

# Будущее Java, грядущие новшества Java 8

---



Андрей Родионов  
Физико-технический институт  
НТУУ «КПИ»

[Andrii.Rodionov@gmail.com](mailto:Andrii.Rodionov@gmail.com)

# О чем будет рассказ

---

- Defender Methods
  - Functional Collection Patterns
  - Lambda Expressions
-

# На сегодня имеем

---

## □ Java 7

- Вышла в конце июля этого года
- Мнения относительно «полезности» изменений разделились

## □ Java 8

- Выход запланирован на октябрь 2012
  - JDK 8 уже доступна для скачивания  
<http://jdk8.java.net/>
  - Внесет революционные изменения
  - Кардинально противоположные мнения
-

---

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

---

---

***Defender Methods***  
***Virtual Extension 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 бишь появление новых методов*

*Необходимо добавлять новые методы в интерфейсы, при этом не нарушая **бинарную совместимость** и **совместимость исходного кода***

---

# Совместимость исходного кода

---

- Компиляция исходного кода с обновленным API без ошибок

**> *javac -cp apiv2.jar client/src/\*.java***

*ClientClass.java:45: cannot find symbol*

*symbol: class RemovedPublicSuperclass*

*public class ClientClass extends RemovedPublicSuper*

---

# Бинарная совместимость

---

- Запуск бинарного кода с обновленным API без ошибок линковки

***>java -cp apiv2.jar:client.jar com.sun.dem***

*Exception in thread "main" java.lang.NoClassDefFoundError:  
com.sun.apide*

*at java.lang.ClassLoader.defineClass1(Native Method)*

*at java.lang.ClassLoader.defineClass(ClassLoader.java:620)*

*...*

---

# Постановка задачи

---

```
public interface NewInterface{  
    void test2();  
    // хотим добавить сюда метод void test() не  
    // реализуя его в классе NewClass  
}
```

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

---

# Defender 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");
    }
}
```

---

# Defender Methods

---

- Открывают ящик Пандоры
  - Получаем множественное наследование

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

---

---

# 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;
        }
    }
}
```

Вручную производим итерации  
Последовательно сравниваем всех студентов  
Можем «случайно» изменить студента

---

# Было бы здорово если бы ...

---

```
SomeCoolList<Student> students = ...
double highestScore =
    students.filter(new Predicate<Student>() {
        public boolean op(Student s) {
            return s.getGradYear() == 2011;
        }
    }).map(new Mapper<Student,Double>() {
        public Double extract(Student s) {
            return s.getScore();
        }
    }).max();
```

---

# Functional Collection Patterns

---

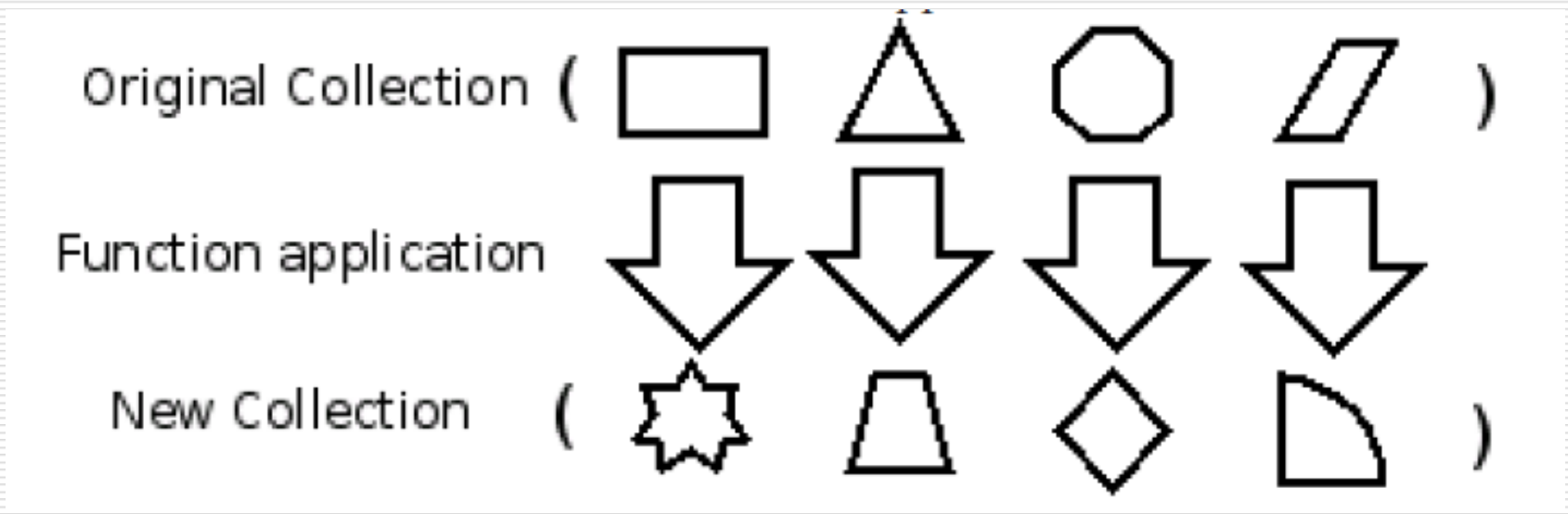
- `map()`
  - `filter()`
  - `reduce()`
  - `forEach()`
-

# Map

---

The Map pattern evaluates a high-order function on all elements of the collection.

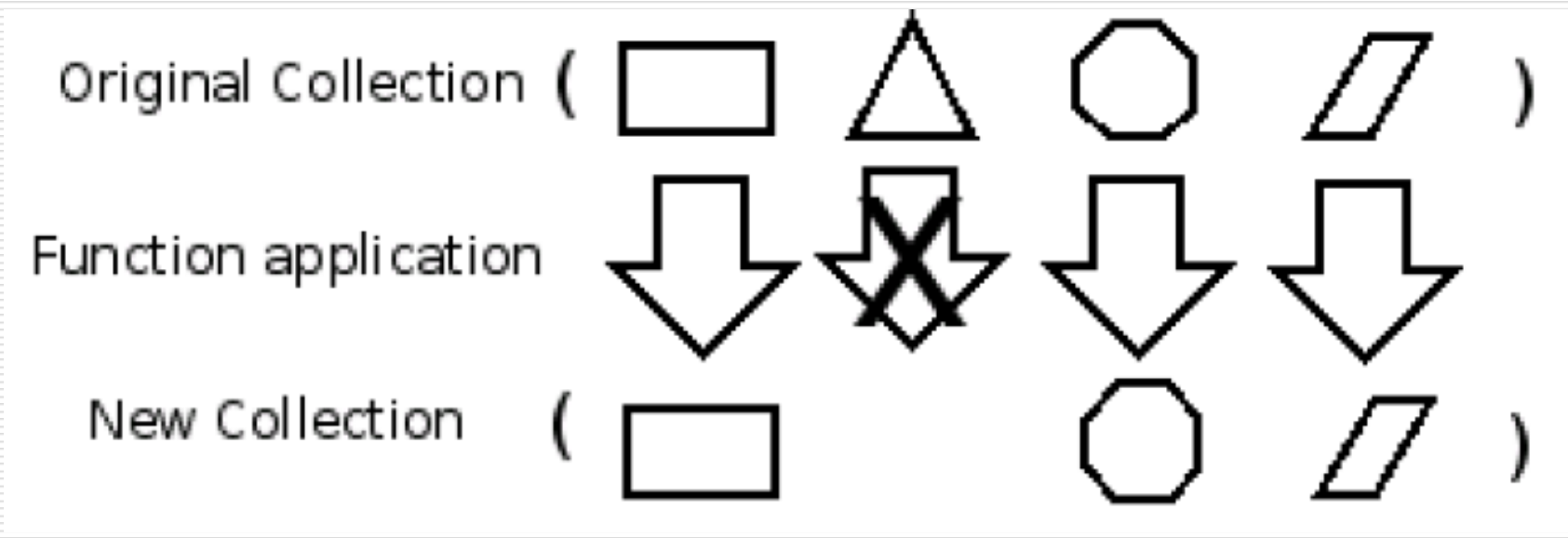
It returns a new collection with the results of each function application.



# Filter

---

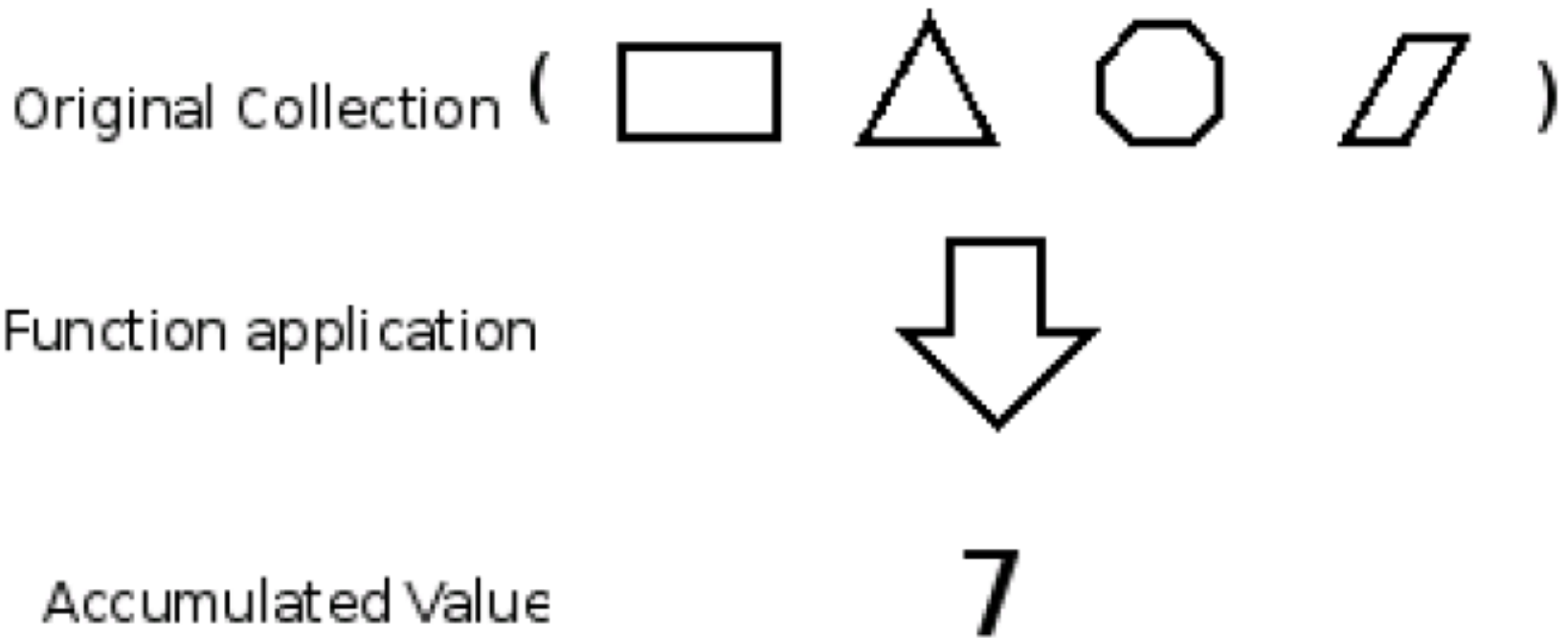
The Filter pattern evaluates a predicate (a function which returns a Boolean) on each of the elements, returning a new collection which is subset of the original collection.



# 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 *Defender 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();
```

---

# 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;  
};
```

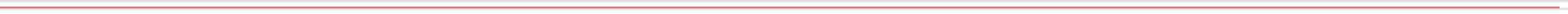
```
highestScore =  
    students.filter((Student s) -> s.getGradYear() == 2010)  
        .map((Student s) -> s.getScore())  
        .reduce(0.0, op);  
//.reduce(0.0, (Double left, Double right) ->  
//    (left > right) ? left : right );
```

---



Зачем это все?

Что кроме удобного синтаксиса?



# Цель — автоматическое распараллеливание операций

---

```
List<Student> students = new ArrayList<>(...);
```

```
...
```

```
double highestScore =
```

```
    students.parallel()
```

```
        .filter(s -> s.getGradYear() == 2011)
```

```
        .map(s -> s.getScore())
```

```
        .max();
```

---

# Что почитать

---

- JDK 8 - <http://jdk8.java.net/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>
  - Defender Methods
    - <http://cr.openjdk.java.net/~briangoetz/lambda/Defender%20Methods%20v3.pdf>
  - State of the Lambda
    - <http://cr.openjdk.java.net/~briangoetz/lambda/lambda-state-3.html>
-