JAYield

How to suppress the Java absence of yield operator using a functional approach to be introduced the new curricular plan for Modeling and Design Patterns in 2018.

JAYield is a *Minimalistic*, *extensible*, *non-parallel* and *lazy* sequence implementation interoperable with Java `Stream` (`toStream` and `fromStream`), which provides an idiomatic `yield` like *generator*.
JAYield `Query` provides similar operations to Java `Stream`, or jOOλ Seq, or StreamEx, or Vavr Stream. Yet, `Query` is **extensible** and its methods can be chained fluently with new operations in a pipeline. Furthermore, `Query` has lower per-element access cost and offers an optimized fast-path traversal, which presents better sequential processing performance in some benchmarks, such as sequences-benchmarks and jayield-jmh.
The core API of `Query` provides well-known query methods that can be composed fluently (*pipeline*), e.g.:

```
// pipeline: iterate-filter-map-limit-forEach
//
Query.iterate('a', prev -> (char) ++prev).filter(n -> n%2 !=
0).map(Object::toString).limit(10).forEach(out::println);
```

Extensibility and chaining

Notice how it looks a JAYield custom `collapse()` method that merges series of adjacent elements. It has a similar shape to that one written in any language providing the `yield` feature such as Kotlin.

```
class Queries {
  private U prev = null;
  <U> Traverser<U>
collapse(Query<U> src) {
    return yield -> {
      src.traverse(item -> {
        if (prev == null ||
!prev.equals(item))
        yield.ret(prev = item);
      });
    };
  }
}
```

```
fun <T> Sequence<T>.collapse() =
sequence {
    var prev: T? = null
    val src =
this@collapse.iterator()
    while (src.hasNext()) {
        val aux = src.next()
        if (aux != null && aux
!= prev) {
            prev = aux
            yield(aux)
        }
    }
}
```

These methods can be chained in queries, such as:

```
Query
    .of(7, 7, 8, 9, 9, 8, 11,
11, 9, 7)
    .then(new
Queries()::collapse)
```

```
sequenceOf(7, 7, 8, 9, 9, 8, 11,
11, 9, 7)
    .collapse()
    .filter { it % 2 != 0 }
    .map(Int::toString)
```

```
    .filter(n -> n%2 != 0)              .forEach(::println)
    .map(Object::toString)
    .traverse(out::println);
```

Internals Overview

`Traverser` is the primary choice for traversing the `Query` elements in bulk and supports all its methods including *terminal*, *intermediate* and *short-circuting* operations. To that end, the traversal's consumer - `Yield` - provides one method to return an element (`ret`) and other to finish the iteration (`bye`). `Advancer` is the alternative iterator of `Query` that provides individually traversal.

Installation

In order to include it to your Maven project, simply add this dependency:

```
<dependency>
    <groupId>com.tinyield</groupId>
    <artifactId>jayield</artifactId>
    <version>1.5.1</version>
</dependency>
```

You can also download the artifact directly from Maven Central Repository

License