Reactive programming

October, 21st 2018

- Sergey Morenets, 2018

DEVELOPER	14 YEARS
TRAINER	6 YEARS
WRITER	4 BOOKS

FOUNDER

ITSimulator

IT DISCOVERY

IT DISCOVERY

SPEAKER

JPRIME

JAVA DAY
MINSK 2013

polish
jug

Dev(Talks):

DEVOXX™
POLAND

JEEConf

JAVA
DAY2015

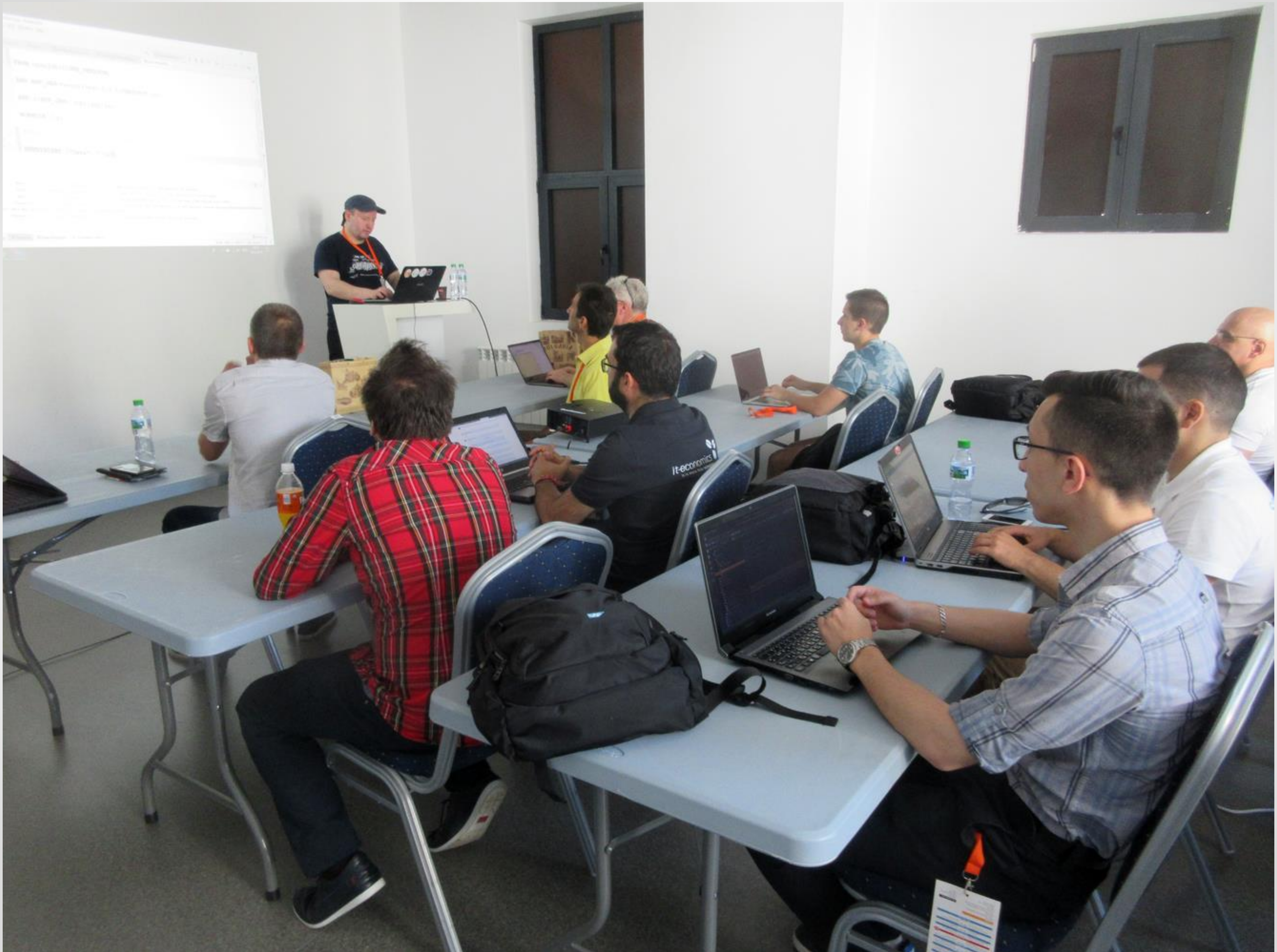
GEEKS LAB

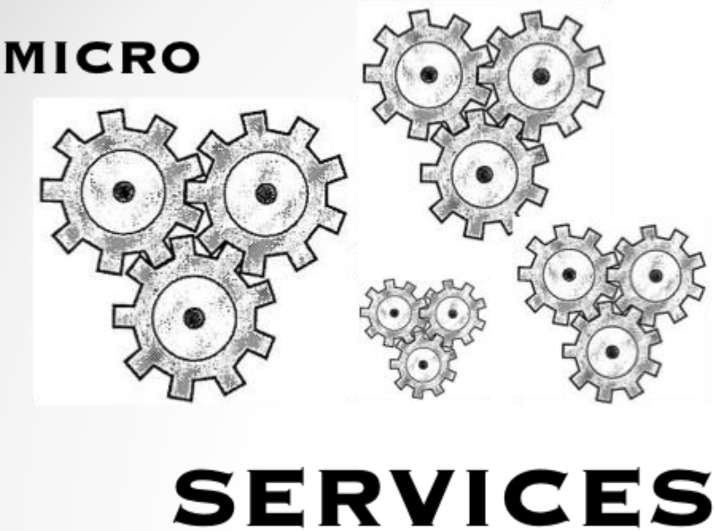
IT talk

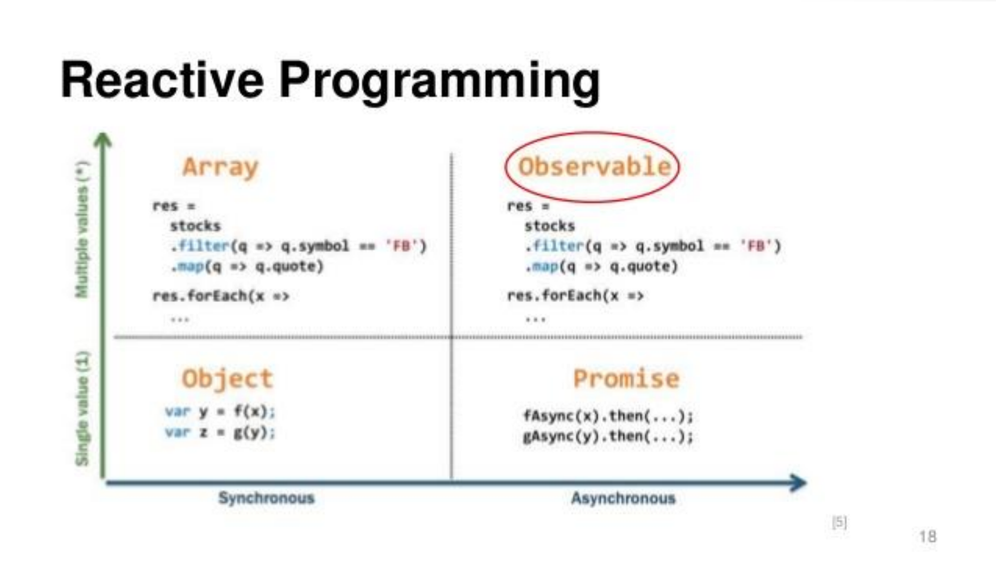
Java
frameworks
day

@sergey-morenets
DEVOXX POLAND

[illegible]


it-economics





Reactive Programming

Multiple values (*)

Array

```
res =
  stocks
  .filter(q => q.symbol == 'FB')
  .map(q => q.quote)
res.forEach(x =>
  ...
```

Observable

```
res =
  stocks
  .filter(q => q.symbol == 'FB')
  .map(q => q.quote)
res.forEach(x =>
  ...
```

Single value (1)

Object

```
var y = f(x);
var z = g(y);
```

Promise

```
fAsync(x).then(...);
gAsync(y).then(...);
```

Synchronous

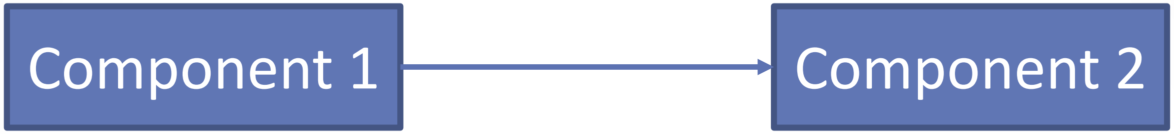
Asynchronous

[5]

18

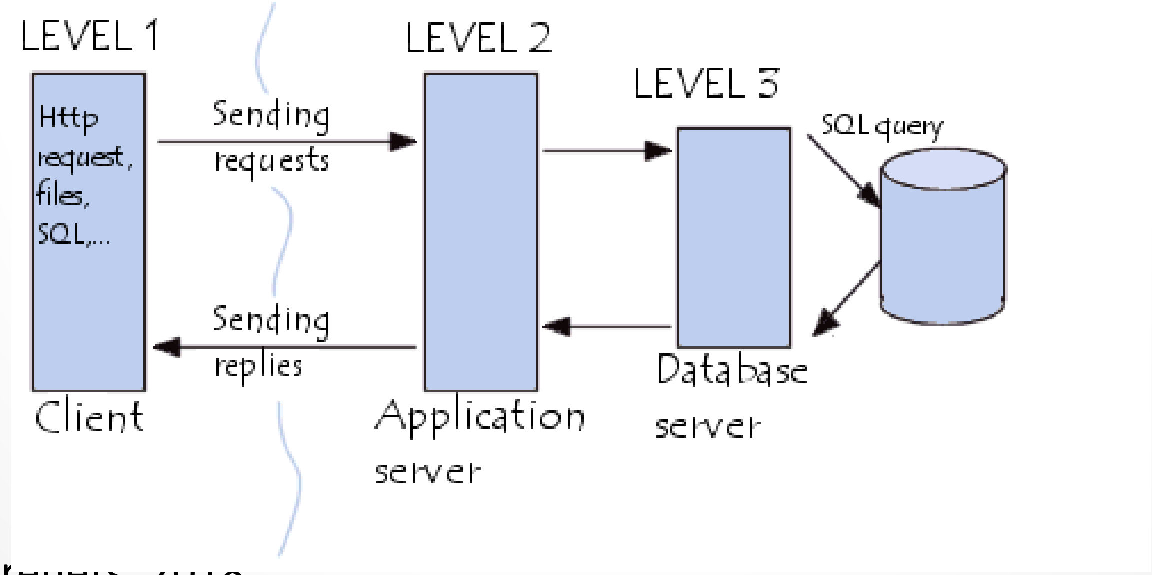
Sergey Morenets, 2018

Software communication



Component 1

Component 2



Software communication rules

Sergey Morenets, 2018

Task #1. Installation & configuration

1. Install and configure your IDE
2. Import training project as **Maven/Gradle** projects.
3. Review project and its structure/content.
4. What are project components? How do they interact with each other?
5. What are advantages/disadvantages of the current component communication?

Java. Asynchronous programming

- Introduced in 1.0 with **Thread**, Runnable and Callable types
- Was refined in Java 5 using Executors library and additional types like Queue, Semaphore, CountDownLatch and CyclicBarrier
- Thread-safe collections like CopyOnWriteArrayList/ **ConcurrentHashMap**
- Uses **synchronized/volatile** keywords or Lock-based types to provide synchronization/locking
- Added ForkJoinPool in Java 7
- Introduced CompletableFuture in Java 8

Asynchronous helpers

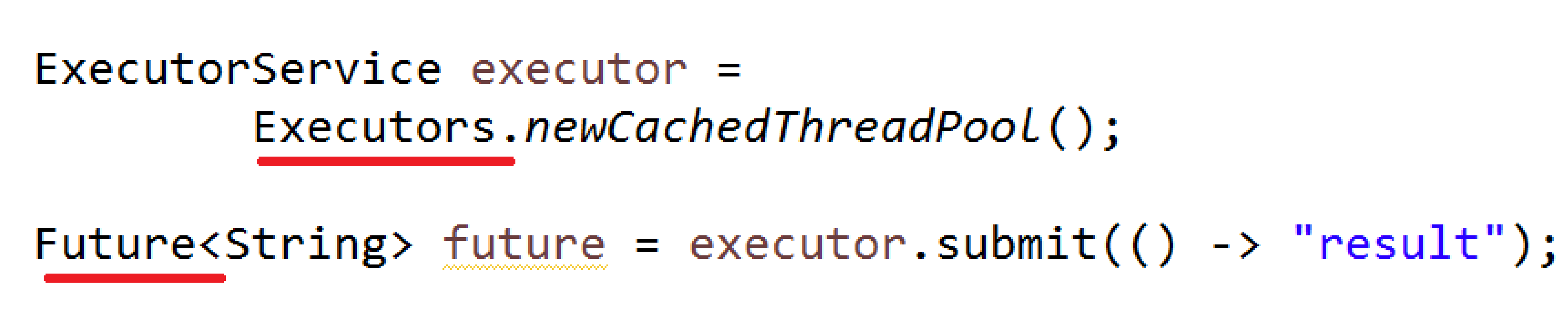
- **Future**

Future is read-only reference to the expression that may be calculated or failed to calculate

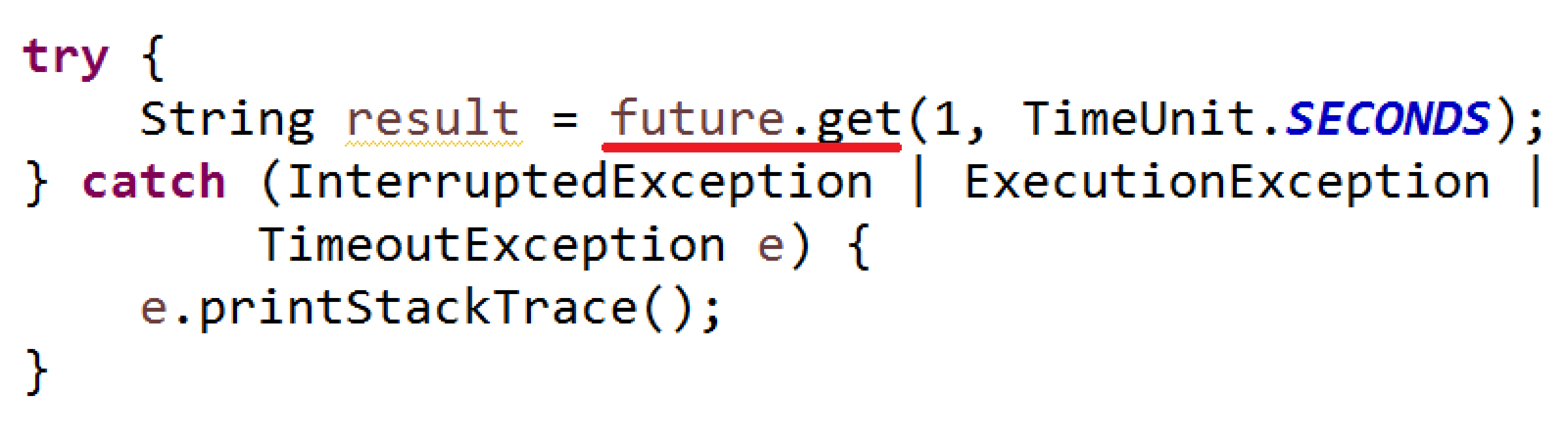
- **Promise**

Promise is write-once reference to the expression that will be provided

Java 5. Futures

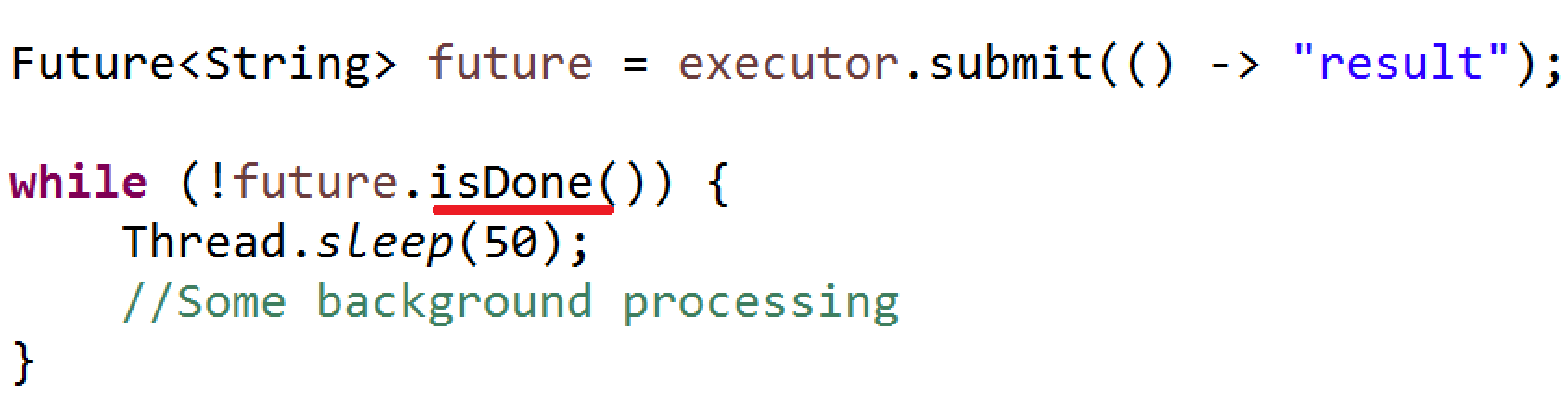


```java
ExecutorService executor =
        Executors.newCachedThreadPool();

Future<String> future = executor.submit(() -> "result");
```

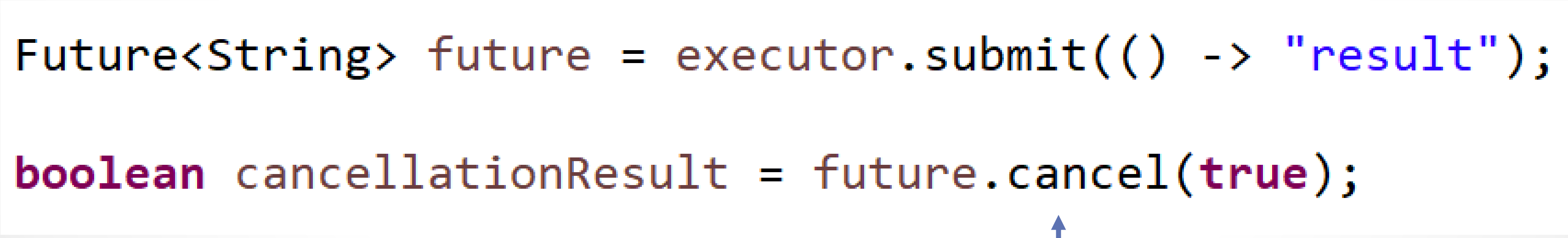


```java
try {
    String result = future.get(1, TimeUnit.SECONDS);
} catch (InterruptedException | ExecutionException |
        TimeoutException e) {
    e.printStackTrace();
}
```

Java 5. Futures workflow



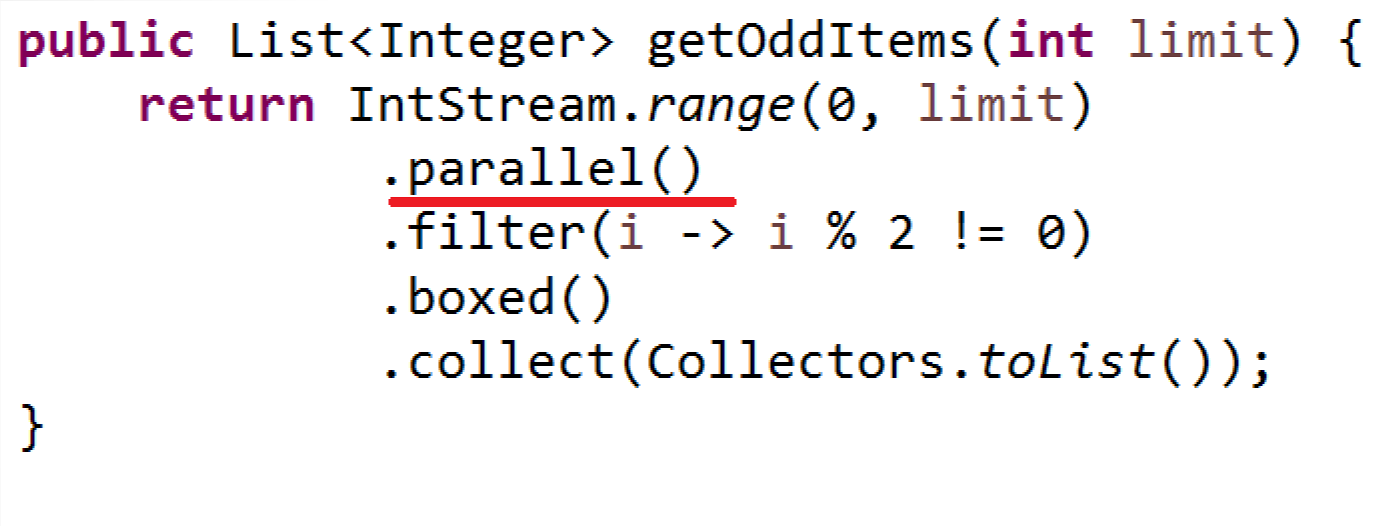
```
Future<String> future = executor.submit(() -> "result");

while (!future.isDone()) {
    Thread.sleep(50);
    //Some background processing
}
```



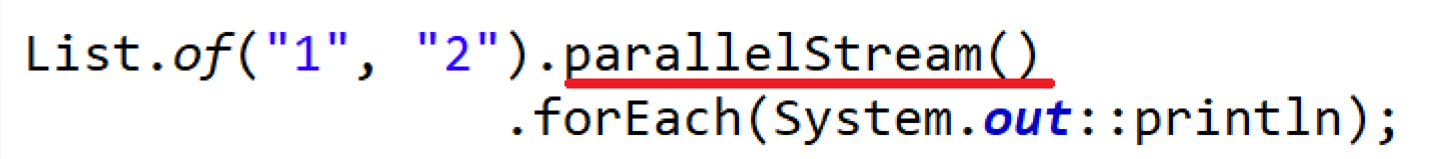
```
Future<String> future = executor.submit(() -> "result");

boolean cancellationResult = future.cancel(true);
```

Cancels current task with interrupting if needed

Java 8. Parallel streams



```
public List<Integer> getOddItems(int limit) {
    return IntStream.range(0, limit)
            .parallel()
            .filter(i -> i % 2 != 0)
            .boxed()
            .collect(Collectors.toList());
}
```



```
List.of("1", "2").parallelStream()
                .forEach(System.out::println);
```

Task #2. Asynchronous programming

1. Review training project. How would you rewrite it using asynchronous (blocking) programming paradigm?
2. Change **WaiterService** so that it operates asynchronously. Try to run Starter class and confirm that application works properly.
3. How would you change Starter/WaiterService to cancel current order on timeout?

CompletableFuture

- Introduced in Java 8
- Implements Future and CompletionStage interfaces
- Allows to combine the results of different asynchronous executions or computations in the chain of steps
- Error handling

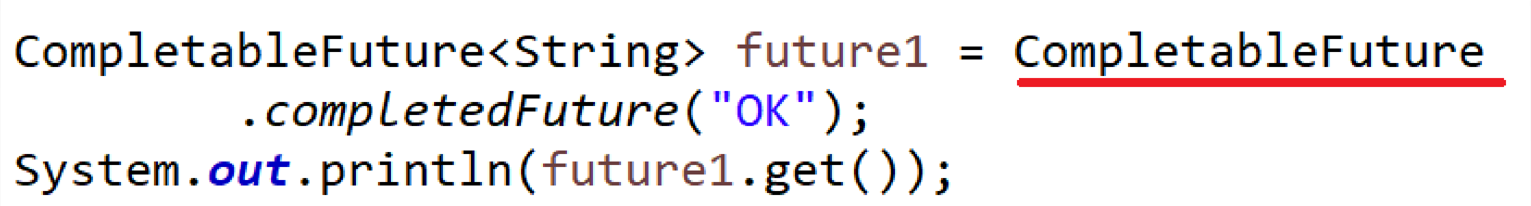
CompletableFuture

Name	Description
runAsync	Asynchronously complete this Runnable
supplyAsync	Asynchronously complete this task
anyOf/allOf	Executes several tasks
acceptEither	Executes current or given task
thenAccept	Allows to execute action using completion result
thenApply	Allows to execute function using completion result
cancel	Cancel current task
runAfterEither	Allows to execute action after current or given task completes

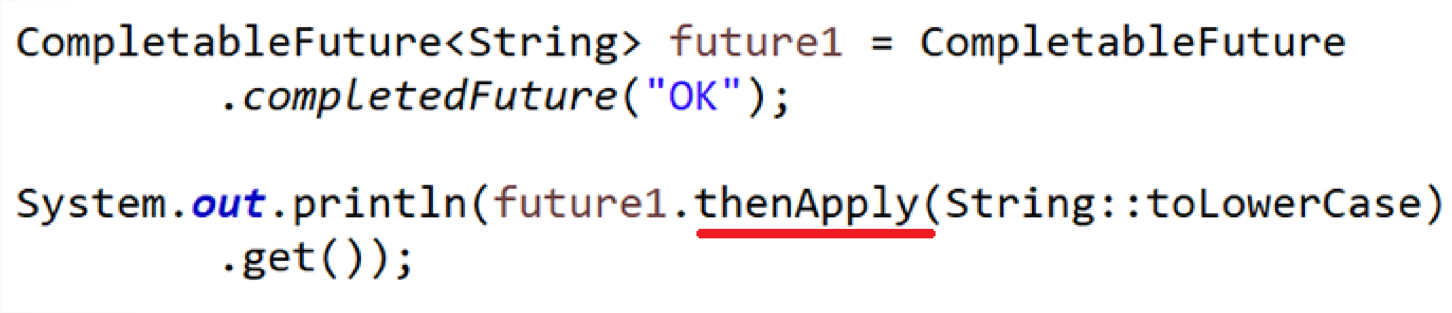
CompletableFuture

Name	Description
thenCombine	Allows to execute in parallel two independent CompletableFuture tasks and combine result
thenCompose	Allows to combine execution of two CompletableFuture tasks
thenRun	Run the action when current task completes
handle	Allows to handle exception/task result in one method
orTimeout	Allows specify timeout for the current task
join	Returns execution result of throws an exception
completeExceptionally	Throws given exception if not completed

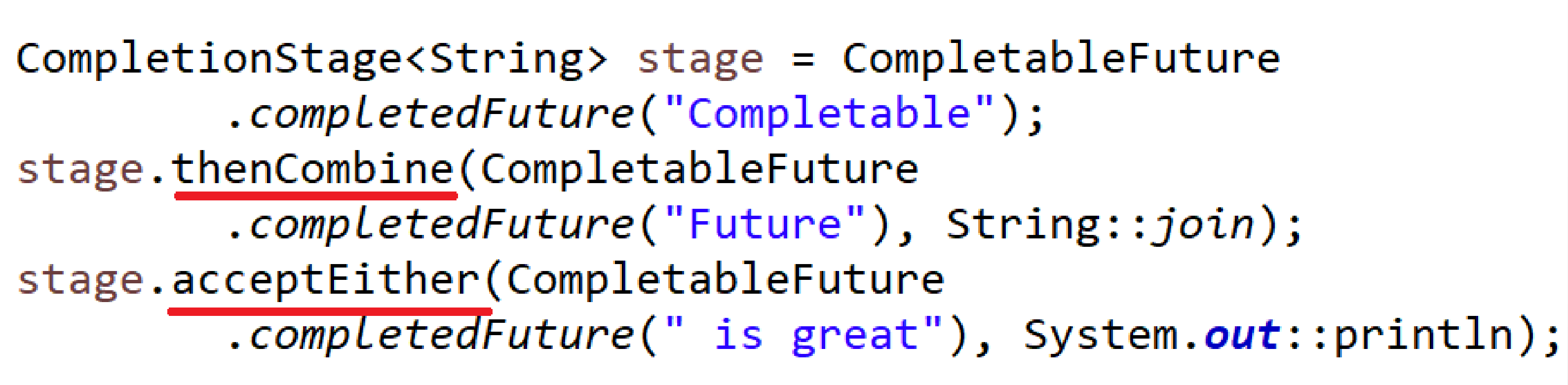
Java 8. CompletableFuture



```
CompletableFuture<String> future1 = CompletableFuture
        .completedFuture("OK");
System.out.println(future1.get());
```

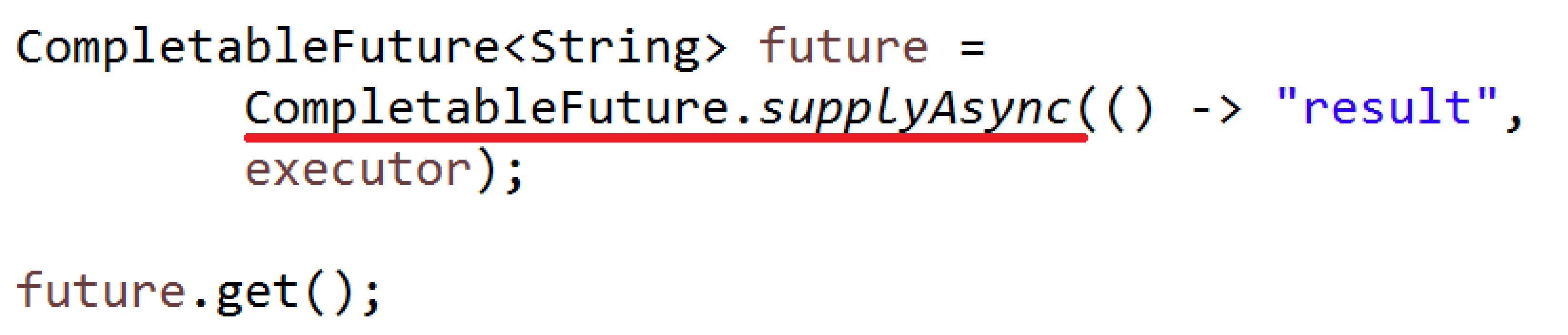


```
CompletableFuture<String> future1 = CompletableFuture
        .completedFuture("OK");

System.out.println(future1.thenApply(String::toLowerCase)
        .get());
```

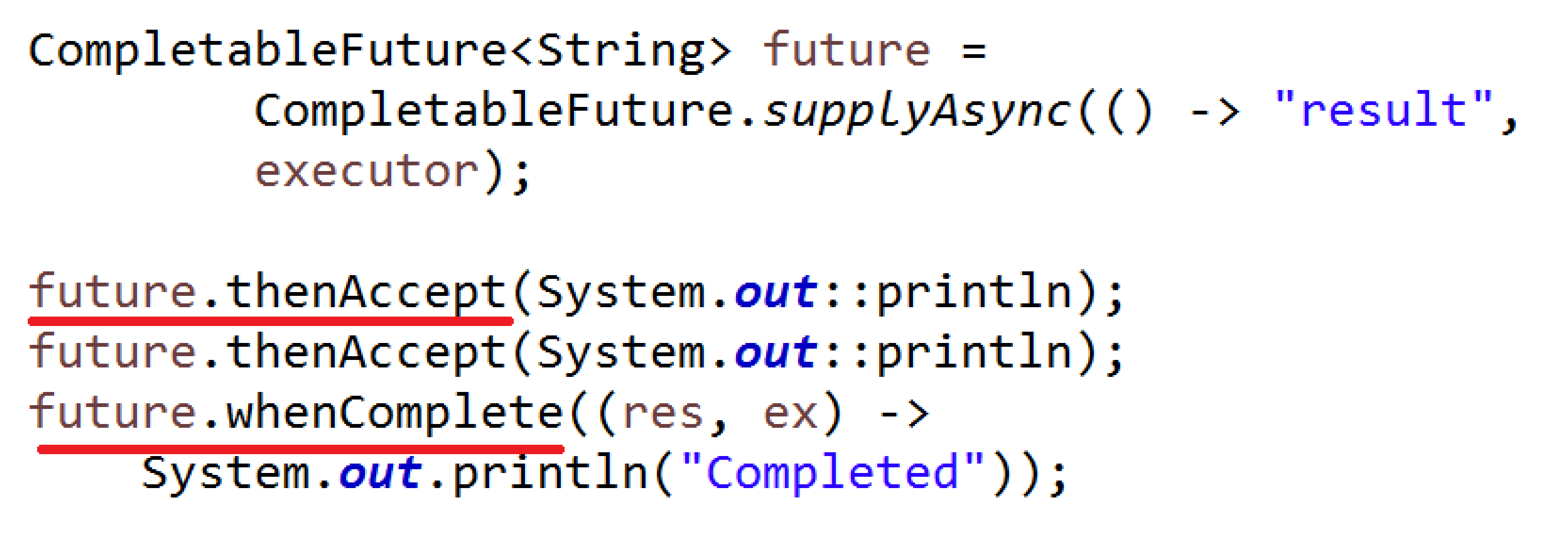


```
CompletionStage<String> stage = CompletableFuture
        .completedFuture("Completable");
stage.thenCombine(CompletableFuture
        .completedFuture("Future"), String::join);
stage.acceptEither(CompletableFuture
        .completedFuture(" is great"), System.out::println);
```

Java 8. CompletableFuture

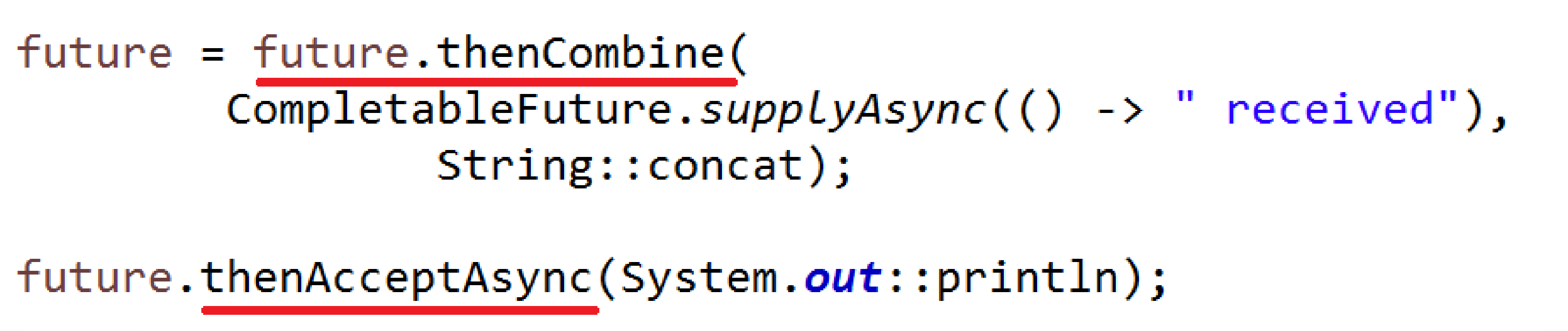


```
CompletableFuture<String> future =
        CompletableFuture.supplyAsync(() -> "result",
        executor);

future.get();
```

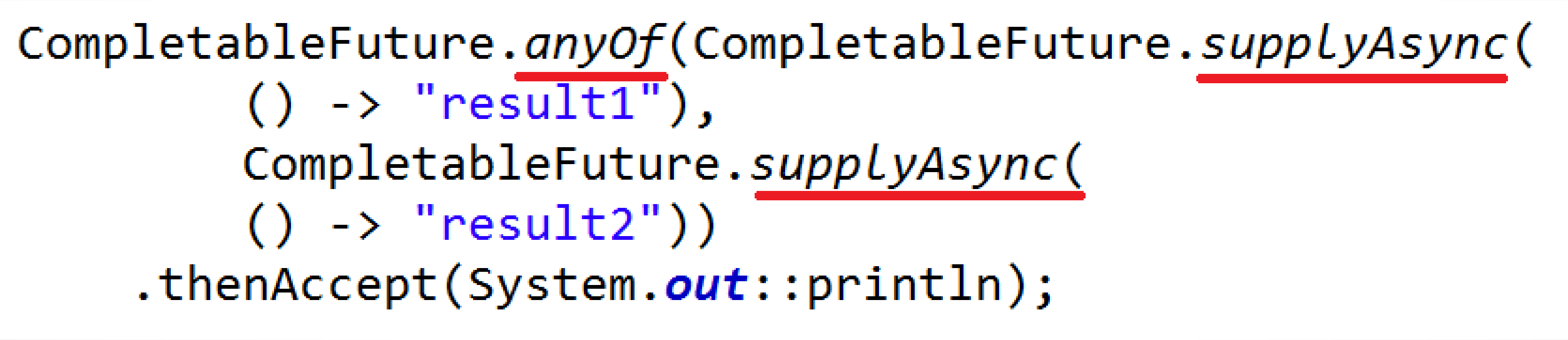


```
CompletableFuture<String> future =
        CompletableFuture.supplyAsync(() -> "result",
        executor);

future.thenAccept(System.out::println);
future.thenAccept(System.out::println);
future.whenComplete((res, ex) ->
    System.out.println("Completed"));
```

Java 8. CompletableFuture

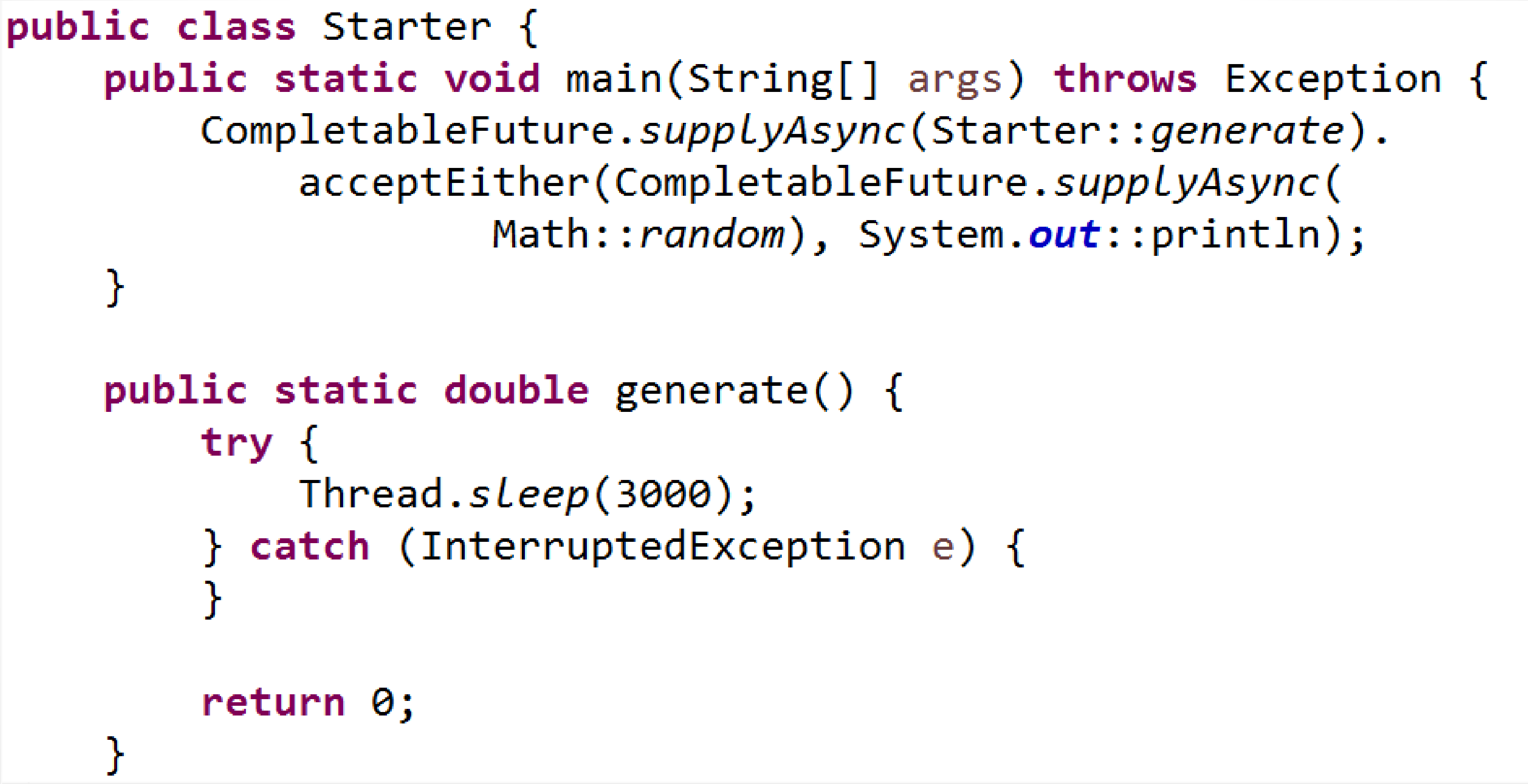


```
future = future.thenCombine(
        CompletableFuture.supplyAsync(() -> " received"),
                String::concat);

future.thenAcceptAsync(System.out::println);
```



```
CompletableFuture.anyOf(CompletableFuture.supplyAsync(
        () -> "result1"),
        CompletableFuture.supplyAsync(
        () -> "result2"))
    .thenAccept(System.out::println);
```

Java 8. CompletableFuture

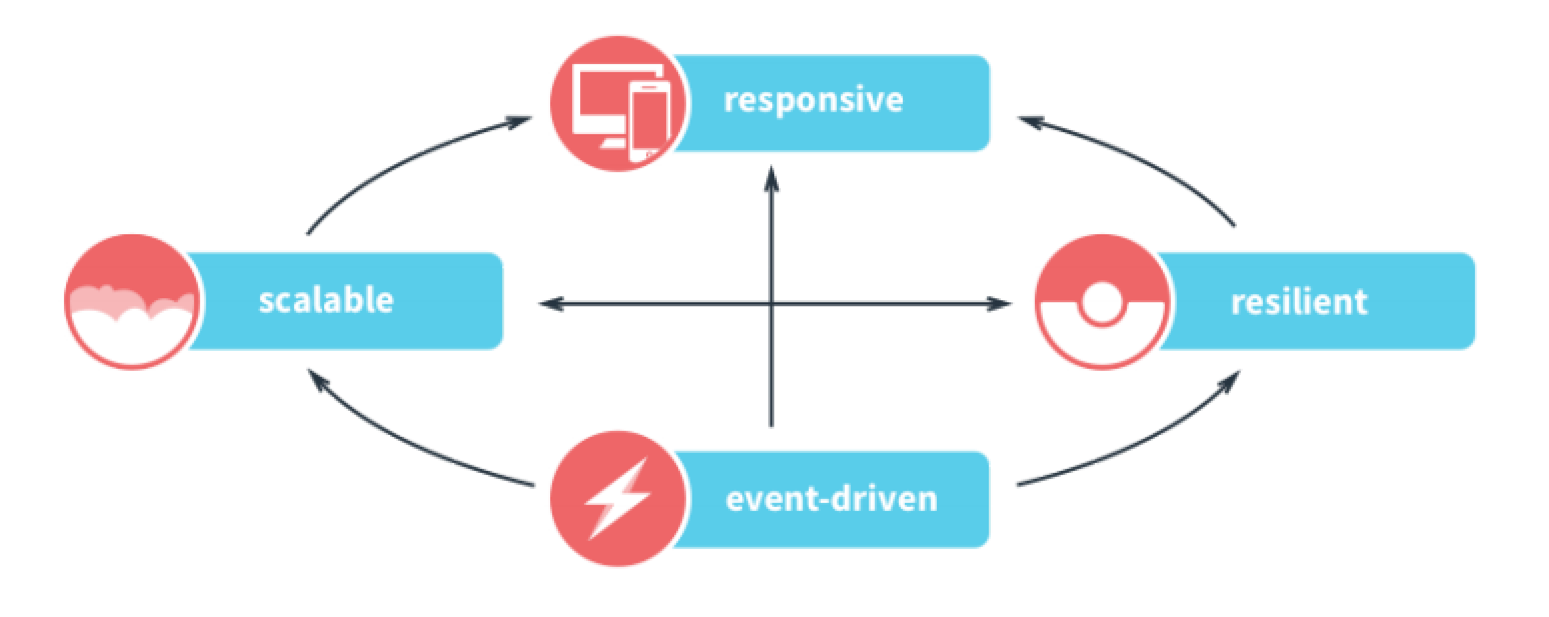


```java
public class Starter {
    public static void main(String[] args) throws Exception {
        CompletableFuture.supplyAsync(Starter::generate).
            acceptEither(CompletableFuture.supplyAsync(
                    Math::random), System.out::println);
    }

    public static double generate() {
        try {
            Thread.sleep(3000);
        } catch (InterruptedException e) {
        }

        return 0;
    }
```

Task #3. CompletableFuture

1. Review methods from CompletableFuture class.
2. Change **WaiterService** so that it returns CompletableFuture now for long-running operations. How would you change Starter class?
3. Try to run Starter class and confirm that application works properly.
4. What is disadvantages of the current solution?

Reactive manifesto



Reactive manifesto

- ✓ **Responsive**

  The system responses in timely manner

- ✓ **Resilient**

  The system is resilient in case of failures(software, hardware, timeout, human errors)

- ✓ **Elastic**

  The system stays responsive under varying workload

- ✓ **Message-driven**

  The system rely on asynchronous messaging to enable communication between components

Reactive manifesto. Patterns

- **Responsive**

  Circuit breaker, offline mode

- **Resilient**

  Fault tolerance, sharding, replication

- **Elastic**

  Vertical and horizontal scalability

- **Message-driven**

  Message queues, location transparency

Reactive programming history

- ✓ **Erlang** created in 1987 by Joe Armstrong and brought Actors for distributed calculations
- ✓ 3 seconds downtime per 100 years
- ✓ Re-involved immutability and functional programming
- ✓ Reactive programming is software paradigm about data flows and propagation of changes
- ✓ Data stream is sequence of ordered events

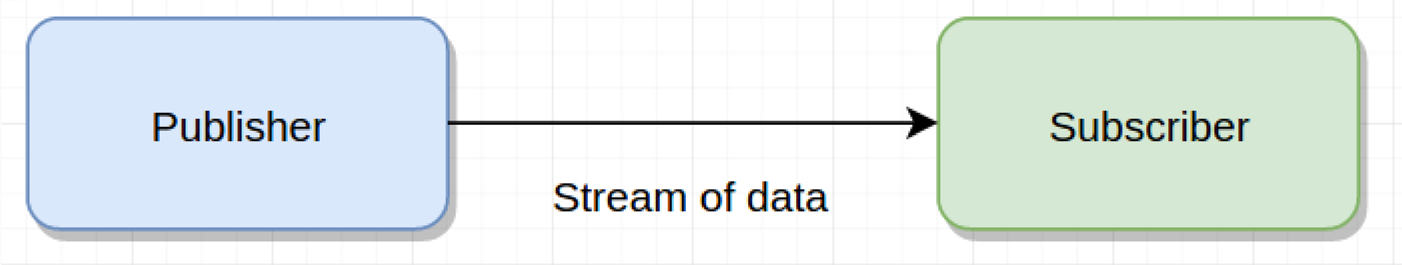
Reactive programming

- **Leads to asynchronous programming**
- Flow of the application is easier to understand
- Provides universal error handling
- Source of information is entity that emits data
- Stream data types are value type, signal of error and signal of completion
- Cold source of information publishes data even if there are no subscribers
- Hot source of information publishes data after first subscription

Developer kit

- ✓ Akka
- ✓ Disruptor
- ✓ Apache Storm
- ✓ ReactiveX

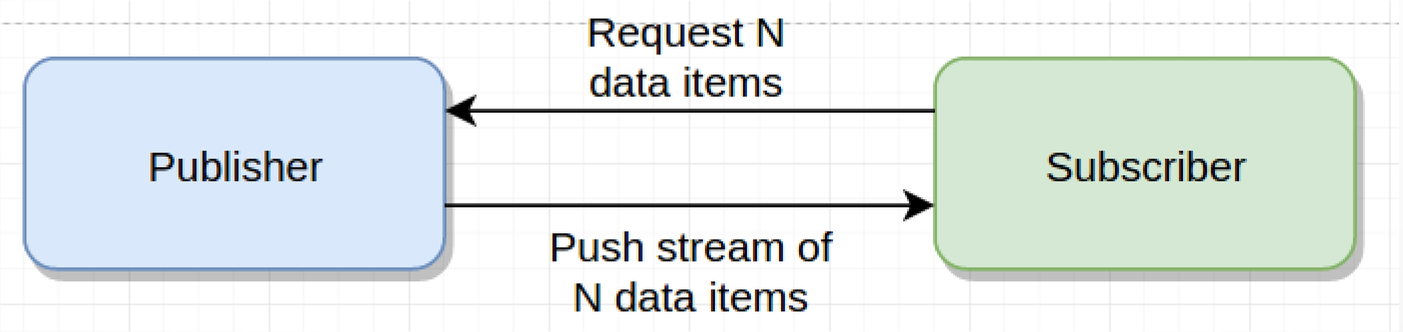
Reactive streams



Publisher

Stream of data

Subscriber



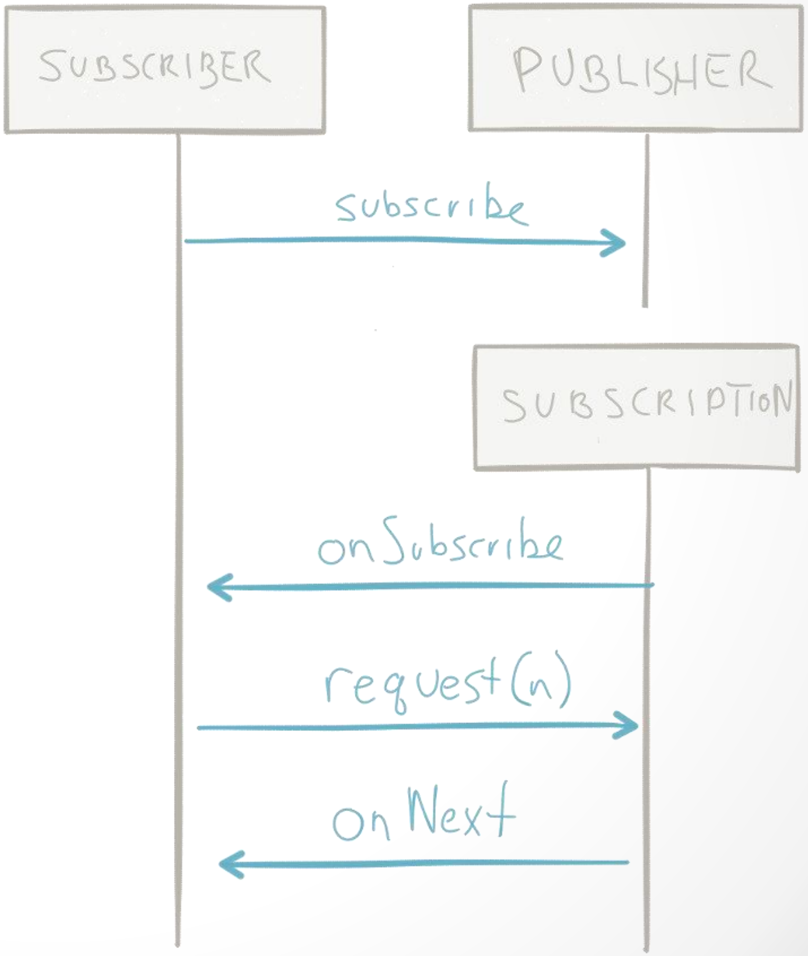
Request N data items

Publisher

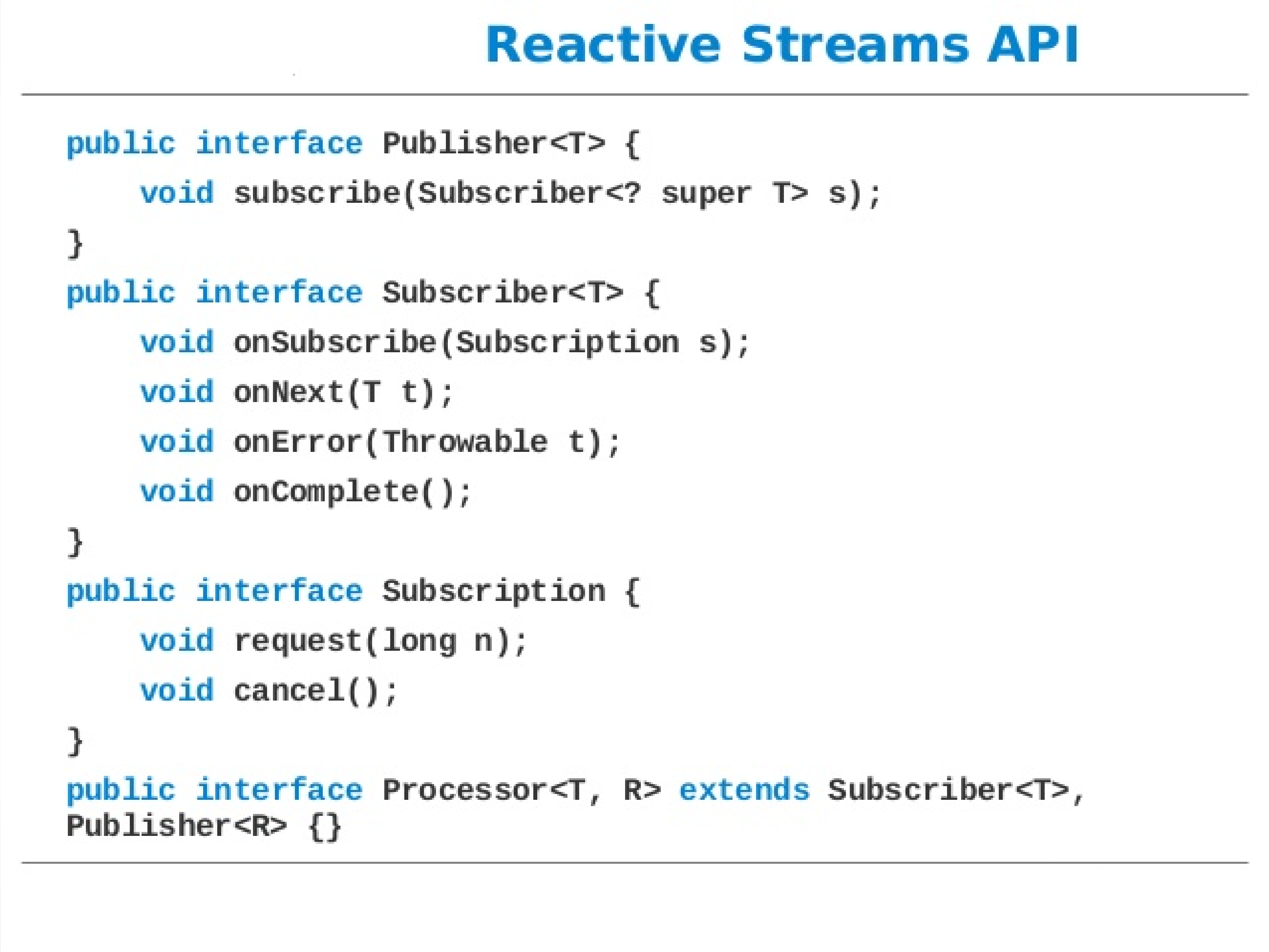
Subscriber

Push stream of N data items

Reactive streams



Sergey Morenets, 2018



Reactive Streams API

```
public interface Publisher<T> {
    void subscribe(Subscriber<? super T> s);
}
public interface Subscriber<T> {
    void onSubscribe(Subscription s);
    void onNext(T t);
    void onError(Throwable t);
    void onComplete();
}
public interface Subscription {
    void request(long n);
    void cancel();
}
public interface Processor<T, R> extends Subscriber<T>,
Publisher<R> {}
```

Reactive streams

- ✓ Version 1.0 released
- ✓ Core types are Subscriber/Publisher/Subscription/Processor
- ✓ Implementations include Akka, MongoDB, **Reactor**, **RxJava** and Slick
- ✓ Incorporated in **Java 9** as Flow API

ReactiveX

- Best ideas from **Observer/Iterator** pattern and functional programming
- Developed by **Netflix**
- Implemented in Java as **RxJava**
- Supports JavaScript, Scala, Clojure, Swift and .NET

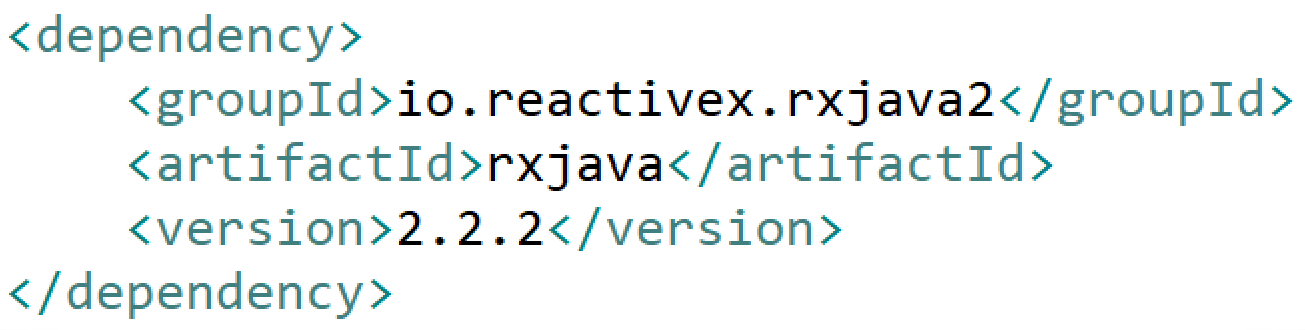
We use ReactiveX

Applied Duality
Microsoft
NETFLIX
GitHub
SOUNDCLOUD
Trello
treehouse
SeatGee
ooVoo
Couchbase
futurice
welbe
airbnb

RxJava

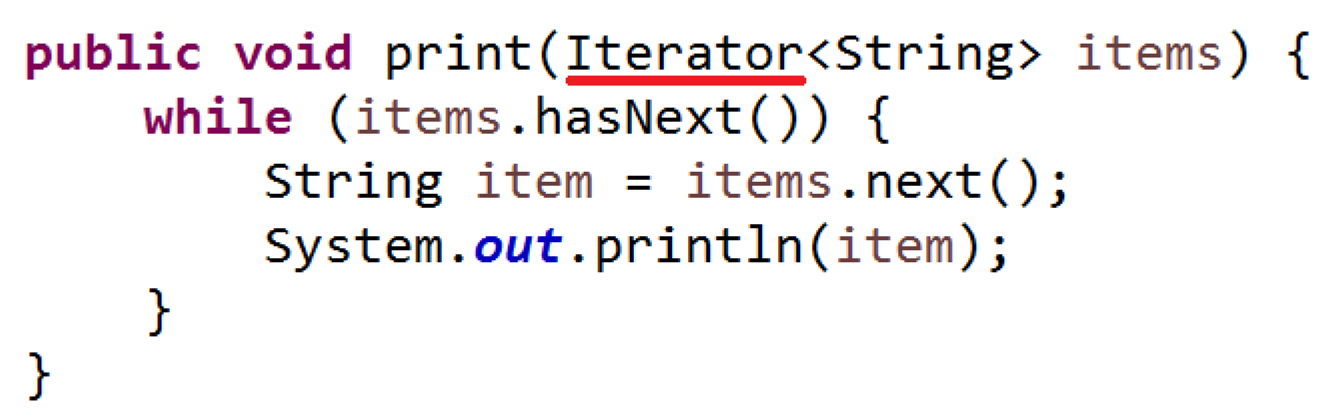
- ✓ Designed for **JDK6+** and Andriod 2.3+
- ✓ Support for Java 8 **lambdas**
- ✓ Developed since 2014
- ✓ Stable 1.x branch and brand new 2.x branch(**18** dev months!)
- ✓ Scala/Kotlin/Clojure support
- ✓ Synchronous and asynchronous execution
- ✓ Reactive and functional programming
- ✓ No 3rd-party libraries
- ✓ Supports back-pressure in 2.x
- ✓ By default single-threaded

RxJava. Maven dependencies

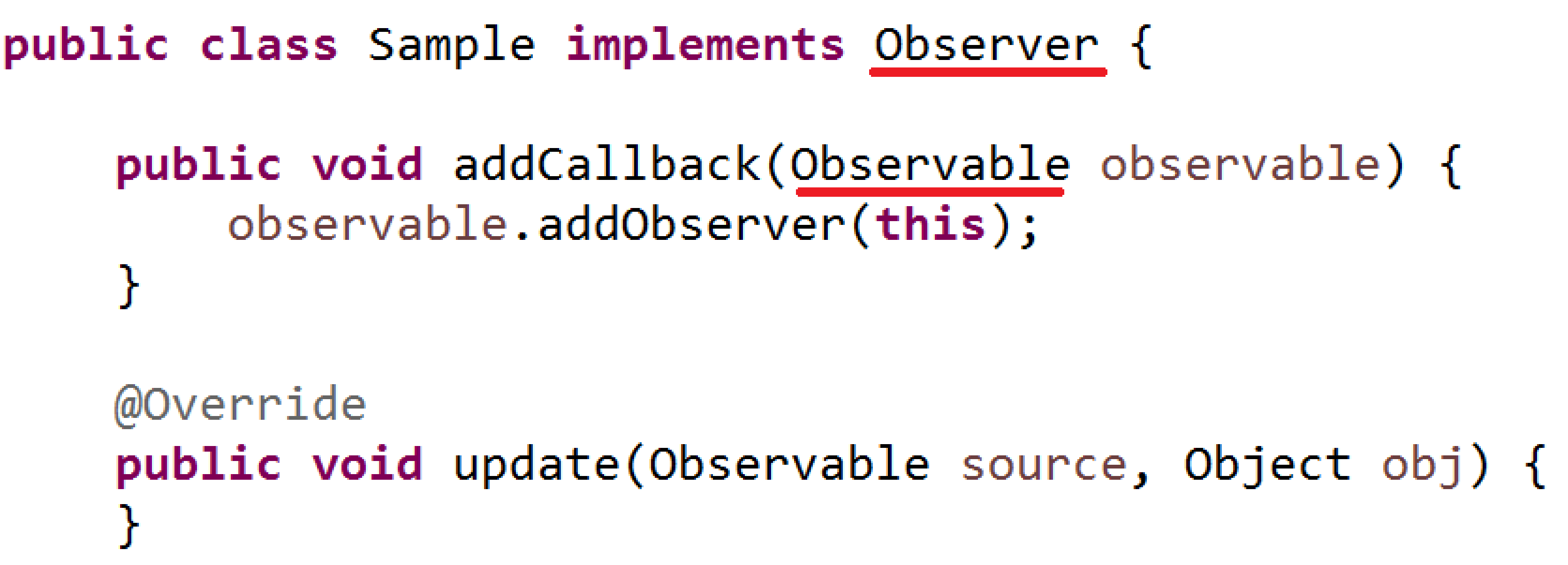


```xml
<dependency>
    <groupId>io.reactivex.rxjava2</groupId>
    <artifactId>rxjava</artifactId>
    <version>2.2.2</version>
</dependency>
```

Java SE. Iterator & Observer



```
public void print(Iterator<String> items) {
    while (items.hasNext()) {
        String item = items.next();
        System.out.println(item);
    }
}
```



```
public class Sample implements Observer {

    public void addCallback(Observable observable) {
        observable.addObserver(this);
    }

    @Override
    public void update(Observable source, Object obj) {
    }
```

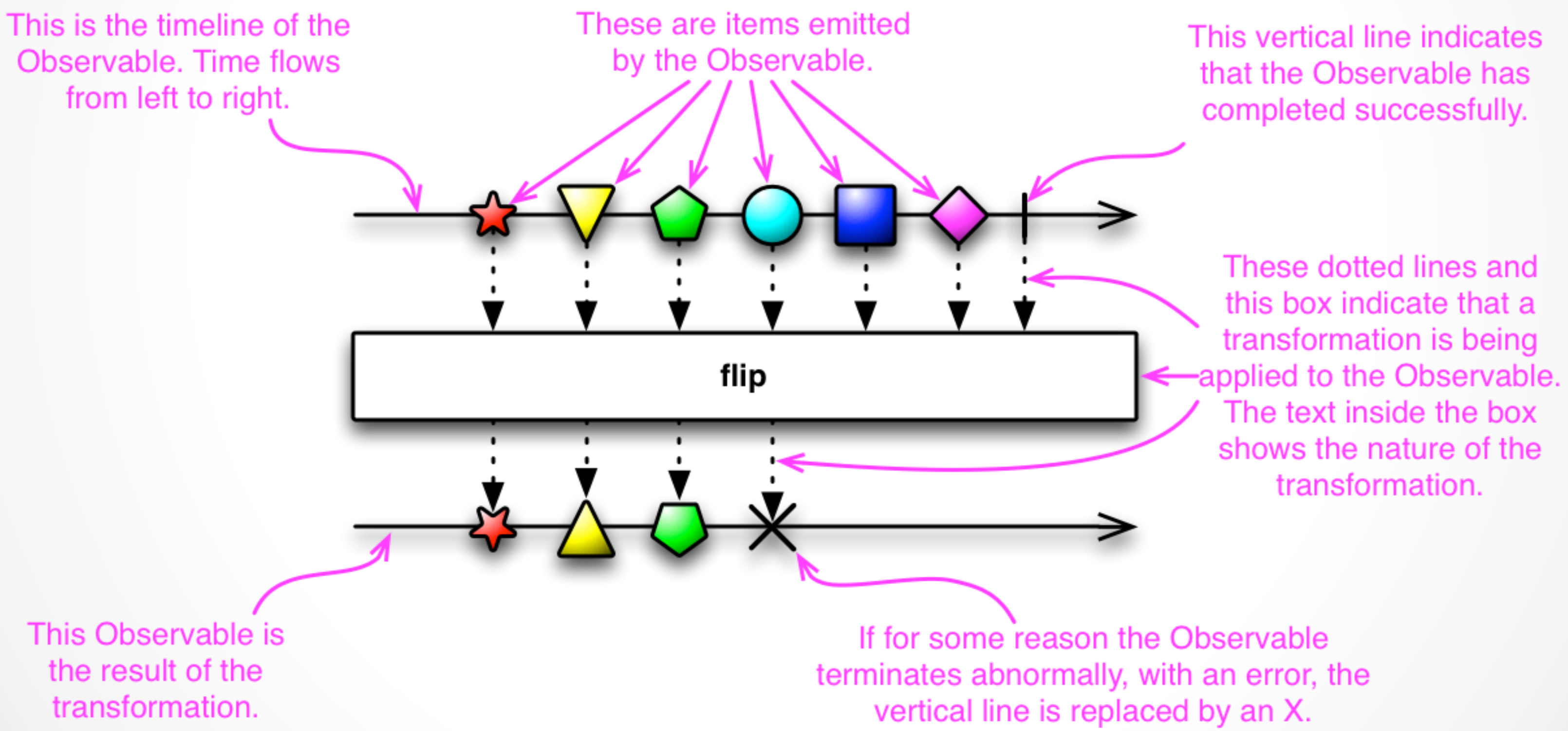
Observable. Creation methods

Method	Description
empty()	Creates Observable with no items but completion event
error()	Creates Observable that invokes **onError** callback
fromArray()	Converts array into Observable
fromCallable()	Creates Observable based on **Callable** instance
fromFuture()	Converts **Future** into Observable
generate()	Creates synchronous stateless generator of values
just()	Creates Obserable that generates given events
never()	Creates Observable with no items
range()	Creates Observable with given range of integers
create()	Creates Observable that wraps non-reactive behavior

RxJava API

- Single – publisher that emits either single event or error with no completion event
- Maybe – publisher that emits single event, no event or error with completion event support
- Completable – computation that can generate only error or completion event

RxJava. Marbles



This is the timeline of the Observable. Time flows from left to right.

These are items emitted by the Observable.

This vertical line indicates that the Observable has completed successfully.

These dotted lines and this box indicate that a transformation is being applied to the Observable. The text inside the box shows the nature of the transformation.

This Observable is the result of the transformation.

If for some reason the Observable terminates abnormally, with an error, the vertical line is replaced by an X.

RxJava. Terms

Emission

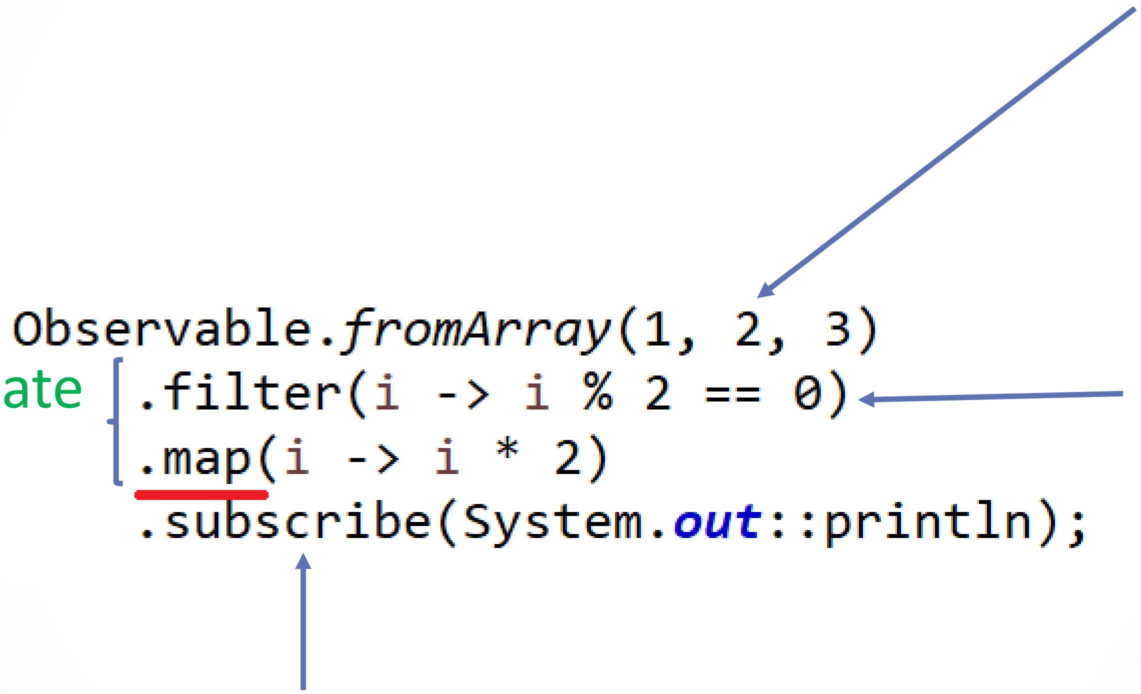
Item

Signal

Event

Message

Data



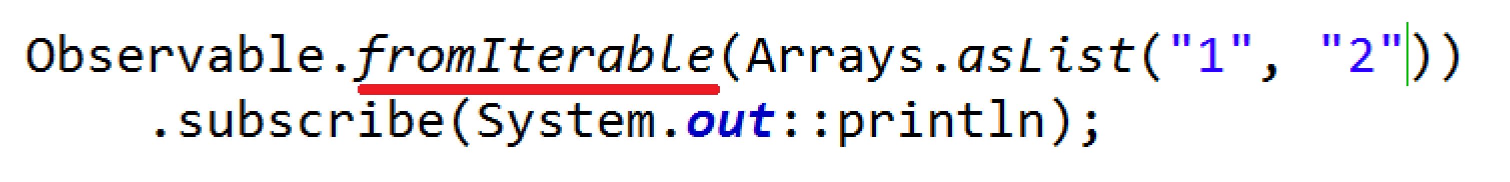
```
Observable.fromArray(1, 2, 3)
          .filter(i -> i % 2 == 0)
          .map(i -> i * 2)
          .subscribe(System.out::println);
```

Intermediate operators

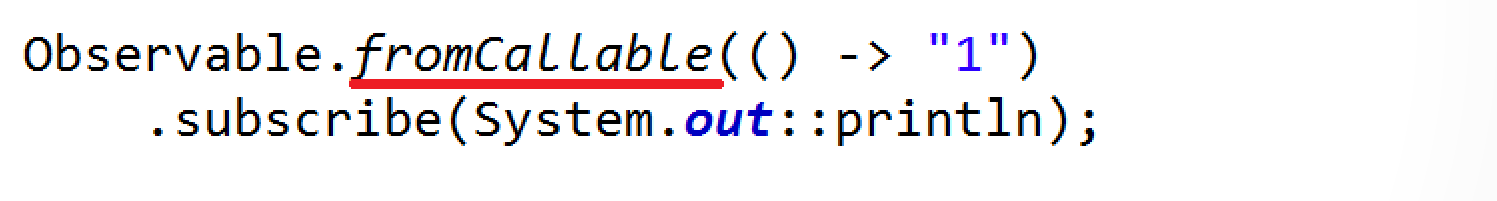
Upstream operators

Downstream operators

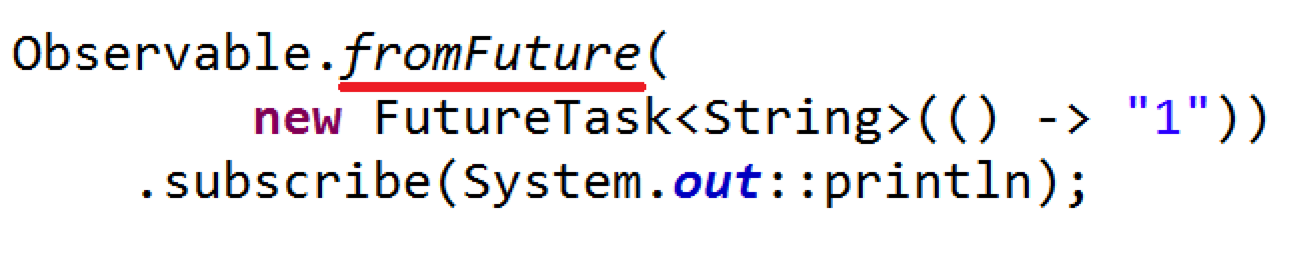
Observable generation



```
Observable.fromIterable(Arrays.asList("1", "2"))
     .subscribe(System.out::println);
```

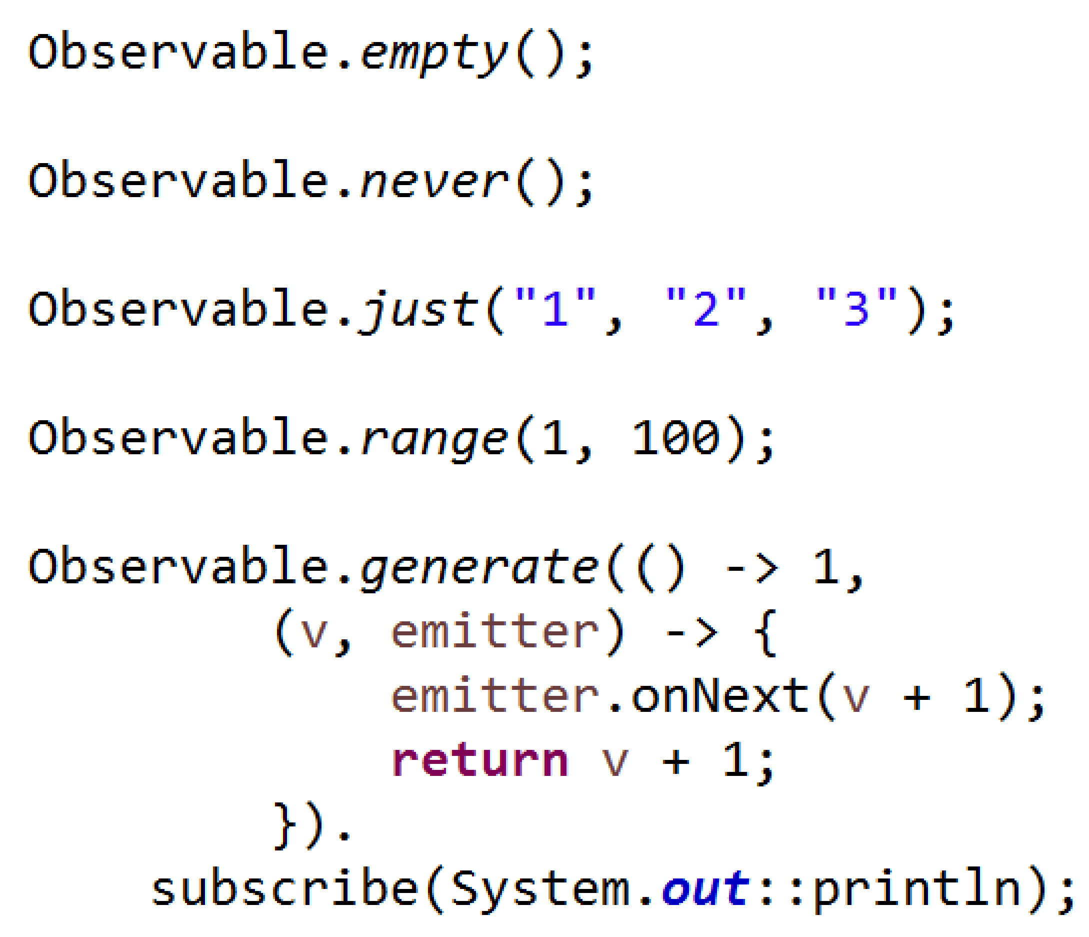


```
Observable.fromCallable(() -> "1")
     .subscribe(System.out::println);
```



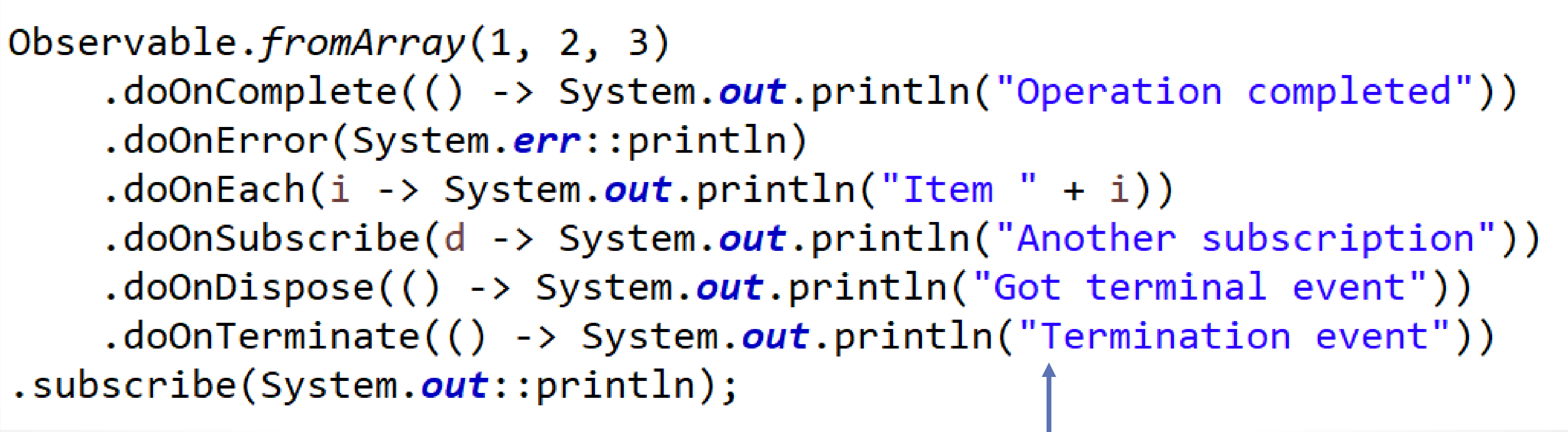
```
Observable.fromFuture(
         new FutureTask<String>(() -> "1"))
     .subscribe(System.out::println);
```

Observable generation



```
Observable.empty();

Observable.never();

Observable.just("1", "2", "3");

Observable.range(1, 100);

Observable.generate(() -> 1,
        (v, emitter) -> {
            emitter.onNext(v + 1);
            return v + 1;
        }).
    subscribe(System.out::println);
```

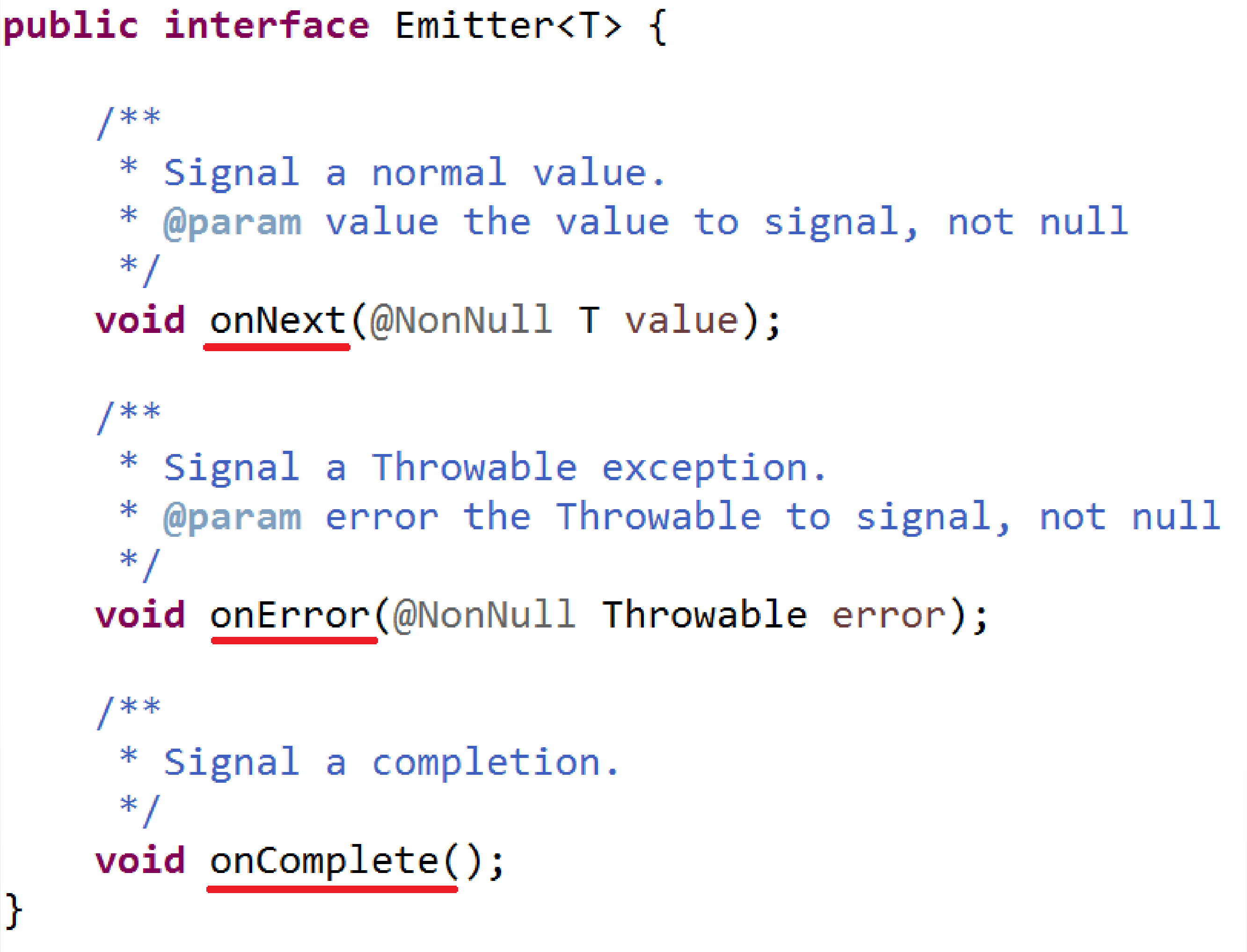
RxJava 2. Callbacks



```
Observable.fromArray(1, 2, 3)
    .doOnComplete(() -> System.out.println("Operation completed"))
    .doOnError(System.err::println)
    .doOnEach(i -> System.out.println("Item " + i))
    .doOnSubscribe(d -> System.out.println("Another subscription"))
    .doOnDispose(() -> System.out.println("Got terminal event"))
    .doOnTerminate(() -> System.out.println("Termination event"))
.subscribe(System.out::println);
```

Calls when onComplete/onError signal received

RxJava 2. Emitter

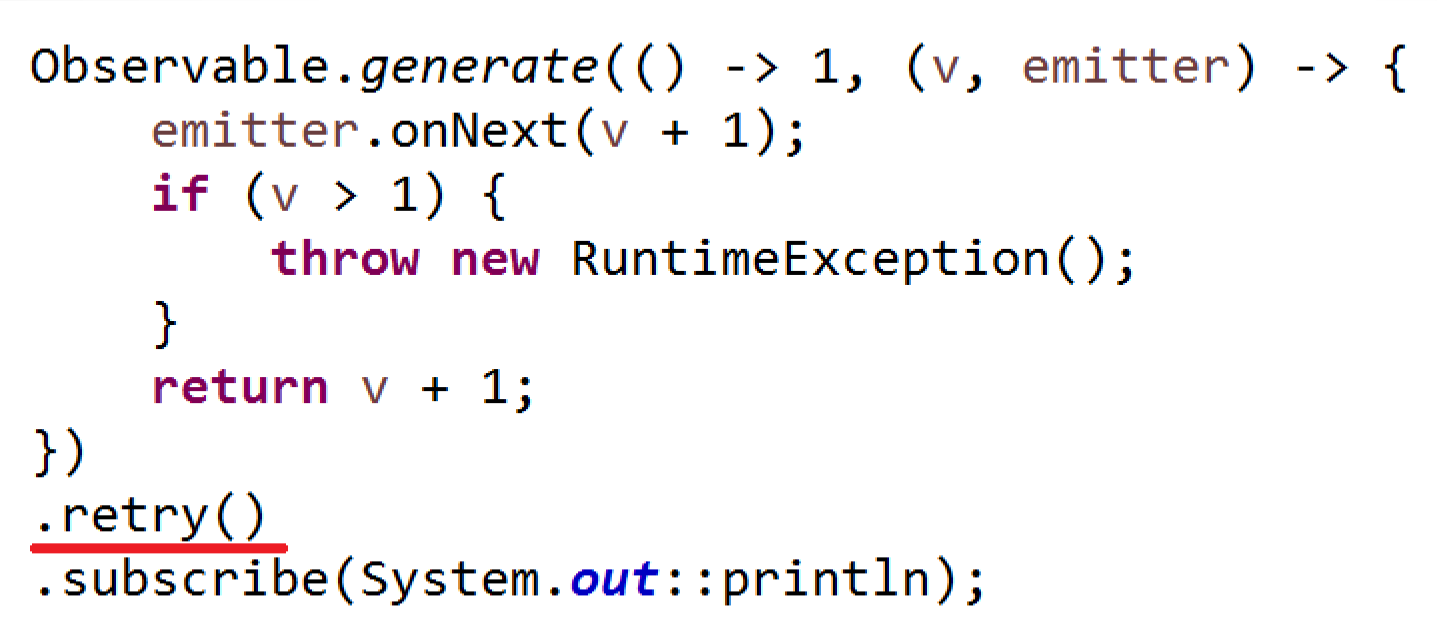


```
public interface Emitter<T> {

    /**
     * Signal a normal value.
     * @param value the value to signal, not null
     */
    void onNext(@NonNull T value);

    /**
     * Signal a Throwable exception.
     * @param error the Throwable to signal, not null
     */
    void onError(@NonNull Throwable error);

    /**
     * Signal a completion.
     */
    void onComplete();
}
```

Observable transformation

Method	Description
interval()	Returns Observable that pushes items with specified interval
delay()	Adds delay before items generation
count()	Calculates total number of generated items
distinct()	Returns Observable with distinct values
doOnEach()	Executes specified consumer for each generated item
elementAt()	Returns item at the specified index or completes if there are less items than specified index
retry()	Retries items generation in case of error
replay()	Replays the generated items for new subscribers

Observable transformation

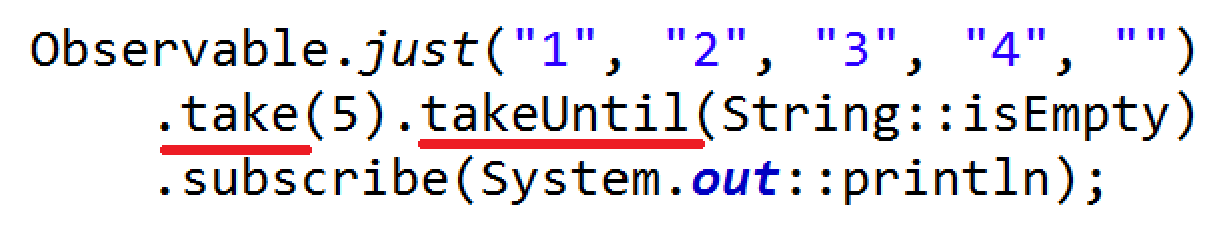


```
Observable.generate(() -> 1, (v, emitter) -> {
    emitter.onNext(v + 1);
    if (v > 1) {
        throw new RuntimeException();
    }
    return v + 1;
})
.retry()
.subscribe(System.out::println);
```

Observable flow control

Method	Description
first()	Returns only the first item
firstOrError()	Returns the first item of signals error if Observable is empty
last()	Returns the last item or the default value
take(N)	Returns only first N items
takeLast(N)	Returns only last N items
takeWhile()	Returns items while condition matches and then completes
takeUntil()	Returns items while condition doesn't match and then completes
skipWhile()	Skip items while certain condition matches
defaultIfEmpty()	Produces item with default value if stream is empty

Observable transformation

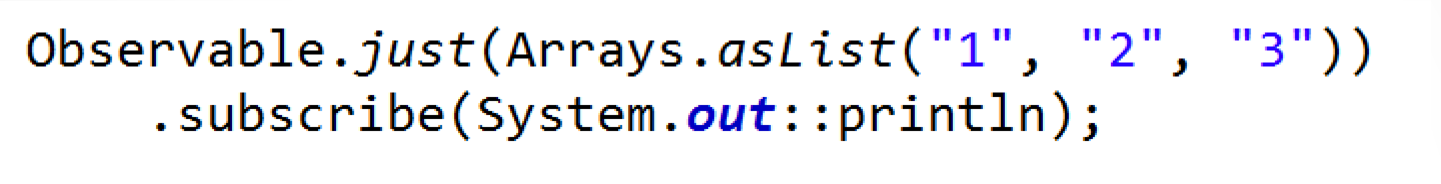


```
Observable.just("1", "2", "3", "4", "")
     .take(5).takeUntil(String::isEmpty)
     .subscribe(System.out::println);
```

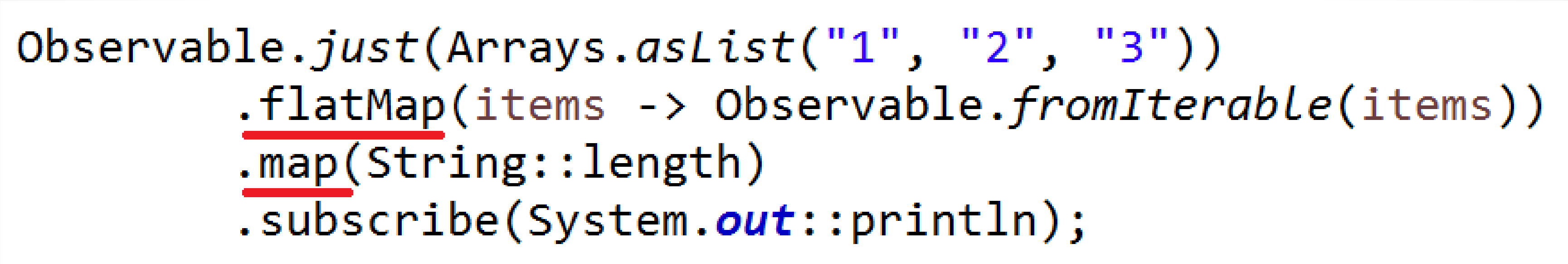
Observable transformation

Method	Description
map()	Performs one-to-one mapping
flatMap()	Performs one-to-many mapping
scan()	Performs scanning of all the items with possible result accumulation
groupBy()	Allows to group by items into groups by some condition

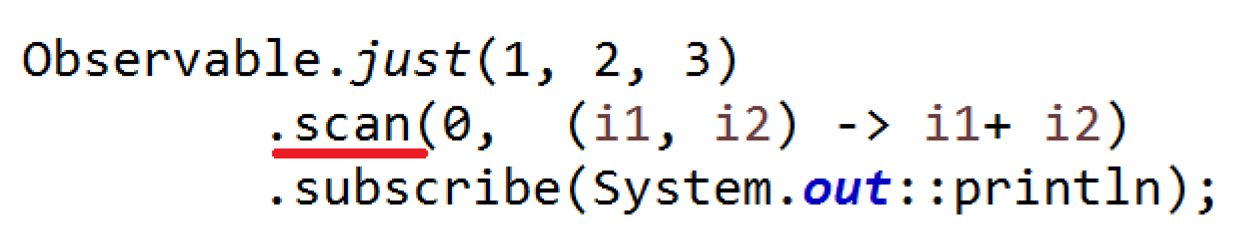
Observable transformation



```
Observable.just(Arrays.asList("1", "2", "3"))
    .subscribe(System.out::println);
```

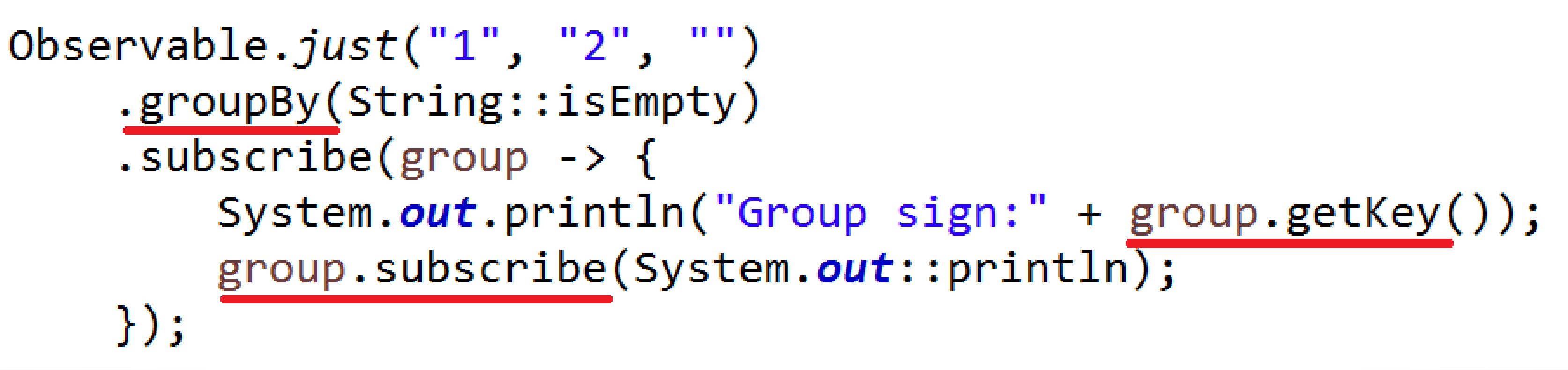


```
Observable.just(Arrays.asList("1", "2", "3"))
        .flatMap(items -> Observable.fromIterable(items))
        .map(String::length)
        .subscribe(System.out::println);
```

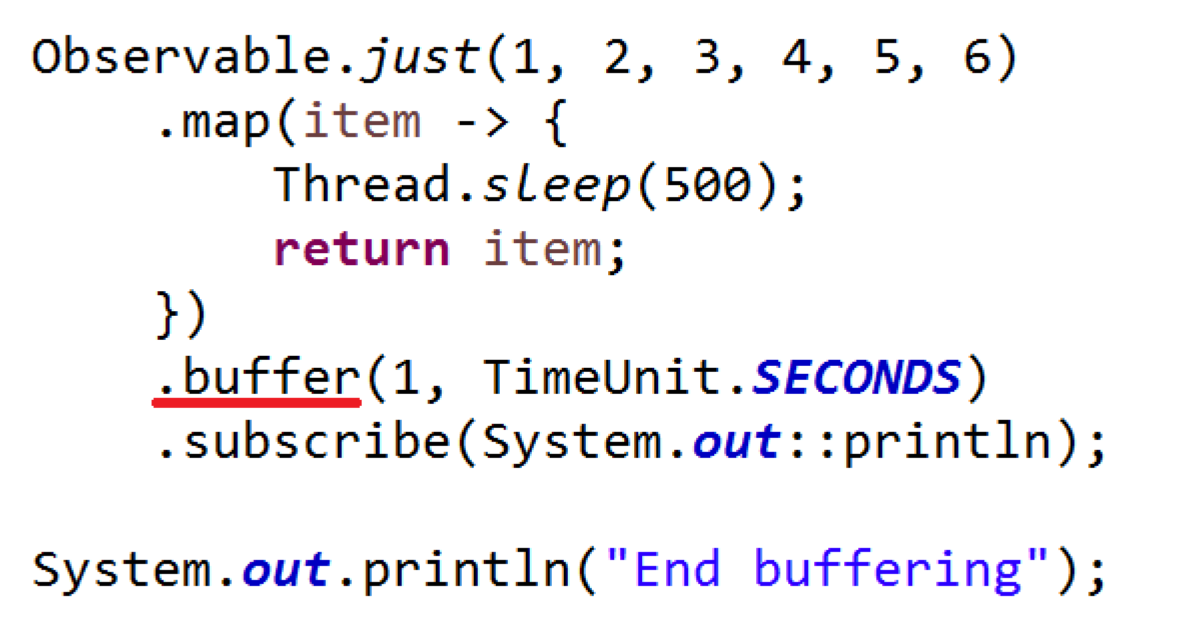


```
Observable.just(1, 2, 3)
        .scan(0,  (i1, i2) -> i1+ i2)
        .subscribe(System.out::println);
```

Observable transformation



```
Observable.just("1", "2", "")
    .groupBy(String::isEmpty)
    .subscribe(group -> {
        System.out.println("Group sign:" + group.getKey());
        group.subscribe(System.out::println);
    });
```

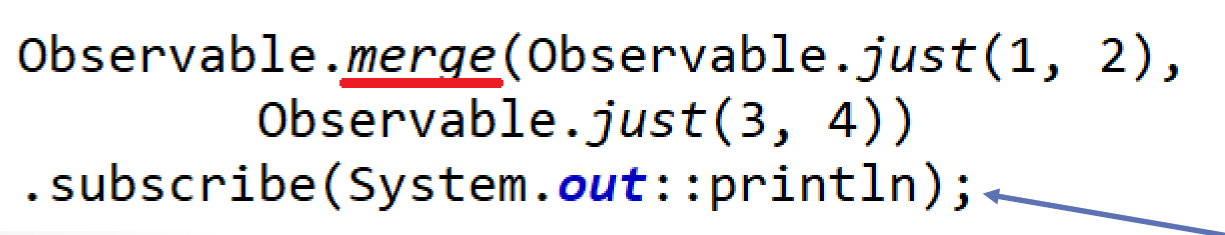


```
Observable.just(1, 2, 3, 4, 5, 6)
    .map(item -> {
        Thread.sleep(500);
        return item;
    })
    .buffer(1, TimeUnit.SECONDS)
    .subscribe(System.out::println);

System.out.println("End buffering");
```

Observable joining

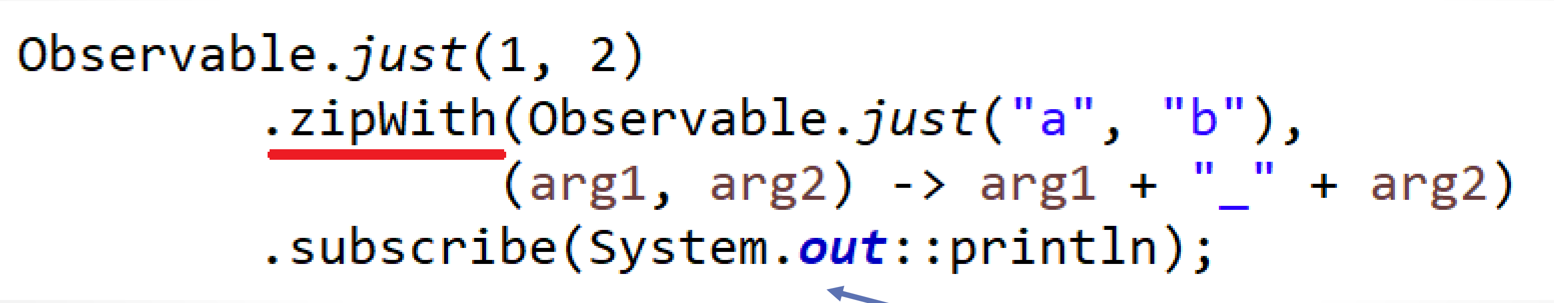
Method	Description
join()	Combine the overlapping items of two Observables
combineLatest()	Combine the latest items of the multiple Observables using given function
merge()	Combine multiple Observables into one stream
mergeWith()	Combine second Observable with first Observable only when the latter complete
zip()	Combines two or more Observables together using given function

Observable joining



```
Observable.merge(Observable.just(1, 2),
        Observable.just(3, 4))
.subscribe(System.out::println);
```

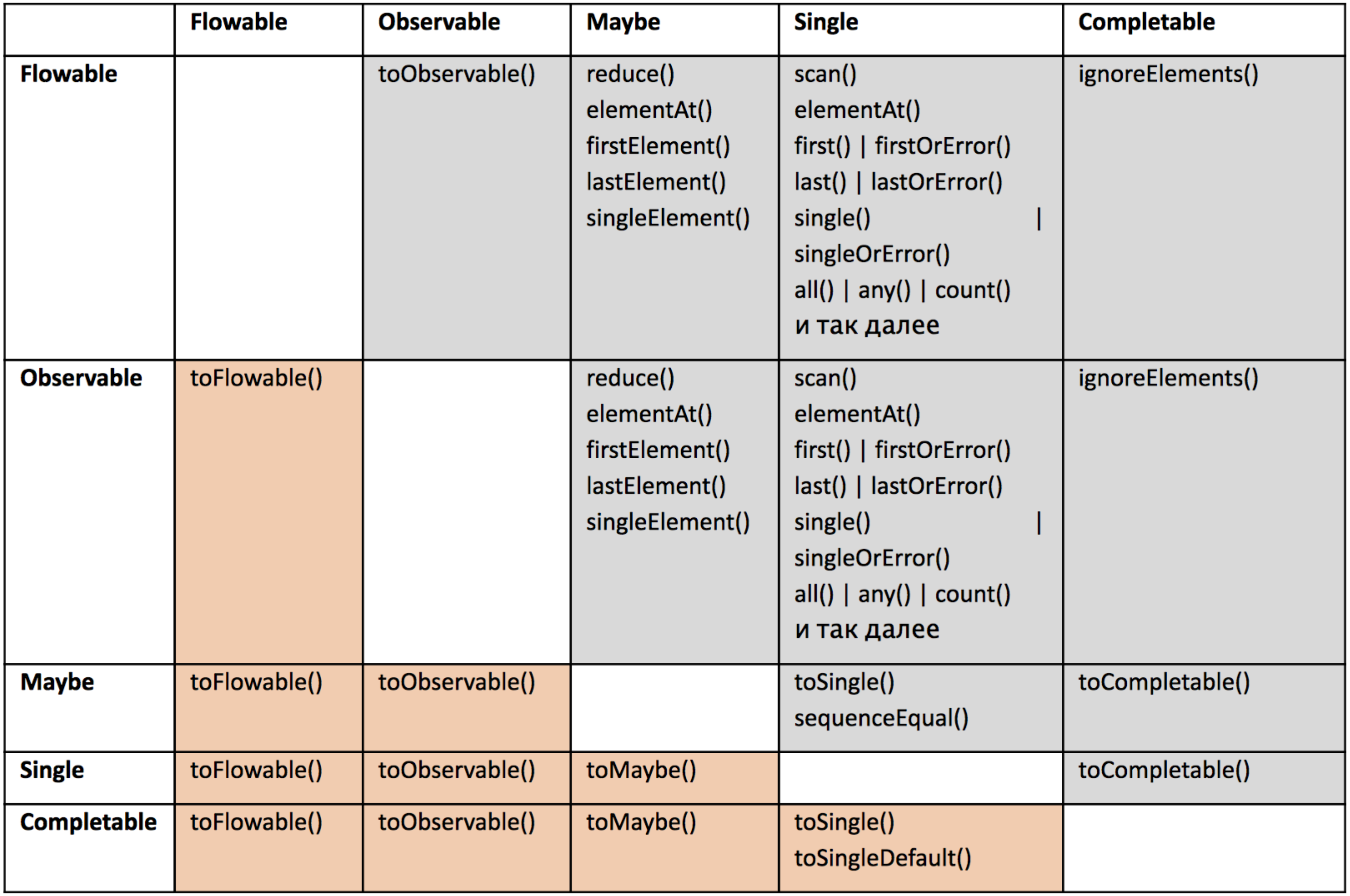
Prints 1,2,3,4



```
Observable.just(1, 2)
        .zipWith(Observable.just("a", "b"),
                (arg1, arg2) -> arg1 + "_" + arg2)
        .subscribe(System.out::println);
```

Prints 1_a,2_b

RxJava types. Conversion



	Flowable	Observable	Maybe	Single	Completable

Flowable		toObservable()	reduce()
			elementAt()
			firstElement()
			lastElement()
			singleElement()
				scan()
				elementAt()
				first() | firstOrError()
				last() | lastOrError()
				single() | singleOrError()
				all() | any() | count()
				и так далее
					ignoreElements()

Observable	toFlowable()		reduce()
			elementAt()
			firstElement()
			lastElement()
			singleElement()
				scan()
				elementAt()
				first() | firstOrError()
				last() | lastOrError()
				single() | singleOrError()
				all() | any() | count()
				и так далее
					ignoreElements()

Maybe	toFlowable()	toObservable()		toSingle()
				sequenceEqual()
					toCompletable()

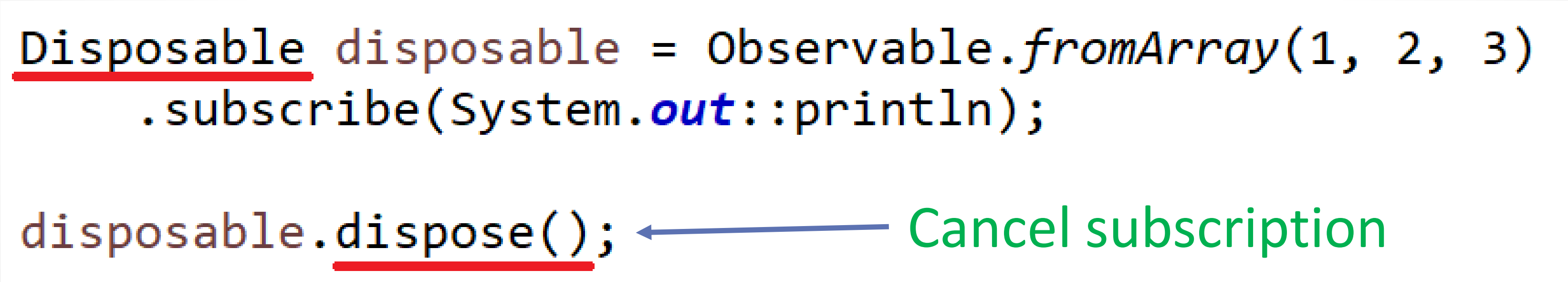
Single	toFlowable()	toObservable()	toMaybe()		toCompletable()

Completable	toFlowable()	toObservable()	toMaybe()	toSingle()
				toSingleDefault()	

Disposable

- ✓ Subscription result
- ✓ Enables to cancel subsription

Method	Description
dispose()	Cancels current task
isDisposed()	Returns true if current task/resource is disposed



```
Disposable disposable = Observable.fromArray(1, 2, 3)
        .subscribe(System.out::println);

disposable.dispose();
```

Cancel subscription

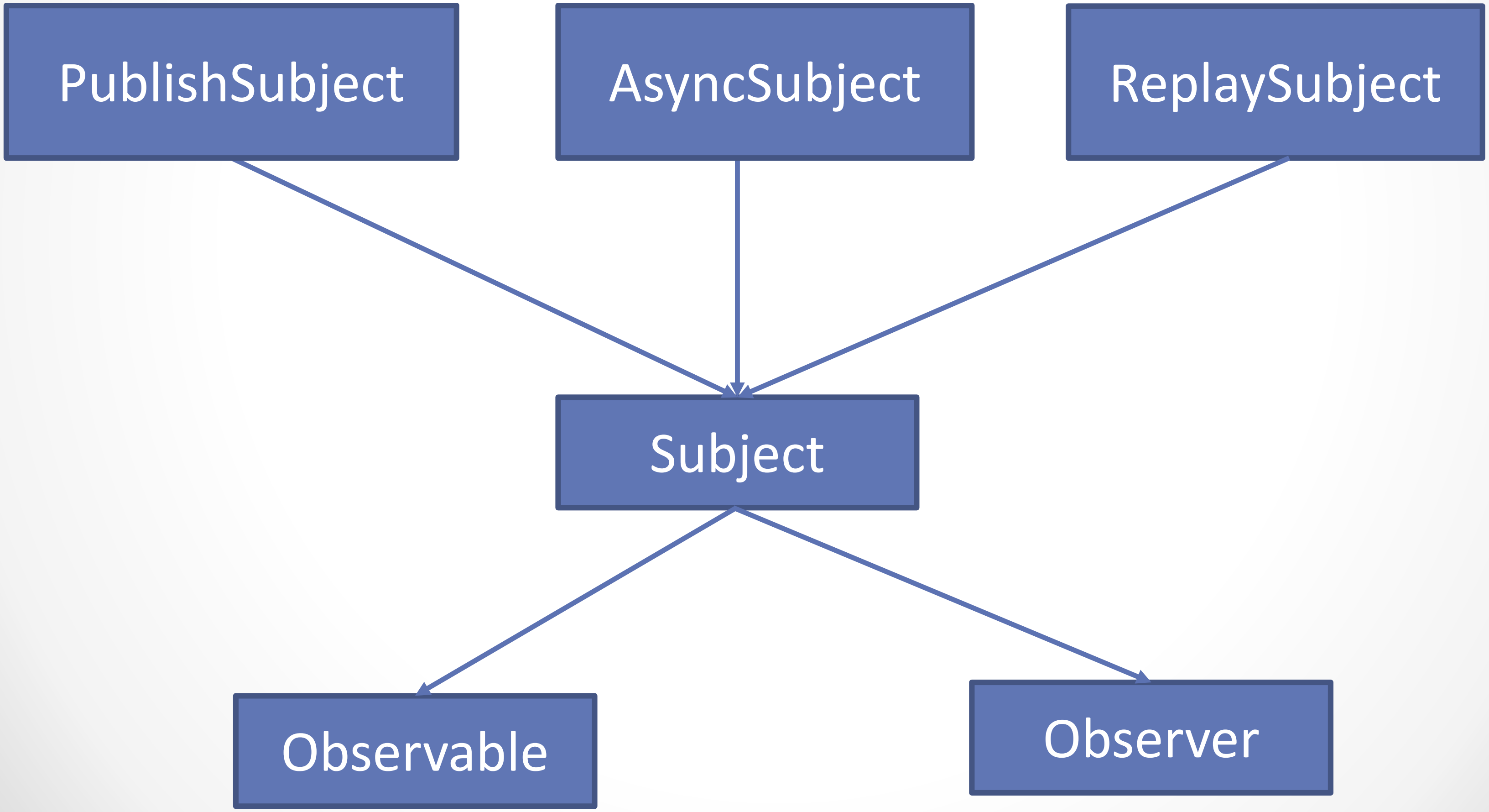
Task #4. Introduction into RxJava 2

1. Add **RxJava 2** dependency to your project:
2. Create new class **RxJavaStarter** with main() method and try to create Observable object in different ways and provide various intermediate operators.
3. Try to cancel subscription
4. Try joining multiple Observables.
5. Try to create Observable that emits latin lower-case characters.

Task #5. Using Observable

1. Change MealRepository so that its **getMeal** method returns Observable.
2. Update CookService/WaiterService so that it uses Observable as well.
3. Modify **Starter** class and run it.

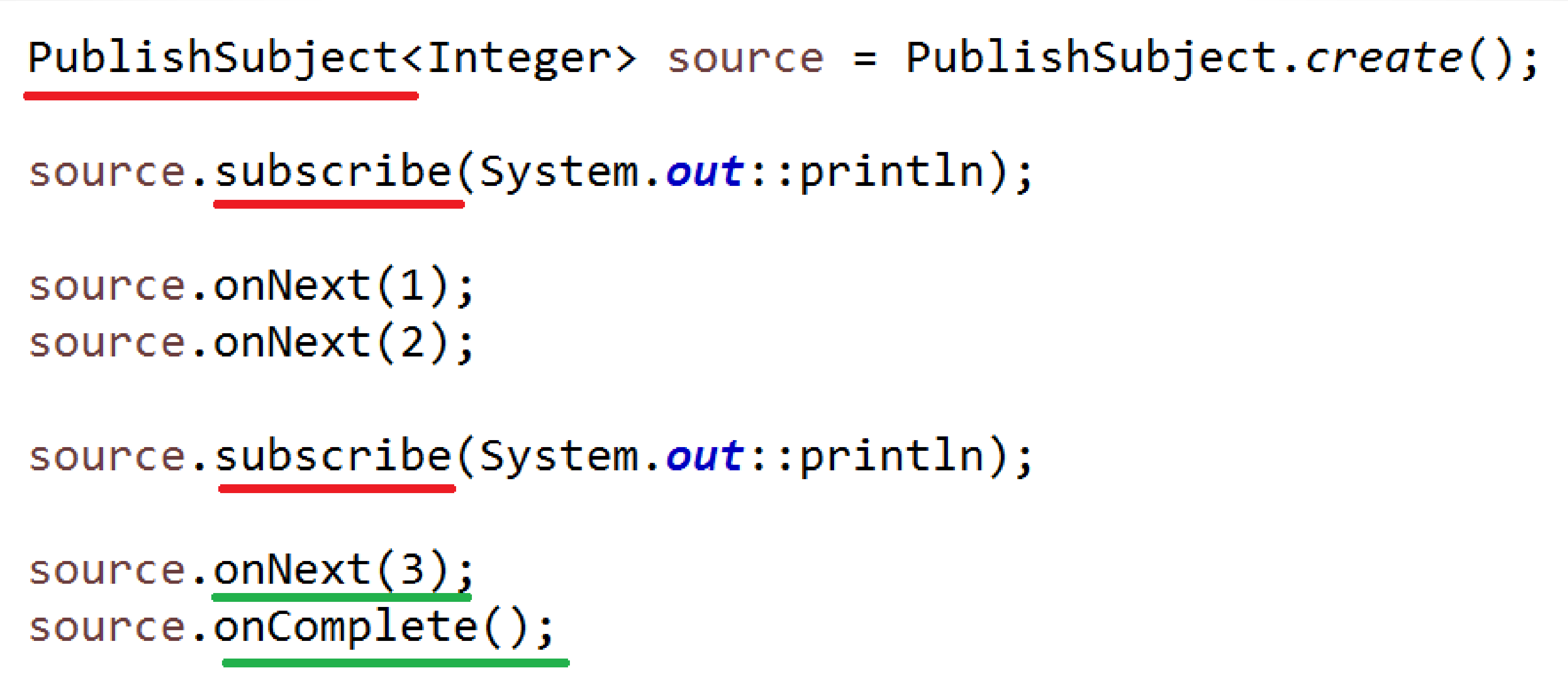
RxJava 2. Bridges



RxJava 2. Bridges

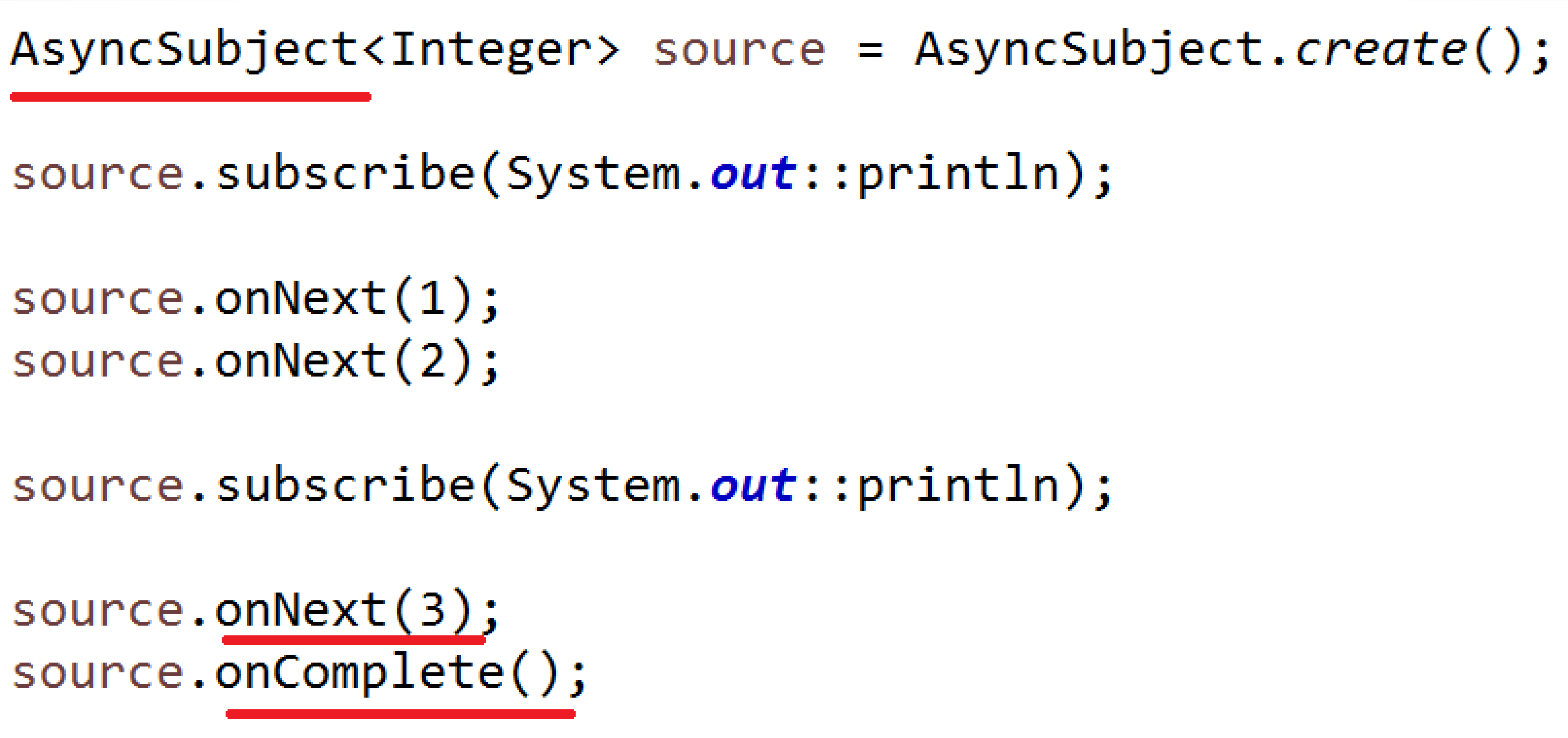
Class	Description
PublishSubject	Emit items to current subscribers and terminal events to current (and late) subscribers
AsyncSubject	Emits the last item and terminal events to subscribers
ReplaySubject	Replay items to current (and late) subscribers
BehaviorSubject	Emits most recent item and all later items to subscribers

RxJava 2. Bridges



```
PublishSubject<Integer> source = PublishSubject.create();

source.subscribe(System.out::println);

source.onNext(1);
source.onNext(2);

source.subscribe(System.out::println);

source.onNext(3);
source.onComplete();
```

RxJava 2. Bridges



```
AsyncSubject<Integer> source = AsyncSubject.create();

source.subscribe(System.out::println);

source.onNext(1);
source.onNext(2);

source.subscribe(System.out::println);

source.onNext(3);
source.onComplete();
```

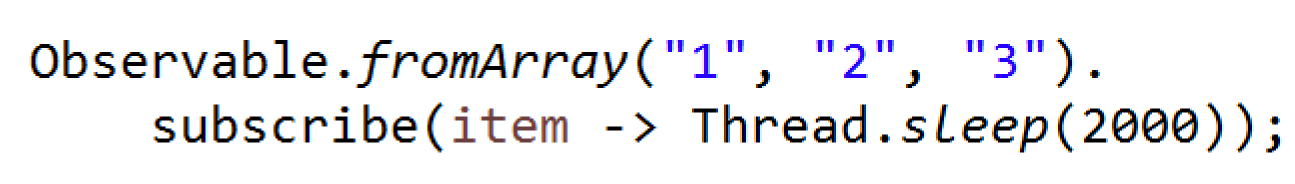
Asynchronous programming

Sergey Morenets, 2018

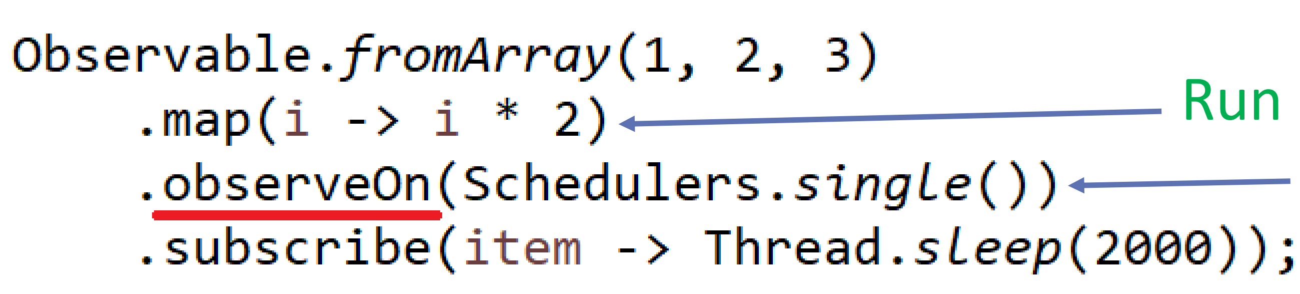
Schedulers

Method	Description
newThread()	Each unit of work will be executed in the new separate thread without further thread reusage
computation()	Each unit of work will be executed in the separate thread pool where number of threads are equal to number of cores. Not recommended for blocking operations
io()	Each unit of work will be executed in the separate reusable thread pool with infinite number of threads. Recommended for slow I/O blocking operations
single()	Each unit of work will be executed sequentially in the same background thread
shutdown()	Shutdown standard schedulers

Schedulers. Subscriptions



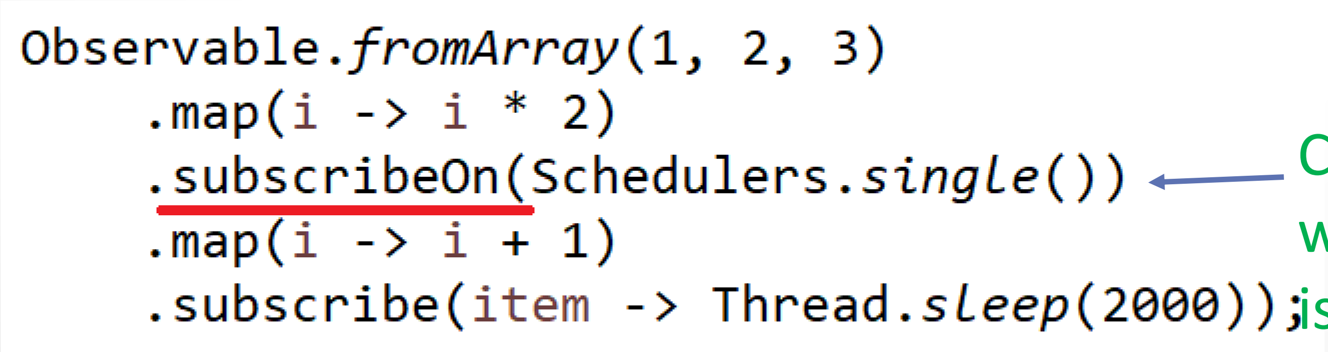
```
Observable.fromArray("1", "2", "3").
    subscribe(item -> Thread.sleep(2000));
```



```
Observable.fromArray(1, 2, 3)
    .map(i -> i * 2)
    .observeOn(Schedulers.single())
    .subscribe(item -> Thread.sleep(2000));
```

Run in main thread

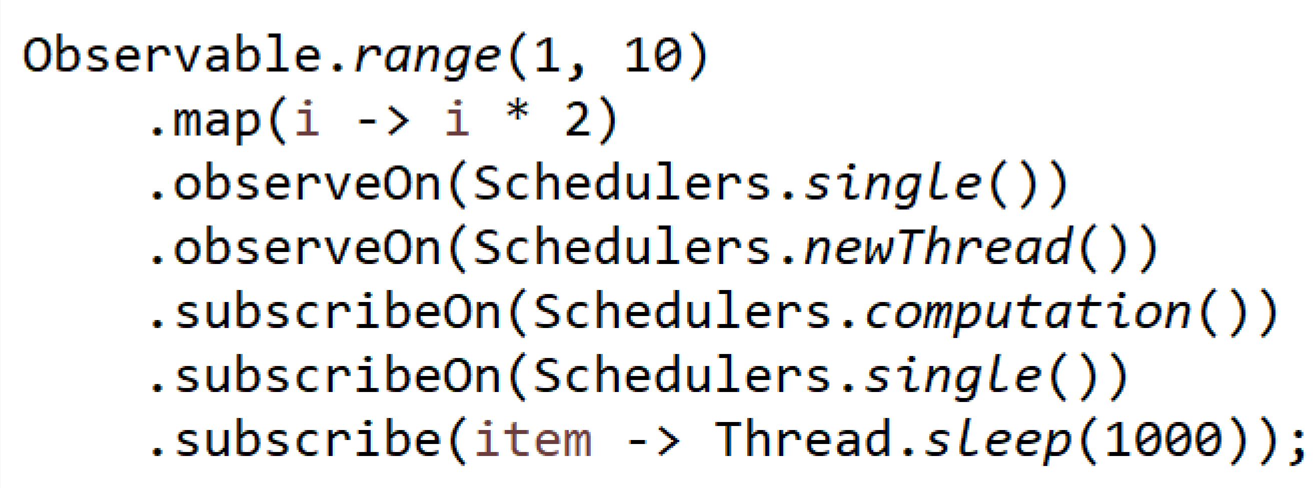
Change thread for downstream operators



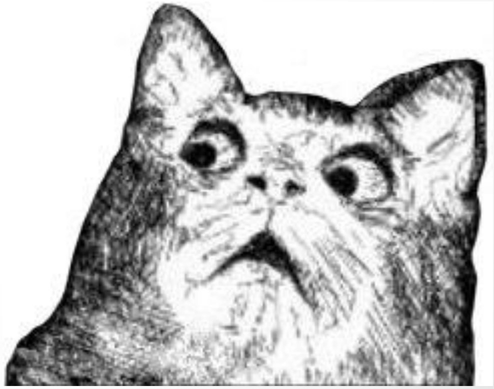
```
Observable.fromArray(1, 2, 3)
    .map(i -> i * 2)
    .subscribeOn(Schedulers.single())
    .map(i -> i + 1)
    .subscribe(item -> Thread.sleep(2000));
```

Change thread when Observable is subscribed

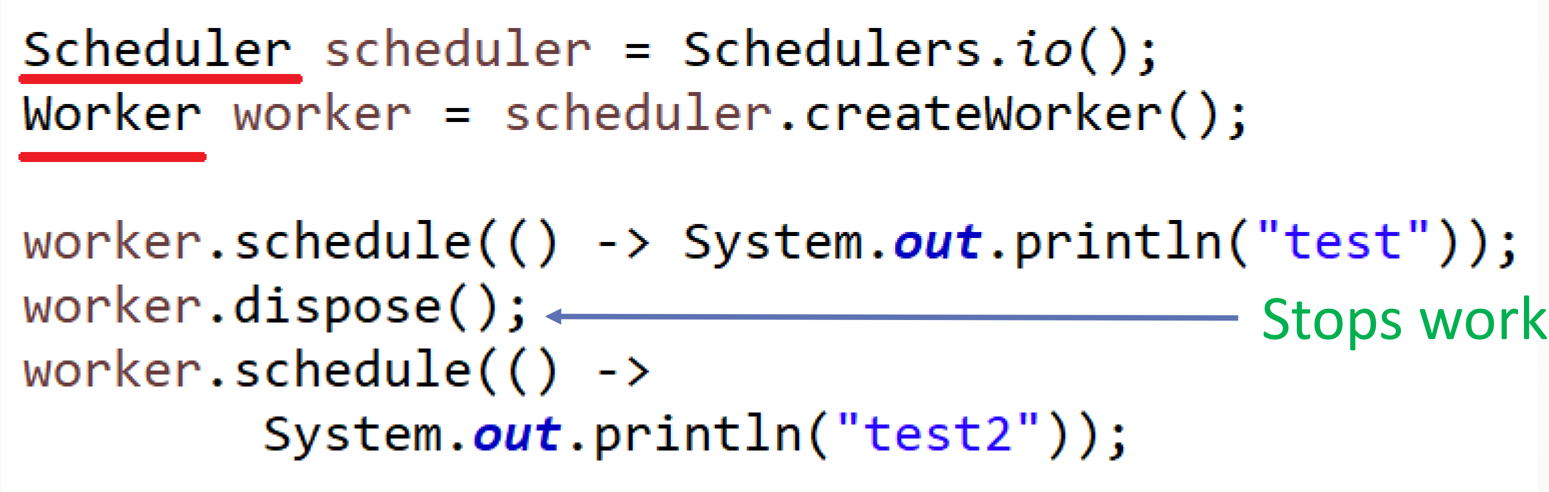
Schedulers. Quiz



```
Observable.range(1, 10)
    .map(i -> i * 2)
    .observeOn(Schedulers.single())
    .observeOn(Schedulers.newThread())
    .subscribeOn(Schedulers.computation())
    .subscribeOn(Schedulers.single())
    .subscribe(item -> Thread.sleep(1000));
```



Schedulers API

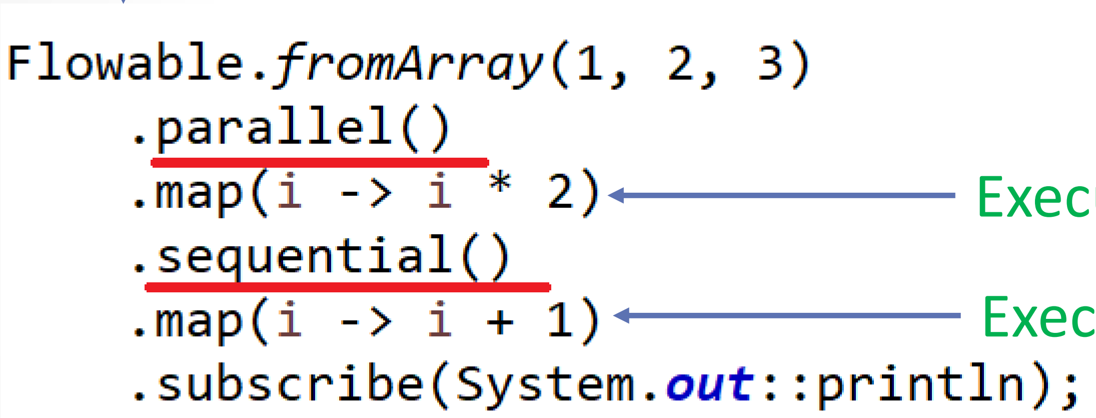


```
Scheduler scheduler = Schedulers.io();
Worker worker = scheduler.createWorker();

worker.schedule(() -> System.out.println("test"));
worker.dispose();
worker.schedule(() ->
        System.out.println("test2"));
```

Stops worker

Parallel execution

New type in RxJava 2.x



```
Flowable.fromArray(1, 2, 3)
    .parallel()
    .map(i -> i * 2)
    .sequential()
    .map(i -> i + 1)
    .subscribe(System.out::println);
```

Executed in parallel

Executed sequentially

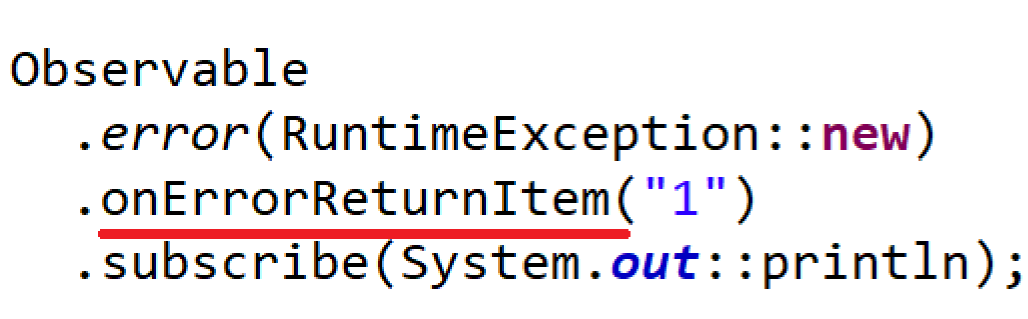
Task #6. Schedulers& Subject

1. Try to create different types of **Subject's** in RxJavaStarter class and observe their behaviors.
2. Try to use different types of **Scheduler's** when emitting items.
3. Try to use parallel/sequential processing in **Flowable**.
4. How would you change CookService/WaiterService so that items were processed **asynchronously**?

Observable. Error handling

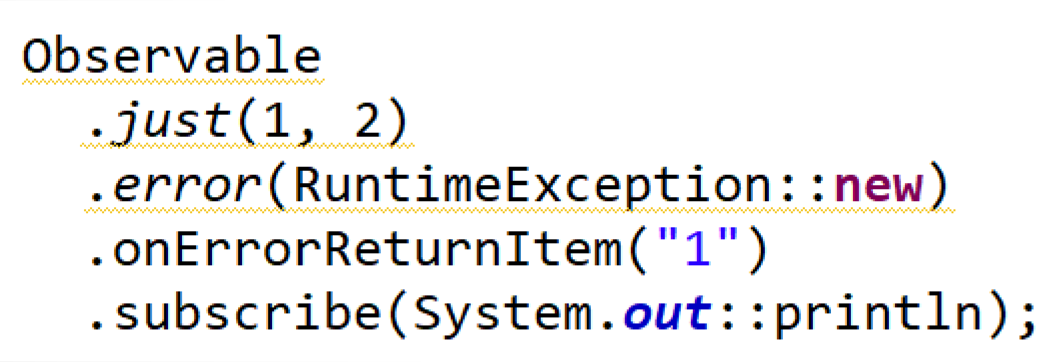
Method	Description
onErrorReturnItem(T)	Instructs Observable to return given item in case of error and don't invoke onError
onErrorReturn(Function)	Instructs Observable to return result of the function execution in case of error and don't invoke onError
onErrorResumeNext	Invokes another Observable in case of error
onExceptionResumeNext	Invokes another Observable in case of Exception
doOnError(Consumer)	Invokes an action if Observable emits onError signal

Observable. Error handling



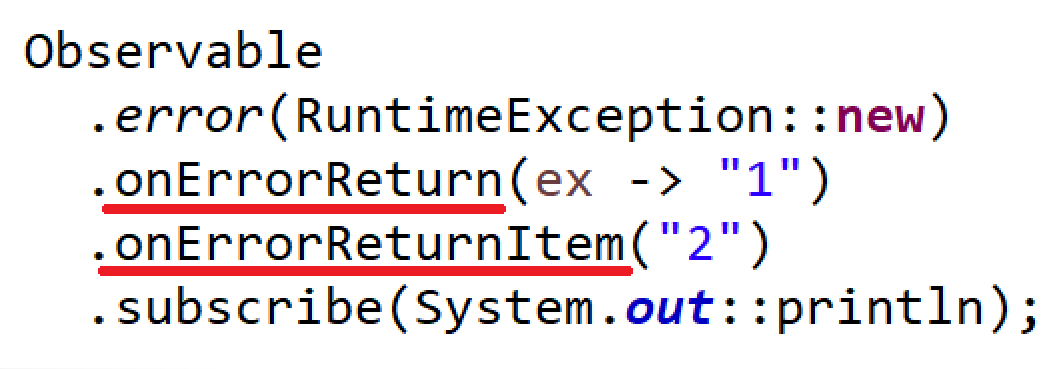
```
Observable
    .error(RuntimeException::new)
    .onErrorReturnItem("1")
    .subscribe(System.out::println);
```

Prints 1



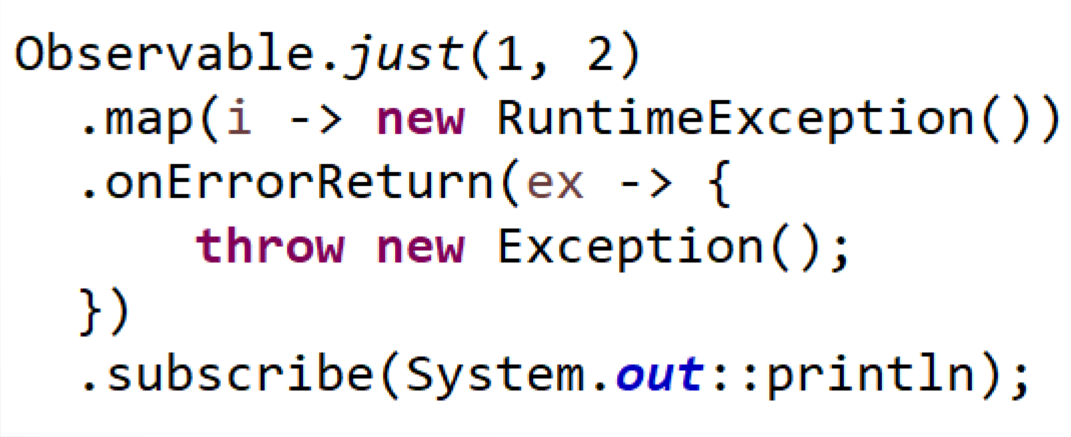
```
Observable
    .just(1, 2)
    .error(RuntimeException::new)
    .onErrorReturnItem("1")
    .subscribe(System.out::println);
```

Observable. Error handling



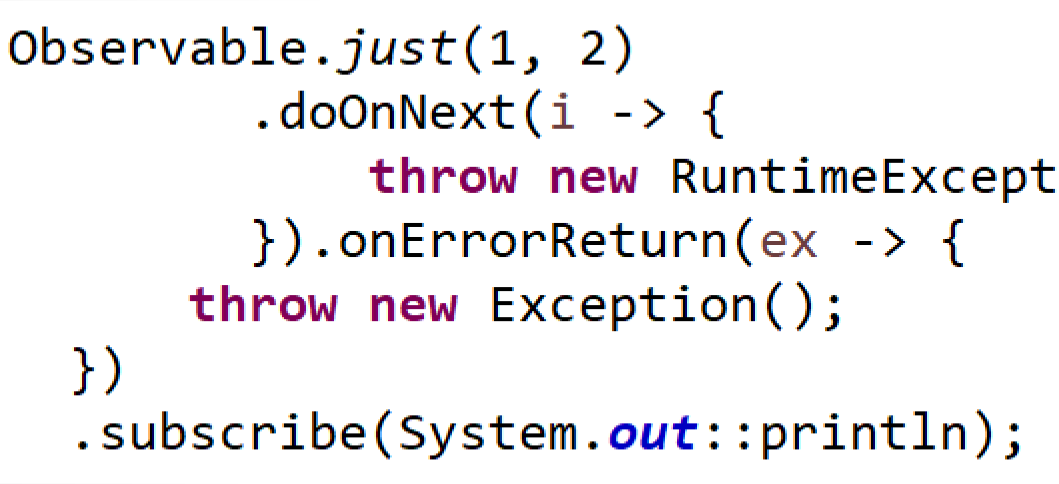
```
Observable
   .error(RuntimeException::new)
   .onErrorReturn(ex -> "1")
   .onErrorReturnItem("2")
   .subscribe(System.out::println);
```

Prints 1



```
Observable.just(1, 2)
   .map(i -> new RuntimeException())
   .onErrorReturn(ex -> {
        throw new Exception();
   })
   .subscribe(System.out::println);
```

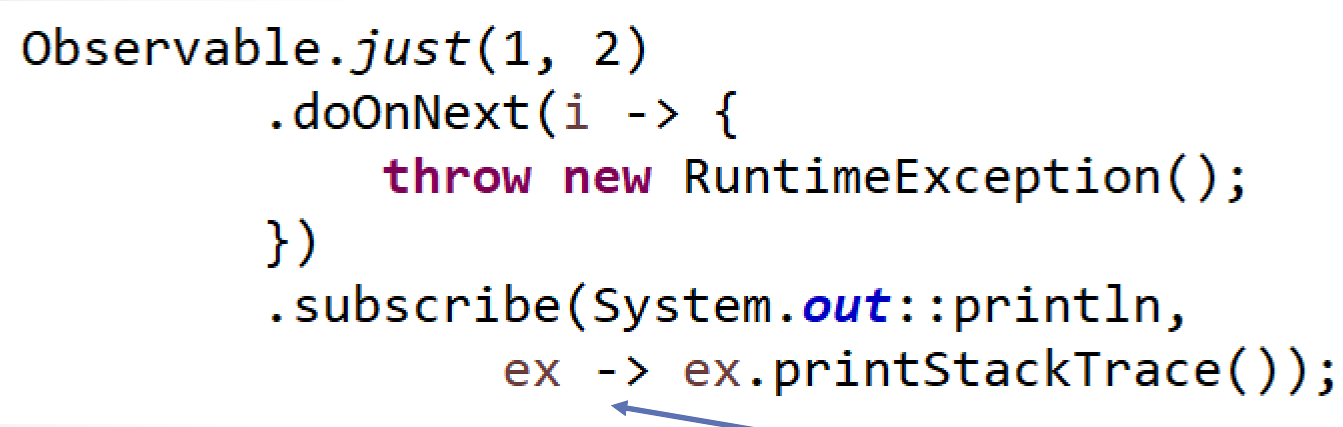
Observable. Error handling



```java
Observable.just(1, 2)
        .doOnNext(i -> {
            throw new RuntimeException();
        }).onErrorReturn(ex -> {
      throw new Exception();
  })
  .subscribe(System.out::println);
```

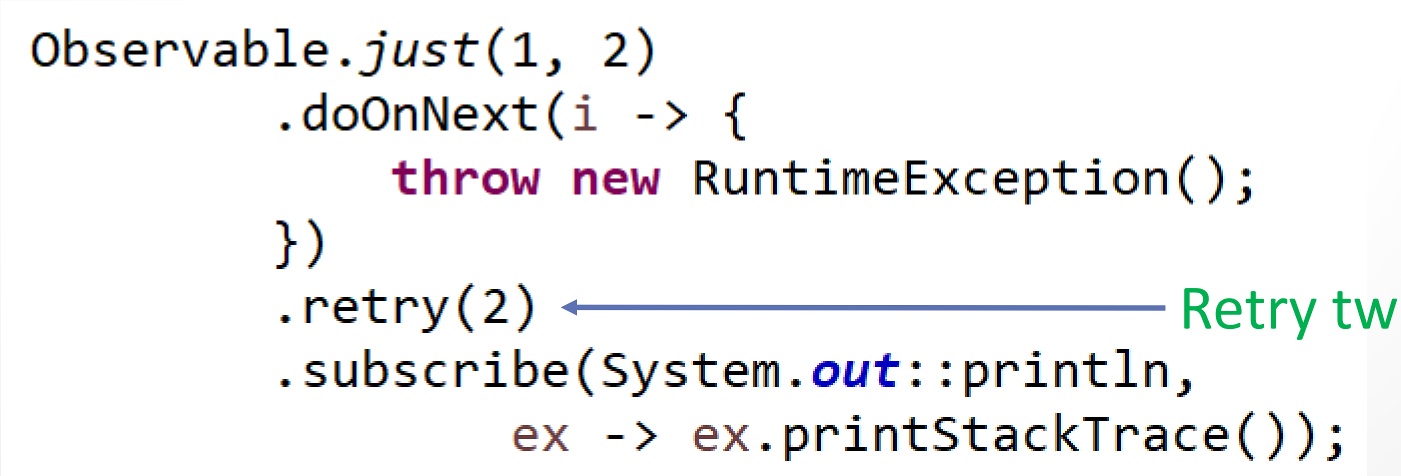
```
io.reactivex.exceptions.OnErrorNotImplementedException: The exception was not
handled due to missing onError handler in the subscribe() method call. Further
reading: https://github.com/ReactiveX/RxJava/wiki/Error-Handling | 2 exceptions
occurred.
        at
io.reactivex.internal.functions.Functions$OnErrorMissingConsumer.accept(Functio
ns.java:704)
```

Observable. Error handling



```
Observable.just(1, 2)
        .doOnNext(i -> {
            throw new RuntimeException();
        })
        .subscribe(System.out::println,
                ex -> ex.printStackTrace());
```

Default error handler



```
Observable.just(1, 2)
        .doOnNext(i -> {
            throw new RuntimeException();
        })
        .retry(2)
        .subscribe(System.out::println,
                ex -> ex.printStackTrace());
```

Retry twice if error

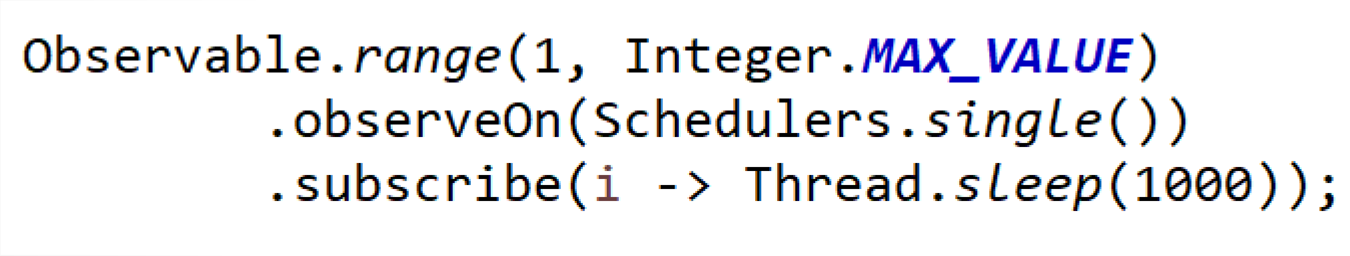
Task #7. Error handling

1. What are the possible errors in the current workflow?
2. Try to implement error handling in the repositories/services of the project.

Overloading and back-pressure

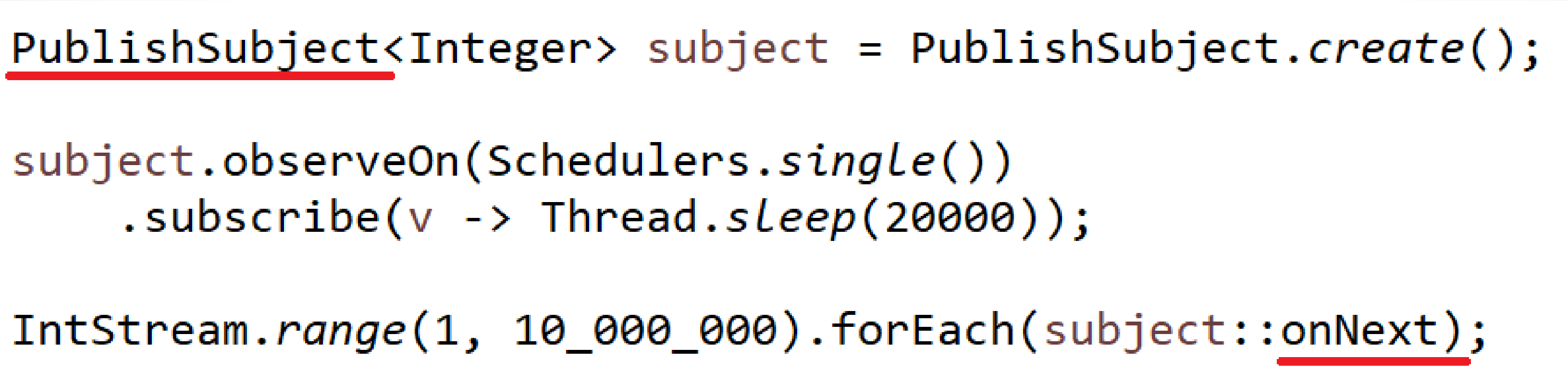
Overloading and back-pressure

Cold Observable



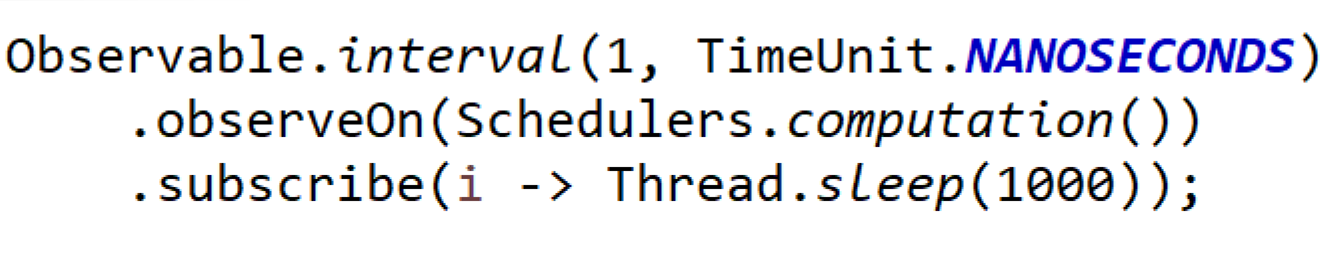
```
Observable.range(1, Integer.MAX_VALUE)
        .observeOn(Schedulers.single())
        .subscribe(i -> Thread.sleep(1000));
```

Hot Observable



```
PublishSubject<Integer> subject = PublishSubject.create();

subject.observeOn(Schedulers.single())
    .subscribe(v -> Thread.sleep(20000));

IntStream.range(1, 10_000_000).forEach(subject::onNext);
```

Overloading. RxJava 1.x

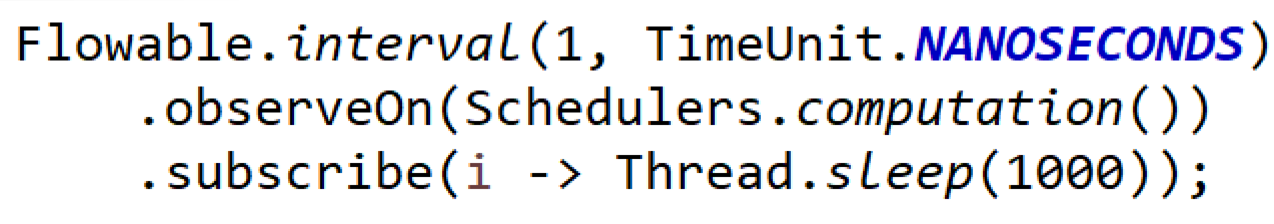


```
Observable.interval(1, TimeUnit.NANOSECONDS)
    .observeOn(Schedulers.computation())
    .subscribe(i -> Thread.sleep(1000));
```

Exception: java.lang.OutOfMemoryError thrown from the UncaughtExceptionHandler in thread "RxSchedulerPurge-1"

Exception: java.lang.OutOfMemoryError thrown from the UncaughtExceptionHandler in thread "RxCachedWorkerPoolEvictor-1"

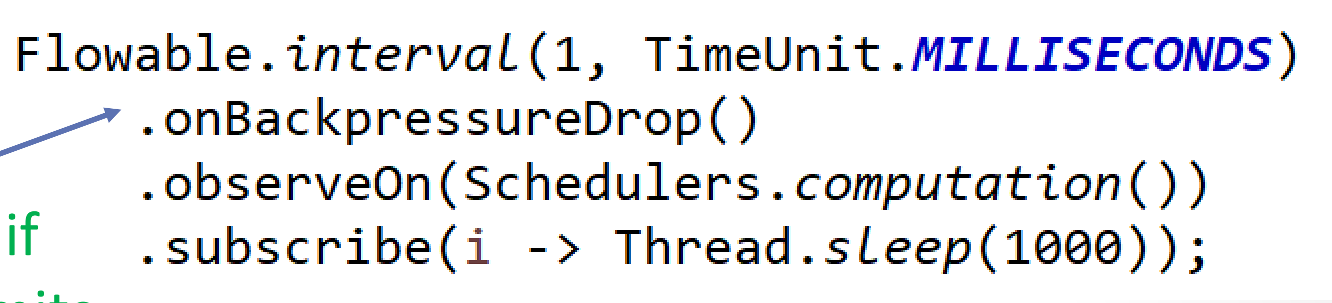
Overloading. RxJava 2.x



```
Flowable.interval(1, TimeUnit.NANOSECONDS)
     .observeOn(Schedulers.computation())
     .subscribe(i -> Thread.sleep(1000));
```

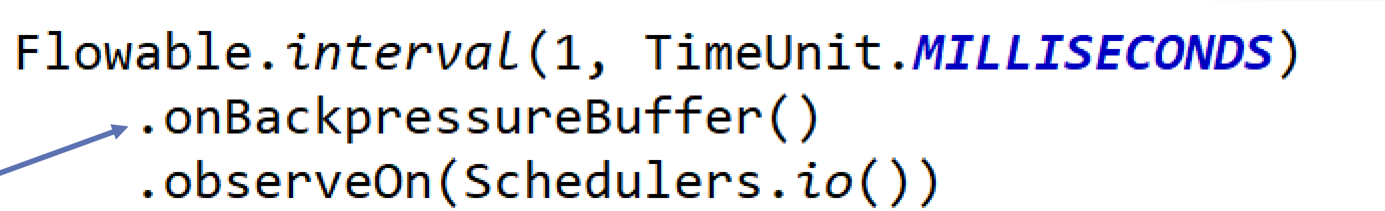
```
Caused by: io.reactivex.exceptions.MissingBackpressureException: Can't deliver value 128 due to lack of requests
	at io.reactivex.internal.operators.flowable.FlowableInterval$IntervalSubscriber.run(FlowableInterval.java:96)
```

Back-pressure. RxJava 2.x



```
Flowable.interval(1, TimeUnit.MILLISECONDS)
        .onBackpressureDrop()
        .observeOn(Schedulers.computation())
        .subscribe(i -> Thread.sleep(1000));
```

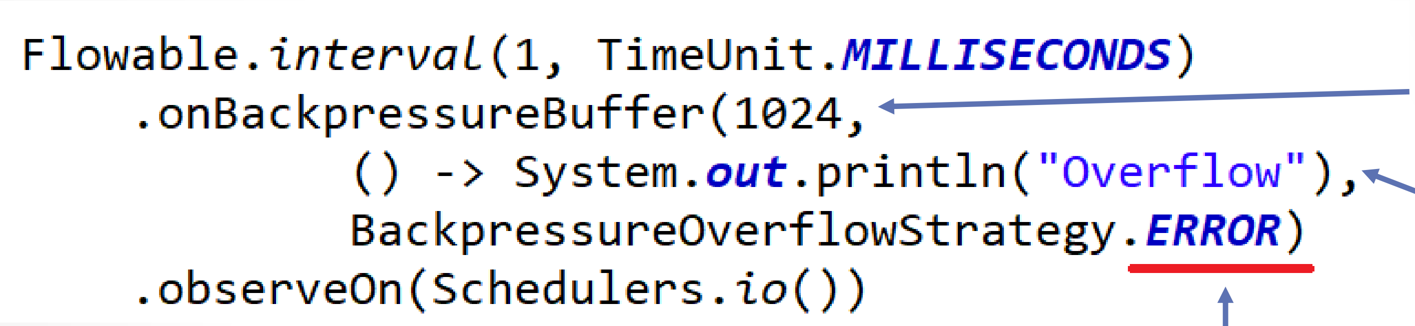
Discard items if
Observable emits
Items faster than observer can process



```
Flowable.interval(1, TimeUnit.MILLISECONDS)
        .onBackpressureBuffer()
        .observeOn(Schedulers.io())
```

Buffer items if
Observable emits
Items faster than observer can process

Back-pressure. RxJava 2.x

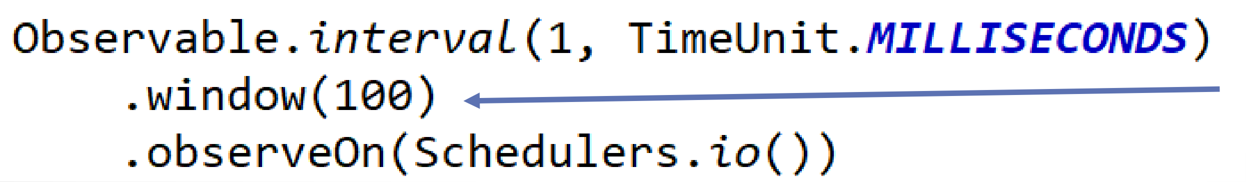


```
Flowable.interval(1, TimeUnit.MILLISECONDS)
    .onBackpressureBuffer(1024,
            () -> System.out.println("Overflow"),
            BackpressureOverflowStrategy.ERROR)
    .observeOn(Schedulers.io())
```

Buffer size

Action on buffer overflow

Also DROP_LATEST or DROP_OLDEST



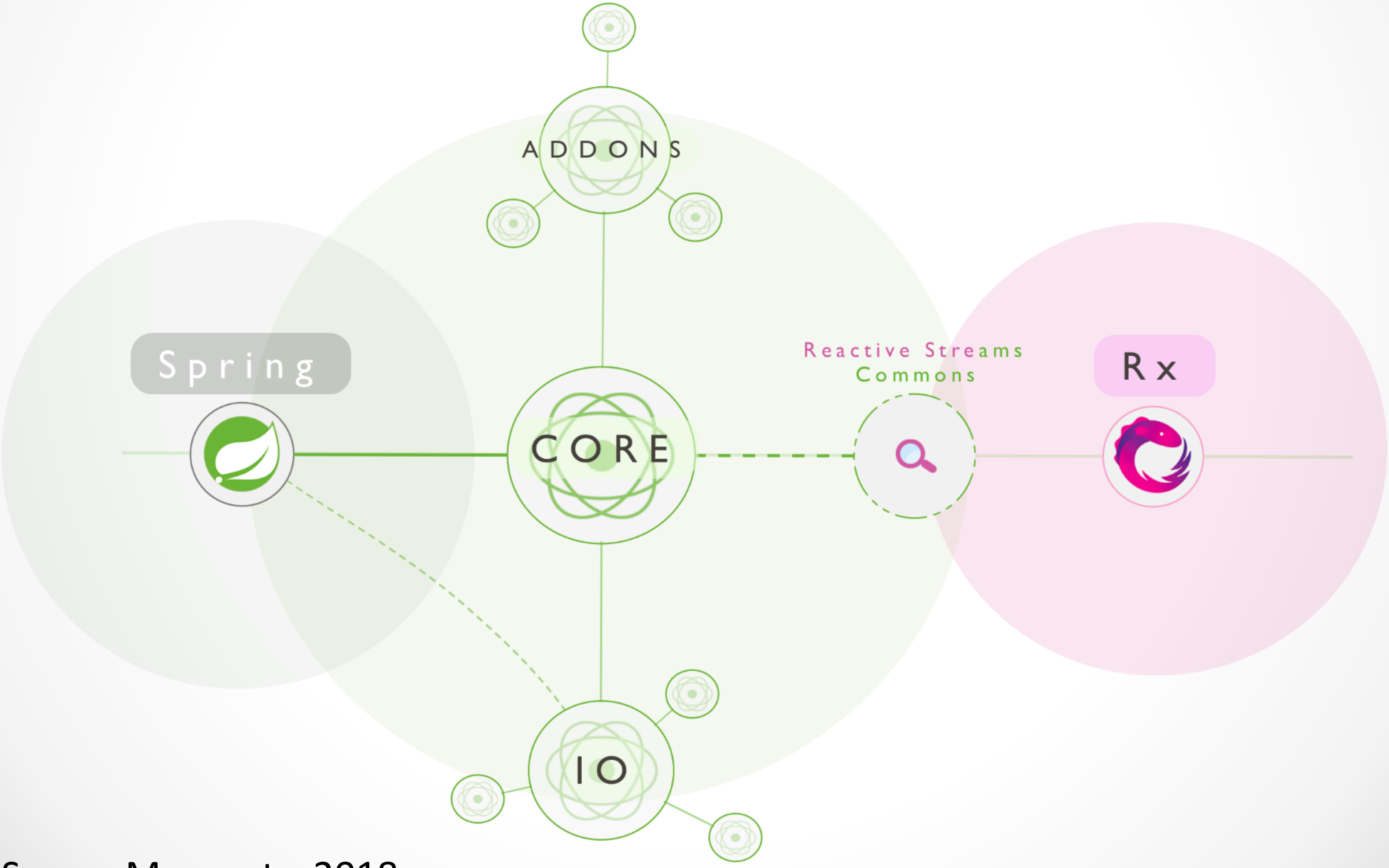
```
Observable.interval(1, TimeUnit.MILLISECONDS)
    .window(100)
    .observeOn(Schedulers.io())
```

Group items in batch

Task #8. Back-pressure

1. What is available places of overloading in the project? What are suitable back-pressure strategies?
2. Try to apply back-pressure in the project workflow.

Projector Reactor

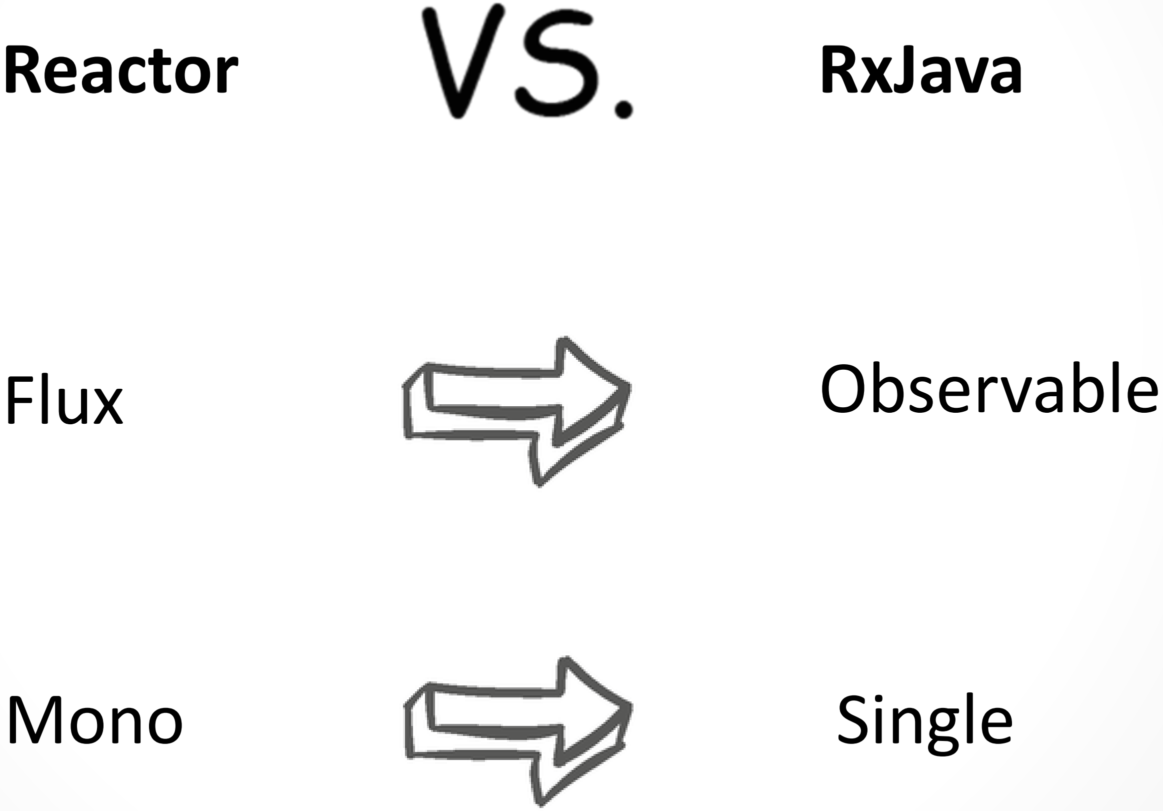


- Sergey Morenets, 2018

Project Reactor

- ✓ Foundational framework for asynchronous applications
- ✓ Developed by Pivotal, Inc since 2013
- ✓ Inspired by **Reactor** pattern
- ✓ Main contributor is Stephane Maldini
- ✓ High-performance platform with Netty/Kafka support

Types. Reactor vs RxJava

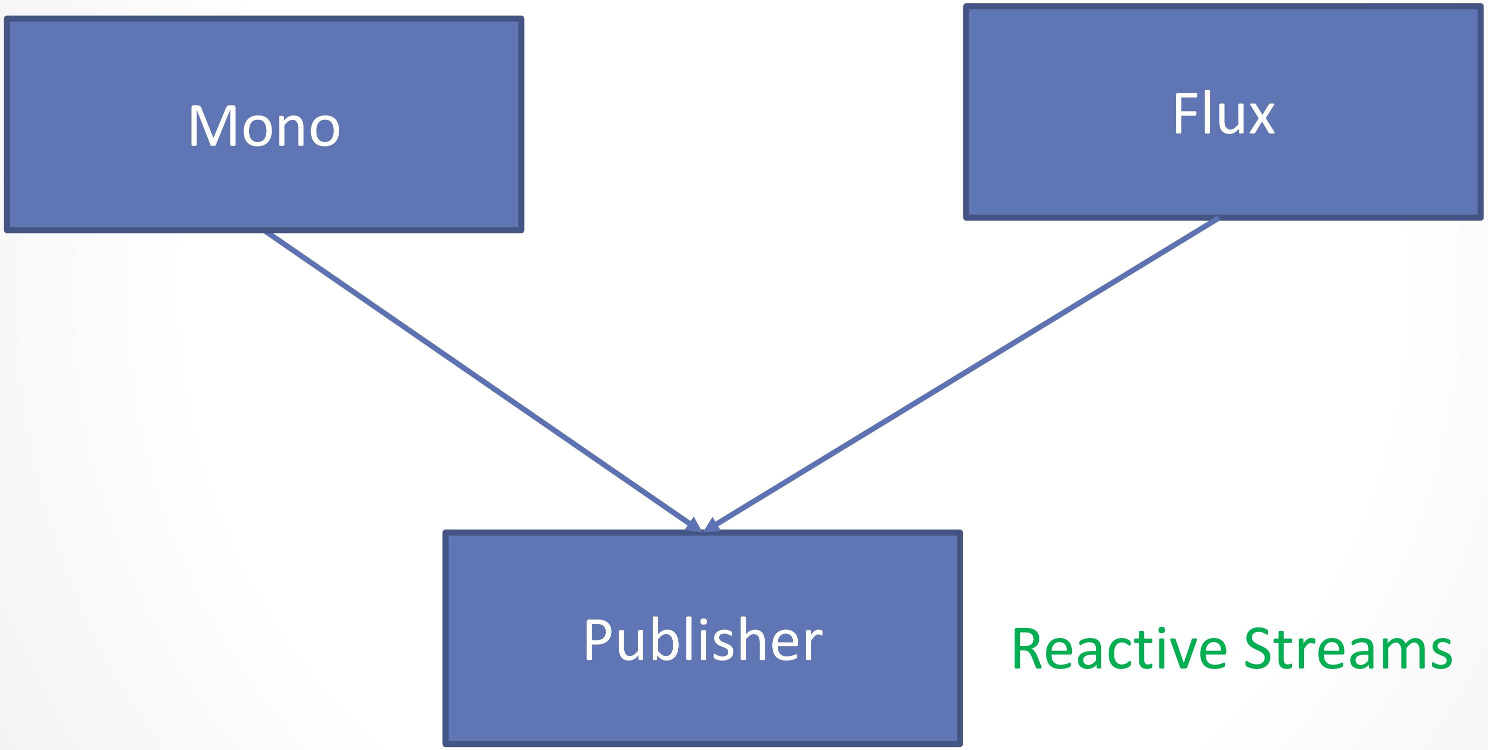


Reactor VS. **RxJava**

Flux → Observable

Mono → Single

Types. Reactor



Reactive Streams

Reactor. Maven dependencies

```xml
<dependency>
    <groupId>io.projectreactor</groupId>
    <artifactId>reactor-core</artifactId>
    <version>3.2.0.RELEASE</version>
</dependency>
```

Mono. Generation

Method	Description
create(Consumer)	Returns Mono based on Consumer
empty()	Returns Mono that emits no items
error(Throwable)	Returns Mono that produces an error
from(Publisher)	Returns Mono based on given Publisher
just(T)	Returns Mono that emits specified item
justOrEmpty(T)	Returns Mono that emits specified item if it's not null
never()	Returns Mono that emits no signals
fromFuture	Returns Mono based on CompletableFuture
fromRunnable	Returns Mono that completes after Runnable has finished
zip	Returns Mono that aggregates Mono(s) into tuple

Mono transformation

Method	Description
delay(Duration)	Delays **onNext** by given duration
block()	Blocks this thread until Mono emits item or completes
cache()	Cache the last emitted item for the next subscribers
doFinally(Consumer)	Triggers Consumer when Mono terminates for any reason
doOnCancel	Invokes Runnable when Mono is cancelled
doOnNext	Triggers Consumer when next item is emitted
doOnError	Triggers Consumer then Mono completes with error
flux()	Converts this Mono into Flux stream
repeat	Repeatedly subscribes to the source
retry(long)	Re-subscribes to the source if it produces an error

Mono transformation

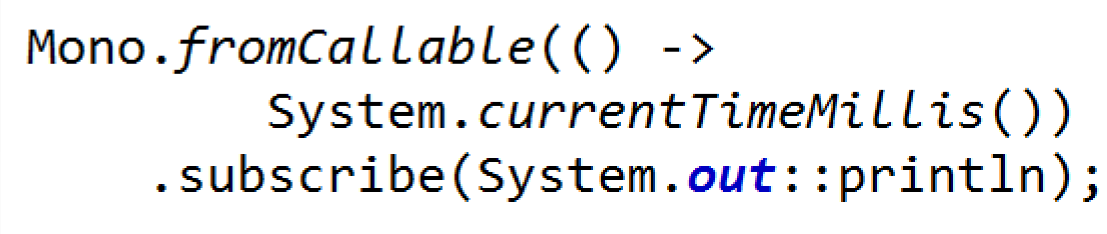
Method	Description
retryBackoff	Returns Mono that makes a retry based on backoff strategy only if error occurres
subscribe()	Subscribes to this source and returns Disposable
take(Duration)	Completes this Mono if no item was emitted during timeout
timeout(Duration)	Raises an error if no item was emitted during timeout
toFuture()	Converts this Mono into CompletableFuture
map(Function)	Converts emitted item synchronously using Function
filter(Predicate)	Emits item only if it Predicates returns true
onErrorReturn(T)	Emits given item in case of error
onErrorResume	Subscribes to fallback in case of error

Disposable

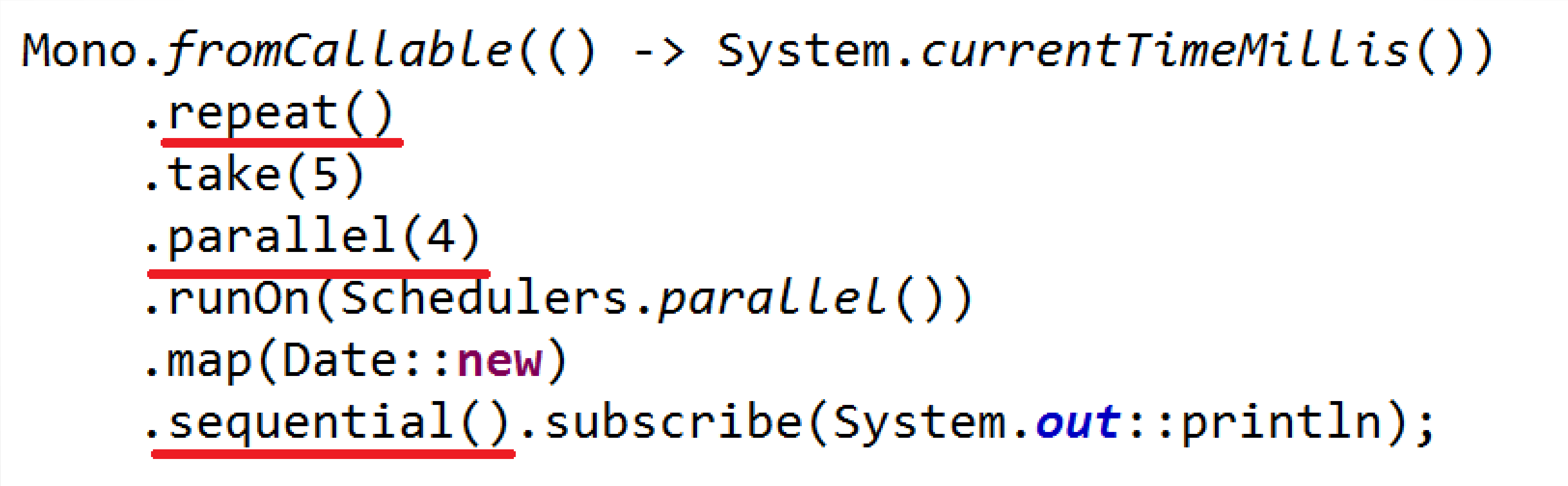
- ✓ Subscription result
- ✓ Enables to cancel subsription

Method	Description
dispose()	Cancels current task
isDisposed()	Returns true if current task/resource is disposed

Projector Reactor. Samples

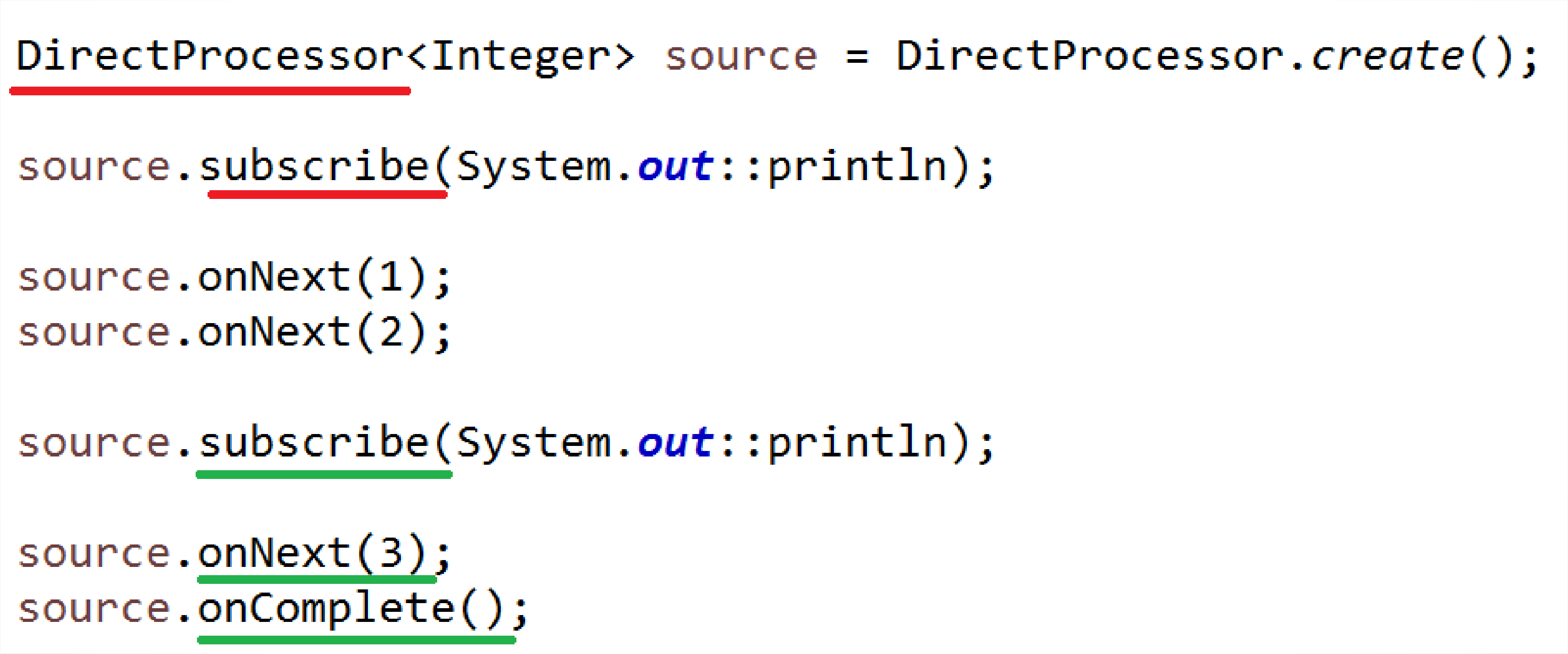


```
Mono.fromCallable(() ->
        System.currentTimeMillis())
    .subscribe(System.out::println);
```



```
Mono.fromCallable(() -> System.currentTimeMillis())
    .repeat()
    .take(5)
    .parallel(4)
    .runOn(Schedulers.parallel())
    .map(Date::new)
    .sequential().subscribe(System.out::println);
```

Projector Reactor. Bridges



```
DirectProcessor<Integer> source = DirectProcessor.create();

source.subscribe(System.out::println);

source.onNext(1);
source.onNext(2);

source.subscribe(System.out::println);

source.onNext(3);
source.onComplete();
```

Task #9. Project Reactor

1. Add new dependency to your project:
2. Create new class ReactorStarter and try to create new instances of types Mono/Flux/DirectProcessor. Try to apply intermediate operators.
3. Change RxJava 2.x types in services/repositories in the project with **Reactor** types.

Spring 5

- ✓ Java 9 and **Jigsaw** project support
- ✓ Java EE 7 support
- ✓ Servlet 4.0 and **HTTP/2** support(Tomcat/Jetty/Undertow)
- ✓ Junit 5 support
- ✓ **Kotlin** 1.0 support
- ✓ Dropped support for **Hibernate 3/4**, Velocity and Guava
- ✓ **Java 8** usage in core functionality
- ✓ **Reactive** architecture (Spring WebFlux)
- ✓ Current version 5.1

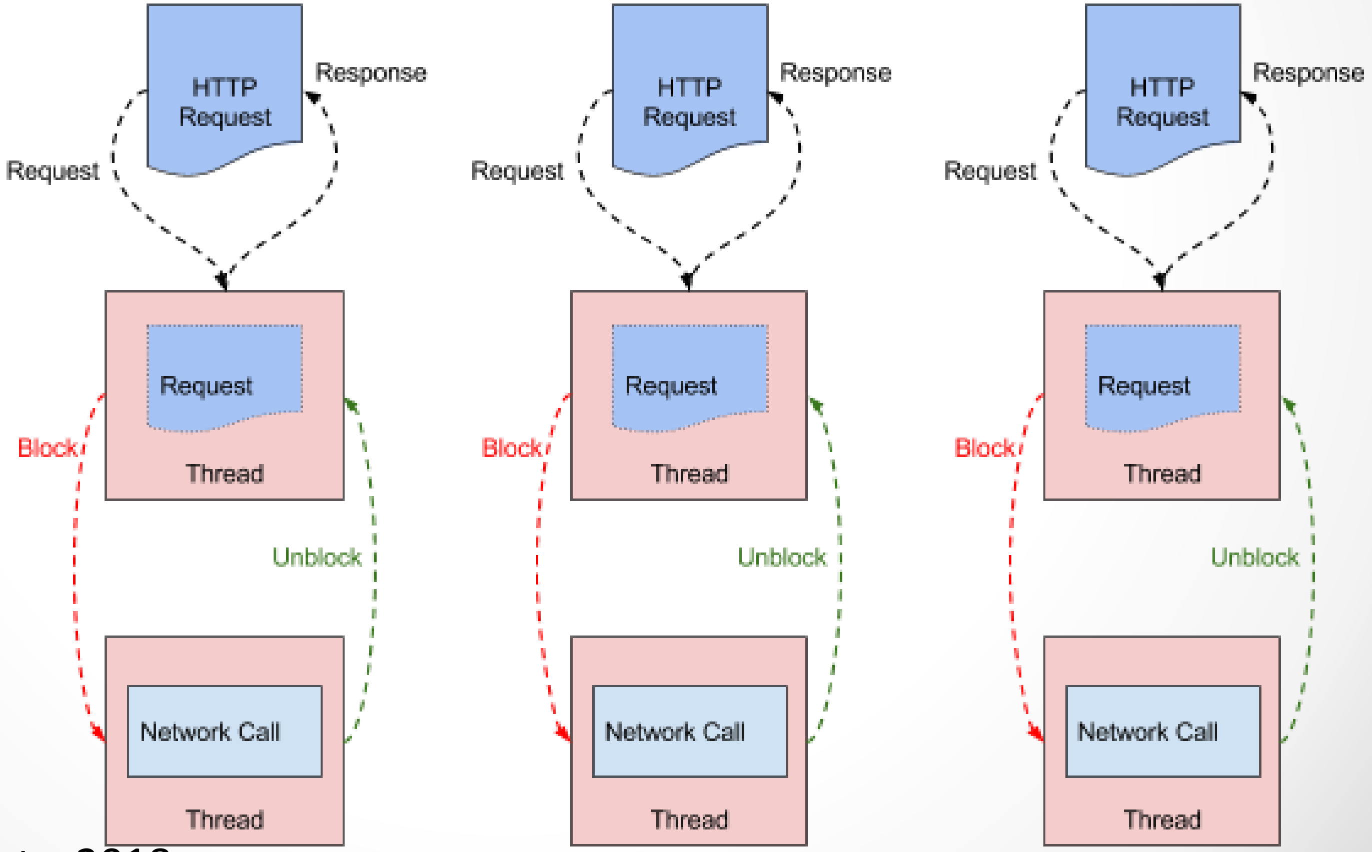
Reactive streams

- Reactive programming provides **asynchronous** data processing in **non-blocking** mode
- Affects **infrastructure**(web servers, database drivers, web frameworks)
- Reactive datastore (Postgres, **MongoDB**, Couchbase, Redis, Cassandra)
- Similar to **CompletableFuture** in Java 8

Think different.

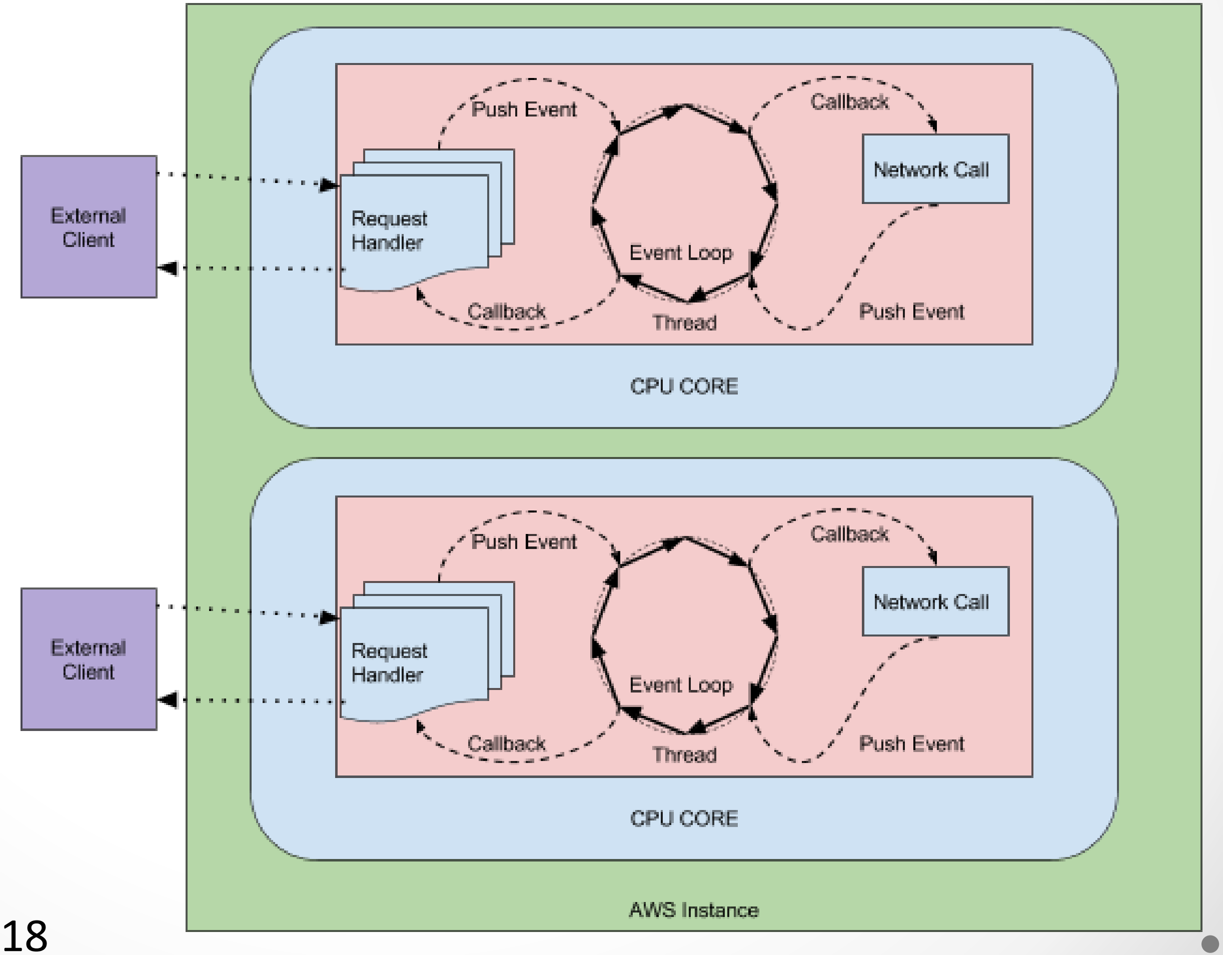
Think Reactive
Big Data... Big Returns...

Blocking I/O



- Sergey Morenets, 2018

Non-blocking I/O



- Sergey Morenets, 2018

Back-pressure

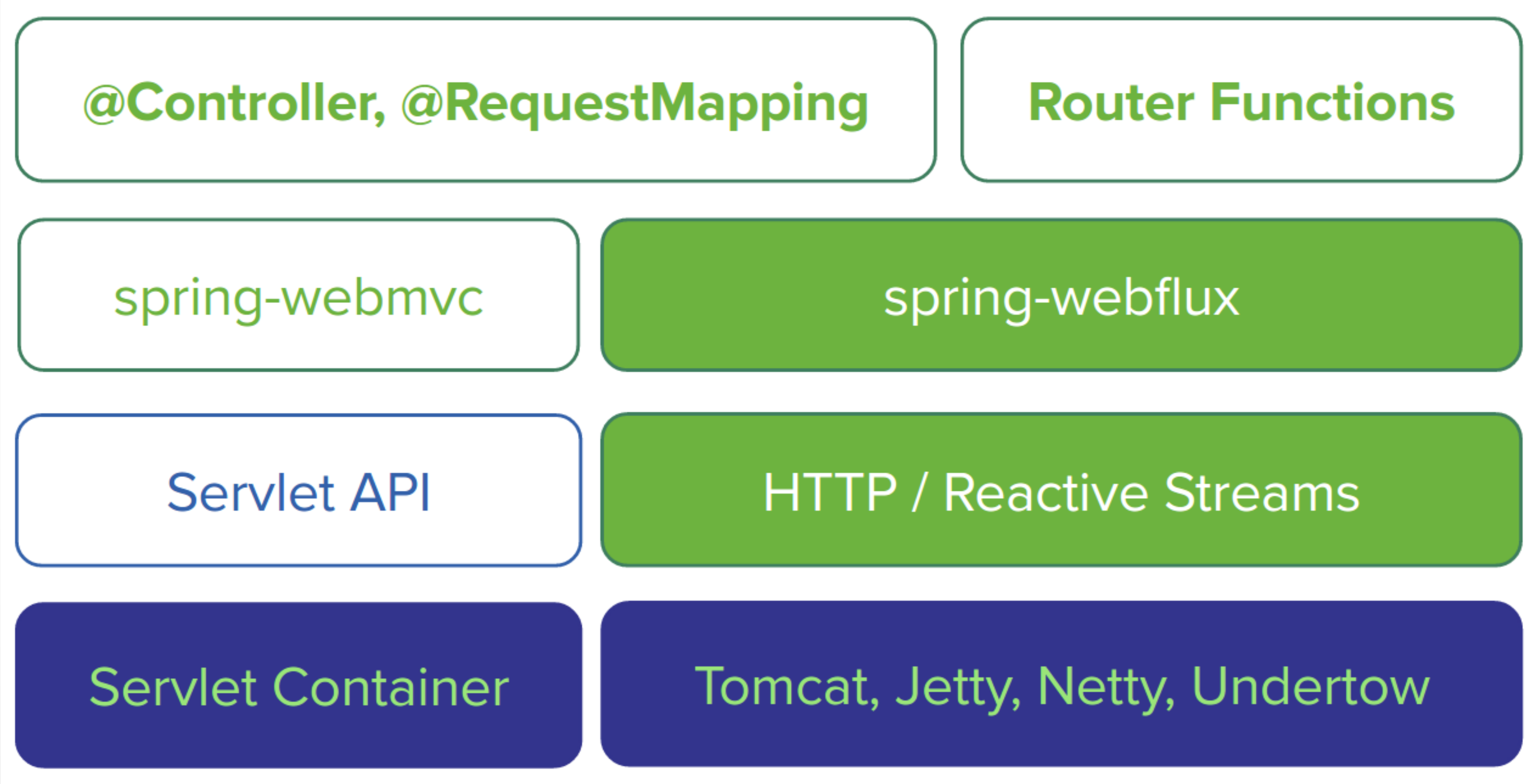
- ✓ Fast publisher shouldn’t overload slow consumer
- ✓ Solutions can be

1. Return back to pull model
2. Increase input buffer(queue)
3. Ignore elements

Spring 5 WebFlux

- New **spring-web-flux** module adapting **Reactive Streams** specification
- Based on **Project Reactor**
- Reusing Spring MVC programming model but in non-blocking mode
- Based on non-blocking Servlet API or native SPI connectors(Netty, Undertow)
- Supported by Tomcat/Jetty/Netty/Undertow
- Reactive client **WebClient**
- Support unit-testing with **WebTestClient**

Spring Web Flux



@Controller, @RequestMapping

Router Functions

spring-webmvc

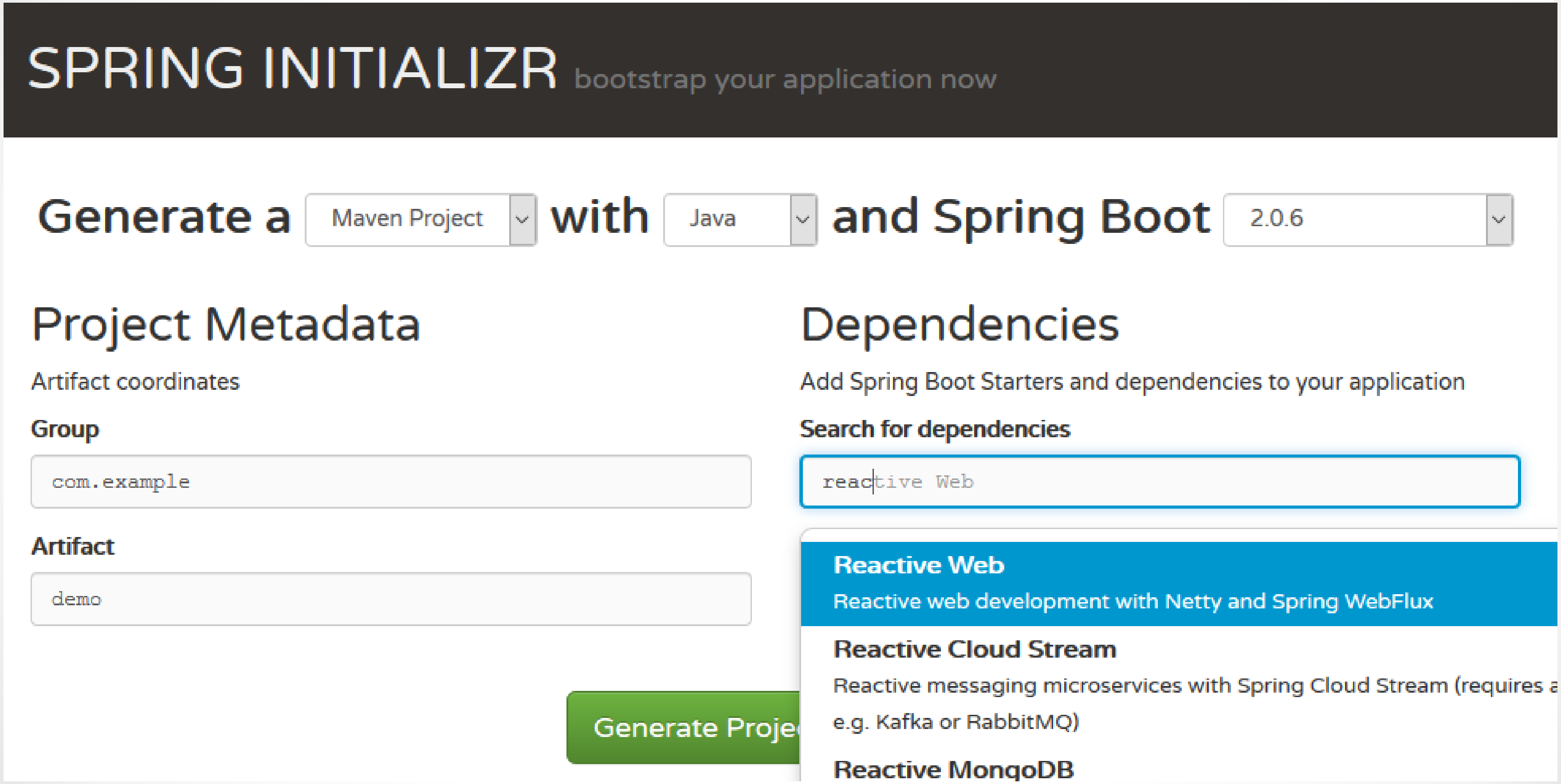
spring-webflux

Servlet API

HTTP / Reactive Streams

Servlet Container

Tomcat, Jetty, Netty, Undertow



SPRING INITIALIZR bootstrap your application now

Generate a Maven Project with Java and Spring Boot 2.0.6

Project Metadata

Artifact coordinates

Group: com.example

Artifact: demo

Dependencies

Add Spring Boot Starters and dependencies to your application

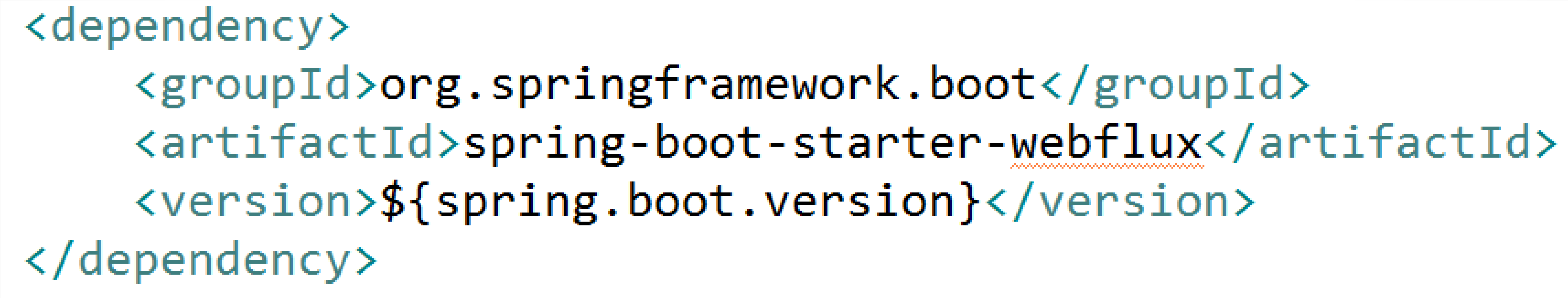
Search for dependencies: reactive Web

Reactive Web
Reactive web development with Netty and Spring WebFlux

Reactive Cloud Stream
Reactive messaging microservices with Spring Cloud Stream (requires a
e.g. Kafka or RabbitMQ)

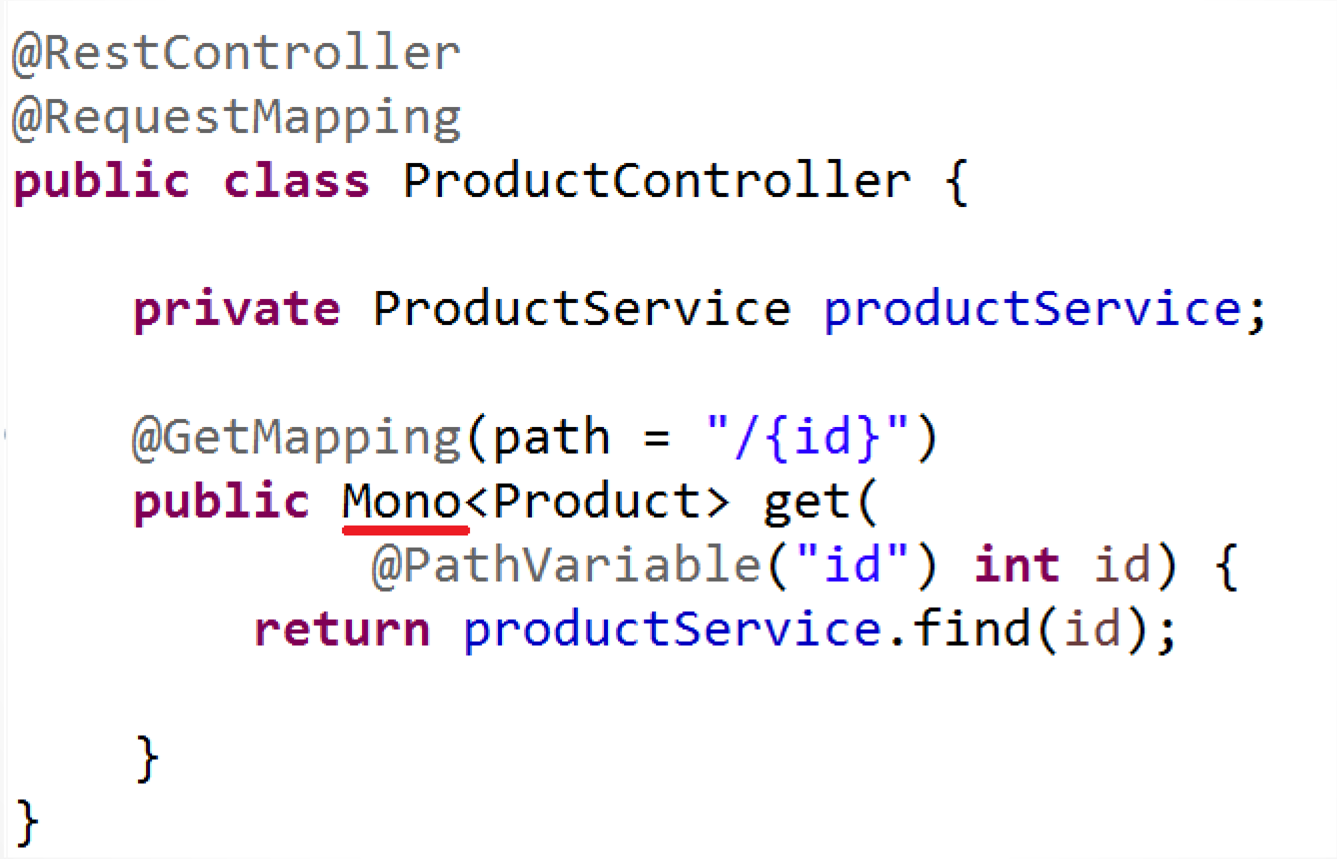
Reactive MongoDB

Spring Web Flux. Maven



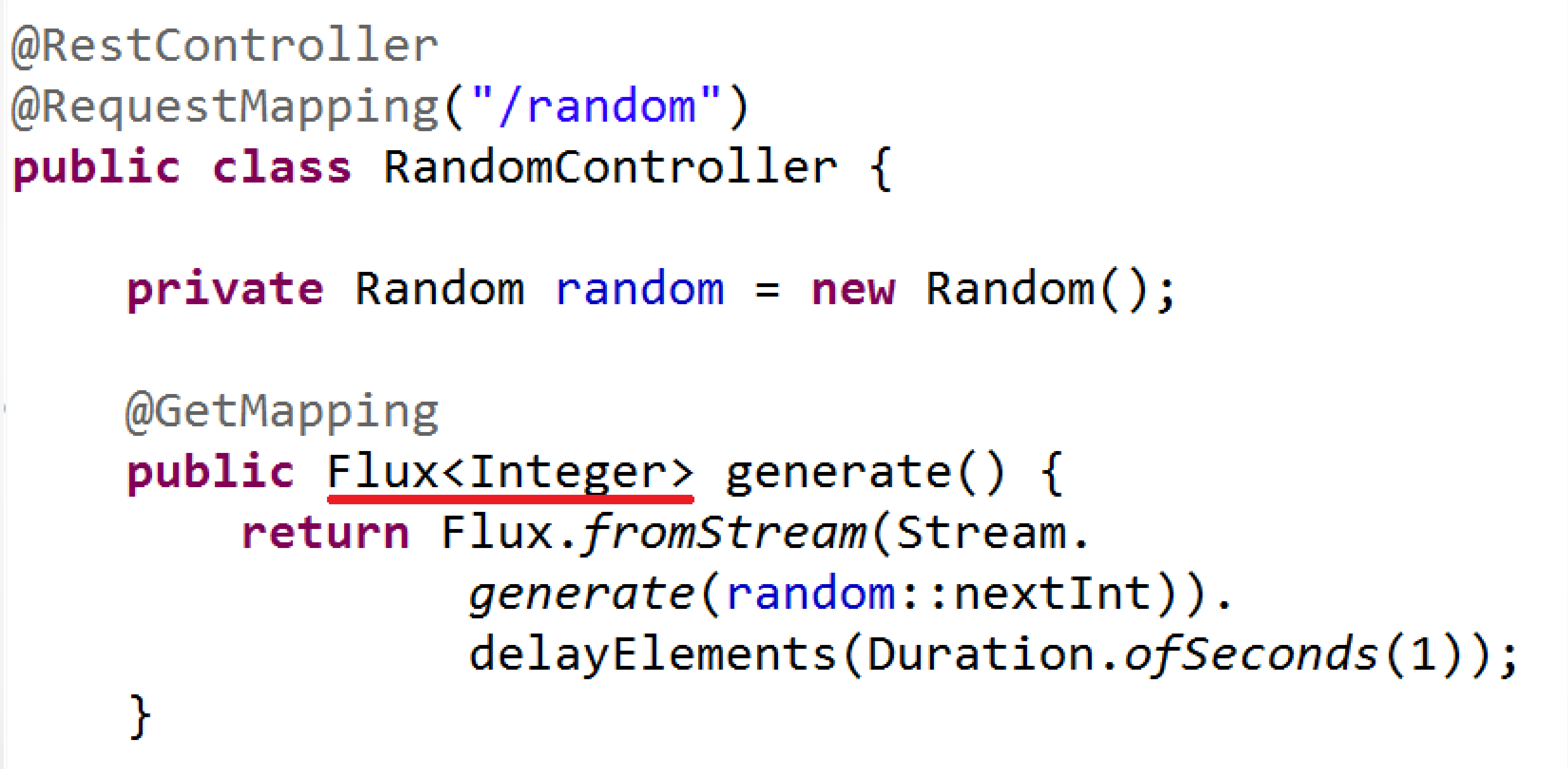
```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-webflux</artifactId>
    <version>${spring.boot.version}</version>
</dependency>
```

REST controller. Mono



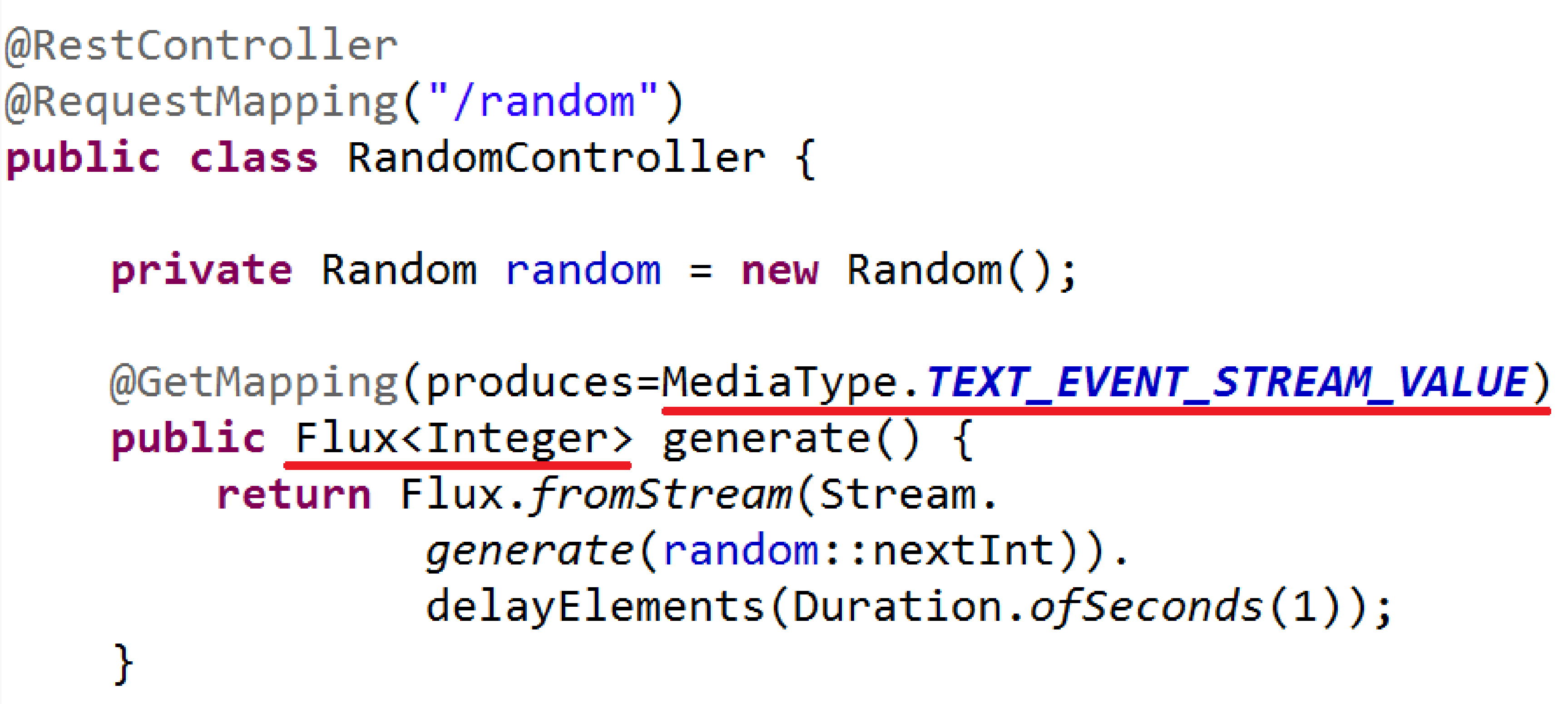
```
@RestController
@RequestMapping
public class ProductController {

    private ProductService productService;

    @GetMapping(path = "/{id}")
    public Mono<Product> get(
            @PathVariable("id") int id) {
        return productService.find(id);

    }
}
```

REST controller. Flux



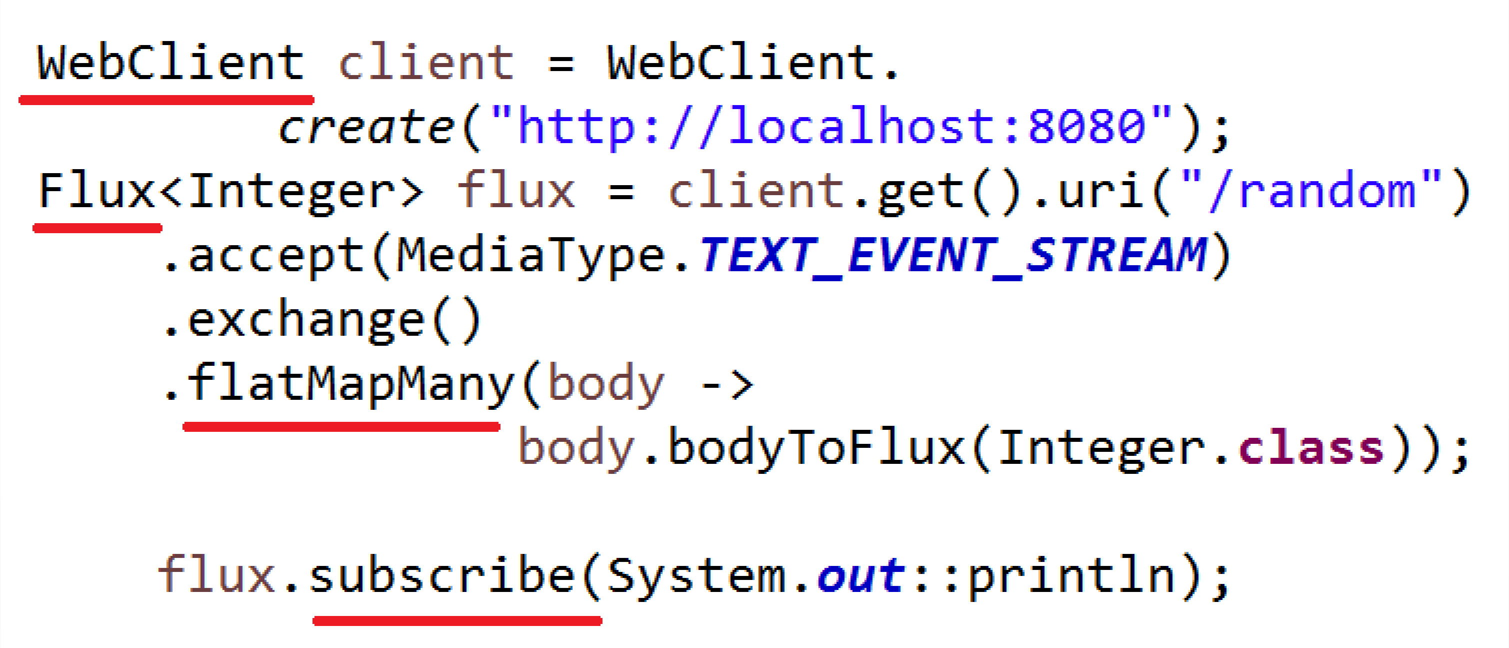
```java
@RestController
@RequestMapping("/random")
public class RandomController {

    private Random random = new Random();

    @GetMapping
    public Flux<Integer> generate() {
        return Flux.fromStream(Stream.
                generate(random::nextInt)).
                delayElements(Duration.ofSeconds(1));
    }
```

REST controller. Flux



```
@RestController
@RequestMapping("/random")
public class RandomController {

    private Random random = new Random();

    @GetMapping(produces=MediaType.TEXT_EVENT_STREAM_VALUE)
    public Flux<Integer> generate() {
        return Flux.fromStream(Stream.
                generate(random::nextInt)).
                delayElements(Duration.ofSeconds(1));
    }
```

Web client



```
WebClient client = WebClient.
            create("http://localhost:8080");
Flux<Integer> flux = client.get().uri("/random")
      .accept(MediaType.TEXT_EVENT_STREAM)
      .exchange()
      .flatMapMany(body ->
                   body.bodyToFlux(Integer.class));

      flux.subscribe(System.out::println);
```

Task #10. Spring Web Flux

1. Add Sping WebFlux dependency to **pom.xml:**
2. Create **REST controller** that returns stream of lower-case Latin characters c as Flux type. Run Spring Boot application and verify in the browser that prime numbers are displayed on the page
3. Create REST client that uses **WebClient** class and prints received prime numbers to the console.

✓ Sergey Morenets, sergey.morenets@gmail.com