

# Iterieren über Kollektionen

Seit Java2:

```
1 List<String> names = Arrays.asList("Peter", "Max",  
    "Moritz", "Lukas");  
2 Iterator<String> i = names.iterator();  
3 for (; i.hasNext(); name=i.next())  
4     System.out.println(name);
```

Seit Java5:

```
1 for (String name : names) {  
2     System.out.println(name);  
3 }
```

Seit Java8:

```
1 names.forEach(n->System.out.println(n));
```

# Streams

```
1 List<String> names =
2     Arrays.asList("Moritz", "Lukas", "Max", "Peter");
3
4 names
5     .stream()
6     .filter(s -> s.startsWith("M"))
7     .map(String::toLowerCase)
8     .sorted()
9     .forEach(System.out::println);
```

# Stream Pipeline

## Eine „source“

- ▶ Z.B.: `List.stream()` erzeugt Stream aus Kollektion `List`;

## Mehrere „intermediate operations“

- ▶ `.filter()`, filtert Objekte aus, für die geg. Prädikat gilt
- ▶ `.map()`, wendet eine Funktion auf jedes Objekt an
- ▶ ...

## Eine „terminal operation“

- ▶ `.forEach()` wendet Funktion auf jedes Element an
- ▶ `.reduce()` berechnet ein Ergebnis aus allen Werten
- ▶ `.collect()` kann z.B. Elemente in Kollektion speichern
- ▶ `.findFirst()`, liefert ein `Optional` mit erstem Element
- ▶ ...

# Streams

```
1 int res =  
2     Stream.of(7, 3, -5, 15)  
3         .filter(j -> j>0)  
4         .mapToInt(j->j)  
5         .sum();
```

# Streams

```
1 List<String> filtered =  
2     names // Kollektion mit Strings  
3         .stream()  
4         .filter(s -> s.startsWith("P"))  
5         .collect(Collectors.toList());
```

# Streams

Eine Version von `.reduce()` erwartet eine (assoziative) Funktion mit zwei Argumenten:

```
1 Optional<Integer> min =  
2     Stream.of(7, 3, -5, 15)  
3         .reduce((x,y)-> x < y ? x : y);
```

Im Prinzip wird diese Funktion in beliebiger Reihenfolge auf die Elemente des Streams angewendet, bis man nur noch einen Wert hat.

Durch die Assoziativität erhält man immer das gleiche Ergebnis.