λambdaj

An internal DSL to manipulate collections without loops

by Mario Fusco
mario@exmachina.ch

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

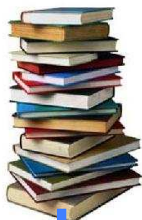
What is lambdaj for?

- 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

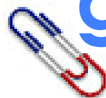
filter

sort



index

aggregate



group

extract

How does lambdaj work?

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

- treat 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");
```

- allow to define a pointer to a java method in a statically typed way

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

That’s all you need to know to start using lambdaj

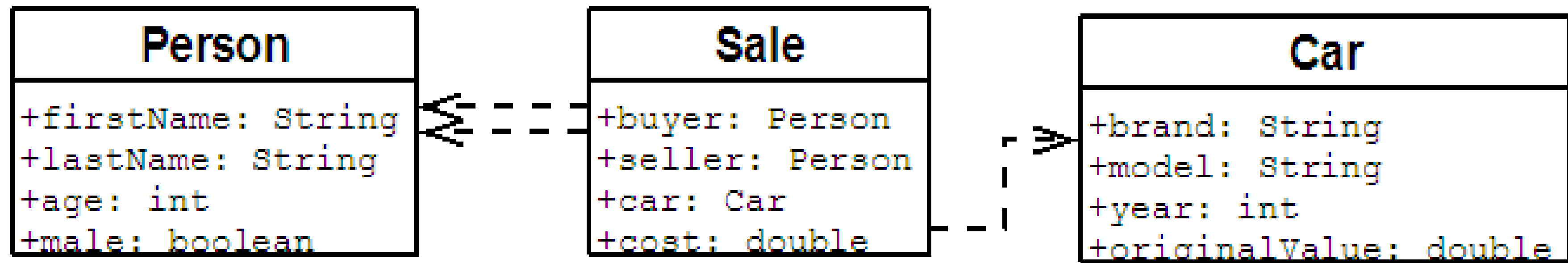
It sounds easy, isn’t it?

That’s all Folks!

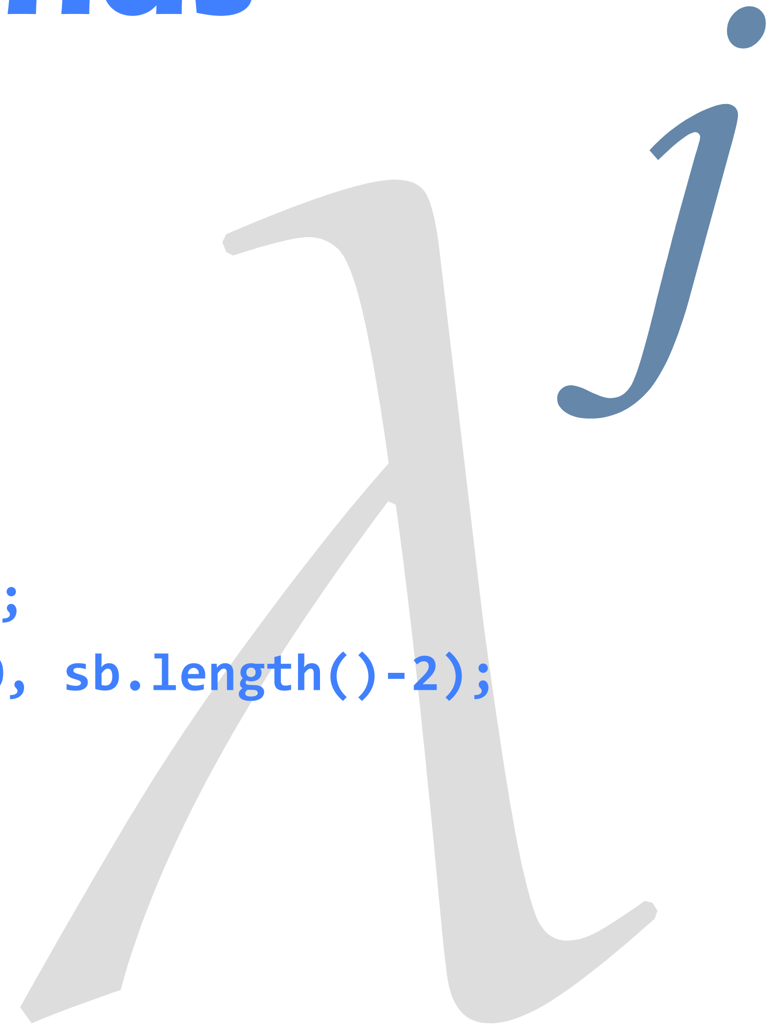
If you don’t believe me let’s see some

EXAMPLEs & DEMOs

The Demo Data Model



Print all cars' brands



Iterative version:

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

lambdaj version:

```
String brands = joinFrom(db.getCars()).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

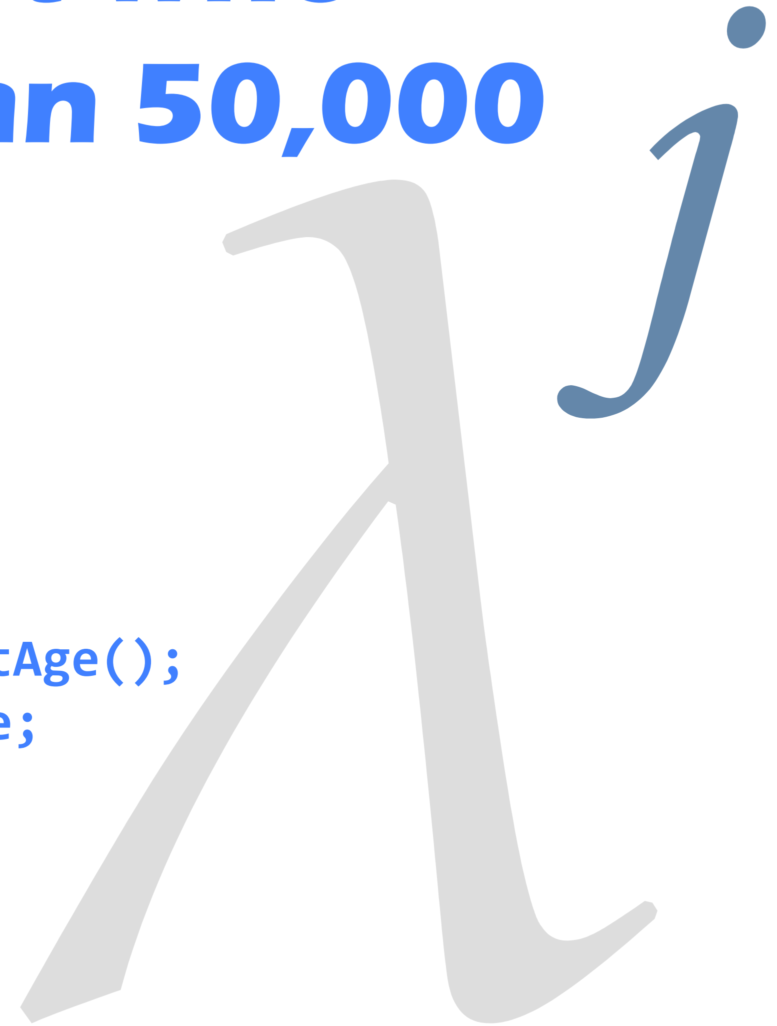
Iterative version:

```
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



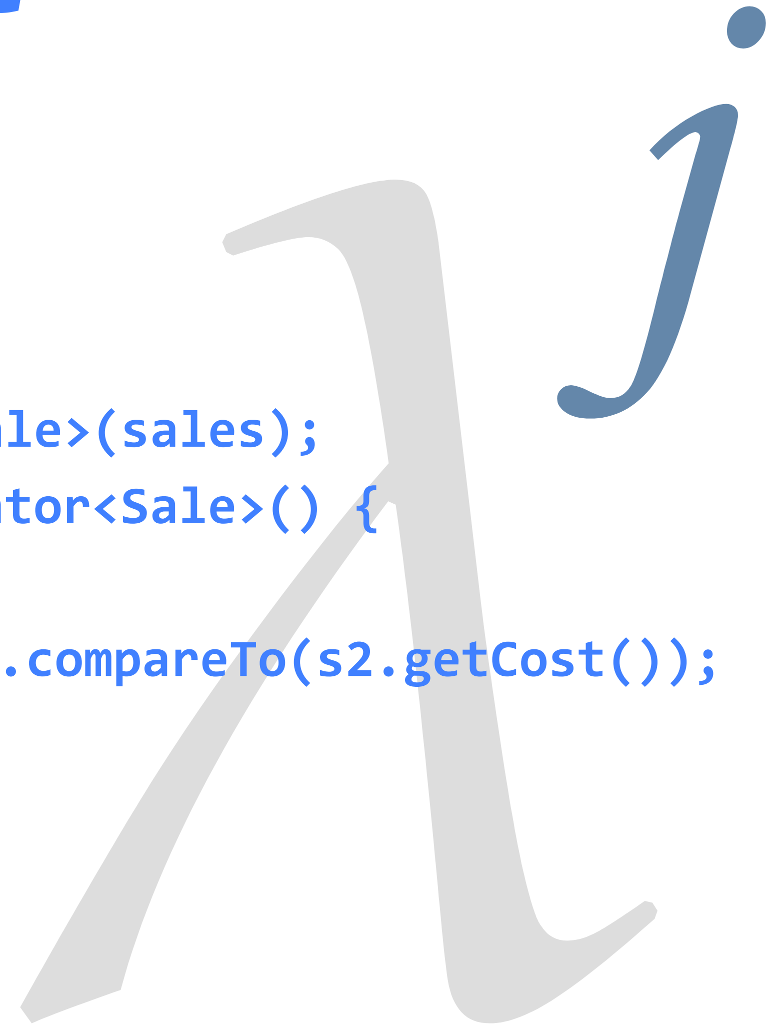
Iterative version:

```
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:

```
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:

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

Extract cars' original cost

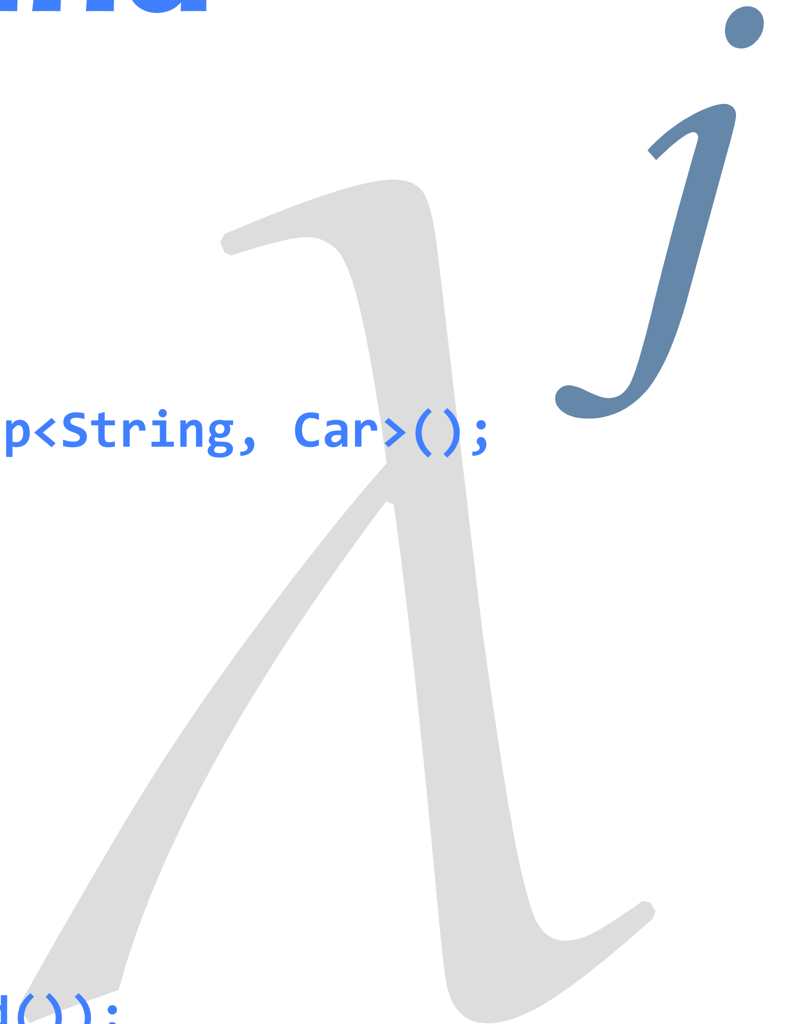
Iterative version:

```
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 : db.getCars())
   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).find(oldest).get(0);
```

Find most bought car

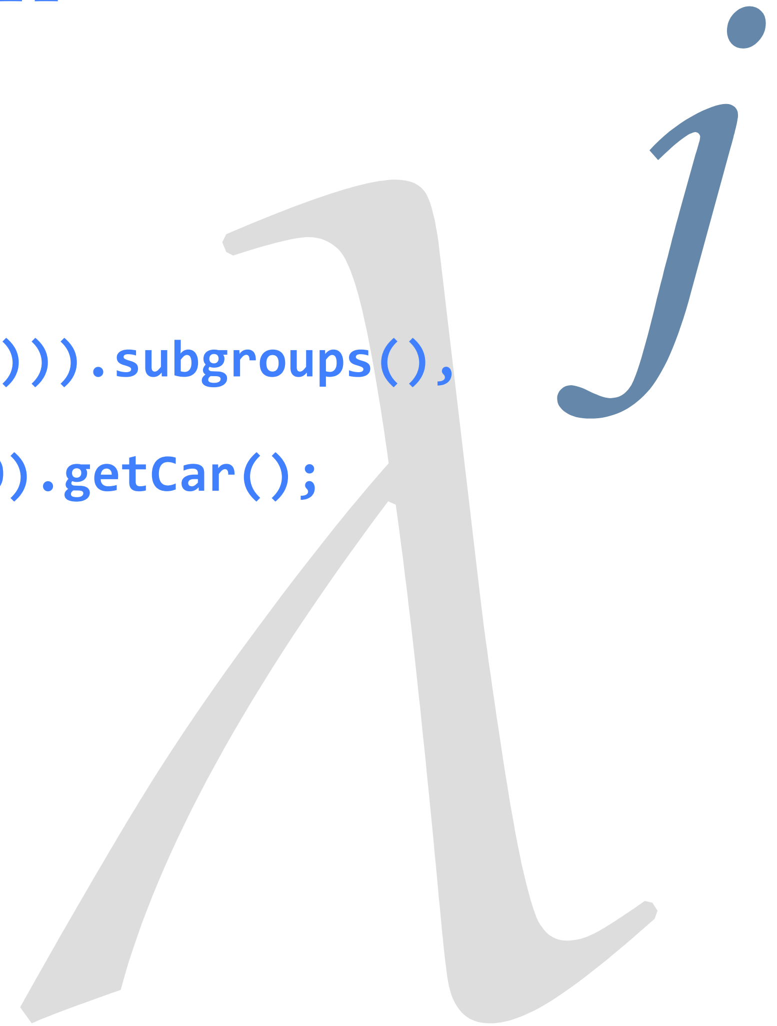
(iterative version)

```
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.findAll().get(0).getCar();
int boughtTimes = group.getSize();
```

Performance analysis

Minimum, maximum and average duration in milliseconds of 10 runs of 100,000 iterations of the former examples

	iterative			lambdaj			
	min	max	avg	min	max	avg	**ratio**
PrintAllBrands	656	828	768	3.797	4.390	3.887	**5,061**
FindAllSalesOfAFerrari	688	766	761	6.172	6.297	6.214	**8,166**
FindAllBuysOfYoungestPerson	13.485	14.969	14.757	18.953	19.375	19.124	**1,296**
FindMostCostlySaleValue	531	547	544	3.546	3.579	3.564	**6,551**
SumCostsWhereBothActorsAreAMale	843	875	859	10.828	11.141	11.021	**12,830**
AgeOfYoungestBuyerForMoreThan50K	12.890	13.063	12.965	24.860	25.234	25.089	**1,935**
SortSalesByCost	3.250	3.500	3.421	21.953	22.312	22.080	**6,454**
ExtractCarsOriginalCost	328	344	338	1.172	1.219	1.206	**3,568**
IndexCarsByBrand	469	500	485	1.468	1.500	1.484	**3,060**
GroupSalesByBuyersAndSellers	23.750	24.203	23.929	36.359	37.250	36.878	**1,541**
FindMostBoughtCar	8.000	8.203	8.074	23.844	24.218	24.053	**2,979**

Average ratio = 4,858

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

Closures

lambdaj's closure

Closures (or more properly first-class functions) 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 free and fixed variables

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

➢ Cast a closure to a one-method interface

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

Check out lambdaj at:

code.google.com/p/lambdaj

Thank you

Mario Fusco mario@exmachina.ch