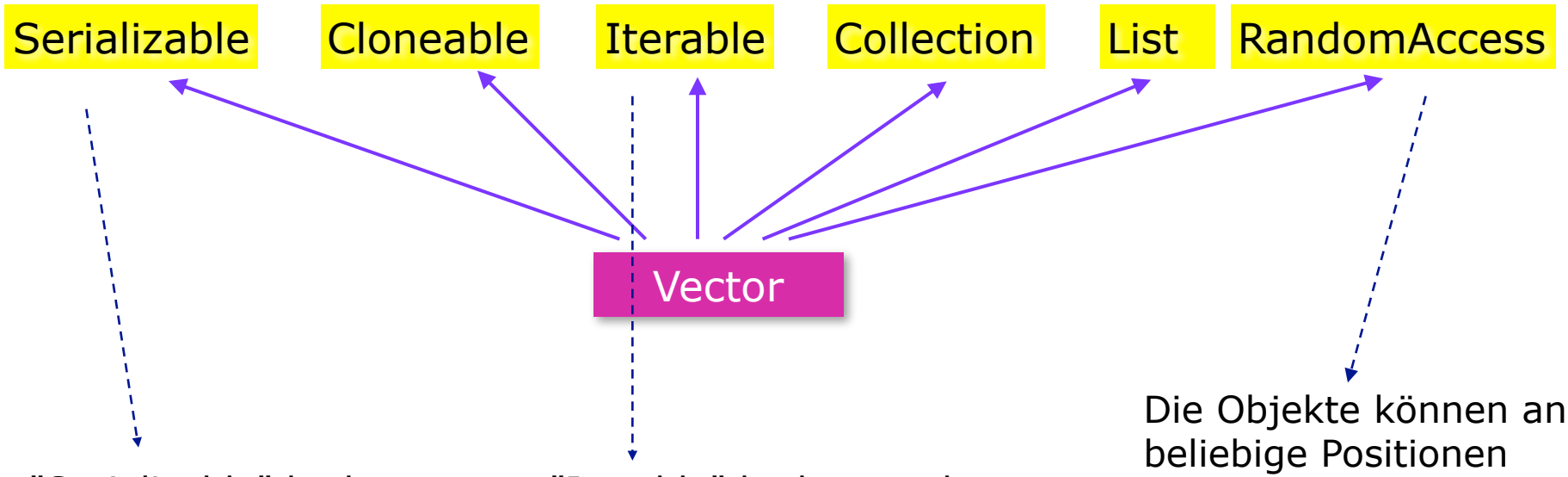


Collection-Klassen



Vektor-Klasse

Dynamische Datenstruktur



"Serializable" bedeutet, dass ein Objekt in eine lineare, sequentielle Darstellung verwandelt werden kann.

"Iterable" bedeutet, dass Methoden zur Verfügung gestellt werden, mit denen eine Datenstruktur durchlaufen werden kann.

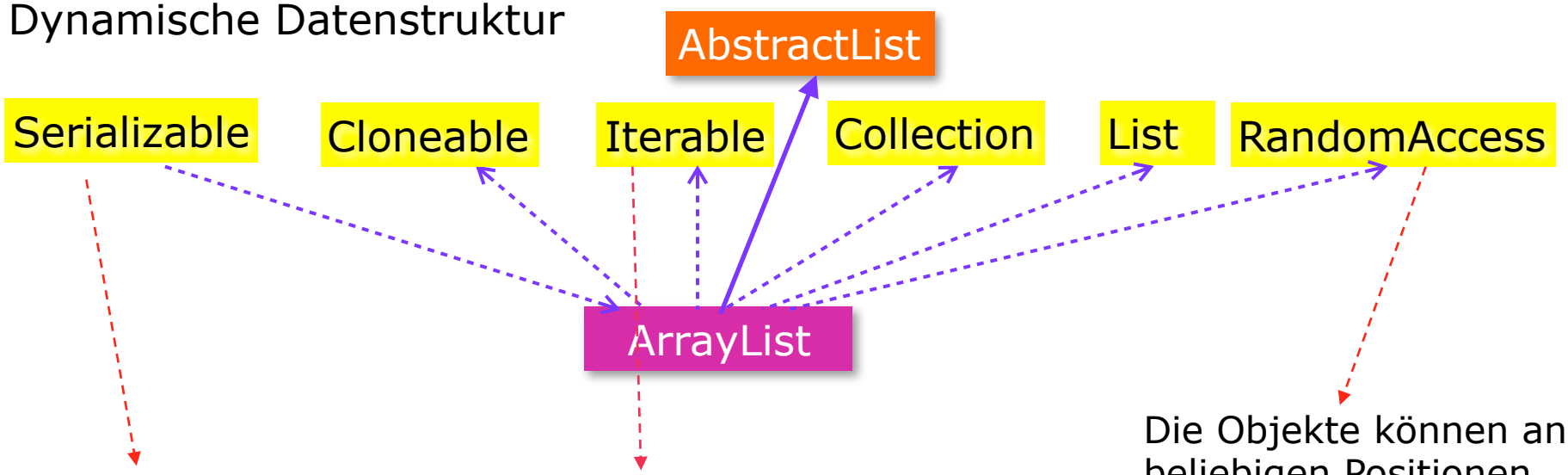
Die Objekte können an beliebige Positionen eingefügt oder gelöscht werden.

Interfaces

Konkrete Klassen

ArrayList-Klasse

Dynamische Datenstruktur



"Serializable" bedeutet, dass ein Objekt in eine lineare, sequentielle Darstellung verwandelt werden kann.

"Iterable" bedeutet, dass Methoden zur Verfügung gestellt werden, mit denen eine Datenstruktur durchlaufen werden kann.

Die Objekte können an beliebigen Positionen eingefügt oder gelöscht werden.

Interfaces
Abstrakte Klassen
Konkrete Klassen

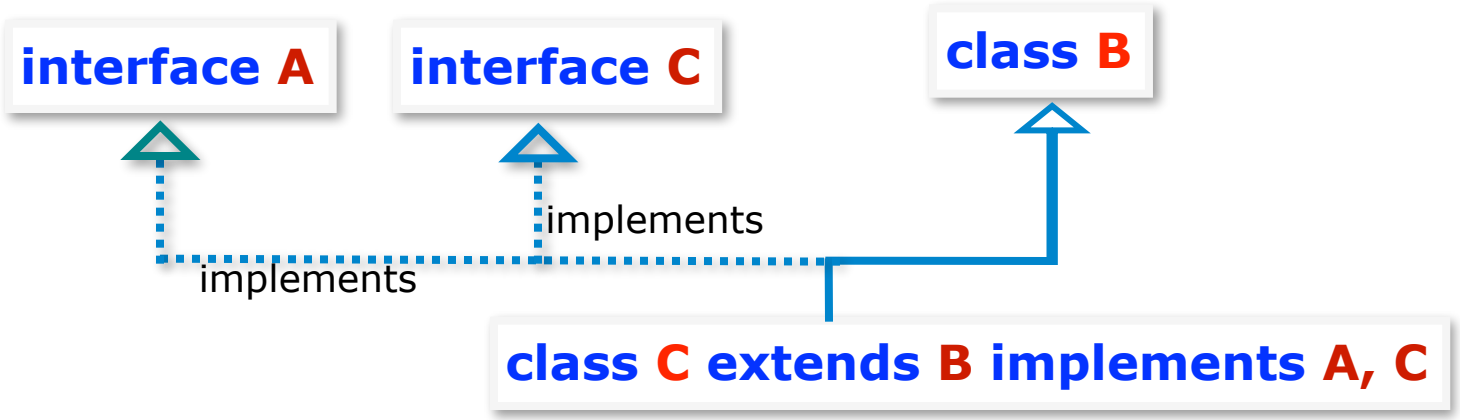
Interfaces als Strukturierungsmittel

Einfache und mehrfache Vererbung

Um die Probleme der Namenskollisionen zu vermeiden wurde in Java mehrfache Vererbung abgeschafft.

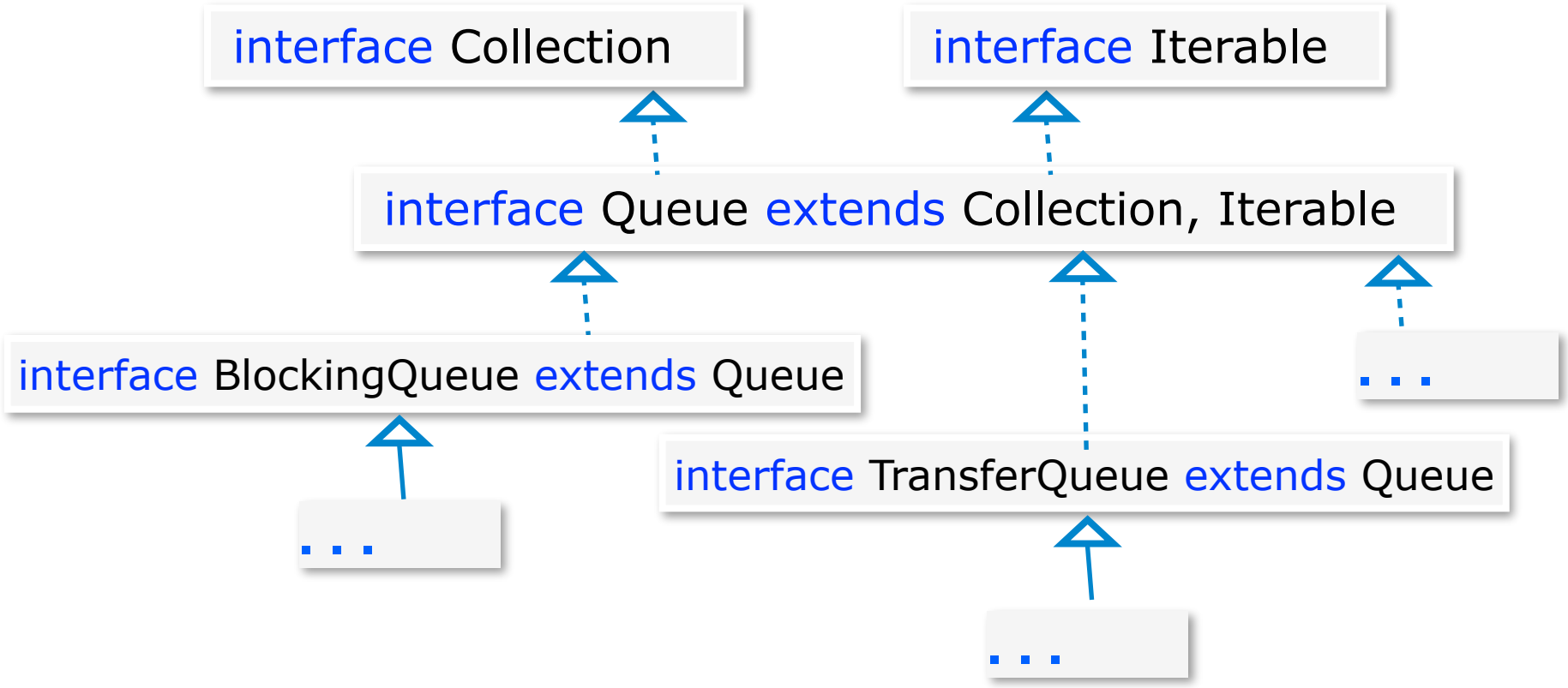
Im Java wird eine beschränkte mehrfache Vererbung mit Hilfe von Interfaces simuliert.

Eine Klasse im Java kann von einer Oberklasse vererben und gleichseitig mehrere Interfaces implementieren.



Interfaces als Strukturierungsmittel

Interfaces in Java können als Unterinterfaces von anderen Interfaces definiert werden.



Wrapper-Klassen

- Referenztypen können nicht auf primitive Datentypen gecastet werden - und umgekehrt.
- Wenn man primitive Datentypen an einer Stelle verwenden will, wo nur Objekttypen erlaubt sind, kann man den Wert eines Basistyps in ein passendes “Wrapper”-Objekt einpacken.
- Für jeden primitiven Datentyp gibt es eine entsprechende Wrapper-Klasse.

z.B für double gibt es die Klasse java.lang.Double

Autoboxing/Unboxing

java.lang.*

Boolean
Byte
Character
Double
Float
Integer
Long
Short

```
…
public static void main(String[] args) {
    Integer num = 3;
    Integer num2 = 3;
    System.out.println( Integer.toBinaryString(num) );
    System.out.println( num==num2);
    System.out.println( num.intValue()==num2.intValue() );
    Boolean bool = true;
    Double zahl = 3.0;
    System.out.println(num);
    System.out.println(bool);
    System.out.println(zahl);
}
…
```

Autoboxing

automatisches Unboxing und Vergleich

```
11
true
true
3
true
3.0
```

Autoboxing/Unboxing

```
...
Integer n = new Integer( 5 );
Integer m = new Integer( 5 );

System.out.println( n >= m ); // Unboxing
System.out.println( n <= m ); // Unboxing
System.out.println( n == m ); // kein Unboxing
...
```

Ausgabe:

```
True
True
False
```

Der Vergleich mit == ist ein Referenzvergleich

Autoboxing/Unboxing

```
...
Integer n = 127;
Integer m = 127;
System.out.println( n == m ); // Unboxing
Integer x = 128;
Integer y = 128;
System.out.println( x == y ); // kein Unboxing
...
```

Ausgabe:

True

False

Unboxing nur bei Objekten die mit automatischen *Boxing* gebildet worden sind und nur innerhalb des Wertbereichs -128 bis 127 (Bytes).