Будущее Java, грядущие новшества Java 8

Андрей Родионов
Физико-технический институт
Лидер Java User Group НТУУ “КПИ”

Andrii.Rodionov@gmail.com
http://jug.ua/

О чем будет рассказ

- Virtual extension methods (~~Defender Methods~~)
- Functional Collection Patterns
- Lambda Expressions
- Parallel computing

На сегодня имеем

- Java 7
  - Вышла в конце июля этого кода
  - Особых нареканий не вызвала, «отцы-основатели» Java остались довольны
- Java 8
  - Выход запланирован на лето 2013
  - JDK 8 уже доступна для скачивания http://jdk8.java.net/lambda
  - Внесет революционный изменения
  - Кардинально противоположный мнения

Поговорим о том, что как раз и вызывает больше всего протестов

Virtual Extension Methods (Defender Methods)

Проблематика

```
interface I{
    void show();
}
```

Интерфейс определяет поведение объекта, «услуги» которые нам предоставляет объект

```
public class A implements I{
    public void show() {
    }
}
```

Развития языка требует и развития его библиотек, то есть появление новых методов

Добавление новых методов в интерфейсы, не нарушает ***бинарную совместимость*** *(при раздельной компиляции), но нарушает* ***совместимость исходного кода***

Бинарная совместимость и совместимость исходного кода

- Совместимость исходного кода (Source compatibility)
  - Компиляция исходного кода с обновленным API проходит без ошибок
- Бинарная совместимость (Binary compatibility)
  - Запуск бинарного кода с обновленным API проходить без ошибок линковки

Source-compatible => Binary-compatible

Binary-compatible ≠> Source-compatible

Binary incompatible => Source incompatible

Постановка задачи

```
public interface NewInterface{
    void test2();
    void test();
/* хотим добавить новый метод в существующий интерфейс не нарушая совместимости исходного кода (без реализации его в классе NewClass) */
}

public class NewClass implements NewInterface{
    public void test2(){ System.out.println("My Hello"); }
}
```

Virtual Extension Methods

```
public interface NewInterface{
    void test2();
    void test() default DefaultClass.test;
            //default { DefaultClass.test(this); };
}
public class DefaultClass {
    public static void test(NewInterface ni){
        System.out.println("Default Hello");
    }
}
```

Получаем следующее

```
public interface NewInterface{
    void test2();
    void test() default DefaultClass.test; }

public class DefaultClass {
   public static void test(NewInterface ni){
            System.out.println("Default Hello");
   }}

public class NewClass implements NewInterface{
    public void test2(){
            System.out.println("My Hello");
    } }
```

Множественное наследование поведения

- Возможность множественного наследование *поведения* (но не *состояния*)

```
interface I1{
    void show() default B.show;
}
interface I2{
    void test() default C.test;
}
public class A implements I1, I2{
    public void someMethod() {...}
}
```

```
class B{
    static void show(I1 i) {...}
}
```

```
class C{
    static void test(I2 i) {...}
}
```

```
A a = new A();
a.show();
a.test();
a.someMethod();
```

Особенности поведения

```
interface A { void m() default X.a; }
interface B extends A { void m() default X.b; }
interface C extends A { }
class D implements B, C { }
```

```
interface A { void m() default X.a; }
interface B extends A { void m() default none; }
class D implements B { }
```

```
interface A { void m() default X.a; }
class C { abstract void m(); }
class D extends C implements A { }
```

```
interface A { void m() default X.a; }
interface B { void m() default X.b; }
interface Q extends A, B {  public void m() default B.super.m;  }
class C implements A, B {
        public void m() { A.super.m(); }
}
```

Functional Collection Patterns

Рассмотрим такой пример

```
List<Student> students = ...
double highestScore = 0.0;
for (Student s : students) {
    if (s.gradYear == 2011) {
            if (s.score > highestScore) {
                highestScore = s.score;
            }
    }
}
```

Производим итерации вручную (external iteration)
Последовательно сравниваем всех студентов
Можем «случайно» изменить студента

Было бы здорово если ...

- Коллекции сами знали, как работать со своими элементами, а мы бы передавали в них лишь ряд критериев и правил

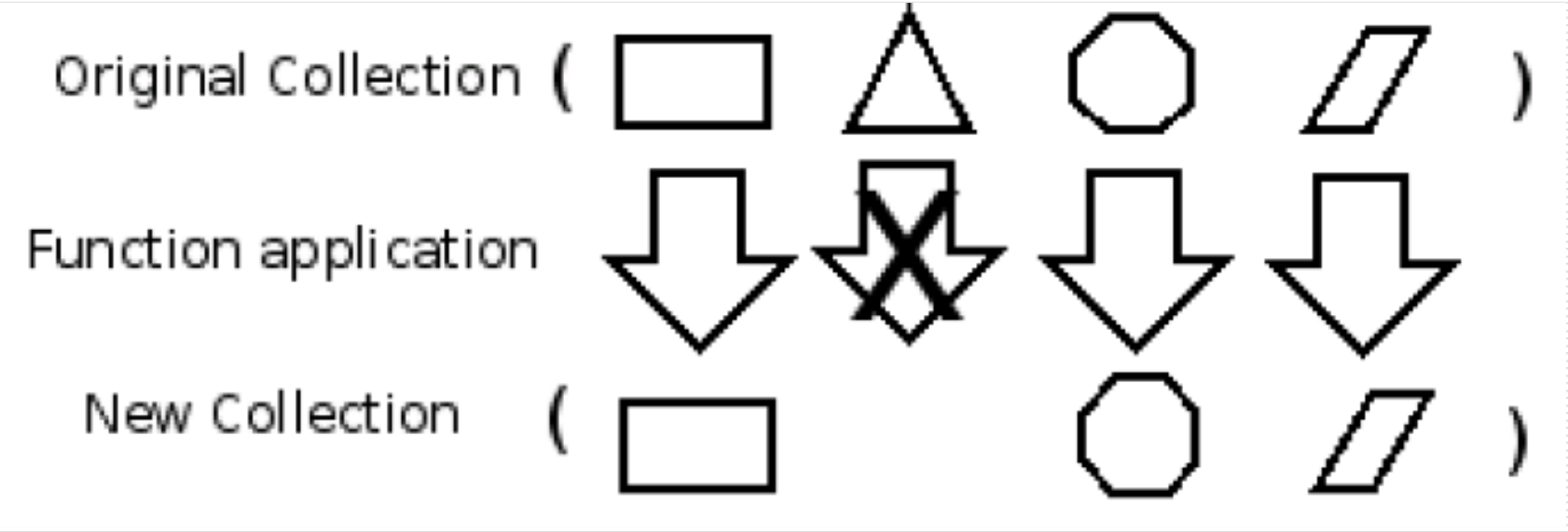
```
SomeCoolList<Student> students = ...
double highestScore =
        students.filter( … )
                .map( … )
                .reduce( … );
```

Functional Collection Patterns

- filter()
- map()
- reduce()
- forEach()

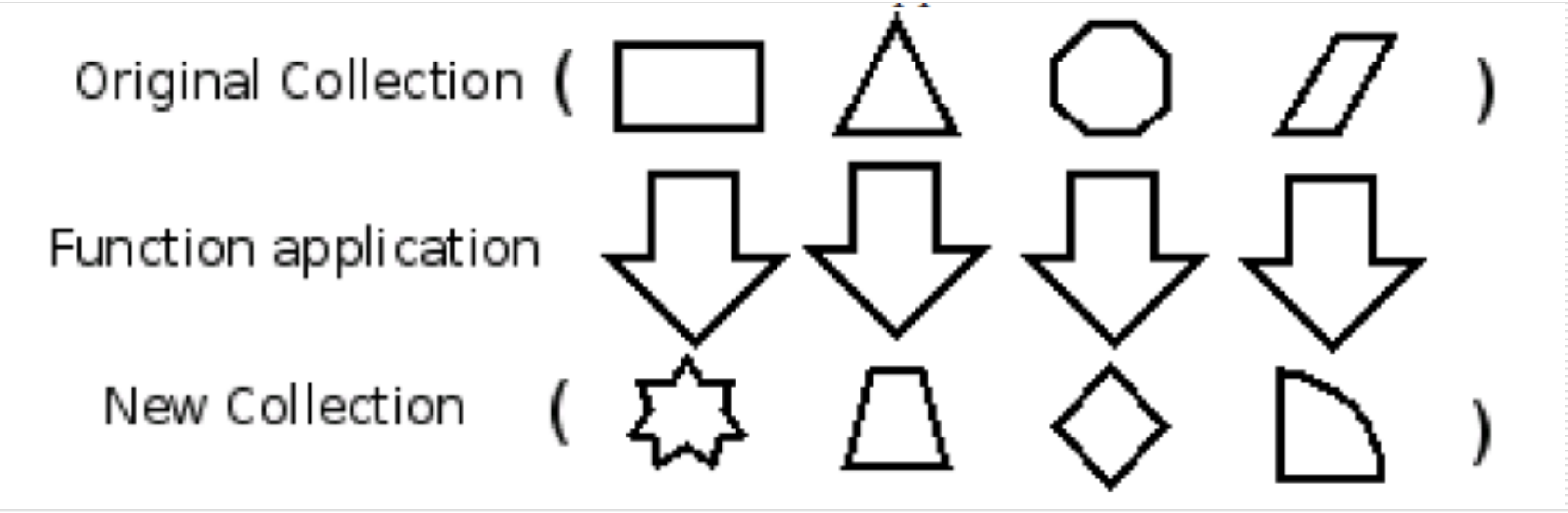
Filter

The Filter pattern evaluates a predicate (a function which returns a Boolean) on each of the elements, returning a new collection which is subset of the original collection.



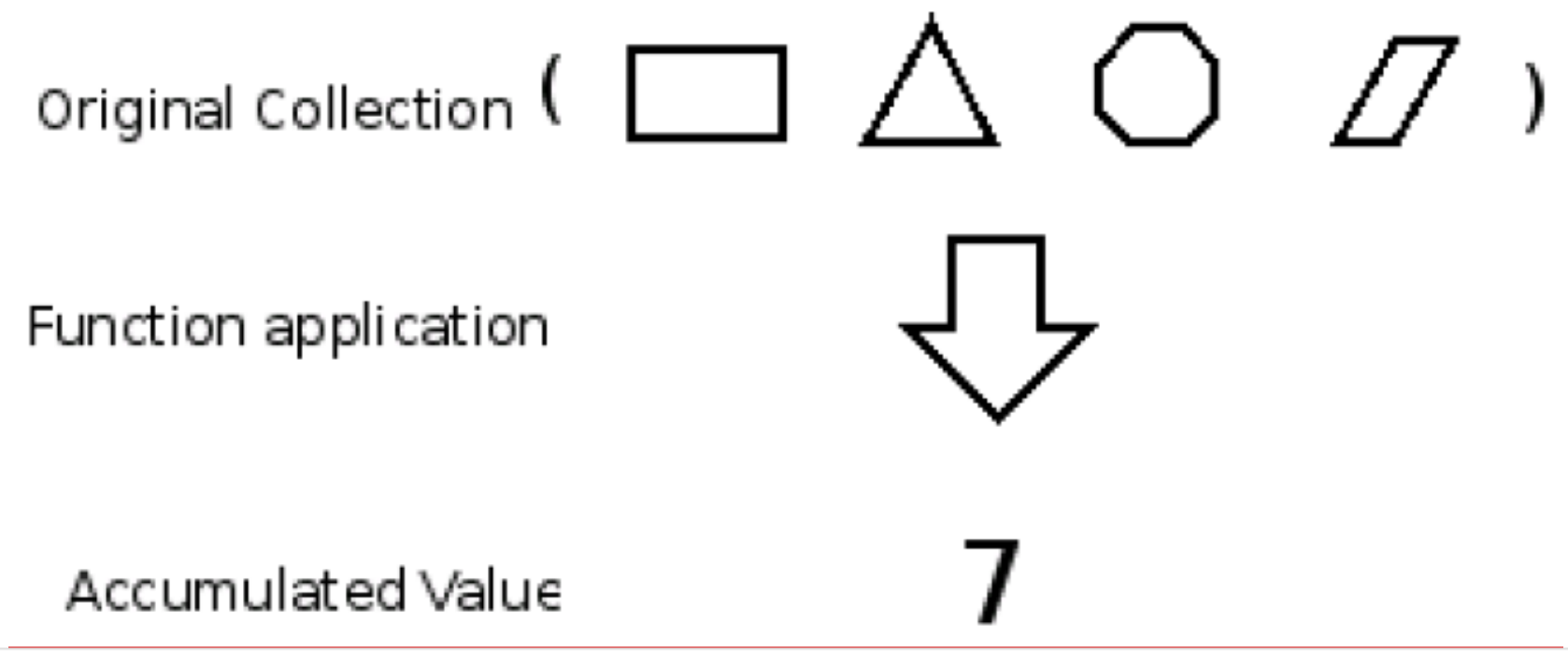
Map

The Map pattern evaluates a high-order function on all elements of the collection.
It returns a new collection with the results of each function application.



Reduce

The Reduce pattern evaluates a function on all elements of the collection, returning a scalar value.



Появились новые методы в List

- Что бы добавить новые методы используются *Virtual Extension Methods*
- Посмотрим на методы в старом интерфейсе ***Iterable*** и на методу в ***Iterable*** в JDK 8

Получаем следующее

```
List<Student> students = ...
highestScore =
        students.filter(new Predicate<Student>(){
                        public boolean eval(Student s){
                                        return s.getGradYear()== 2011;
                        }
                }).map(new Mapper<Student, Double>(){
                        public Double map(Student s){
                                        return s.getScore();
                        }
                }).reduce(0.0, new Operator<Double>(){
                        public Double eval(Double left, Double right) {
                                if (left > right) return left;
                                return right;
                        }
                });
```

Правда выглядит жутковато по сравнению с первоначальным вариантом?!

Что бы не было так страшно, вводятся Lambda Expressions

Lambda Expressions

- Lambda expressions are anonymous functions
  - Like a method, has a typed argument list, a return type, a set of thrown exceptions, and a body

```
double highestScore =
    students.filter(Student s -> s.getGradYear() == 2011)
            .map(Student s -> s.getScore())
            .max();
```

Lambda Expressions

- A *functional interface* is an interface that has just one abstract method
- A *lambda expression* is a way to create an instance of a *functional interface*

```
interface F{
        int f();
}

interface F1{
        int add(int x, int y);
}
```

```
F func = () -> 2011;
func.f();

F funcan = new F() {
   public int f(){  return 2011; }
};
funcan.f();

F1 func1 = (a,b) -> a+b;
int i = 5, j = 13;
func1.add(i,j);
```

Передача лямба-выражения в качестве параметра

```
interface F2{
        int eval(int a, int b);
}

class A{
        static int max(int a, int b, F2 f){
                return f.eval(a,b);
        }
}
```

```
F2 fn = (x, y) -> (x > y) ? x : y;
   int max = A.max(i, j, fn);
```

```
int max = A.max(i, j, (x, y) -> (x > y) ? x : y );
```

Пример реализации Comparator

```
Comparator<String> c = new Comparator<String>() {
        public int compare(String x, String y) {
                return x.length() - y.length();
        }
};
```

```
Comparator<String> c =
        (String x, String y) -> x.length() - y.length();
```

```
Collections.sort(ls, (String x, String y) -> x.length() - y.length());
```

Как выглядит наш пример

```
Operator<Double> op = (Double x, Double y) -> {
                          if (x > y) return x;
                          return y;
                          };

highestScore =
    students.filter((Student s) -> s.getGradYear() == 2010)
            .map((Student s) -> s.getScore())
            .reduce(0.0, op);
          //.reduce(0.0,  (Double left, Double right) ->
          //                (left > right) ? left : right );
```

Зачем это все?

Что кроме удобного синтаксиса?

Virtual Extension Methods

+

Functional Collection Patterns

+

Lambda Expressions

=

Parallel computing

Цель — автоматическое распараллеливание операций

```
List<Student> students = new ArrayList<>(…);
…
double highestScore =
    students.parallel()
            .filter(s -> s.getGradYear() == 2011)
            .map(s -> s.getScore())
            .max();
```

Как работает сейчас

- Операции применяются последовательно

X0 — начальное значение

Шаги:
1. filter → map → reduce(**X0**, eval) → **X1**
2. filter → map → reduce(**X1**, eval) → **X2**
3. filter
4. filter → map → reduce(**X2**, eval) → **X3**

...

n. filter → map → reduce(**Xn-1**, eval) → **Xn**

Xn — конечный результат

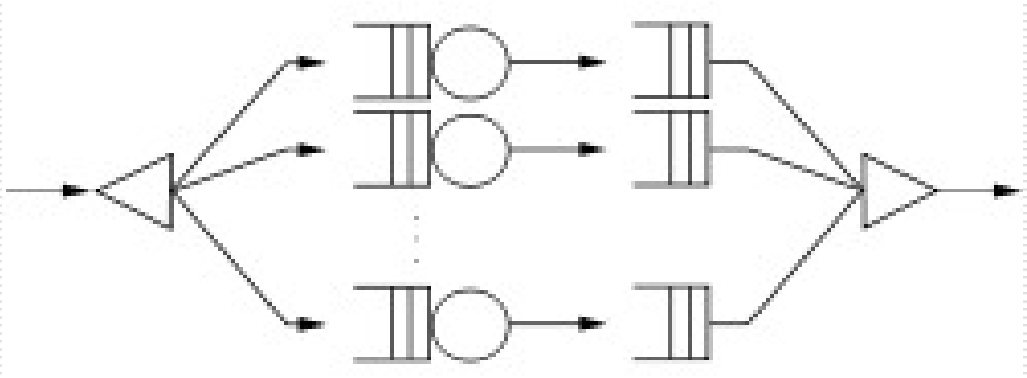
- Пример

1. Anna → Filter
2. Andrii → Filter→ Map → Reduce: eval l: **0.0** r: 4.5
3. Nik → Filter → Map → Reduce: eval l: **4.5** r: 4.2
4. Ann → Filter → Map → Reduce: eval l: 4.5 r: **4.9**
5. Serg → Filter

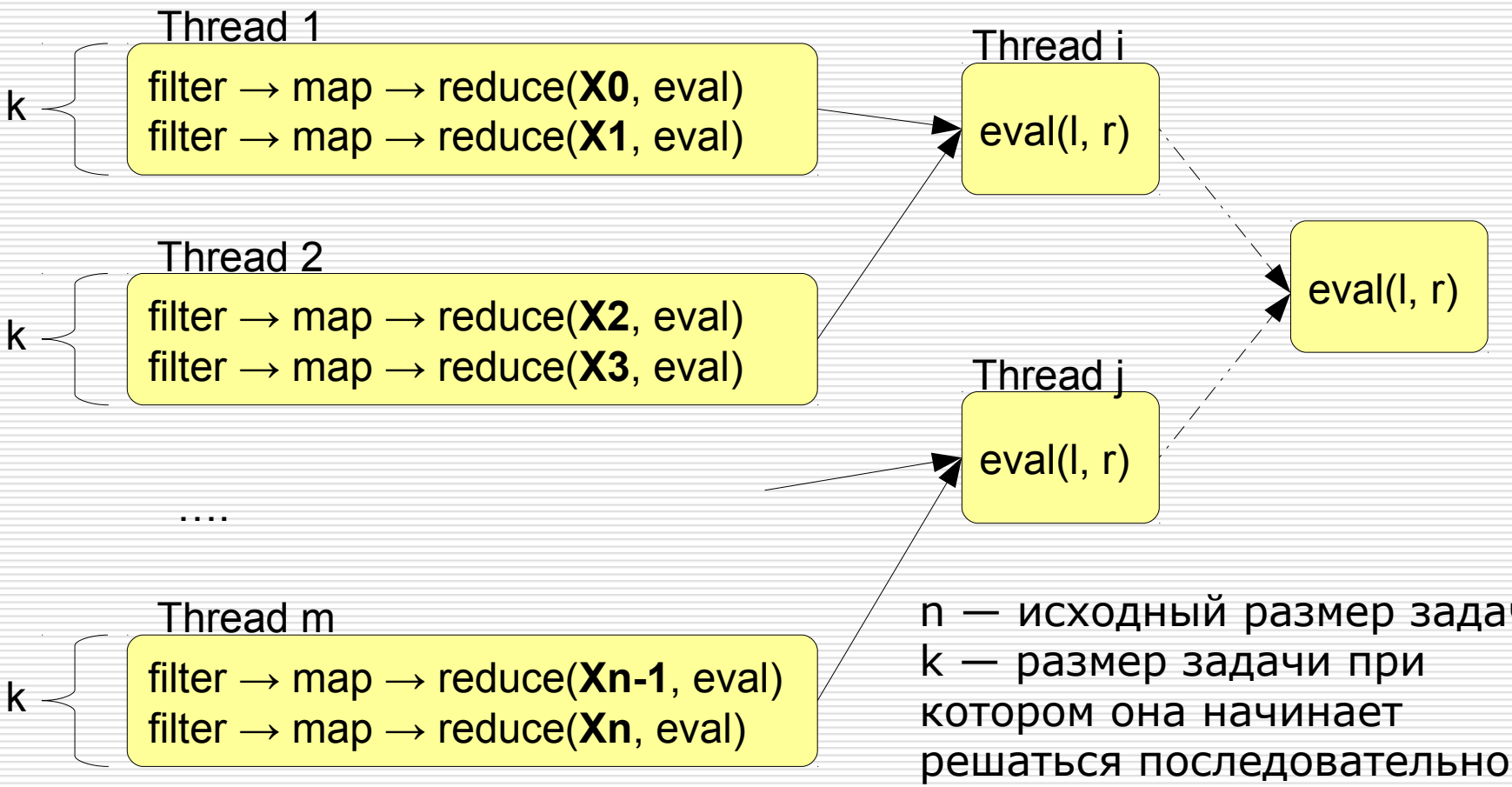
answer = ***4.9***

Хотим сделать чтобы работало параллельно

- Операции ***filter*** и ***map*** могут выполняться параллельно, но операция ***reduce*** зависит от предыдущего значения
- Но можем применить подход «divide-and-conquer» с использование Fork-Join framework



Алгоритм «divide-and-conquer»



Как выглядит в коде сейчас

```
class ScoreProblem {
    final List<Student> students;
    final int size;
    ScoreProblem(List<Student> ls) {
        this.students = ls;
        this.size = this.students.size();
    }

    public double solveSequentially() {
        double highestScore =
            students.filter(s -> s.getGradYear() == 2011)
                .map(s -> s.getScore())
                .reduce(0.0, (x,y) -> Math.max(x,y));
                            return highestScore;
    }
    public ScoreProblem subproblem(int start, int end) {
        return new ScoreProblem(students.
                            subList(start, end));
    }
}

class ScoreFinder extends RecursiveAction {
    private final ScoreProblem problem;
    double highestScore = 0;
    private int THRESHOLD = 2;
    public ScoreFinder(ScoreProblem p){
        problem = p;
    }
```

```
    protected void compute() {
     if (problem.size < THRESHOLD) {
         highestScore = problem.solveSequentially();
     } else {
         int m = problem.size / 2;
         ScoreFinder left, right;
         left = new ScoreFinder(problem.
                subproblem(0, m));
         right = new ScoreFinder(problem.
                 subproblem(m, problem.size));
         invokeAll(left, right);
         Operator<Double> op =
                             (x,y) -> Math.max(x,y);
         highestScore = op.eval(left.highestScore,
                             right.highestScore);
     }
    }
 }
public class ScoreParallel{
  private static int nThreads=3;
  public static void main(String [] args){
    ScoreProblem problem =
        new ScoreProblem(students);
    ForkJoinPool pool =
        new ForkJoinPool(nThreads);
    ScoreFinder finder =
        new ScoreFinder(problem);
    pool.invoke(finder);
 }
```

- Выглядит ужасно …
- Но, всю эту работу за нас могут выполнять параллельные версии соответствующих методов
- Мы лишь будем передавать в них лямбда-выражения

Можно передавать лямбда выражения

```
class ScoreProblem {
    final List<Student> students;
    final int size;
    ScoreProblem(List<Student> ls) {
        this.students = ls;
        this.size = this.students.size();
    }

    public double solveSequentially() {
        double highestScore =
            students.filter(s -> s.getGradYear() == 2011)
                .map(s -> s.getScore())
                .reduce(0.0, (x,y) -> Math.max(x,y));
                                return highestScore;
    }
    public ScoreProblem subproblem(int start, int end) {
        return new ScoreProblem(students.
                                subList(start, end));
    }
}

class ScoreFinder extends RecursiveAction {
    private final ScoreProblem problem;
    double highestScore = 0;
    private int THRESHOLD = 2;
    public ScoreFinder(ScoreProblem p){
        problem = p;
    }
```

```
    protected void compute() {
     if (problem.size < THRESHOLD) {
          highestScore = problem.solveSequentially();
     } else {
          int m = problem.size / 2;
          ScoreFinder left, right;
          left = new ScoreFinder(problem.
                subproblem(0, m));
          right = new ScoreFinder(problem.
                 subproblem(m, problem.size));
          invokeAll(left, right);
          Operator<Double> op =
                                (x,y) -> Math.max(x,y);
          highestScore = op.eval(left.highestScore,
                                right.highestScore);
     }
    }
 }
public class ScoreParallel{
  private static int nThreads=3;
  public static void main(String [] args){
     ScoreProblem problem =
          new ScoreProblem(students);
     ForkJoinPool pool =
          new ForkJoinPool(nThreads);
     ScoreFinder finder =
          new ScoreFinder(problem);
     pool.invoke(finder);
}
```

Что почитать

- JDK 8 - http://jdk8.java.net/lambda/
- Project Lambda
  - http://openjdk.java.net/projects/lambda/
- Language / Library / VM Co-Evolution in Java SE 8
  - http://blogs.oracle.com/briangoetz/resource/devoxx-lang-lib-vm-co-evol.pdf
- JSR 335: Lambda Expressions for the Java
  - http://jcp.org/en/jsr/detail?id=335
- JDK Enhancement Proposals
  - http://openjdk.java.net/jeps/

- Презентация и коды примеров
  - http://jug.ua

Спасибо!

Andrii.Rodionov@gmail.com