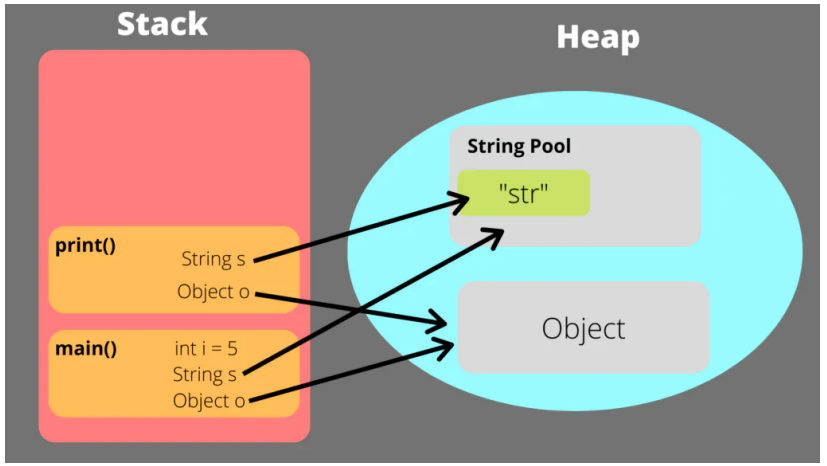
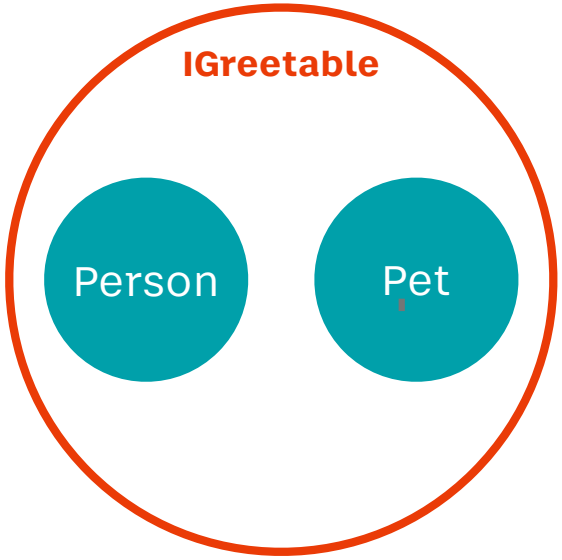
05. COLLECTIONS

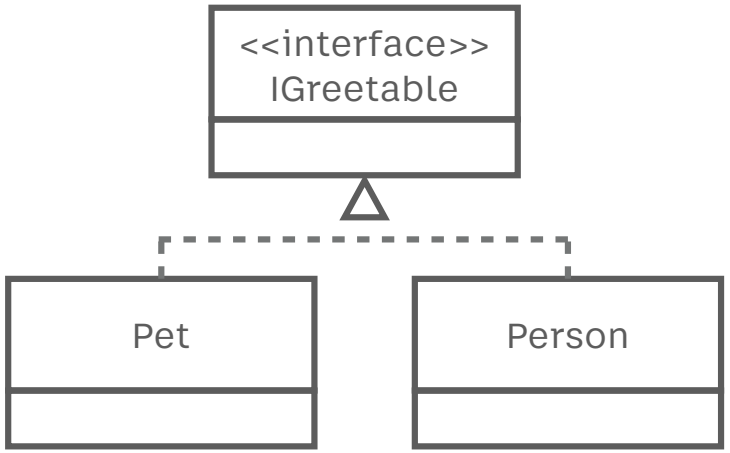
B2.1 - Angewandte Programmierung

WIEDERHOLUNG

- Datentypen
- Schnittstellen
- Datenkapselung
- Listen

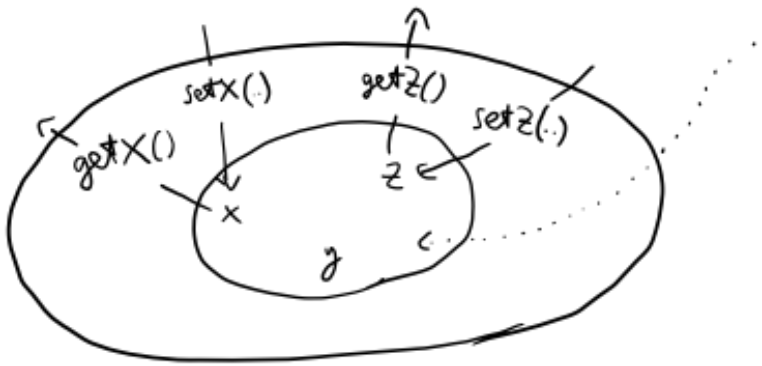




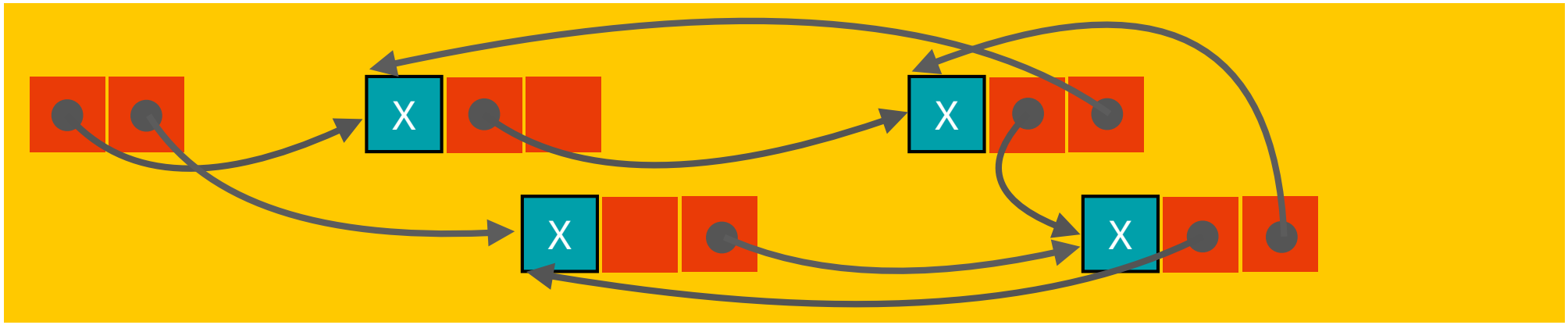


INHALT

- Zustand & Datenkapselung
- Das Collection Framework
- Listen
- Maps
- Generische Typen



`LinkedList<>`

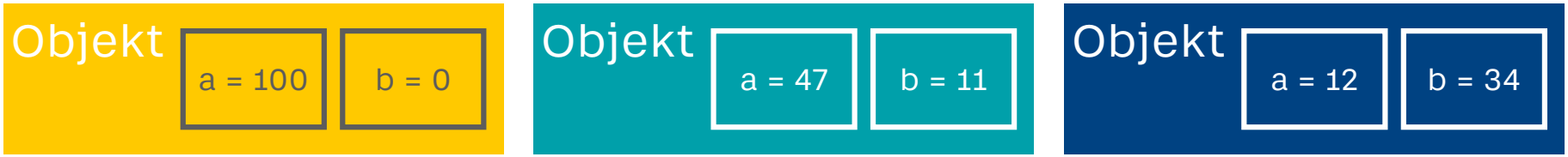


WIEDERHOLUNG ZUSTAND

Was ist der Zustand eines Objekts?

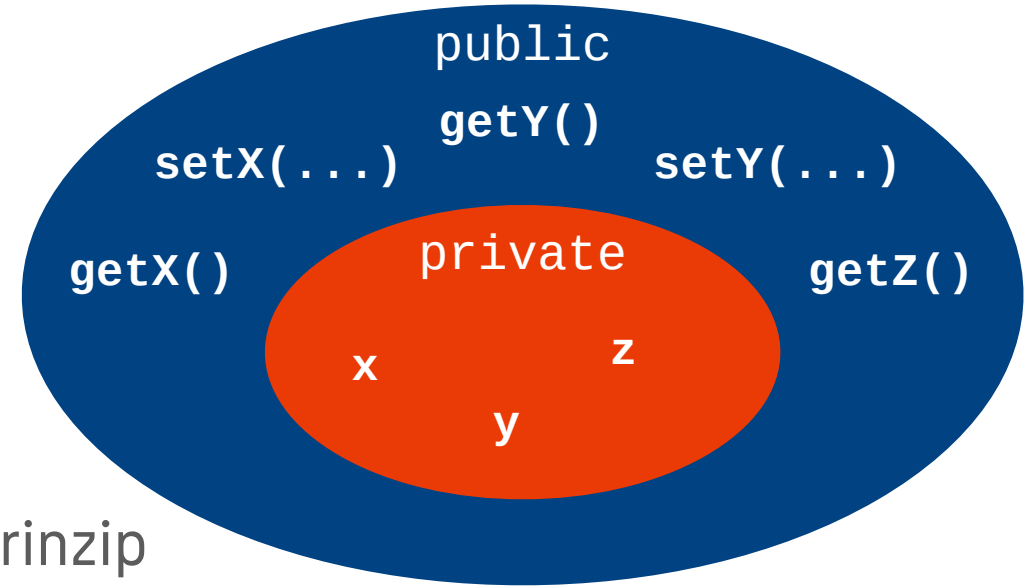
DATENKAPSELUNG

- Was ist Zustand? (engl. state)
  - Beschreibt die Menge aller Attribute eines Objekts, die sich ändern können, und Einfluss auf dessen Verhalten haben.
  - Einfache Funktionsbibliotheken haben keinen Zustand, da sie keine Daten speichern, es gehen nur Daten rein und direkt wieder raus. Sie verhalten sich immer gleich.
  - Komplexere Systeme haben praktisch immer einen Zustand (Warenkorb etc.) und man will i.d.R. vermeiden, dass dieser einen nicht zugelassenen Wert annimmt



DATENKAPSELUNG IN JAVA

- Grundregel:
  - Instanzvariablen `private` machen und Zugriff nur über öffentliche Methoden (z.B. `getter/setter`)



- Erfüllt das Information Hiding Prinzip
- Definiert eine Schnittstelle
- Sichert einen sauberen / kontrollierten Zustand

NEGATIVBEISPIEL: KEINE DATENKAPSELUNG

- age ist public
- Zugriff von außerhalb der Klasse möglich
- Attribut kann verändert werden ohne Kontrolle der Werte

```java
class Person {

    public String name;
    public int age;

    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }
}


class Scratch {
    public static void main(String[] args) {
        Person p = new Person( name: "Arif",  age: 37);
        p.age = -1; // unkontrollierte Zustandsänderung
    }
}
```

EINFACHE DATENKAPSELUNG

- Zugriff auf Attribut `age` nur über getter- und setter-Methode
- Zusätzliche Checks in Methode `setAge`
- Zustand nimmt nun nur noch kontrollierte Werte an.

```
class Person {
    private String name;
    private int age;

    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }

    public String getName() { return name; }
    public void setName(String name) { this.name = name; }

    public int getAge() { return age; }
    public void setAge(int age) { if(age>0) this.age = age; }
}

class Scratch {
    public static void main(String[] args) {
        Person p = new Person( name: "Arif",  age: 37);
        p.setAge(-1); // kontrollierte Zustandsänderung
    }
}
```

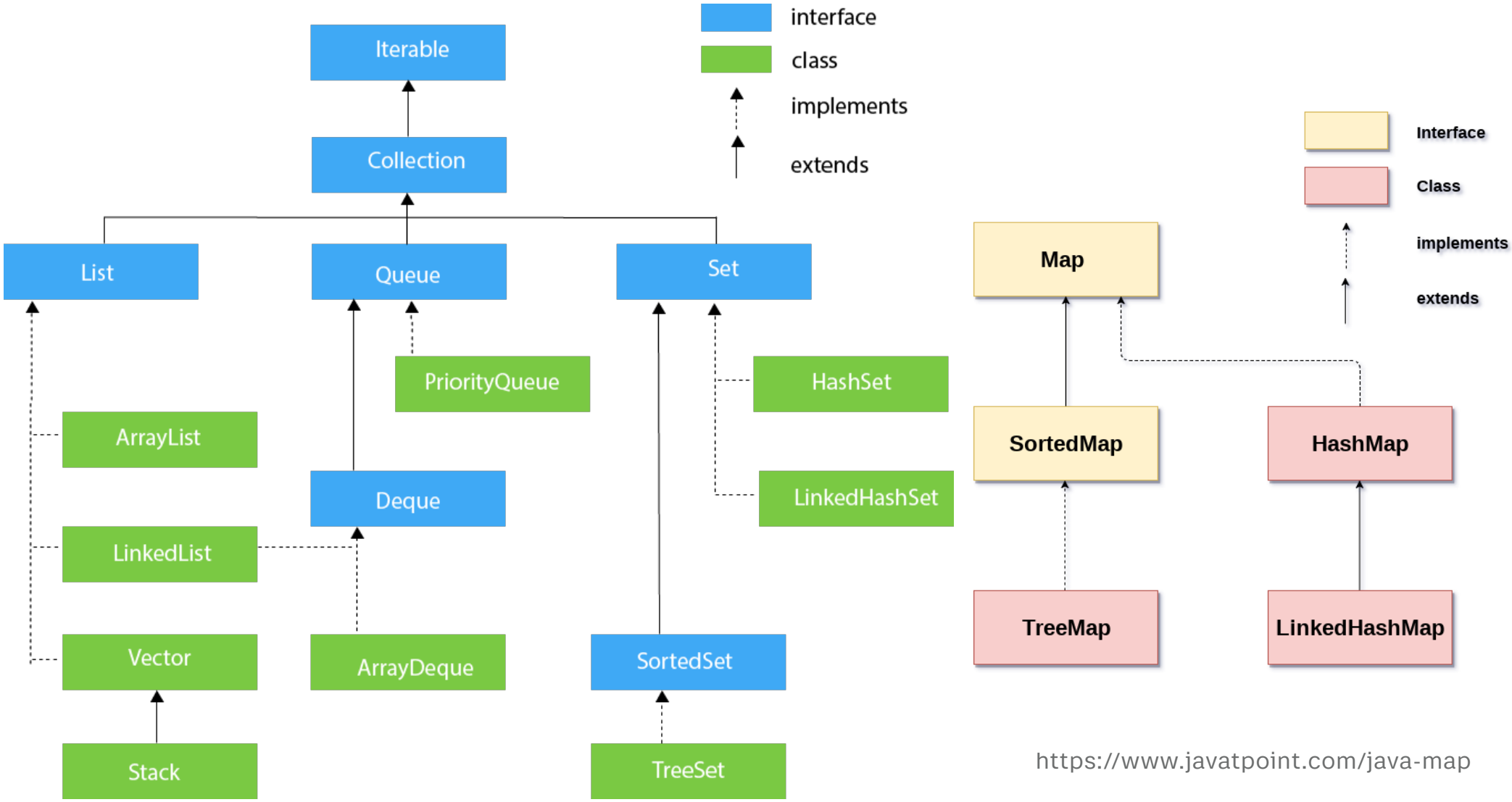
ECHTE DATENKAPSELUNG MIT FACHLICHER SCHNITTSTELLE

```java
class Person {
    private String name;
    private int age;

    public Person(String firstName, String lastName, int age) {
        name = firstName + " " + lastName;
        if(age>0) this.age = age;
    }

    public String getFullName() { return name; }

    public void changeLastName(String lastName) {
        if(lastName == null || lastName.isEmpty()) throw new IllegalArgumentException();
        this.name = name.substring(0,name.indexOf(" ")) + lastName;
    }
    public void changeFirstName(String firstName) {
        if(firstName == null || firstName.isEmpty()) throw new IllegalArgumentException();
        this.name = firstName + name.substring(name.indexOf(" "));
    }

    public int getAge() { return age; }

    public void celebrateBirthday() { age = age+1; }
}
```

ZUSAMMENFASSUNG DATENKAPSELUNG

- Information Hiding bedeutet, dass man **Implementierungs-entscheidungen**, die sich ändern könnten, so versteckt, dass andere sich nicht fälschlicherweise darauf verlassen können; Entscheidungen, die sich nicht ändern, werden über verlässliche Schnittstellen dokumentiert.
- Ein **zustandsbehaftetes** Objekt reagiert ggf. unterschiedlich, je nach dem, was davor damit gemacht wurde, da es sich Dinge „merken“ kann; Ein Objekt ohne Instanzvariablen kann sich nichts merken und ist daher **zustandslos**.
- **Datenkapselung** sorgt dafür, dass der Zustand eines Objekts (also dessen Attribute) nur auf kontrollierte Weise geändert werden kann.

JAVA COLLECTIONS FRAMEWORK



https://www.javatpoint.com/java-map

https://www.javatpoint.com/collections-in-java

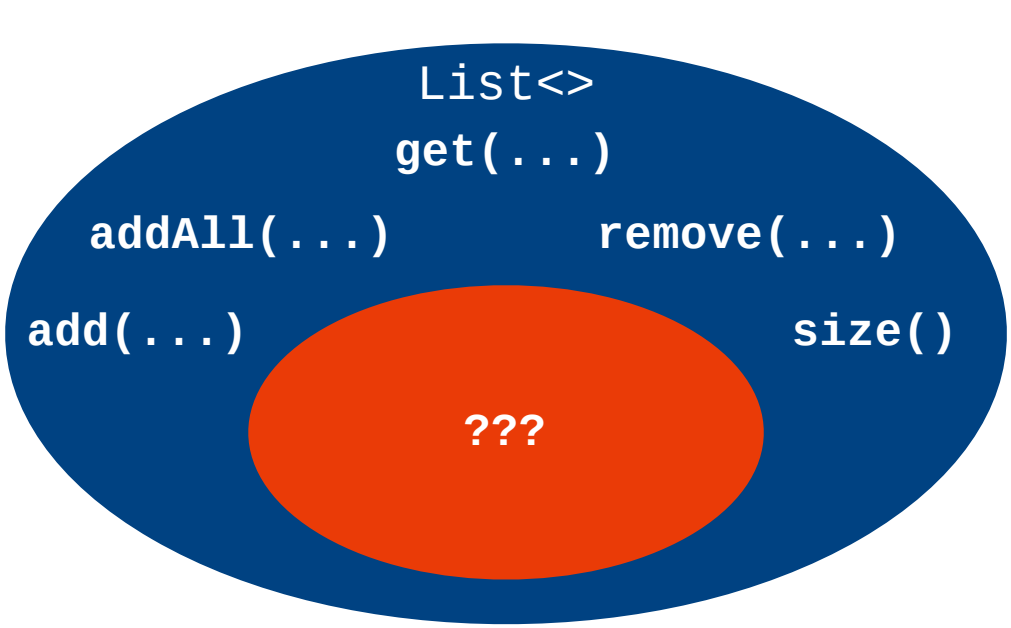
KURZÜBERBLICK COLLECTIONS

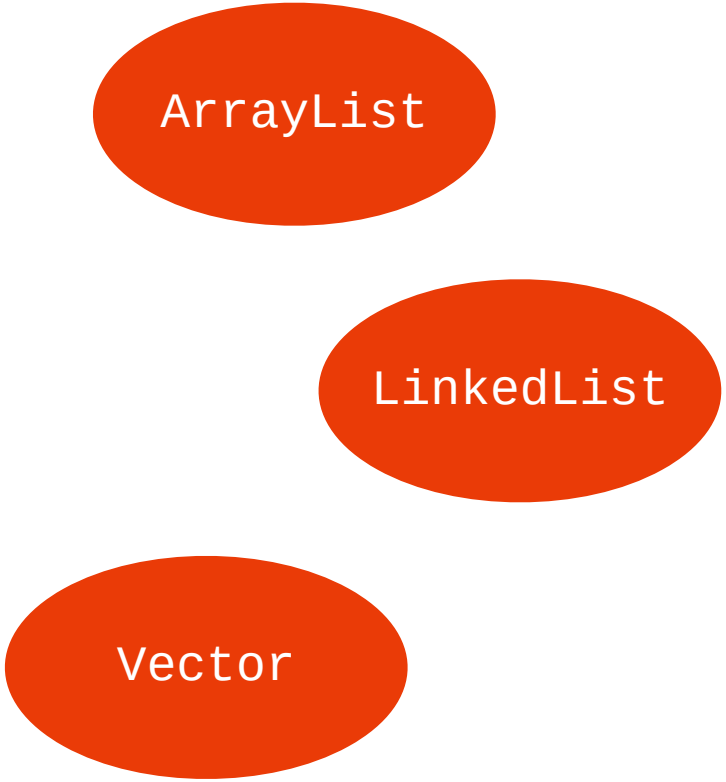
- Collections
  - sind Datenstrukturen,
  - die "beliebig" viele Elemente speichern können,
  - verbessern die Nachteile von Arrays.

Datenstruktur	Java-Interface	Einsatzgebiet
Liste	`List`	Speicherung mehrere Elemente; Einfügen, Löschen, Zugriff basierend auf **Index**; Reihenfolge der Elemente wählbar.
Warteschlange	`Queue`	Speicherung mehrere Elemente; Einfügen nur am Ende, Löschen nur am Anfang (FIFO)
Menge	`Set`	Speicherung **eindeutiger** Elemente (keine Dopplungen); Einfügen, Löschen basierend auf **Objekt**; Reihenfolge festgelegt.
Tabelle / Map	`Map`	Speicherung mehrere Elemente als Schüssel-Wert-Paar; keine Dopplungen; Einfügen, Löschen, Zugriff über Schlüssel

LISTEN

- "Sequentielle" Speicher -> speichern mehrere Elemente;
  Reihenfolge kannst festgelegt werden (bspw. Einfügereihenfolge)

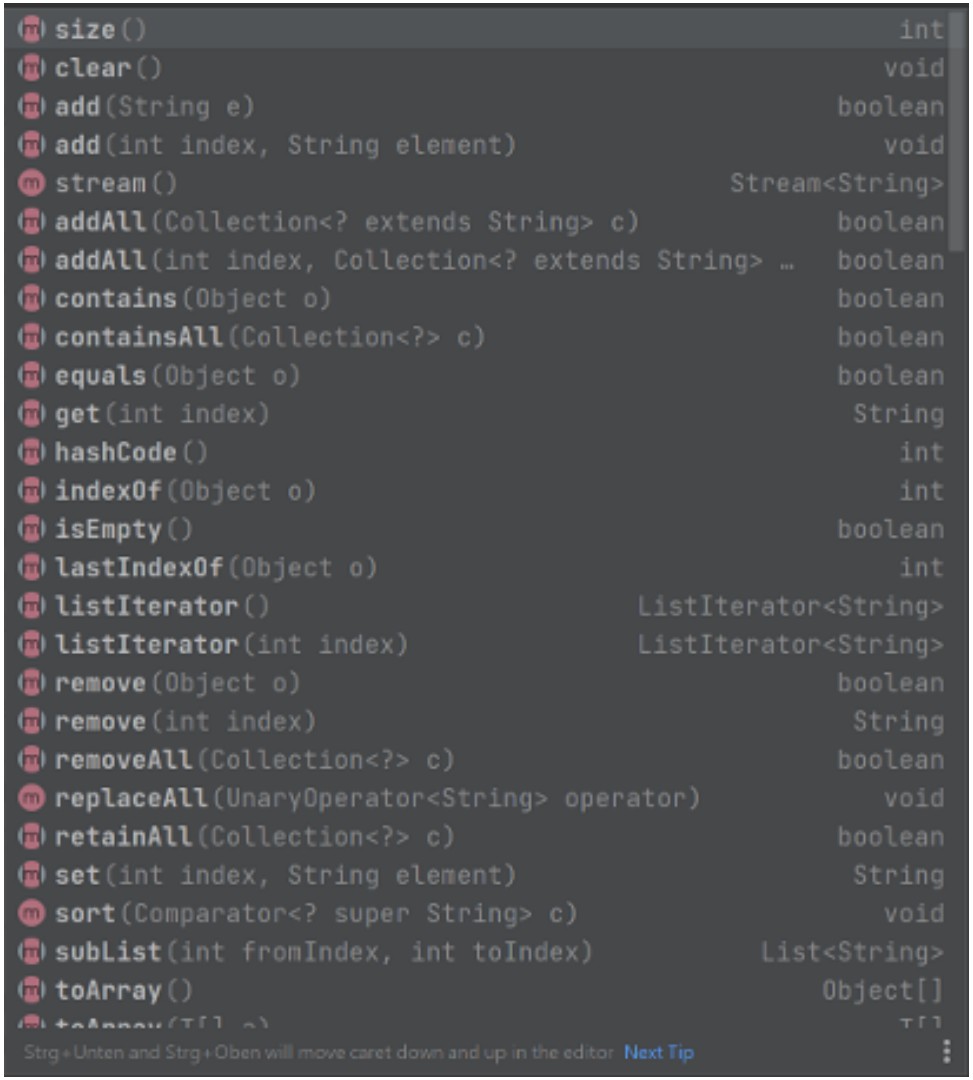




SEQUENTIELLE COLLECTIONS

	Zugriff	Einfügen	Löschen	Iterator	Suche
`ArrayList`	O(1)	Ende: O(1) sonst: O(n)	Ende: O(1) sonst: O(n)	wahlfrei	O(n)
`LinkedList`	Anfang/ Ende: O(1) sonst: O(n)	O(1)	O(1)	vorwärts rückwärts	O(n)
`Vector`	O(1)	Ende: O(1) sonst: O(n)	Ende: O(1) sonst: O(n)	wahlfrei	O(n)

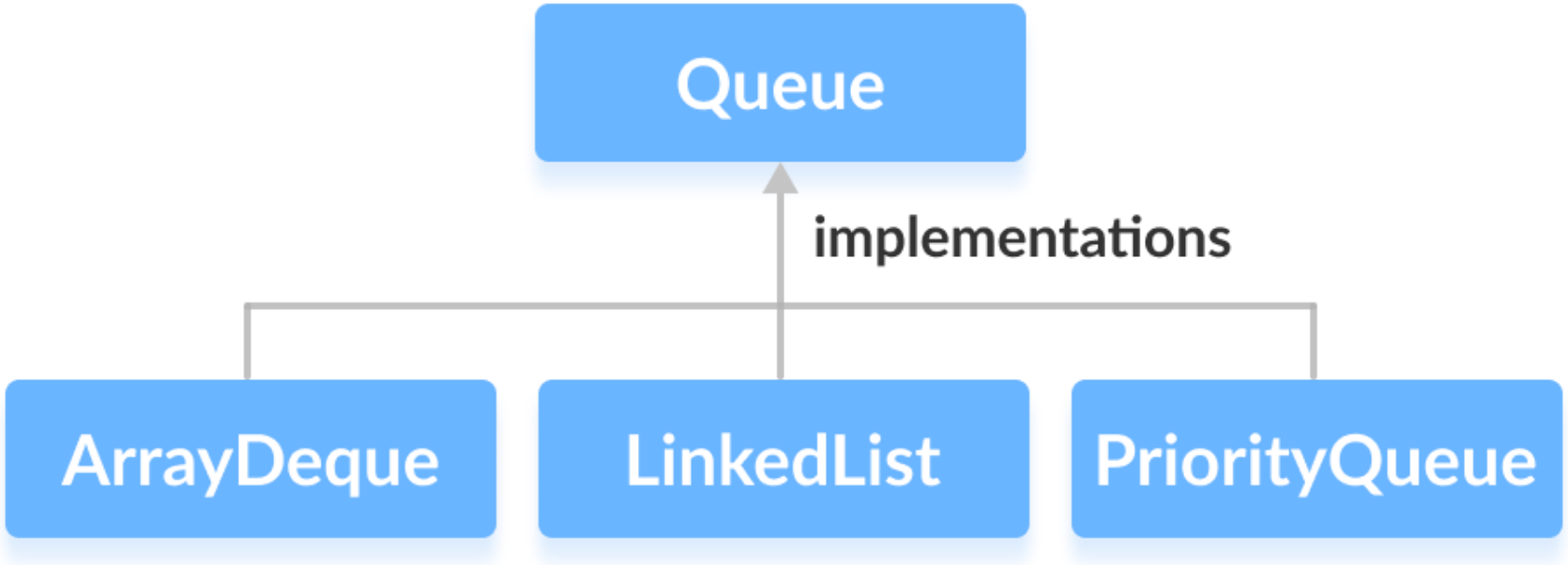
WICHTIGE INSTANZMETHODEN LIST



- `size()` - Anzahl Elemente
- `add()`, `addAll()` - Einfügen
- `contains()`, `constainsAll()` - prüft ob Elemente enthalten sind
- `indexOf()` - gibt Index von Element zurück
- `remove()`, `removeAll()` - Löschen
- `clear()` - alle Elemente löschen
- `get()` - Element zugreifen

WARTESCHLANGEN - QUEUE

- FIFO-Prinzip
- Einfügen am Anfang mit `add()`
- Entfernen am Ende mit `remove()` oder `poll()`
- Auslesen eines Elements mit `element()` oder `peek()`

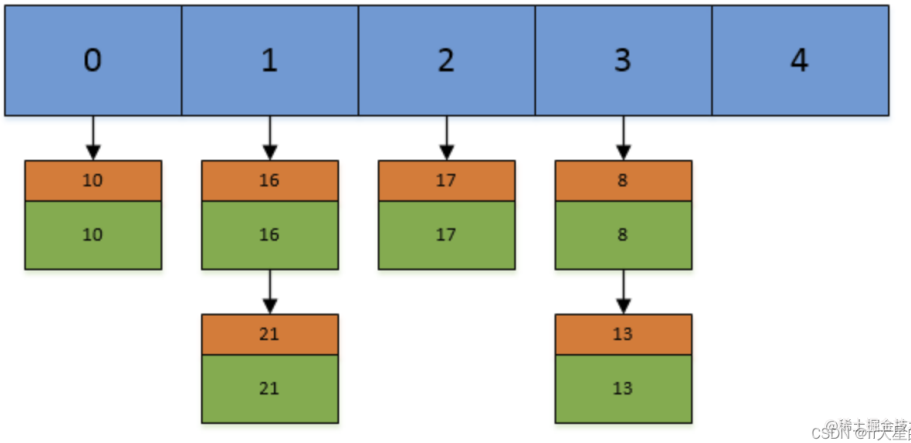


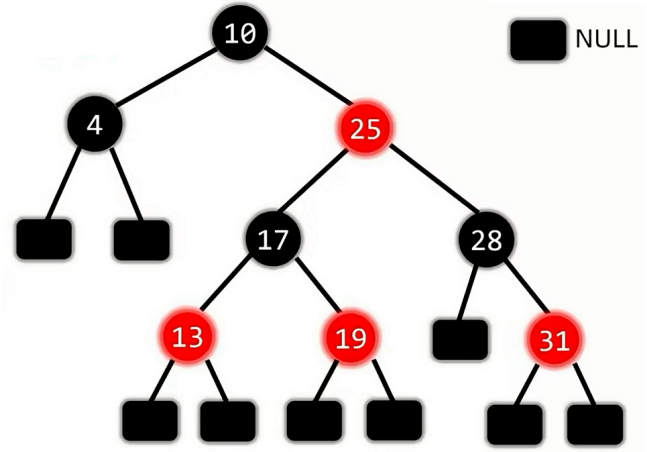
INTERFACES ERMÖGLICHEN AUSTAUSCHBARKEIT

```java
/**
 * Sum all elements in the queue by removing all elements
 * @param q Queue to read elements from
 * @return Total sum of elements
 */
public static Integer sumQueue(Queue<Integer> q) {
    Integer sum = 0;
    while (!q.isEmpty()) {
        sum += q.poll();
    }
    return sum;
}

public static void main(String[] args) {
    Queue<Integer> q1 = new LinkedList<>();
    Queue<Integer> q2 = new PriorityQueue<>();
    Queue<Integer> q3 = new ArrayDeque<>();

    System.out.println("Summe Q1: " + sumQueue(q1));
    System.out.println("Summe Q2: " + sumQueue(q2));
    System.out.println("Summe Q3: " + sumQueue(q3));
}

```

MENGEN - INTERFACE SET

- Speichern **eindeutige** Elemente -> Doppelte Elemente werden nicht eingefügt
- Einsatzgebiet: eindeutige Elemente, häufiges Suchen von Elementen
- Klassen, welche das Interface implementieren:
  - `HashSet` - speichert Elemente basierend auf einer Hash-Tabelle
  - `TreeSet` - speichert Elemente **sortiert** in einem Binärbaum





BEISPIEL HASHSET

```
01 package de.htwberlin;
02
03 import java.util.*;
04
05 public class SetDemo {
06     public static void addNames(Collection<String> c, String ...args) {
07         c.addAll(Arrays.asList(args));
08     }
09
10     public static void main(String[] args) {
11         List<String> test1 = new ArrayList<>();
12         Set<String> test2 = new HashSet<>();
13
14         addNames(test1, "michael", "sophia", "ahmed",
15                         "ebru", "sophia", "peter");
16         addNames(test2, "michael", "sophia", "ahmed",
17                         "ebru", "sophia", "peter");
18
19         System.out.println("Elemente in test1: " + test1.size());
20         System.out.println("Elemente in test2: " + test2.size());
21     }
22 }
23
```

TABELLEN / MAPS / DICTIONARY

- Speichern zu einem Schlüssel (Referenztyp) einen Wert (Referenztyp)
- Anders als bei sequentiellen Containern werden Elemente nicht über den Index angesprochen, sondern über den **Schlüssel**.
- Klassen, welche das Interface implementieren:
  - `HashMap` - speichert Elemente basierend auf einer Hash-Tabelle
  - `TreeMap` - speichert Elemente **sortiert** in einem Binärbaum

```
// 2 Datentypen sind nötig (Schlüssel und Wert)
Map<String, Integer> map = new HashMap<>();
// Wert zu Schlüssel speichern
map.put("Anzahl", 60);
// Wert abrufen
System.out.println("Wert ist: " + map.get("Anzahl"));
// Gleicher Schlüssel überschreibt alten Wert
map.put("Anzahl", 30);
System.out.println("Wert ist: " + map.get("Anzahl"));
```

WICHTIGE INSTANZMETHODEN MAPS

- `size()` - Anzahl Paare
- `put()` - Einfügen
- `containsKey()`, `constainsValue()` - prüft ob Schlüssel/Wert enthalten
- `keySet()` - Menge aller Schlüssel
- `entrySet()` - Menge aller Schlüssel/Wert-Paare
- `remove()` - Löschen
- `clear()` - alle Elemente löschen
- `get()` - Element zugreifen
- `value()` - Menge aller Werte

WELCHES COLLECTION INTERFACE FÜR WELCHEN ZWECK

Java-Interface	Reihenfolge	Duplikate	Besonderheit
`List`	Wie eingefügt	Ja	Zugriff auf Postionen
`Queue`	Wie eingefügt	Ja	FIFO-Prinzip
`Set`	Ggf. sortiert sonst unbestimmt	Nein	Eindeutige Elemente, schnelle Prüfung ob Elemente enthalten sind
`Map`	Ggf. sortiert sonst unbestimmt	Nein	Eindeutige Schlüssel

JAVA GENERICS

- Verwendung von < >-Klammern um Generics zu erstellen
- Generics sind verfügbar für
  - Klassen
  - Methoden
- Zwischen <> stehen die Template-Parameter bzw. sog. Typ-Parameter, mit denen die Platzhalter realisiert werden.

Template-Parameter

```
public class ListElement<T> {
  private T data;
  private ListElement next;
}
```

GENERICS VERWENDEN

```
01    // Generische Klasse mit einem Template-Parameter mit Namen T
02    public class ListElement<T> {
03      private T data;
04      private ListElement next;
05
06      // Template-Parameter T kann als Typ in Klasse verwendet werden
07      public ListElement(T d, ListElement parent)
08      {
09        this.data = d;
10        if (parent != null) { parent.next = this; }
11      }
12
13      public T getData() { return this.data; };
14    };
15
16    public static void main(String[] arguments) {
17      ListElement<Integer> begin = new ListElement<Integer>(10, null);
18      ListElement<Integer> next = new ListElement<Integer>(20, begin);
19
20      Integer data = begin.getData(); // Typensicherheit gegeben!!!
21      ...
22    }
23
```

ÜBERSETZUNG VON GENERICS

- Generics werden vom Java-Compiler übersetzt
- Verwendung eines Generics mit konkreten *Template-Argumenten* wird **Spezialisierung** genannt.

```
new ListElement<Integer>(10, null);
```

Spezialisierung mit **Integer**

  - Compiler übersetzt Generics zur Laufzeit in Klassen, welche auf dem Basis-Datentyp `Object` arbeiten (Type Erasure)
  - Da alle Klassen von Object erben funktioniert Zuweisung durch die ist-ein-Relation
  - Implizierte Typen-Casts durch die JVM
  - Platzsparend jedoch mit Overhead zur Laufzeit

ZUSAMMENFASSUNG

- Collections sind Container für Objekte und imple-mentieren je nach Fähigkeiten unterschiedliche Interfaces.
- Sie müssen sich entscheiden, welche Fähigkeiten (d.h. Methoden) Sie benötigen und demensprechend das Interface auswählen.
- Collections speichern nur Objekt-Referenzen, haben daher keine feste Größe, und bieten eine Vielzahl von Methoden zur Verwendung.
- Primitive Werte werden durch Wrapper-Klassen als Objekte repräsentiert.
- Die meisten Collections-Klassen sind generisch, d.h. sie müssen mit dem Typ Ihrer Elemente parametrisiert werden.

VIELEN DANK