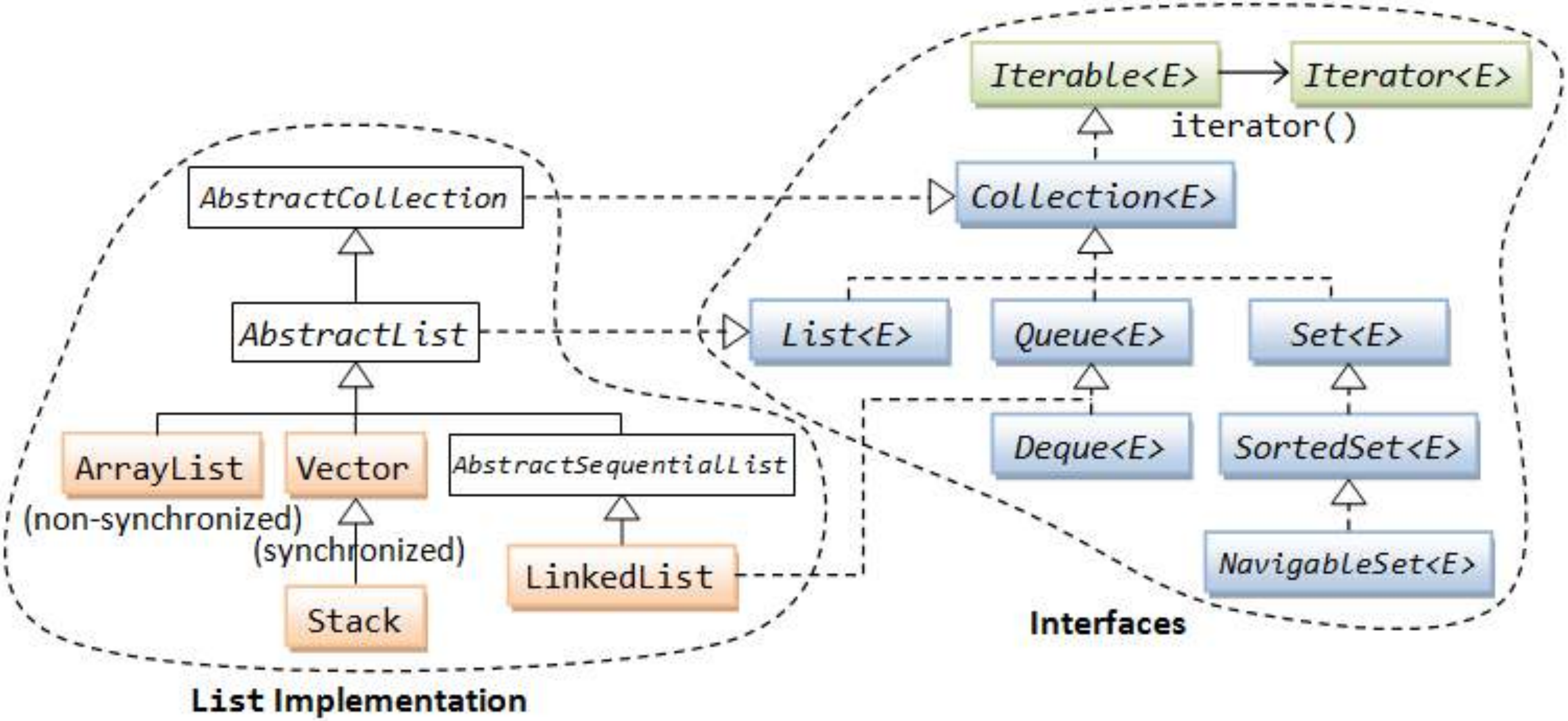
Stream API

Martijn Blankestijn

The new world

```java
Iterable<Integer> numbers =
        Arrays.asList(1);

numbers.forEach(System.out::println);
```

Collection API



```
public interface Iterable<T> {

 Iterator<T> iterator();

 default void forEach(Consumer<> action){
    for (T t : this) {
      action.accept(t);
    }
  }
  …
```

And while we’re at it, why not add static methods as well then…

Interfaces & Companion classes

Collection | Collections

Path | Paths

```
@FunctionalInterface
public interface Comparator<T>


public static Comparator<T> reverseOrder() {
      return Collections.reverseOrder();
}


public static Comparator<T> naturalOrder() {
      return  ...;
}
```

Streams

Why ?

- More readable code
- Easy path to parallelism
  - Aggregate operations < JDK 1.7
    - Fundamental sequential
    - Frustrating imperative

Map <CurrencyCode, Countries>

```
Map<String, Set<String>> map = new HashMap<>();
for (Locale locale : Locale.getAvailableLocales()) {

  if (locale.getCountry().isEmpty()) continue;

  String curr = this.getCurrencyCode(locale);

  if ( ! map.containsKey(curr))
    map.put(curr, new TreeSet<>());

  map.get(curr).add(locale.getDisplayCountry());
}
```

What is a Stream ?

1) Create

2) Intermediary operations
Stateless transformations
Stateful transformations
Extract / combine

3) Terminal operations
Reduction
Collecting

1) Creating streams

1) Call `Collection.stream();`

```
numbers.stream();
```

List<Integer> numbers;

2) Use a static factory

```
Stream.of("stream", "of", "strings");
```

3) Generators

```
Stream.generate( () -> 42 );
Stream.iterate( 1, i -> i * 2 );
```

4) Roll your own Stream Source

```
public interface Spliterator<T>
```

Stream Sources

Source	Decomposibility	Characteristics
ArrayList	Excellent	Sized, Ordered
LinkedList	Poor	Sized, Ordered
HashSet	Good	Sized, Distinct
IntStream.range	Excellent	Sized, Distinct, Sorted, Ordered
Stream.iterate()	Poor	Ordered

2) Transformations - stateless

```
Stream<String> words =
  Stream.of("stream", "of", "strings");

// { "streams", "strings" }
Stream<String> longWords =
  words.filter(s -> s.length() > 4);

// { 6, 2, 7 };
Stream<Integer> lengths =
  words.map(s -> s.length());
```

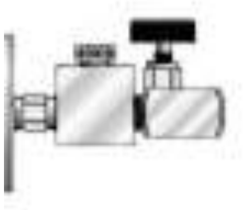
2) Transformations - stateful

```
Stream<String> chars =
  Stream.of("A","B","D","A","B");

// { "A","B","D" }
Stream<String> distinctChars =
  chars.distinct();

// {"A","A","B","B","D" };
Stream<String> sorted =
  chars.sorted(); // natural order
```

Intermediate Operations

Operation	Effect
filter()	Removes SIZED
map()	Removes DISTINCT, SORTED
sorted()	Injects SORTED, ORDERED
distinct()	Injects DISTINCT
limit()	Preserves All
peek()	Preserves All

3) Terminal operation-reduction



```
Stream<String> chars =
   Stream.of("AB","CDE","FGHI");

long numberOfChars = chars.count(); // 3


// "FGHI"
Optional<String> max =
   chars.max(comparing(String::length));
```

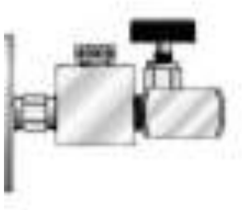
Optional values before JDK 8

```
Person person;

String street = "Unknown";
if (person != null
    && person.getAddress() != null
    && person.getAddress()
             .getStreet() != null) {
  street = person.getAddress().getStreet();
}
```

Optional

```
Optional<Person> person;

String street =
  person.map(Person::getAddress)
        .map(Address::getStreet)
        .orElse("Unknown");
```

3) Terminal operation-collect



```
List<String> list =
   stream.collect(Collectors.toList());

Set<String> set =
   stream.collect(Collectors.toSet());

String joined =
   stream.collect(joining(","));
```

Terminal Operations

Family	Operation
	toArray
Reduction	reduce
	sum, min, max, count
	anyMatch, allMatch
Collection	collect
Iteration	forEach
Searching	findFirst
	findAny

Transitioning

Map <CurrencyCode, Countries>

```
Map<String, Set<String>> map = new HashMap<>();
for (Locale locale : Locale.getAvailableLocales()) {

  if (locale.getCountry().isEmpty()) continue;

  String curr = this.getCurrencyCode(locale);

  if ( ! map.containsKey(curr))
    map.put(curr, new TreeSet<>());

  map.get(curr).add(locale.getDisplayCountry());
}
```

Map <CurrencyCode, Countries>

```
Predicate<Locale> hasCountry =
                    l -> ! l.getCountry().isEmpty();

Map<String, Set<String>> m =
    Stream.of(Locale.getAvailableLocales())
        .filter(hasCountry)
        .collect(
            groupingBy(this::getCurrencyCode,
                mapping(Locale::getDisplayCountry,
                    toCollection(TreeSet::new))
            )
        );
```

Streams Characteristics

- No data-structure
- Functional
- Lazy
- Parallelizable
- Can be Infinite

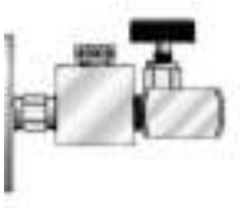
Parallel

```
List<String> result =
  numbers.parallelStream()
         .map(this::slowOp)
         .collect(toList());

 numbers.stream()
     .parallel()
     …
```

? ? ?

{ live coding; }

Collecting



```
<R> R collect(
  Supplier<R> supplier,
  BiConsumer<R, ? super T> accumulator,
  BiConsumer<R, R> combiner);

Set<String> s = stream.collect(
  HashSet::new,
  HashSet::add,
  HashSet::addAll
);
```

Imperative	Stream
Code deals with individual data items	Code deals with the data set
Focused on *how*	Focused on *what*
Code look ….	Code reads like the problem statement
Steps mashed together	Well-factored
Leaks extraneous details	No ‘ garbage variables’
Inherently sequential	Can be Sequential or parallel