Java 8: Lambdas and Collection Handling

Lukasz Wrobel

About me

- Software architect, team leader
- Recruiter, teacher
- High-traffic, scalable systems
- lukaszwrobel.pl / @lukaszwrobel
- eBook: „Memoirs of a Software Team Leader”

New features

- To me—the first breakthrough since generics.
- Lambdas
  - @FunctionalInterface
  - default methods
- Optional

Implications

- Readable collection operations
  - other languages:
    Ruby (→ Groovy), Scala, Clojure
- DSLs
- Parallelism
- ~~NullPointerException~~

Collections
+
lambdas

A matter of personal importance to me

Event handling

At best—anonymous inner class

```
final Button button = (Button) findViewById(R.id.button_id);

button.setOnClickListener(new View.OnClickListener() {
    public void onClick(View v) {
        // Perform action on click
    }
});
```

How about a cleaner syntax?

```
final Button button = (Button) findViewById(R.id.button_id);

button.setOnClickListener((View v) -> {
    // Perform action on click
});
```

Collection manipulation

Old approach

```
private static int sumOfSquaresGreaterThan50(List<Integer> numbers) {
    Iterator<Integer> it = numbers.iterator();
    int sum = 0;

    while (it.hasNext()) {
        int number = it.next();
        int square = Math.pow(number, 2);

        if (square > 50) {
            sum += square;
        }
    }

    return sum;
}
```

Stream API

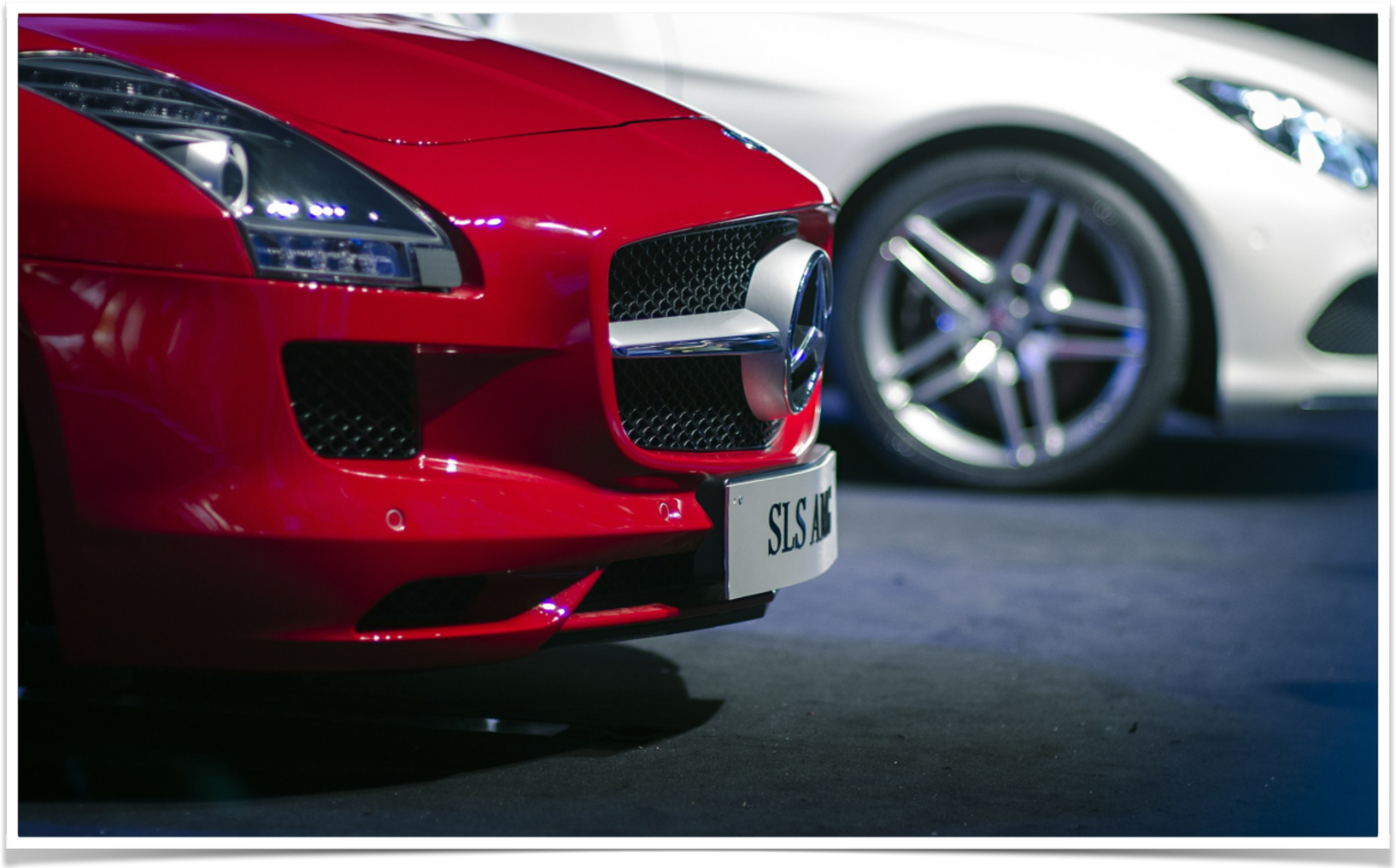
```
private static int sumOfSquaresGreaterThan50(List<Integer> numbers) {
    return numbers
        .stream()
        .mapToInt(number -> Math.pow(number, 2))
        .filter(square -> square > 50)
        .sum();
}
```

Intermediate vs. terminal

Finish right now or add another operations the chain?

Terminal operations

- forEach[Ordered]
- toArray
- sorted
- reduce
- collect
- min, max
- count
- [any|all|none]Match
- find[First|Any]



Domain: cars

Common use cases

Filter

```
public static List<Car> getInsurancedCars(ArrayList<Car> cars) {
    return cars
        .stream()
        .filter(car -> car.isInsuranced())
        .collect(Collectors.toList());
}
```

Method reference

```
public static List<Car> getInsurancedCars(ArrayList<Car> cars) {
    return cars
        .stream()
        .filter(Car::isInsuranced)
        .collect(Collectors.toList());
}
```

Map

```
public static List<Plates> getPlates(ArrayList<Car> cars) {
    return cars
        .stream()
        .map(Car::getPlates)
        .collect(Collectors.toList());
}
```

Max

```
public static int getTheBiggestPrice(ArrayList<Car> cars) {
    return cars
        .stream()
        .max(Car::getPrice); // Terminal operation
}
```

Primitives → Performance

```
public static int getAverageNumberOfSeats(ArrayList<Car> cars) {
    return cars
        .stream()
        .mapToInt(Car::getNumberOfSeats)
        .average();
}

getAverageNumberOfSeats(allCarsInTheEntireUniverse);
```

Variable capturing

Either final or effectively final

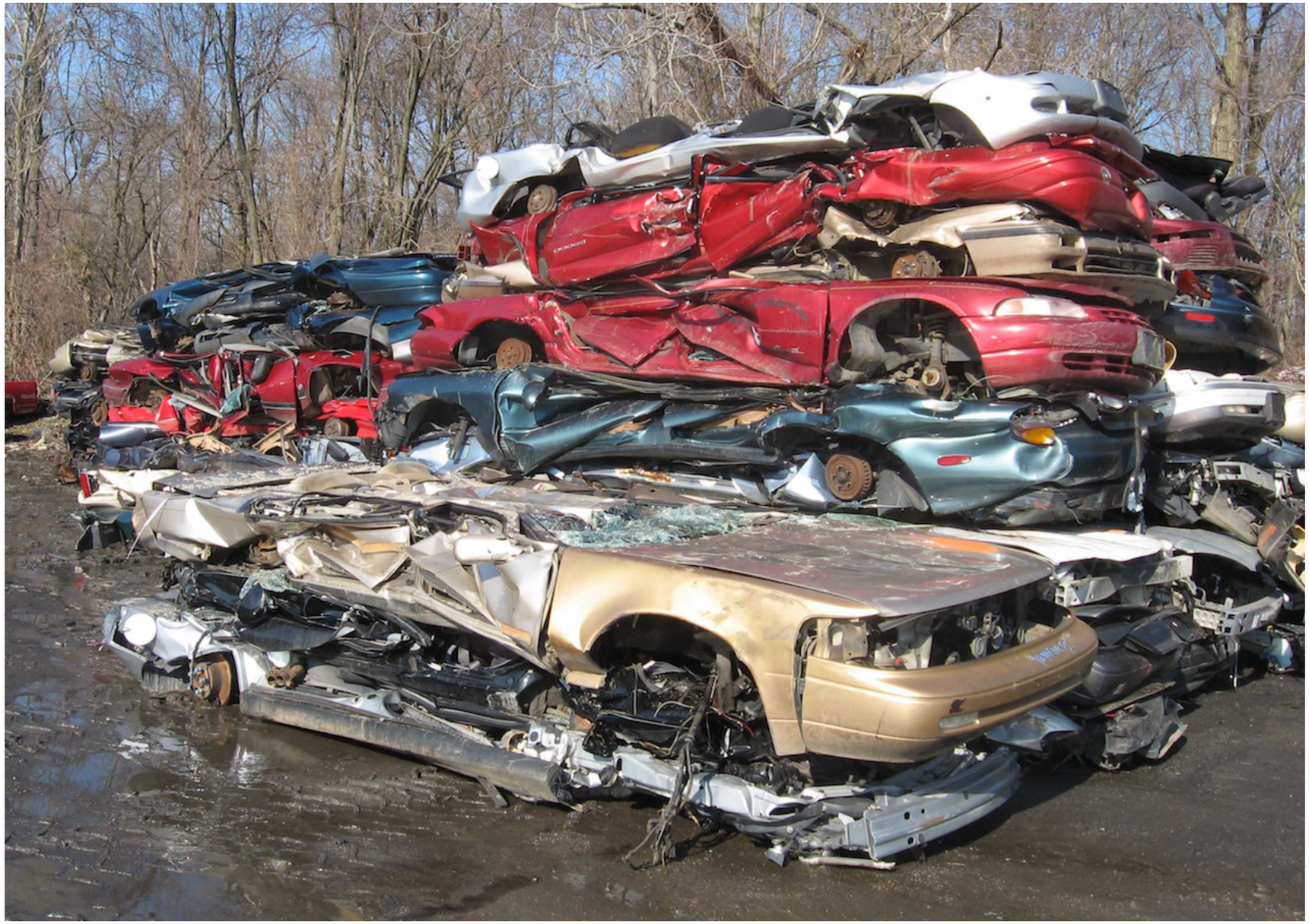
Final

```
final Car car = new AMG("WQ 42684");

button.setOnClickListener((View v) -> {
    displayMessage("Plates: " + car.getPlates());
});
```

Effectively final

```
Car car = new AMG("WQ 42684");
car = new Fiat126P("DFB 93453");

button.setOnClickListener((View v) -> {
    displayMessage("Plates: " + car.getPlates());
});
```

Lambda type

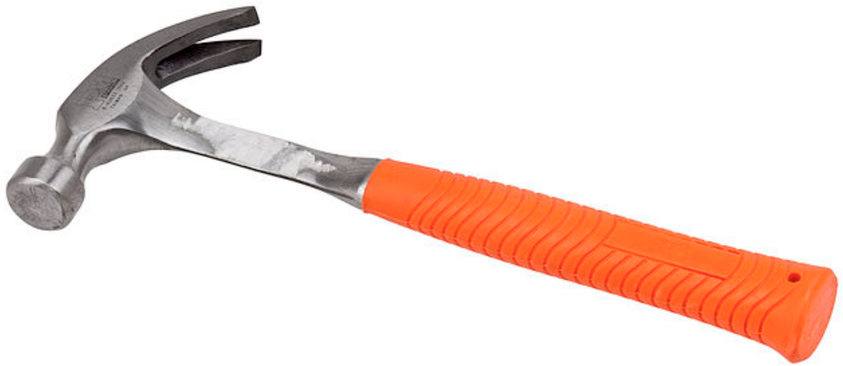


Inline?

```
public class Scrapyard {
    public List<ScrapMetal> scrap(
        List<Car> cars,
        (Car -> ScrapMetal) scrapper) {
        // Scrap all the given cars
    }
}
```

What about type reuse?

@FunctionalInterface



```
@FunctionalInterface
public interface Scrapper {
    ScrapMetal scrap(Car car);
}
```

Functional interface applied

```
public class Scrapyard {
    public List<ScrapMetal> scrap(
        List<Car> cars,
        Scrapper scrapper) {
    }
}
```

Useful functional interfaces

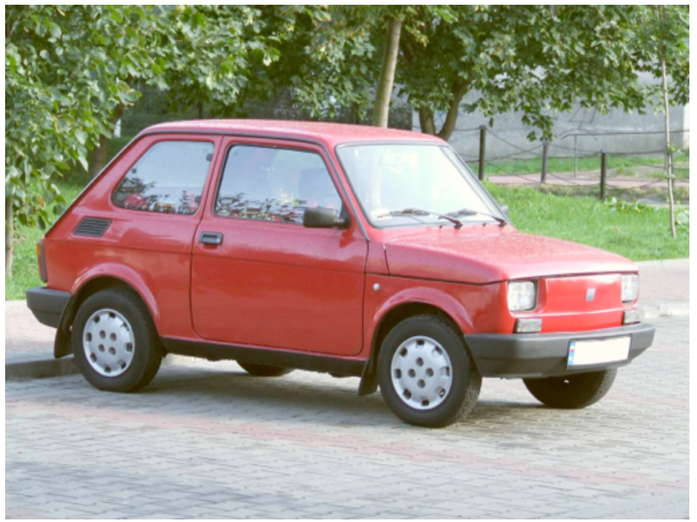
Name	Arguments	Return type
Function<T, R>	T	R
Predicate<T>	T	boolean
UnaryOperator<T>	T	T
BinaryOperator<T>	T, T	T
Supplier<T>	-	T
Consumer<T>	T	void

Default methods

Binary compatibility
Java 8 → 7 → … 1

„default” keyword

```
public interface AirConditioned {
    default void coolTheInterior() {
        // Perform the cooling process
    }
}

public class Fiat126P implements AirConditioned {
    // Thanks to the "default" keyword, Fiat 126p
    // is now being air-conditioned.
}
```




IT'S DANGEROUS TO GO ALONE!
TAKE ONE OF THESE!

What for?

<<*Custom*>> collections.
No stream() method? Take the one provided.

„default” precedence

class > interface

subtype > supertype
(interfaces extending other interfaces)

Parallelism

Effortless

```
public static List<Car> getInsurancedCars(ArrayList<Car> cars) {
    return cars
        .parallelStream()
        .filter(Car::isInsuranced)
        .collect(Collectors.toList());
}
```

However…

Amdahl’s law

The speedup of a program using multiple processors in parallel computing is limited by the time needed for the **sequential fraction** of the program.

http://en.wikipedia.org/wiki/Amdahl%27s_law

Is it worth it?

- Number of elements:
  large enough?
- Time spent processing each element
  substantial?
- Performance tests.

Parallel reduce

- Initial value can be combined with any element.
- Arbitrary order of evaluation.

Data structure splitting

Performance	Access type	Example
Good	random	ArrayList
Bad	sequential	LinkedList

State

Performance	State	Example
Good	stateless	filter
Bad	stateful	sorted

How is your life going to change?

DSLs

```
public class HelloWorld {
    public static void main(String[] args) {
        get("/hello", (req, res) -> „Hello, World!”);
    }
}
```

Caching— the old way

```
public Car getCarCached(Plate plate) {
    Map<Plate, Car> cache = this.getCarCache();

    Car car = cache.get(plate);
    if (car == null) {
        car = this.getCarFromDatabase(plate);
        cache.put(plate, car);
    }

    return car;
}
```

Convenience

```
public Car getCarCached(Plate plate) {
    return this.getCarCache()
        .computeIfAbsent(plate, this::getCarFromDatabase);
}
```

Thanks for listening!