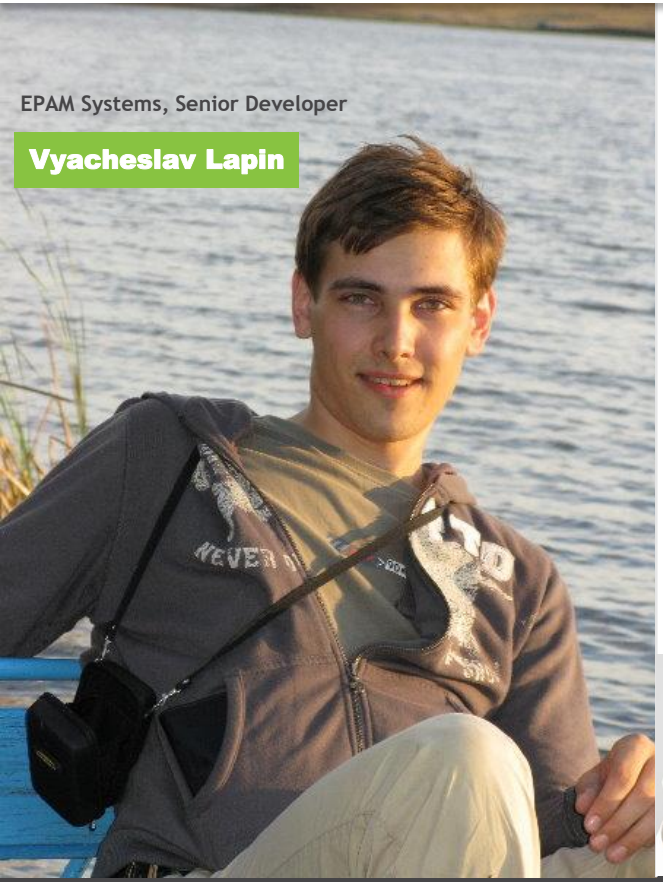
Jpoint Student Day

<epam>

Functional Reactive full-stack development in Java&JS

March 27, 2016

Vyacheslav Lapin



- 10+ years experience in IT
- 7+ Java-development experience
- 5+ trainer experience
- 3+ system analysis

Interests:

- Messaging (Nanomsg, Kafka)
- Functional programming (Clojure, Scala)

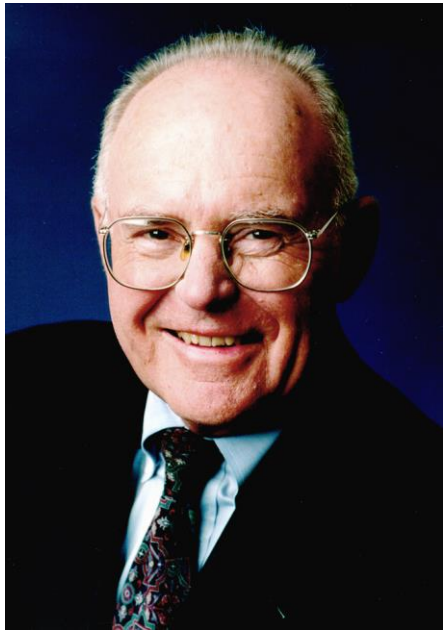
Agenda

1. **FRP**
   - Проблема
   - Развенчание мифов о функциональном программировании
   - Анекдоты
2. **FRP в Java**
   - Optional, Stream`ы, CompletableFeature
   - RxJava, дружба с Apache Camel`ом
   - Speedment
3. **Вести с фронта…**
   - RxJS

Part 1

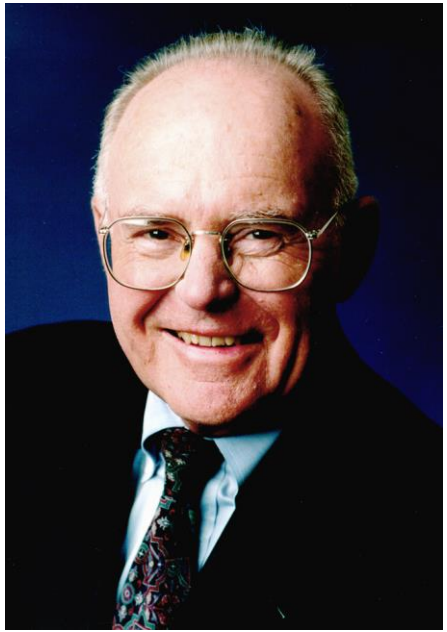
FUNCTIONAL **REACTIVE PROGRAMMING**

Functional Reactive Programming – Проблема



Гав? Мур? Иа? Ку-ка-ре-куу?

Functional Reactive Programming – Проблема



Гордон Эрл Мур

«…количество транзисторов, размещаемых на кристалле интегральной схемы, удваивается каждые 24 месяца[1]»

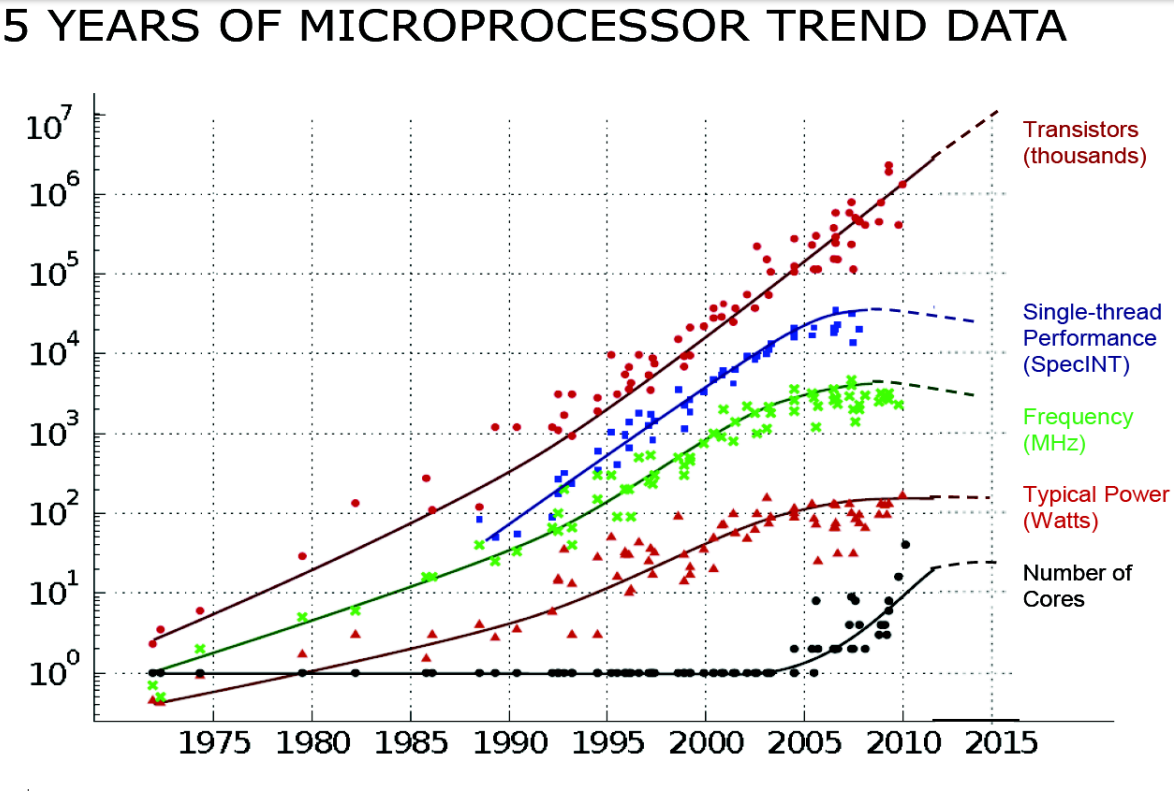
Закон Мура

[1] **Давид Хаус** из Intel: «т.к. растёт ещё и быстродействие транзисторов, *производительность* процессоров должна удваиваться каждые *18 месяцев*»

Functional Reactive Programming – Проблема

«The free lunch is over»

Herb Sutter
http://www.gotw.ca/publications/concurrency-ddj.htm
(Ru: https://habrahabr.ru/post/145432/)



Functional Reactive Programming – Проблема

Джин Амдал

«В случае, когда задача разделяется на несколько частей, суммарное время её выполнения на параллельной системе не может быть меньше времени выполнения самого длинного фрагмента, при условии одинаковой скорости всех вычислителей.»

Functional Reactive Programming – Проблема

Джин Амдал

$$S_p = \frac{1}{\alpha + \frac{1-\alpha}{p}}$$

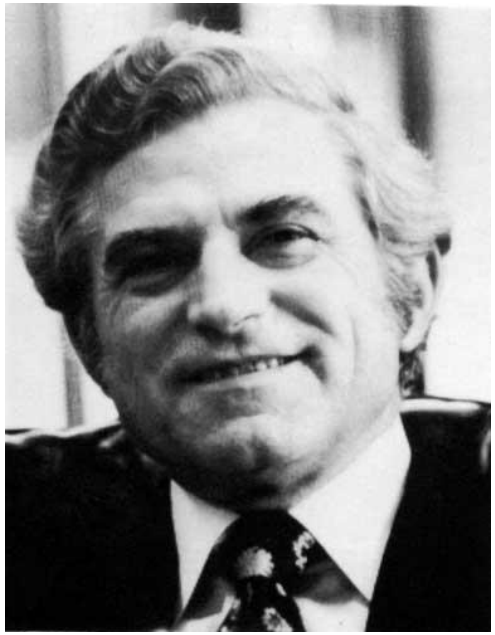
p - число задействованных независимых процессоров

α - последовательная доля рассчётов

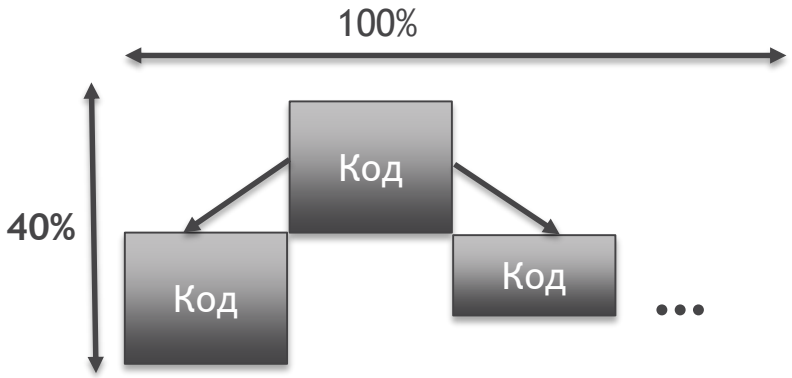
$1-\alpha$ - распараллеливаемая доля рассчётов

Максимально-возможное ускорение, которое может быть получено на вычислительной системе из p процессоров, по сравнению с однопроцессорным решением

Functional Reactive Programming – Проблема



Джин Амдал



Один процессор: X

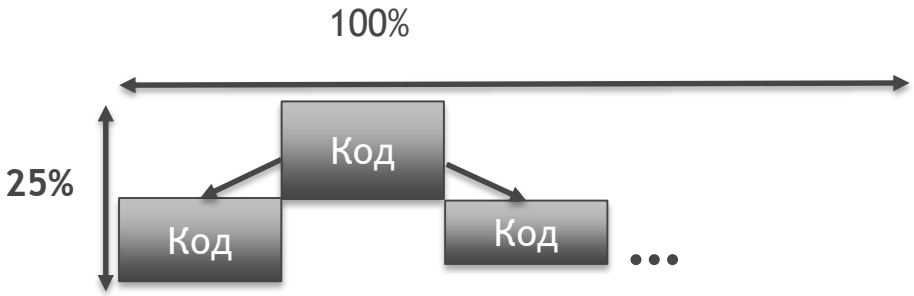
10 процессоров: **2.174** * X

100 процессоров: **2.263** * X

1000 процессоров: **2.496** * X

Functional Reactive Programming – Проблема

Джин Амдал



Один процессор: X

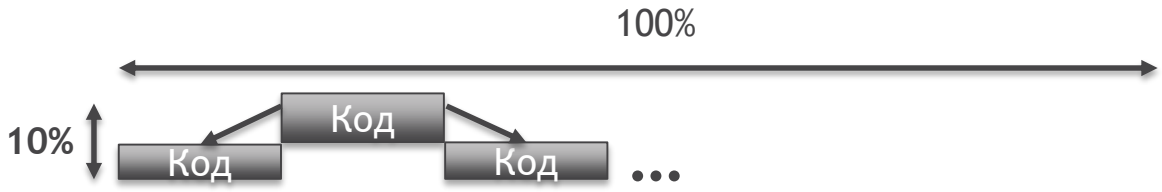
10 процессоров: **3.077** * X

100 процессоров: **3.883** * X

1000 процессоров: **3.988** * X

Functional Reactive Programming – Проблема

Джин Амдал



Один процессор: X

10 процессоров: **5.263** * X

100 процессоров: **9.174** * X

1000 процессоров: **9.910** * X

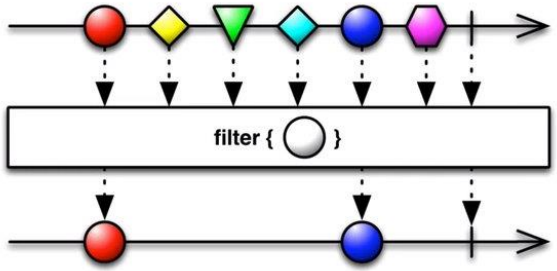
Functional Reactive Programming – Определения

Functional programming

- Осмысление предметной области в терминах неизменяемых значений и функций, которые их преобразуют.

Reactive Programming

- Программирование асинхронной обработки и посылки данных
- Observer и EDA (Event-Driven Architecture)
- Back-pressure: «Wow, wow, wow!.. НЕ ТАК БЫСТРО!!!»



Functional Reactive Programming – Определения

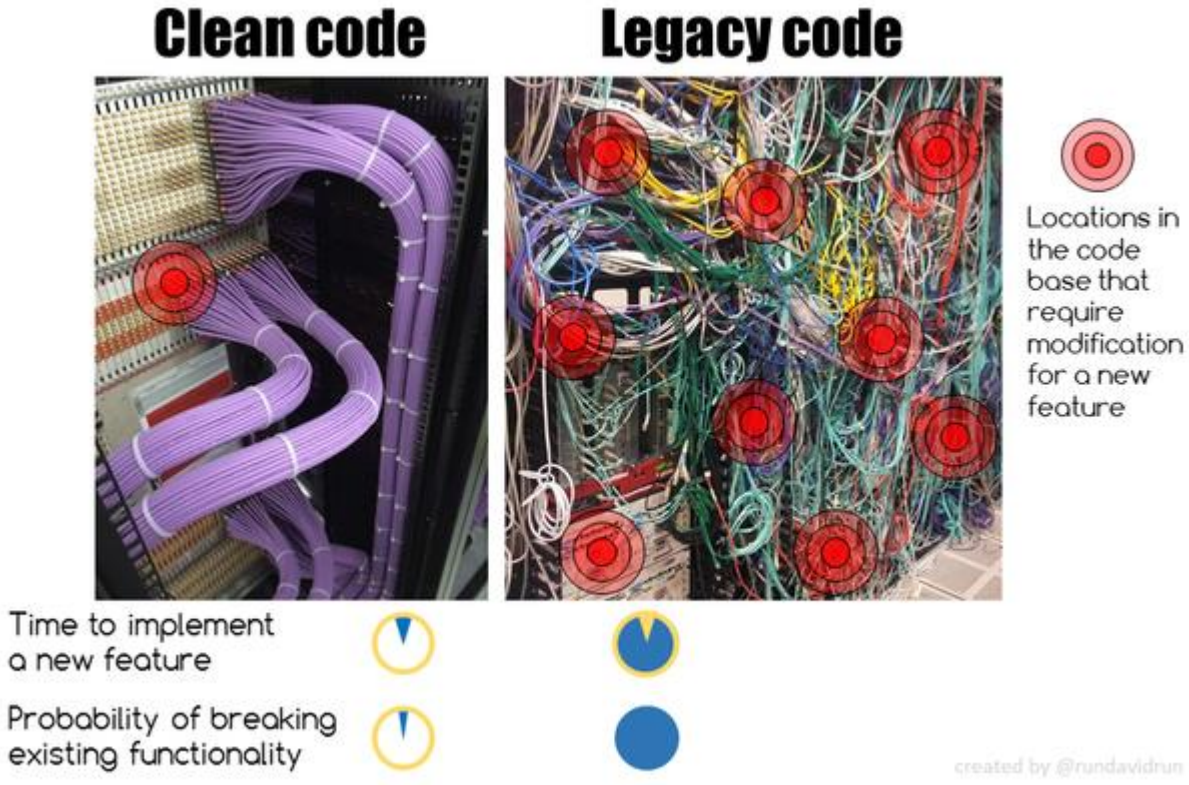
Приемущества

- Просто тестировать

Functional Reactive Programming – Определения

Приемущества

- Просто тестировать
- **Меньше зависимостей - проще читать.**



Functional Reactive Programming – Определения

Приемущества

- Просто тестировать
- Меньше зависимостей - проще читать.
- **Меньше ошибок**

Functional Reactive Programming – Определения

Приемущества

- Просто тестировать
- Меньше зависимостей - проще читать.
- Меньше ошибок

Недостатки

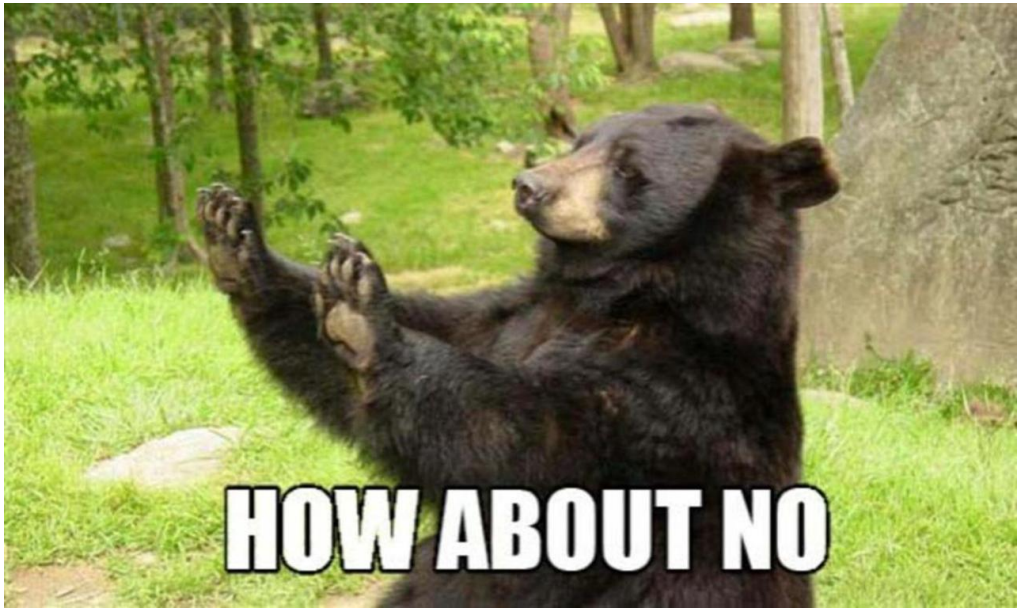
- **Дольше (интереснее и сложнее) писать**

Functional Reactive Programming - Мифы

1. Функциональное программирование способен понять далеко не каждый…

2. Писать в функциональном стиле можно лишь на **true-функциональных** языках!..

- Lisp
  - Clojure
- Haskell
- Erlang
- Scala
- …



Functional Reactive Programming - Мифы

3. Чтобы писать в функциональном стиле, обязательно изучить и использовать:

Моноиды

Функторы

Ленивые вычисления

Каррирование

Pattern matching

Монады

Map/reduce /fold/merge

Functional Reactive Programming – Особенности

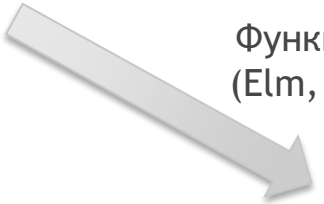
Ленивые вычисления

```
b = 1;
c = 2;
a = b + c;
b = 3;
print(a); //?
```

Императивный подход
(Java, C, C++, C#, JS)

3

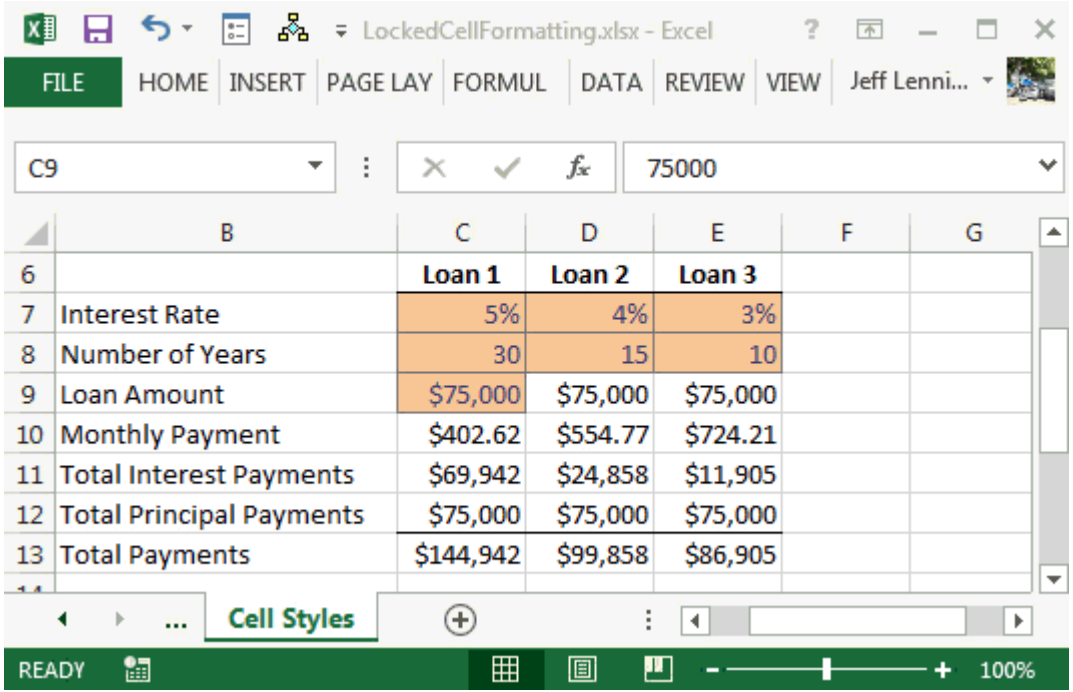
Функциональный подход
(Elm, Lisp, Haskel, Erlang)



5

Functional Reactive Programming – Особенности

Ленивые вычисления



	B	C	D	E	F	G
6		Loan 1	Loan 2	Loan 3		
7	Interest Rate	5%	4%	3%		
8	Number of Years	30	15	10		
9	Loan Amount	$75,000	$75,000	$75,000		
10	Monthly Payment	$402.62	$554.77	$724.21		
11	Total Interest Payments	$69,942	$24,858	$11,905		
12	Total Principal Payments	$75,000	$75,000	$75,000		
13	Total Payments	$144,942	$99,858	$86,905		

Functional Reactive Programming – Особенности

Ленивые вычисления

Сколько будет 5 + 2 ?

7, профессор!

Похвально!

```
@Test
public void
CorrectCalculateFivePlusTwo() {
    assertThat(5.plus(2), is(7));
}
```

OK CorrectCalculateFivePlusTwo

Functional Reactive Programming – Особенности

Ленивые вычисления

А сколько будет 5 + 3 ?

8, профессор!

Ставлю «неуд»!

Functional Reactive Programming – Особенности

Ленивые вычисления

А сколько?..

10

Кааааак???

Потому что на предыдущем шаге «5» мы уже (`5.plus(2)`) превратили в «7»!

```
@Test
public void CorrectCalculateFivePlusThree() {
    assertThat(5.plus(2).plus(3), is(8));
}
```

CorrectCalculateFivePlusThree

Functional Reactive Programming – Особенности

ПЕРЕМЕННЫЕ

Императивный (ООП) подход

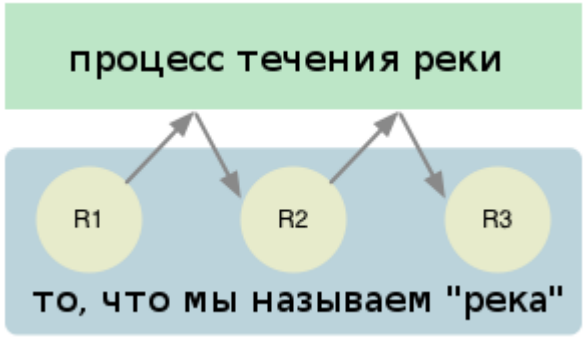
```
@Mutable
public class Река {

    private long state;

    public long getState() {
        return state;
    }

    public void setState(long state) {
        this.state = state;
    }
}
```

Что есть река?

Функциональный подход

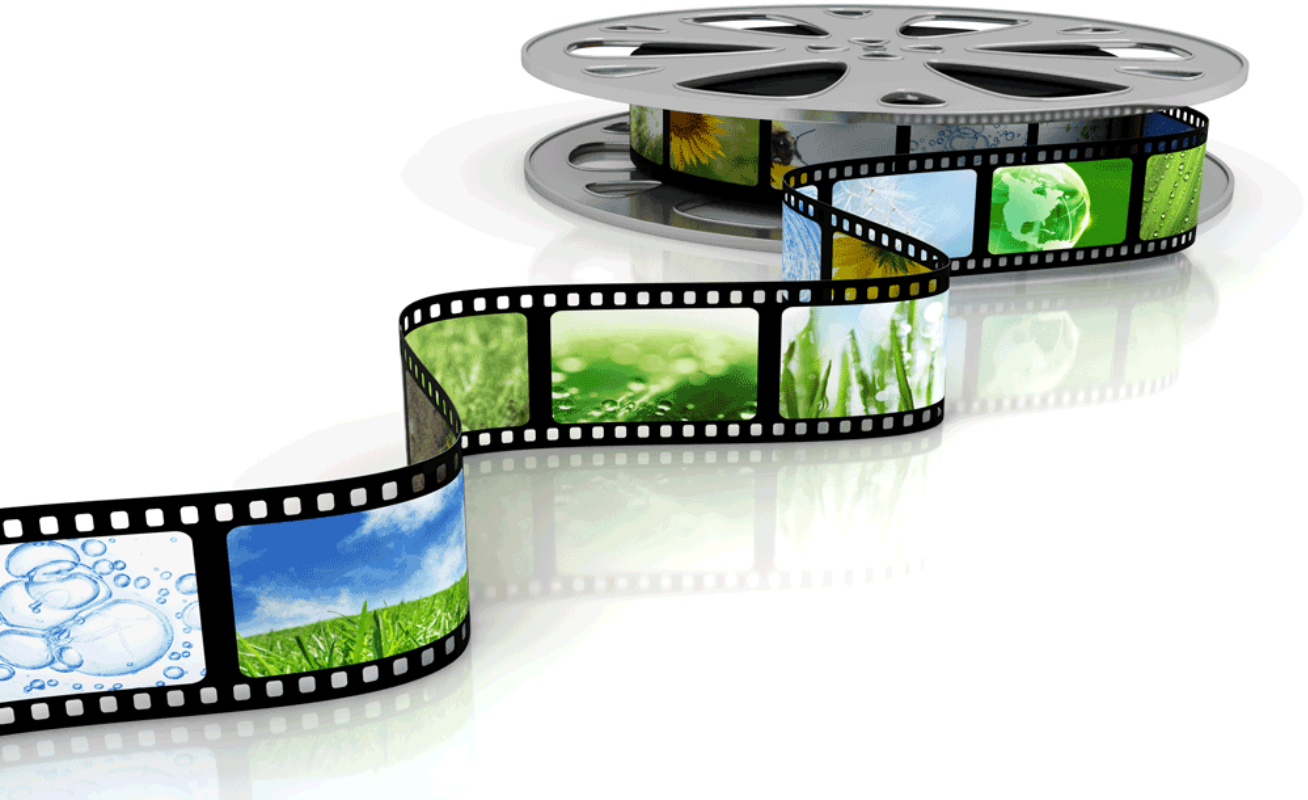
«Дважды не войти в одну и ту же реку, ибо притекает другая вода.»

Гераклит Эфесский,
544—483 гг. до н. э.



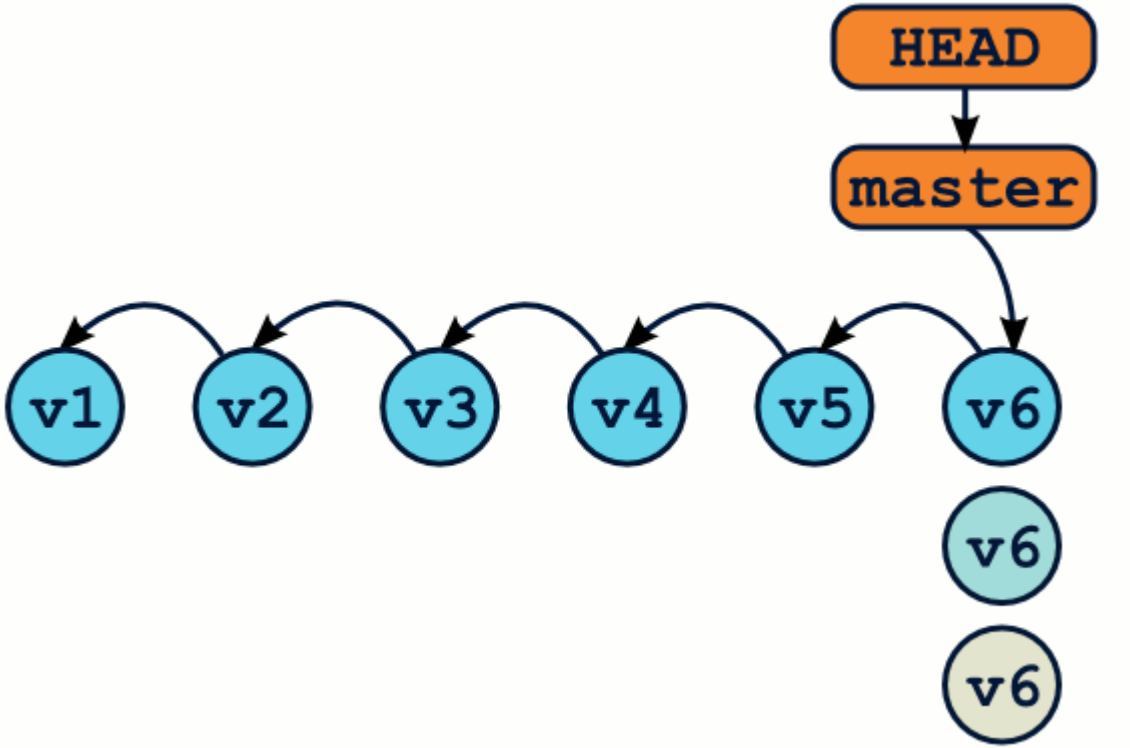
Functional Reactive Programming – Особенности

ПЕРЕМЕННЫЕ



Functional Reactive Programming – Особенности

ПЕРЕМЕННЫЕ



Functional Reactive Programming – Особенности

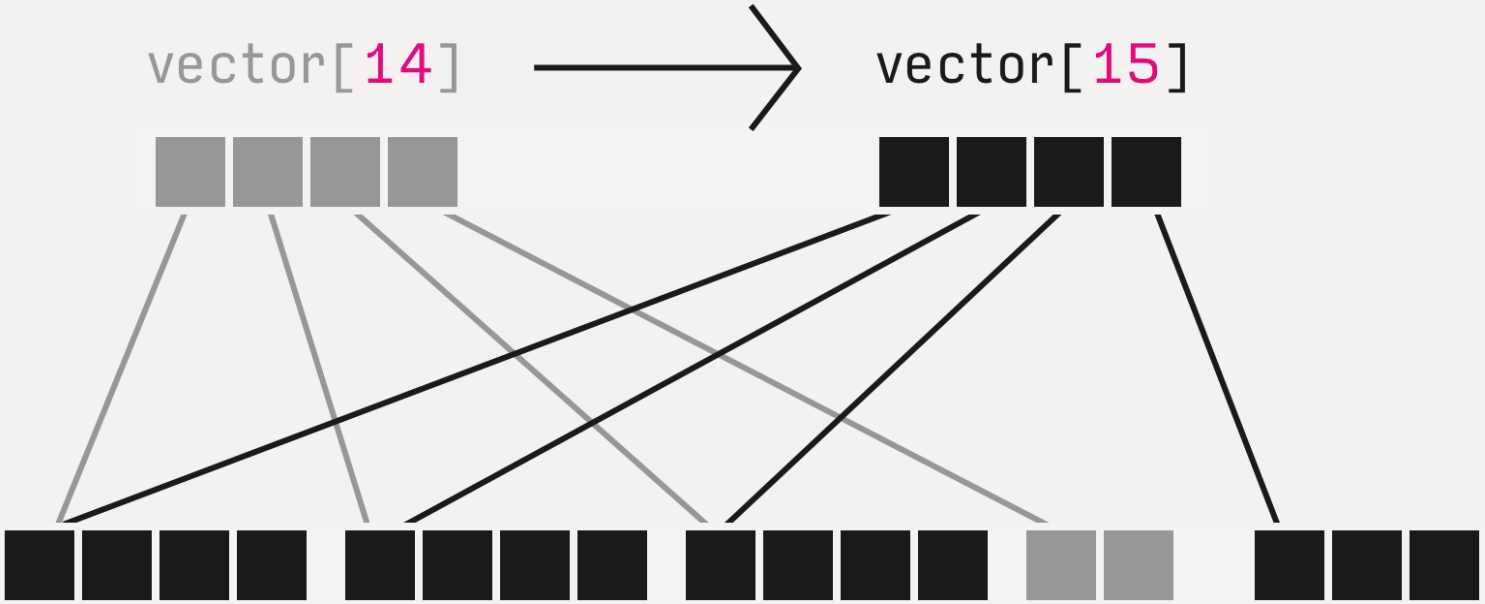
ПЕРЕМЕННЫЕ

Как, совсем без переменных???

```
public static int factorial(final int n) {
    if (n == 0) {
        return 1;
    } else {
        return n * factorial(n - 1);
    }
}
```

Functional Reactive Programming – Особенности

ПЕРЕМЕННЫЕ



Functional Reactive Programming – Особенности

ЧИСТАЯ ФУНКЦИЯ

- Детерминированная
- Нет side-effect`ов

Преимущества:

- Кеширование
- Удалённые вызовы
- Параллельность выполнения
- Тестируемость

Functional Reactive Programming – Особенности

ЧИСТАЯ ФУНКЦИЯ

```
public static int square(int x) {
    updateXToDb(x);
    return x * x;
}

public static void squareAll(int... items) {
    for (int i = 0; i < items.length; i++) {
        items[i] = square(i);
    }
}
```

Functional Reactive Programming – Особенности

ЧИСТАЯ ФУНКЦИЯ

```
public static int square(int x) {
    return x * x;
}

public static int[] squareAll(int... items) {
    return Arrays.stream(items)
            .map(operand -> square(operand))
            .toArray();
}
```

Functional Reactive Programming – Особенности

ЧИСТАЯ ФУНКЦИЯ

```
public Program getCurrentProgram(TvGuide tvGuide, Chennel
channel) {

    Schedule schedule = tvGuide.getSchedule(channel);
    return schedule.programAt(new Date());
}
```

Functional Reactive Programming – Особенности

ЧИСТАЯ ФУНКЦИЯ

```
public Program getCurrentProgram(TvGuide tvGuide, Chennel
channel, Date time) {

    Schedule schedule = tvGuide.getSchedule(channel);
    return schedule.programAt(time);
}
```

Functional Reactive Programming – Особенности

АСИНХРОННОСТЬ

Синхронность	Асинхронность
Взаимодействие	Воздействие
Запрос	Подписка на изменения
Ответ	Получение уведомления
Генерация ID в БД	INSERT со сгенерированным ID
HTTP POST	HTTP GET

Part 2

BACK-END

Java SE 8 java.util.Optional

- Какой самый распространённый `Exception` в Java?

Java SE 8 java.util.Optional

- Какой самый распространённый `Exception` в Java?
  - `NullPointerException`
- Как мы от него защищаемся?

Java SE 8 java.util.Optional

- Какой самый распространённый `Exception` в Java?
  - `NullPointerException`
- Как мы от него защищаемся?
  - Проверки на `null`…

Java SE 8 java.util.Optional

```
public String getUsersCityName() {
    User user = findUser();
    if (user != null) {
        Address address = user.getAddress();
        if (address != null) {
            String zipCode = address.getZipCode();
            if (zipCode != null) {
                City city = findCityByZipCode(zipCode);
                if (city != null)
                    return city.getName();}}}
    throw new RuntimeException();
}
```

Java SE 8 java.util.Optional

- Либо `null`, либо какой-то объект
- Получение:
  - `Optional<Integer> integer = Optional.of(5);`
  - `Optional<Object> empty = Optional.empty();`
  - `Optional<Object> o = Optional.ofNullable(nullableFn());`

Back-end - Java SE 8 java.util.Optional

СЦЕНАРИИ ИСПОЛЬЗОВАНИЯ БЕЗ ВОЗВРАТА ЗНАЧЕНИЯ

- Примитивный сценарий:

```
Integer integer = getIntOptional().get();
```

СЦЕНАРИИ ИСПОЛЬЗОВАНИЯ БЕЗ ВОЗВРАТА ЗНАЧЕНИЯ

- Сценарий получше:

```
Optional<Integer> optionalInt = getOptionalInt();
if (optionalInt.isPresent()) {
    Integer integer = optionalInt.get();
    // do something with integer variable
}
```

Back-end - Java SE 8 java.util.Optional

СЦЕНАРИИ ИСПОЛЬЗОВАНИЯ БЕЗ ВОЗВРАТА ЗНАЧЕНИЯ

- Ещё лучше:

```
getOptionalInt().ifPresent(integer ->
        // do something with integer variable
);
```

ВАРИАНТЫ СЦЕНАРИЕВ С ВОЗВРАТОМ ЗНАЧЕНИЯ

- Со значением: по умолчанию

```
Integer integer = getOptionalInt().orElse(0);

Integer integer = getOptionalInt().orElseGet(
  new Supplier<Integer>() {
    @Override
    public Integer get() {
      return 0;
    }
  });
```

Back-end - Java SE 8 java.util.Optional

ВАРИАНТЫ СЦЕНАРИЕВ С ВОЗВРАТОМ ЗНАЧЕНИЯ

- Со значением: по умолчанию

```
Integer integer = getOptionalInt().orElse(0);

Integer integer = getOptionalInt().orElseGet(() -> 0);
```

ВАРИАНТЫ СЦЕНАРИЕВ С ВОЗВРАТОМ ЗНАЧЕНИЯ

- С отсечением:

```
Integer integer = getOptionalInt().orElseThrow(
        () -> new RuntimeException("{some msg...}"));

// do something with integer variable
```

ВАРИАНТЫ СЦЕНАРИЕВ С ВОЗВРАТОМ ЗНАЧЕНИЯ

- С преобразованием и значением по умолчанию:

```
String s = getOptionalInt()
        .map(Integer::toHexString)
        .orElse("NaN");

// do something with s variable
```

ВАРИАНТЫ СЦЕНАРИЕВ С ВОЗВРАТОМ ЗНАЧЕНИЯ

- С фильтрацией, преобразованием и значением по умолчанию:

```
String s = getOptionalInt()
        .filter(integer -> integer > 0 && integer < 100)
        .map(Integer::toHexString)
        .orElse("NaN");

// do something with s variable
```

ВАРИАНТЫ СЦЕНАРИЕВ С ВОЗВРАТОМ ЗНАЧЕНИЯ

- С фильтрацией, преобразованием и значением по умолчанию:

```
String s = getOptionalInt()
        .filter(integer -> integer > 0 && integer < 100)
        .map(Integer::toHexString)
        .flatMap(s1 -> getOptionalString().map(s2 -> s1
+ s1))
        .orElse("NaN");

// do something with s variable
```

Back-end - Java SE 8 java.util.Optional

ВАРИАНТЫ СЦЕНАРИЕВ С ВОЗВРАТОМ ЗНАЧЕНИЯ

- Считать сразу два свойства из property-файла и вывести рез-т на консоль лишь в случае, если они оба заданы:

```
Properties properties = new Properties();
properties.load(new FileInputStream("props.properties"));

ofNullable(properties.getProperty("key"))
        .flatMap(key -> ofNullable(properties.getProperty("value"))
                .map(value -> key + " = " + value))
        .ifPresent(System.out::println);
```

Функциональные интерфейсы

- Только один абстрактный (без реализации) метод
  - => можно реализовывать при помощи лямбд
- Для самоконтроля полезно ставить аннотацию `@FunctionalInterface`
- Есть достаточно много предопределённых
- Почти все дублируются для типов `int`, `double`, `long`

Функциональные интерфейсы

- `Runnable { void run(); }`

Функциональные интерфейсы

- `Callable<V> { V call() throws Exception; }`
  - `Supplier<T> { T get(); }`

Функциональные интерфейсы

- `Consumer<T> { void accept(T t); }`
  - `BiConsumer<T, U> { void accept(T t, U u); }`

Функциональные интерфейсы

- `Function<T, R> { R apply(T t); }`
  - `UnaryOperator<T> extends Function<T, T>`
  - `BiFunction<T, U, R> { R apply(T t, U u); }`
    - `BinaryOperator<T> extends BiFunction<T,T,T>`
  - `Predicate<T> { boolean test(T t); }`
    - `BiPredicate<T, U> { boolean test(T t, U u); }`

Back-end - Java SE 8 Stream API

`java.util.stream.Stream<T>`

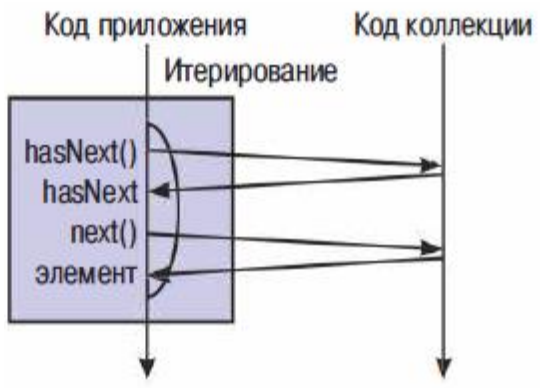
- Абстракция потока данных
- «Внутреннее» итерирование
- «Декларативная параллельность»

Back-end - Java SE 8 Stream API

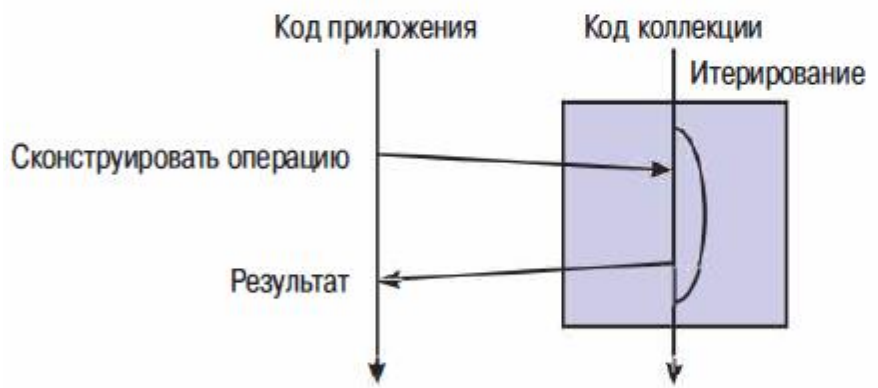
java.util.stream.Stream<T>

```
ArrayDeque<Artist> allArtists;
```

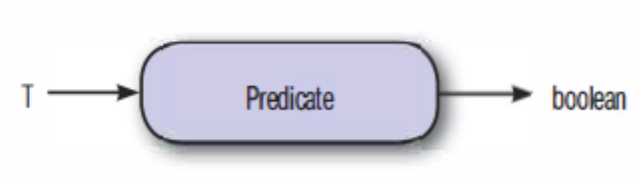
```
long count = 0L;
for (Artist artist : allArtists) {
    if (artist.isFrom("Moscow")) {
        count++;
    }
}
```

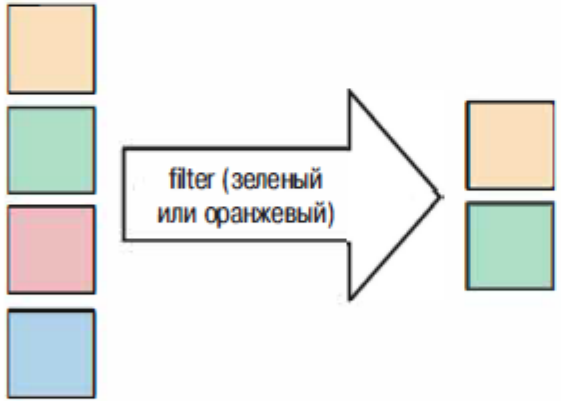



```
long count = allArtists.stream()
        .parallel()
        .filter(artist ->
            artist.isFrom("Moscow"))
        .count();
```



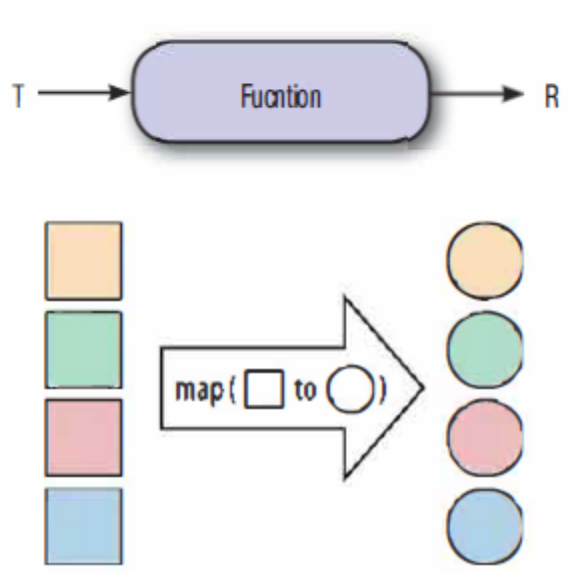
filter



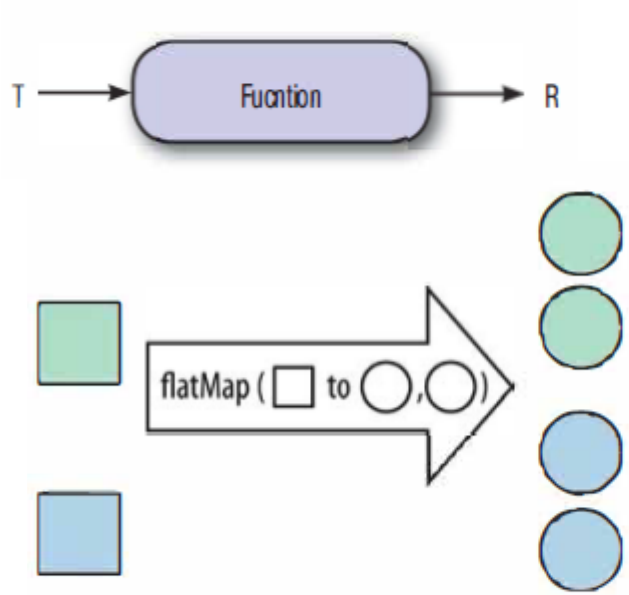


Back-end - Java SE 8 Stream API

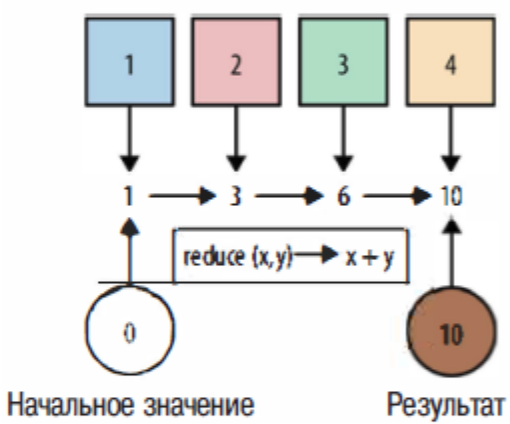
map



flatMap



reduce

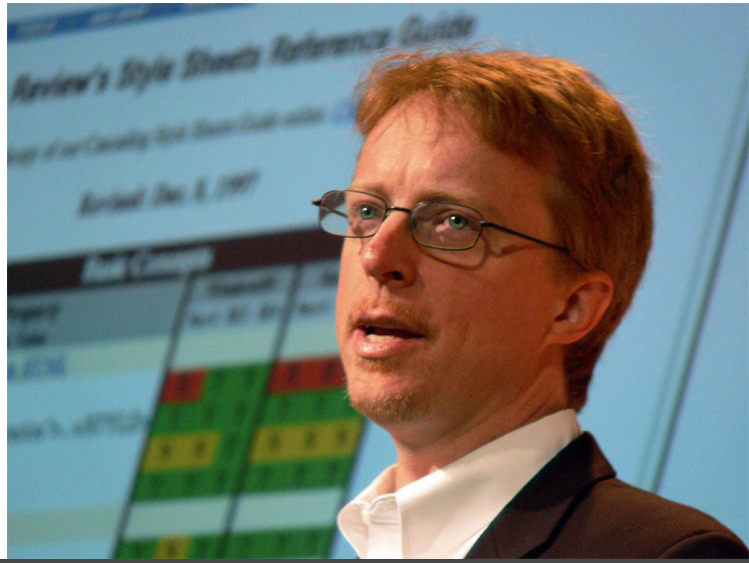


А как же интеграция?

Rx Basics

- ReactiveX – это комбинация Observer pattern, Iterator pattern, functional programming
- Вдохновлена «reactive manifesto» (http://www.reactivemanifesto.org):
  - Отзывчивые (responsive)
  - Отказоустойчивые (resilient)
  - Масштабируемые (elastic)
  - Ориентированные на события (message-driven) системы

Эрик Мейер



Rx basics

- `Subscriber` (*Observer*) подписывается на `Observable`
- `Observable` вызывает методы `Subscriber`а:
  - `onNext(T)` - доступен следующий элемент
  - `onError(Throwable)` - произошла ошибка
  - `onCompleted()` - поток удачно завершился
- `Subscriber` обрабатывает соответствующие вызовы.
- Контракт - порядок вызова методов должен быть следующий:
  - `onNext(T)*, onError(Throwable) | onCompleted()`

Rx basics

- Простейший способ создания `Observable`а:
  - `Observable.just(5);`

Rx basics

- Подписываемся и обрабатываем входящие значения:

```
Observable.just(5)
          .subscribe(integer ->
  System.out.println(integer));
```

Rx basics

- Подписываемся и обрабатываем входящие значения:

```
– Observable.just(5)
            .subscribe(System.out::println,
                       throwable ->
                               System.err.println(throwable));
```

Rx basics

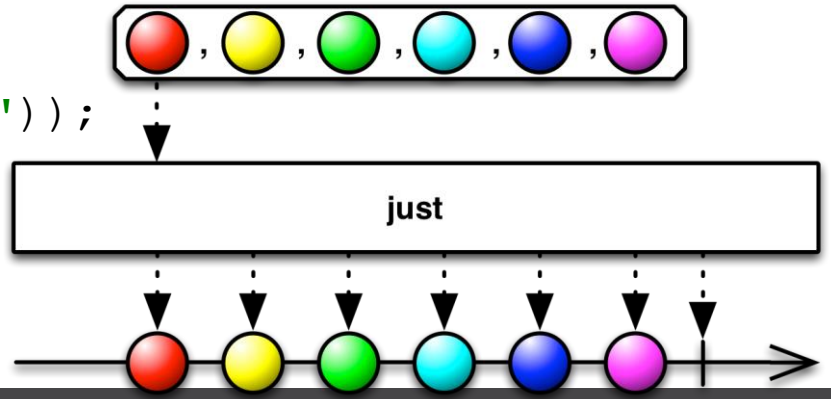
- Подписываемся и обрабатываем входящие значения:

```
– Observable.just(5)
            .subscribe(System.out::println,
                       System.err::println,
                       () ->
  System.out.println("Completed!"));
```

Rx basics

- Подписываемся и обрабатываем входящие значения:

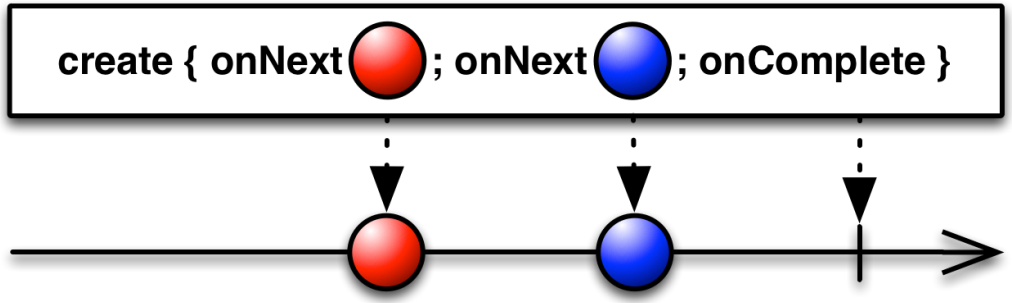
```
Observable.just(5, 4, 3, 2, 1)
          .subscribe(System.out::println,
                     System.err::println,
                     () ->
  System.out.println("Completed!"));
```



Rx basics

- Создаём `Observable` из вызовов соотв. методов `Subscriber`а:

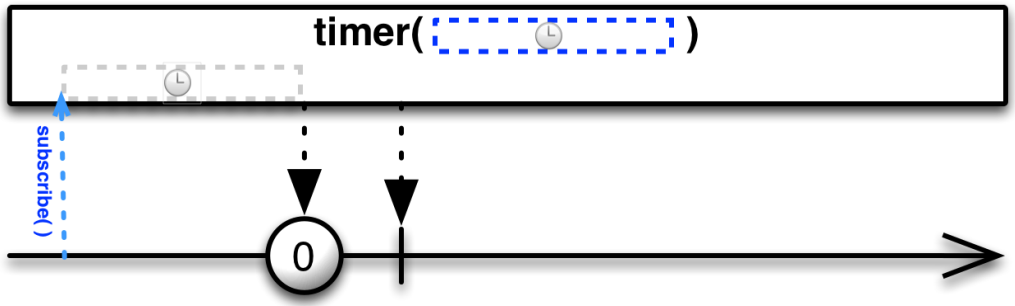
```
– Observable.create(subscriber -> {
        subscriber.onNext(5);
        subscriber.onNext(4);
        subscriber.onNext(3);
        subscriber.onNext(2);
        subscriber.onNext(1);
        subscriber.onCompleted();
  }).subscribe(System.out::println,
               System.err::println,
               () -> System.out.println("Completed!"));
```



Rx basics

- Создаём `Observable` из вызовов соотв. методов `Subscriber`а:

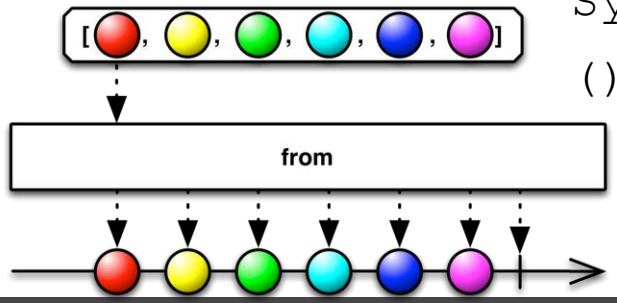
```
– Observable.timer(15, TimeUnit.SECONDS)
            .subscribe(System.out::println,
                       System.err::println,
                       () -> System.out.println("Completed!"));
```



Rx basics

Создаём `Observable` из вызовов соотв. методов `Subscriber`а:

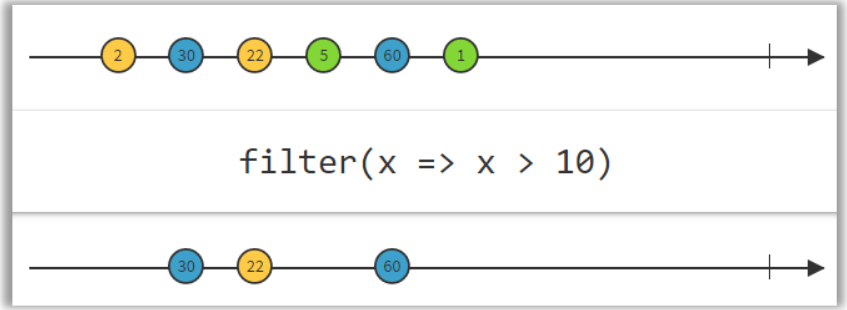
```
final Iterable<Integer> integers = Arrays.asList(5, 4, 3,
2, 1);
Observable.from(integers)
        .subscribe(System.out::println,
                System.err::println,
                () -> System.out.println("Completed!"));
```

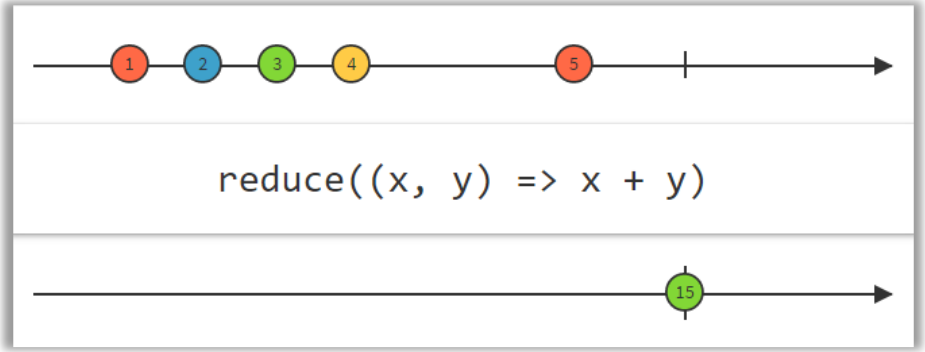


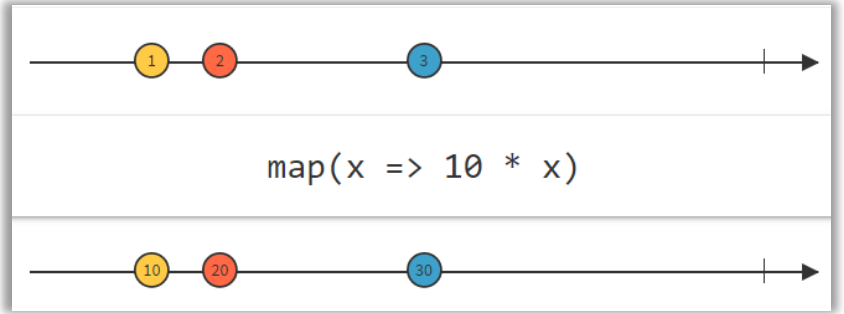
Back-end – marble diagrams

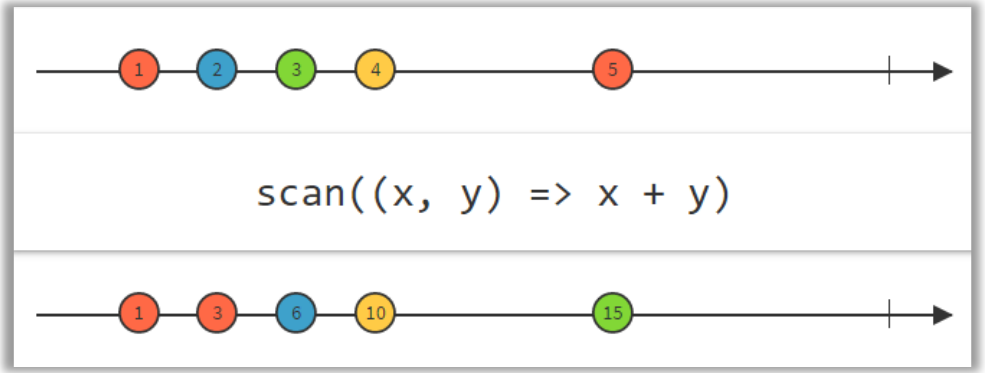
http://rxmarbles.com

Back-end

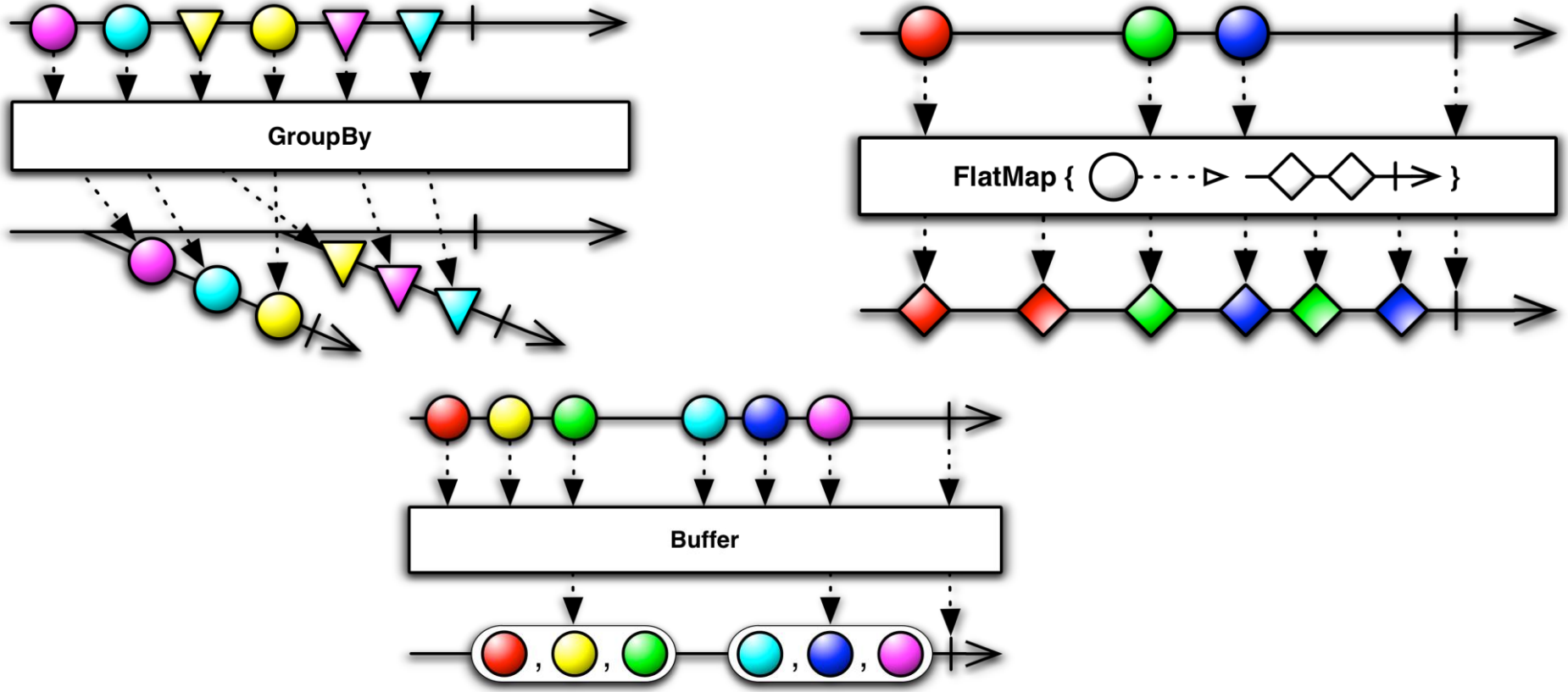

2
30
22
5
60
1
filter(x => x > 10)
30
22
60


1
2
3
4
5
reduce((x, y) => x + y)
15


1
2
3
map(x => 10 * x)
10
20
30


1
2
3
4
5
scan((x, y) => x + y)
1
3
6
10
15

Back-end



А как же существующие интеграционные решения?

Основы Apache Camel

- Маршруты (`Routes`) можно конфигурировать
  - статически (например, в файле Spring-контекста)
  - во время работы приложения.

Основы Apache Camel

- По маршрутам ходят караваны сообщений, попутно попадая в различные
  - обработчики,
  - конверторы,
  - агрегаторы,
  - прочие трансформеры

Основы Apache Camel


```
public static void main(String... args) throws Exception {
    CamelContext context = Route.set(
            FILE_FROM_URI,
            fromRoute -> fromRoute
                    .process(CamelExample::process)
                    .to(FILE_TO_URI));

    context.start();
    Thread.sleep(10_000);
    context.stop();
}

private static void process(Exchange exchange) {
    System.out.println(
            "Найден и загружен файл: " +
                    exchange.getIn().getHeaders().entrySet().stream()
                            .map(Object::toString)
                            .collect(Collectors.joining("\n")));}
```

Camel RX


```
ReactiveCamel rx = new ReactiveCamel(context);
Observable<Order> observable = rx.toObservable("seda:orders",
Order.class);

Observable<String> largeOrderIds = observable
        .filter(order -> order.getAmount() > 100.0)
        .map(Order::getId);
```

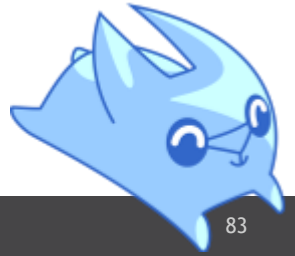
http://camel.apache.org/rx

Back-end - Speedment

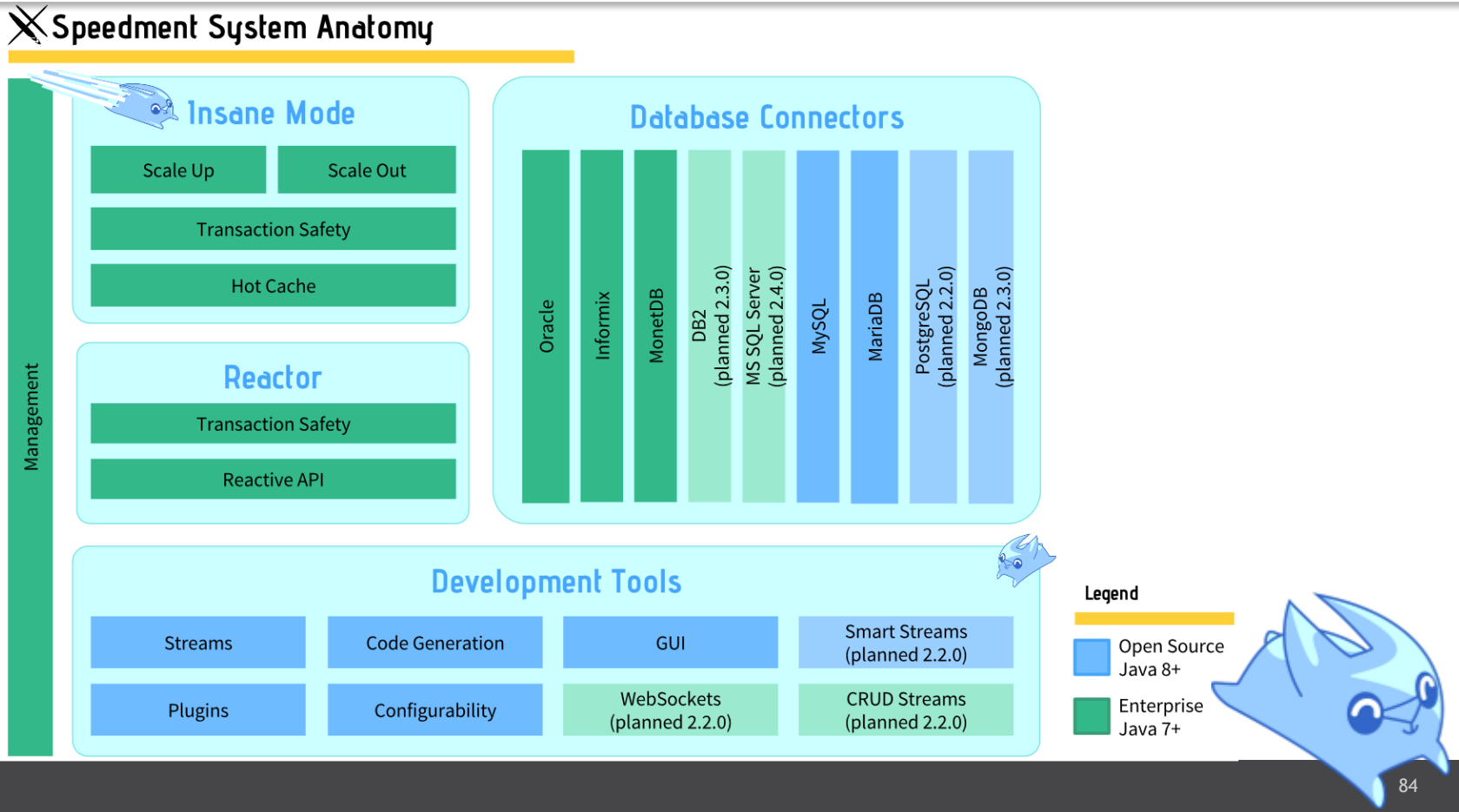
```
mysql> explain hare;
+-------+-------------+------+-----+---------+----------------+
| Field | Type        | Null | Key | Default | Extra          |
+-------+-------------+------+-----+---------+----------------+
| id    | int(11)     | NO   | PRI | NULL    | auto_increment |
| name  | varchar(45) | NO   |     | NULL    |                |
| color | varchar(45) | NO   |     | NULL    |                |
| age   | int(11)     | NO   |     | NULL    |                |
+-------+-------------+------+-----+---------+----------------+
4 rows in set (0.01 sec)
```

```
List<Hare> oldHares = hares.stream()
        .filter(AGE.greaterThan(8))
        .collect(toList());
```

```
long noOldHares = hares.stream()
        .filter(AGE.greaterThan(8))
        .mapToInt(Hare::getAge)
        .sorted()
        .count();
```



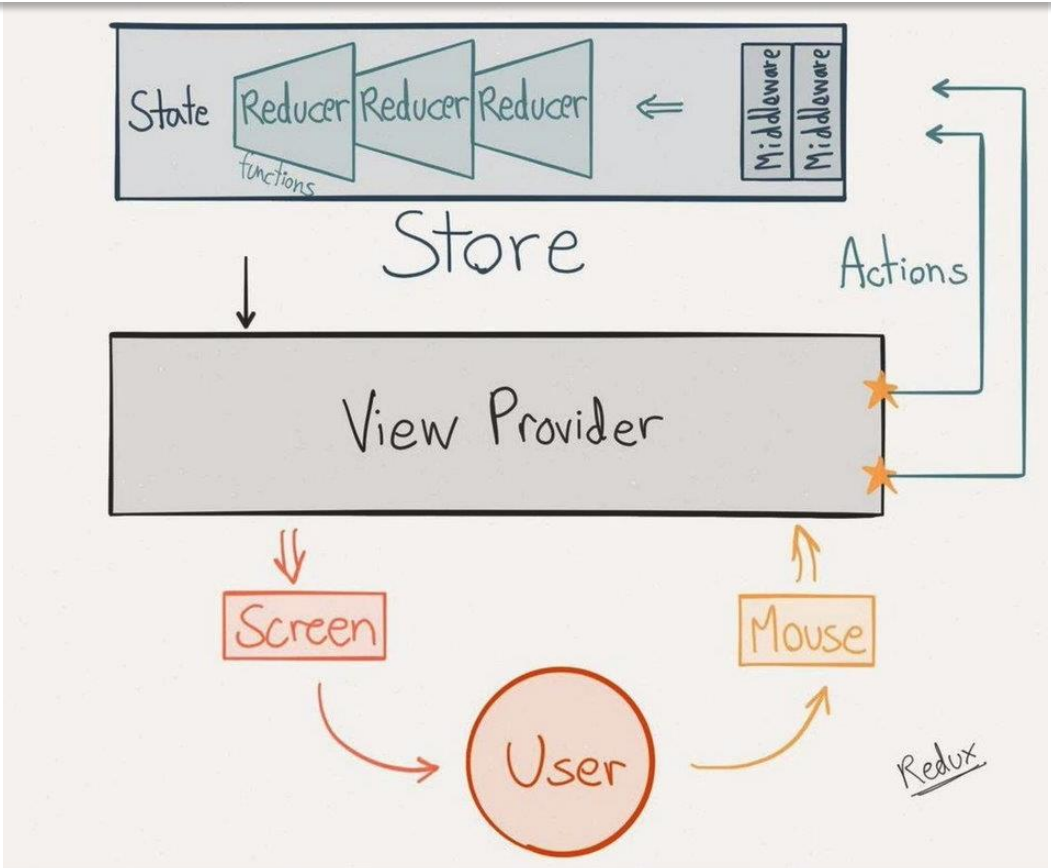
Back-end - Speedment



Part 3

FRONT-END

Functional Reactive Programming – Redux

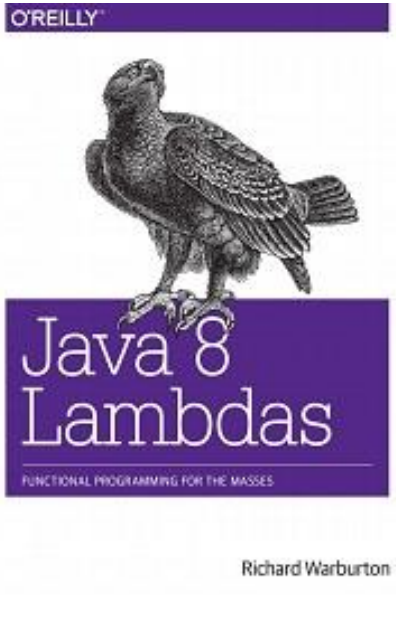


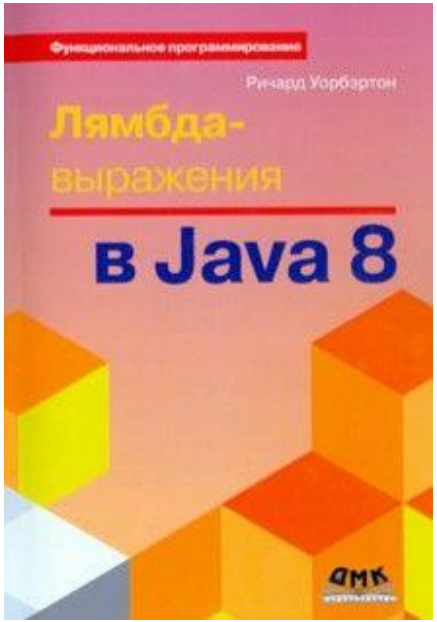
Functional Reactive Programming - RxJS

```
/* With an interval time */
var source = Rx.Observable.interval(1000)
    .sample(5000)
    .take(2);

var subscription = source.subscribe(
    function (x) {
        console.log('Next: ' + x);
    },
    function (err) {
        console.log('Error: ' + err);
    },
    function () {
        console.log('Completed');
    });
```

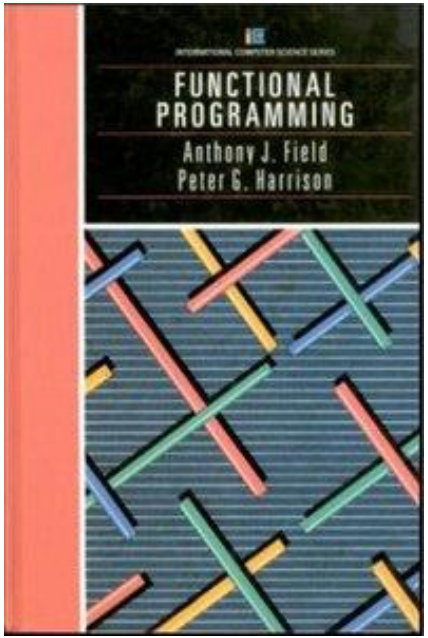
Functional Reactive Programming - Рекомендации

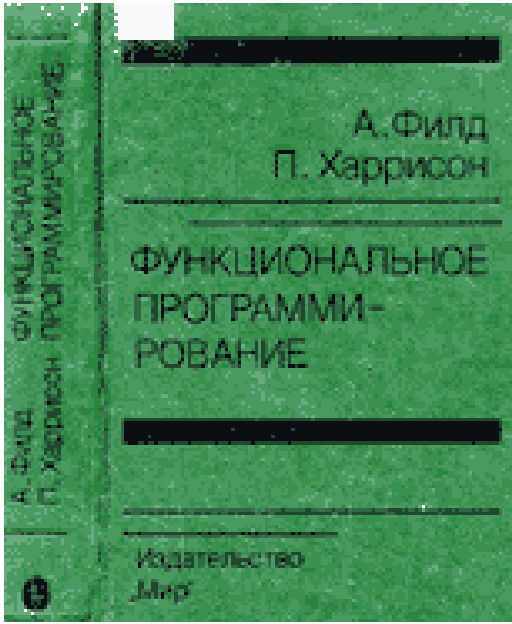
Р.Уорбертон
«Лямбда-выражения в Java 8»





Р.Филд, П.Харрисон
«Функциональное программирование»





THANK YOU

CONTACT ME

selavy

Vyacheslav Lapin

Vseslavur