No more loops with λambdaj

An internal DSL to manipulate collections without loops

by Mario Fusco
mario.fusco@gmail.com
twitter: @mariofusco

Why is lambdaj born?

The best way to understand what lambdaj does and how it works is to start asking why we felt the need to develop it:

- We were on a project with a complex data model
- The biggest part of our business logic did almost always the same: iterating over collections of our business objects in order to do the same set of tasks
- Loops (especially when nested or mixed with conditions) are harder to be read than to be written
- We wanted to write our business logic in a less technical and closer to business fashion

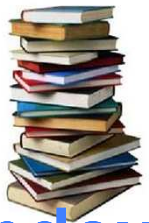
What is lambdaj for?

- It provides a DSL to manipulate collections in a pseudo-functional and statically typed way.
- It eliminates the burden to write (often poorly readable) loops while iterating over collections.
- It allows to iterate collections in order to:

convert

filter

sort



index

aggregate

group

extract

How does lambdaj work?

lambdaj is a thread safe library of static methods based on 2 main features:

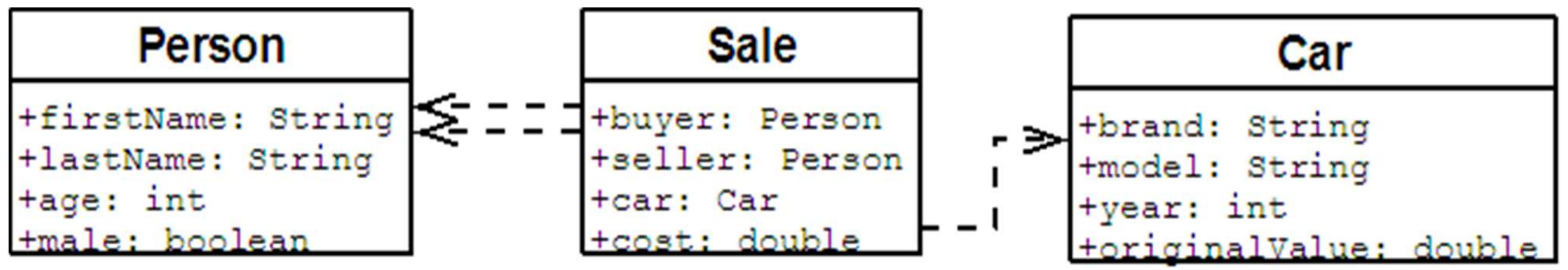
- treating a collection as it was a single object by allowing to propagate a single method invocation to all the objects in the collection

```
forEach(personsInFamily).setLastName("Fusco");
```

- allowing to define a reference to a java method in a statically typed way

```
sort(persons, on(Person.class).getAge());
```

The Demo Data Model



Print all cars' brands

Iterative version:

```
StringBuilder sb = new StringBuilder();
for (Car car : cars)
   sb.append(car.getBrand()).append(", ");
String brands = sb.toString().substring(0, sb.length()-2);
```

lambdaj version:

```
String brands = joinFrom(cars).getBrand();
```

Select all sales of a Ferrari

Iterative version:

```
List<Sale> salesOfAFerrari = new ArrayList<Sale>();
for (Sale sale : sales) {
   if (sale.getCar().getBrand().equals("Ferrari"))
        salesOfAFerrari.add(sale);
}
```

lambdaj version:

```
List<Sale> salesOfAFerrari = select(sales,
  having(on(Sale.class).getCar().getBrand(),equalTo("Ferrari")));
```

Find buys of youngest person

Iterative version:

```
Person youngest = null;
for (Person person : persons)
   if (youngest == null || person.getAge() < youngest.getAge())
        youngest = person;
List<Sale> buys = new ArrayList<Sale>();
for (Sale sale : sales)
   if (sale.getBuyer().equals(youngest)) buys.add(sale);
```

lambdaj version:

```
List<Sale> sales = select(sales,having(on(Sale.class).getBuyer(),
   equalTo(selectMin(persons, on(Person.class).getAge()))));
```

Find most costly sale

Iterative version:

```
double maxCost = 0.0;
for (Sale sale : sales) {
     double cost = sale.getCost();
     if (cost > maxCost) maxCost = cost;
}
```

lambdaj version:

```
Sol. 1 -> double maxCost = max(sales, on(Sale.class).getCost());
Sol. 2 -> double maxCost = maxFrom(sales).getCost();
```

Sum costs where both are males

Iterative version:

```
double sum = 0.0;
for (Sale sale : sales) {
   if (sale.getBuyer().isMale() && sale.getSeller().isMale())
        sum += sale.getCost();
}
```

lambdaj version:

```
double sum = sumFrom(select(sales,
   having(on(Sale.class).getBuyer().isMale()).and(
   having(on(Sale.class).getSeller().isMale())))).getCost();
```

Find age of youngest who bought for more than 50,000

Iterative version:

```
int age = Integer.MAX_VALUE;
for (Sale sale : sales) {
   if (sale.getCost() > 50000.00) {
       int buyerAge = sale.getBuyer().getAge();
       if (buyerAge < age) age = buyerAge;
   }
}
```

lambdaj version:

```
int age = min(forEach(select(sales,
   having(on(Sale.class).getCost(), greaterThan(50000.00))))
   .getBuyer(), on(Person.class).getAge());
```

Sort sales by cost

Iterative version:

```java
List<Sale> sortedSales = new ArrayList<Sale>(sales);
Collections.sort(sortedSales, new Comparator<Sale>() {
   public int compare(Sale s1, Sale s2) {
      return Double.valueOf(s1.getCost()).compareTo(s2.getCost());
   }
});
```

lambdaj version:

```java
List<Sale> sortedSales = sort(sales, on(Sale.class).getCost());
```

Extract cars' original cost

Iterative version:

```
List<Double> costs = new ArrayList<Double>();
for (Car car : cars)
    costs.add(car.getOriginalValue());
```

lambdaj version:

```
List<Double> costs =
        extract(cars, on(Car.class).getOriginalValue());
```

Index cars by brand

Iterative version:

```
Map<String, Car> carsByBrand = new HashMap<String, Car>();
for (Car car : cars)
    carsByBrand.put(car.getBrand(), car);
```

lambdaj version:

```
Map<String, Car> carsByBrand =
        index(cars, on(Car.class).getBrand());
```

Group sales by buyers and sellers (iterative version)

```
Map<Person,Map<Person,Sale>> map = new HashMap<Person,Map<Person,Sale>>();
for (Sale sale : sales) {
   Person buyer = sale.getBuyer();
   Map<Person, Sale> buyerMap = map.get(buyer);
   if (buyerMap == null) {
        buyerMap = new HashMap<Person, Sale>();
        map.put(buyer, buyerMap);
   }
   buyerMap.put(sale.getSeller(), sale);
}
Person youngest = null;
Person oldest = null;
for (Person person : persons) {
   if (youngest == null || person.getAge() < youngest.getAge())
        youngest = person;
   if (oldest == null || person.getAge() > oldest.getAge())
        oldest = person;
}
Sale saleFromYoungestToOldest = map.get(youngest).get(oldest);
```

Group sales by buyers and sellers (lambdaj version)

```
Group<Sale> group = group(sales,
   by(on(Sale.class).getBuyer()),by(on(Sale.class).getSeller()));
Person youngest = selectMin(persons, on(Person.class).getAge());
Person oldest = selectMax(persons, on(Person.class).getAge());
Sale sale = group.findGroup(youngest).findGroup(oldest).first();
```

Find most bought car

(iterative version)

```
Map<Car, Integer> carsBought = new HashMap<Car, Integer>();
for (Sale sale : sales) {
   Car car = sale.getCar();
   Integer boughtTimes = carsBought.get(car);
   carsBought.put(car, boughtTimes == null ? 1 : boughtTimes+1);
}

Car mostBoughtCarIterative = null;
int boughtTimesIterative = 0;
for (Entry<Car, Integer> entry : carsBought.entrySet()) {
   if (entry.getValue() > boughtTimesIterative) {
        mostBoughtCarIterative = entry.getKey();
        boughtTimesIterative = entry.getValue();
   }
}
```

Find most bought car

(lambdaj version)

```
Group<Sale> group = selectMax(
   group(sales, by(on(Sale.class).getCar())).subgroups(),
   on(Group.class).getSize());
Car mostBoughtCar = group.first().getCar();
int boughtTimes = group.getSize();
```

How does lambdaj work?

The getCar() invocation is propagated by the first proxy to all the sales. The cars returned by these invocations are put again in a list and another proxy, similar to the first one, is created to wrap this list, allowing to repeat this type of invocation once more (proxy concatenation).

```
Car fastestCar = max(
        forEach(sales).getCar(),
            on(Car.class).getSpeed());
```

A proxy wraps the list of sales. The returned object is of class Sale but dynamically implements the Iterable interface too

A proxy of the Car class is created in order to register the invocations on it. These invocations will be then reapplied to the cars of the former list

Lambdaj's extensibility

```java
List<Double> speeds = extract(cars, on(Car.class).getSpeed());
```

is the short form for:

```java
List<Double> speeds = convert(cars, new Car2SpeedConverter());
```

where the Car2SpeedConverter is defined as:

```java
class Car2SpeedConverter implements Converter<Car, Double> {
    public Double convert(Car car) {
        return car.getSpeed();
    }
}
```

Performance analysis

Minimum, maximum and average duration in milliseconds of 20 runs with 100,000 iterations of the former examples

	iterative			lambdaj			
	min	max	avg	min	max	avg	**ratio**
PrintAllBrands	265	312	283	1,310	1,591	1,377	**4.866**
FindAllSalesOfAFerrari	281	437	366	1,528	1,607	1,566	**4.279**
FindAllBuysOfYoungestPerson	5,585	5,975	5,938	6,895	6,989	6,936	**1.168**
FindMostCostlySaleValue	218	234	227	655	702	670	**2.952**
SumCostsWhereBothActorsAreAMale	358	452	375	2,199	2,637	2,247	**5.992**
AgeOfYoungestBuyerForMoreThan50K	5,257	5,319	5,292	9,625	9,750	9,696	**1.832**
SortSalesByCost	1,388	1,482	1,448	3,213	3,245	3,231	**2.231**
ExtractCarsOriginalCost	140	156	141	234	249	236	**1.674**
IndexCarsByBrand	172	203	186	327	343	336	**1.806**
GroupSalesByBuyersAndSellers	9,469	9,766	9,507	12,698	12,838	12,753	**1.341**
FindMostBoughtCar	3,744	3,884	3,846	4,181	4,259	4211	**1.095**

Average ratio = 2.658

Known limitations

- ***Lack of reified generics*** → lambdaj cannot infer the actual type to be returned when a null or empty collection is passed to forEach()

```
List<Person> persons = new ArrayList<Person>();
forEach(persons).setLastName("Fusco");
```

↖
Exception

- ***Impossibility to proxy a final class*** → the on() construct cannot register an invocation after a final Class is met

```
List<Person> sortedByNamePersons =
    sort(persons, on(Person.class).getName().toLowerCase());
```

↖
Exception

Let’s write it fluently

Fluent Interface Collections

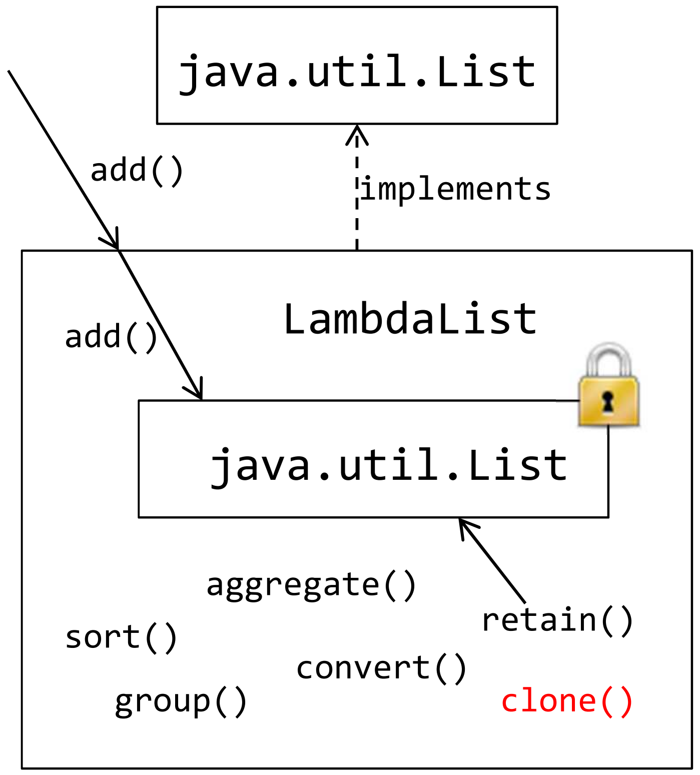
Why Fluent Interfaces

```
List<Person> richBuyersSortedByAge =
    sort(
        extract(
            select(sales,
                having(on(Sale.class).getValue(),
                greaterThan(50000)))
        ), on(Sale.class).getBuyer()
    ), on(Person.class).getAge());
```

```
List<Person> richBuyersSortedByAge = with(sales)
    .retain(having(on(Sale.class).getValue(),greaterThan(50000)))
    .extract(on(Sale.class).getBuyer())
    .sort(on(Person.class).getAge());
```

LambdaCollections

- LambdaCollections implement the corresponding Java interface (i.e. LambdaList is a java.util.List) so you can use them in all other API
- They enrich the Java Collection Framework API with a fluent interface that allows to use the lambdaj's features
- Invoking the methods of this fluent interface also change the state of the original wrapped collection



- The instance of the wrapped collection doesn't change so its characteristics are always reflected even by the wrapping lambdaj counterpart
- If you need to leave the original collection unchanged clone it:

```
      with(sales).clone().remove(…)  …
```

Let's go functional

Closures

(actually lambda expressions)

lambdaj's closure

Closures (or more properly lambda expressions) can be defined through the usual lambdaj DSL style

```
Closure println = closure(); {
    of(System.out).println(var(String.class));
}
```

and then invoked by "closing" its free variable once:

```
println.apply("one");
```

or more times:

```
println.each("one", "two", "three");
```

Closure's features

- Typed closure

```
Closure2<Integer,Integer> adder = closure(Integer.class, Integer.class); {
    of(this).sum(var(Integer.class), var(Integer.class));
}
```

- Curry

```
Closure1<Integer> adderOf10 = adder.curry2(10);
```

- Mix variables and fixed values

```
Closure1<Integer> adderOf10 = closure(Integer.class); {
    of(this).sum(var(Integer.class), 10);
}
```

- Cast a closure to a one-method interface (SAM)

```
Converter<Integer,Integer> converter = adderOf10.cast(Converter.class);
```

Closure's features (2)

- Keep unbound the object on which the closure is invoked

```
Closure2<Person, Integer> ageSetter = closure(Person.class, Integer.class); {
        of(Person.class).setAge(var(Integer.class));
    }
```

- Define a closure without using a ThreadLocal

```
Closure2<Person, Integer> ageSetter = new Closure2<Person, Integer>() {{
        of(Person.class).setAge(var(Integer.class));
    }};
```

- Define the invocation of a static method …

```
Closure1<String> intParser = closure(String.class)
         .of(Integer.class, "parseInt", var(String.class));
```

- … or of a constructor

```
Closure2<String, Integer> personCreator = closure()
        .of(Person.class, CONSTRUCTOR, var(String.class), var(Integer.class));
```

Switcher

```
public List<String> sortStrings(List<String> list) {
       // a sort algorithm suitable for Strings
}
public List<T> sortSmallList(List<T> list) {
       // a sort algorithm suitable for no more than 100 items
}
public List<String> sort(List<String> list) {
       // a generic sort algorithm
}


Switcher<List<T>> sortStrategy = new Switcher<List<T>>()
   .addCase(having(on(List.class).get(0), instanceOf(String.class)),
        new Closure() {{ of(this).sortStrings(var(List.class)); }})
   .addCase(having(on(List.class).size(), lessThan(100)),
        new Closure() {{ of(this).sortSmallList(var(List.class)); }})
   .setDefault(new Closure() {{ of(this).sort(var(List.class)); }});
```

Check out lambdaj at:

http://lambdaj.googlecode.com

Thank you

Mario Fusco
mario.fusco@gmail.com
twitter: @mariofusco