RxAda: An Ada implementation of the ReactiveX API

Alejandro R. Mosteo

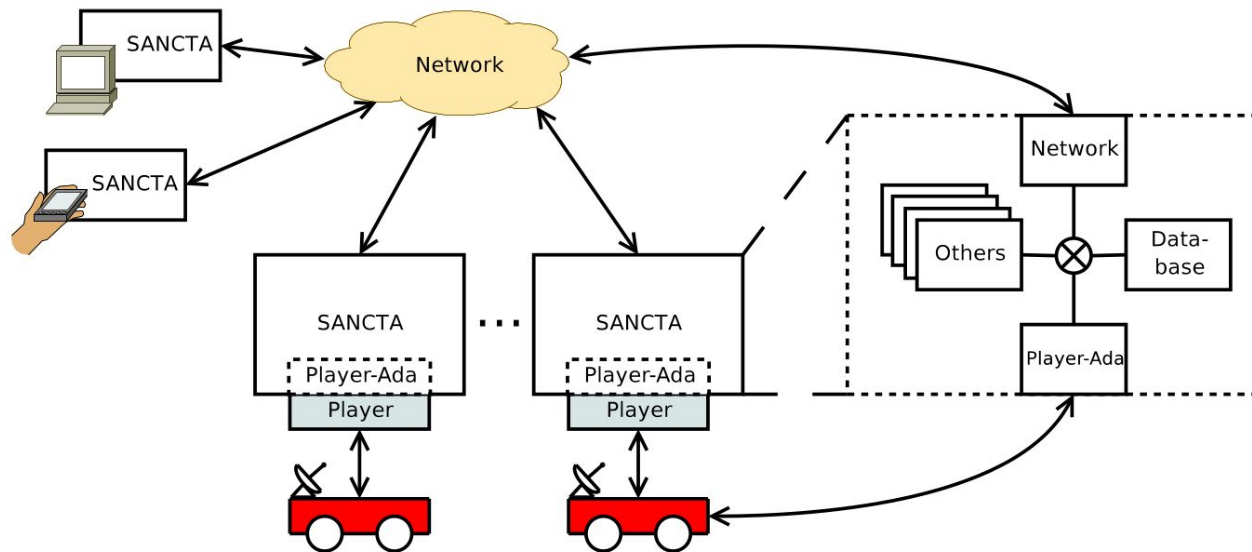
2017-jun-13

CUD

Centro Universitario de la Defensa Zaragoza

PREVIOUSLY, IN ADA-EUROPE 2007...

SANCTA: An Ada 2005 General-Purpose Architecture for ***Mobile Robotics Research***



ABOUT ME

Robotics, Perception, and Real-Time group (RoPeRT)

http://robots.unizar.es/

Universidad de Zaragoza, Spain

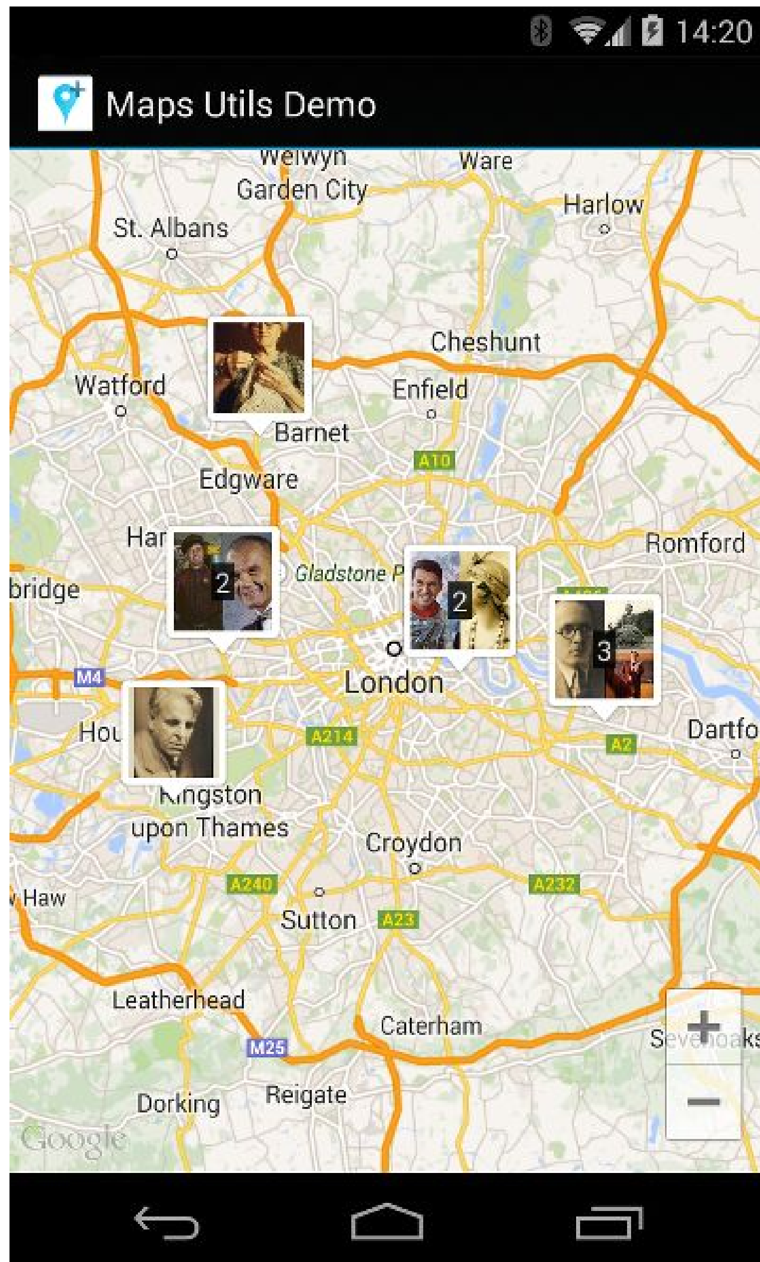
CONTENTS

- Motivation
- What is ReactiveX
  - Language agnostic
  - Java
  - Ada
- RxAda
  - Design challenges/decisions
  - Current implementation status
  - Future steps

PERSONAL MOTIVATION

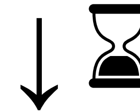
- Android development
  - Questionable design decisions for background tasks that interact with the GUI
- Found RxJava
  - Simpler, saner way of doing multitasking
  - Documented comprehensively
  - Very active community in the Rx world
- Achievable in Ada?
  - Aiming for the RxJava simplicity of use

EVENT-DRIVEN / ASYNCHRONOUS SYSTEMS

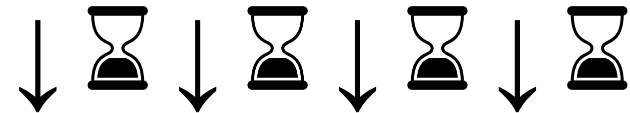


<User drags map>

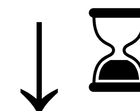
Find nearby items



Request images



Crop/Process image



Update GUI markers

OVERVIEW

Event-driven systems

Reactive Programming (philosophy)

ReactiveX / Rx (specification)

Rx.Net, RxJava, RxJS, RxC++, …

RxAda

REACTIVE MANIFESTO (2014-sep-16 v2.0)

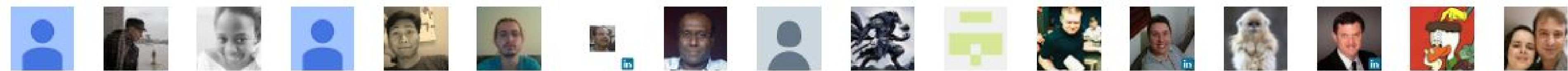
www.reactivemanifesto.org

"(...) we want systems that are ***Responsive****,* ***Resilient****,* ***Elastic*** *and* ***Message Driven****. We call these* ***Reactive Systems****."*

Jonas Bonér
Dave Farley
Roland Kuhn
Martin Thompson

Sign the manifesto

19082 people already signed (Full list)



REACTIVE PROPERTIES

- Responsive
  - “the cornerstone of usability and utility”
  - “reliable upper bounds [on response time]”
  - “consistent quality of service”
- Resilient
  - “responsive in the face of failure”
  - “achieved by replication, containment, isolation and delegation”
  - “The client of a component is not burdened with handling its failures.”

REACTIVE PROPERTIES

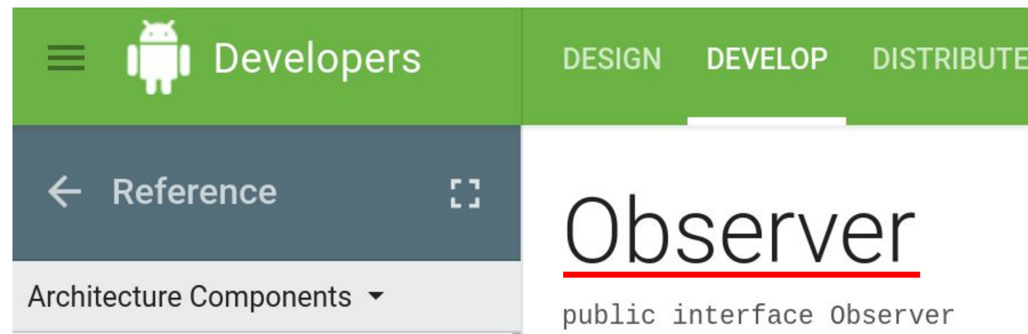
- Elastic
  - "responsive under varying workload"
  - "react to changes in the input rate by increasing or decreasing the resources allocated"
  - "no contention points or central bottlenecks"
- Message driven
  - "rely on asynchronous message-passing (...) ensures loose coupling"
  - "enables load management, elasticity, and flow control"

Courtesy of Washington University course CSE481C

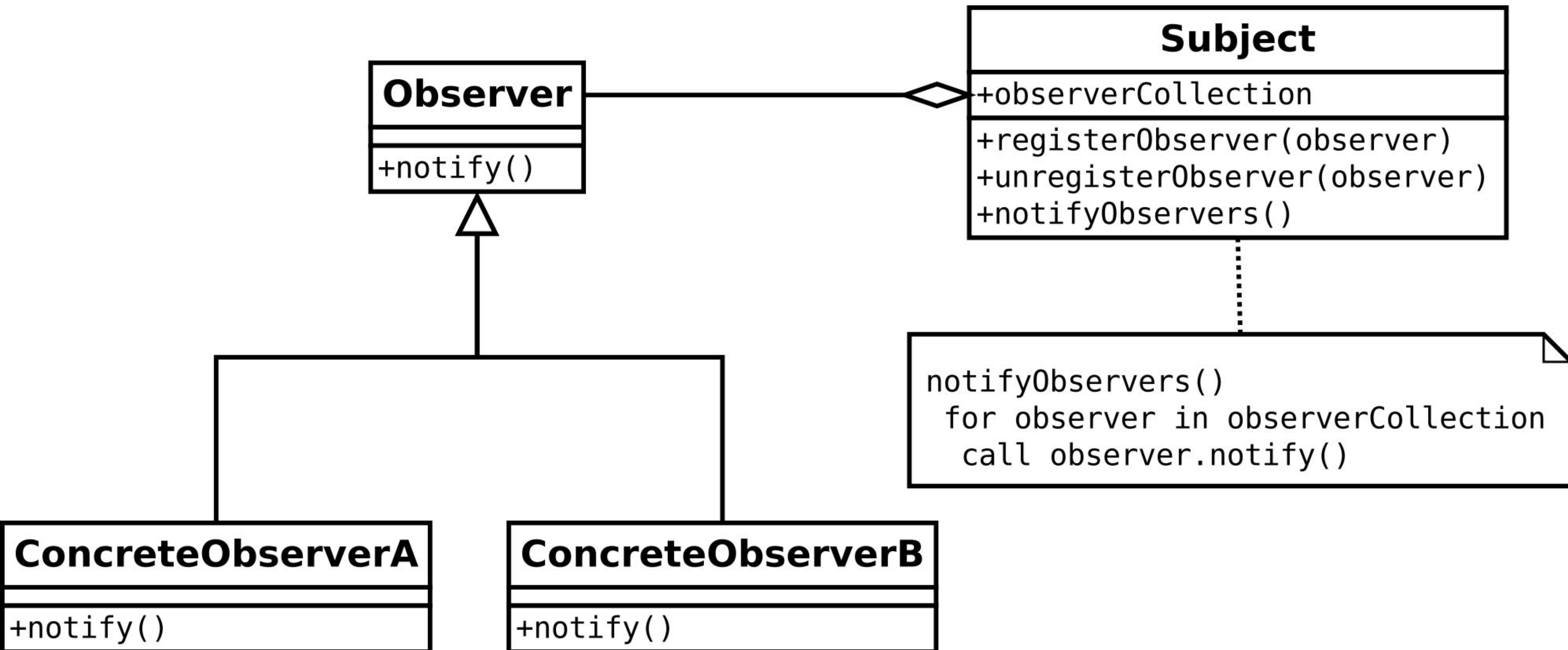
DISTRIBUTED CONSENSUS ALGORITHMS

ReactiveX ORIGINS

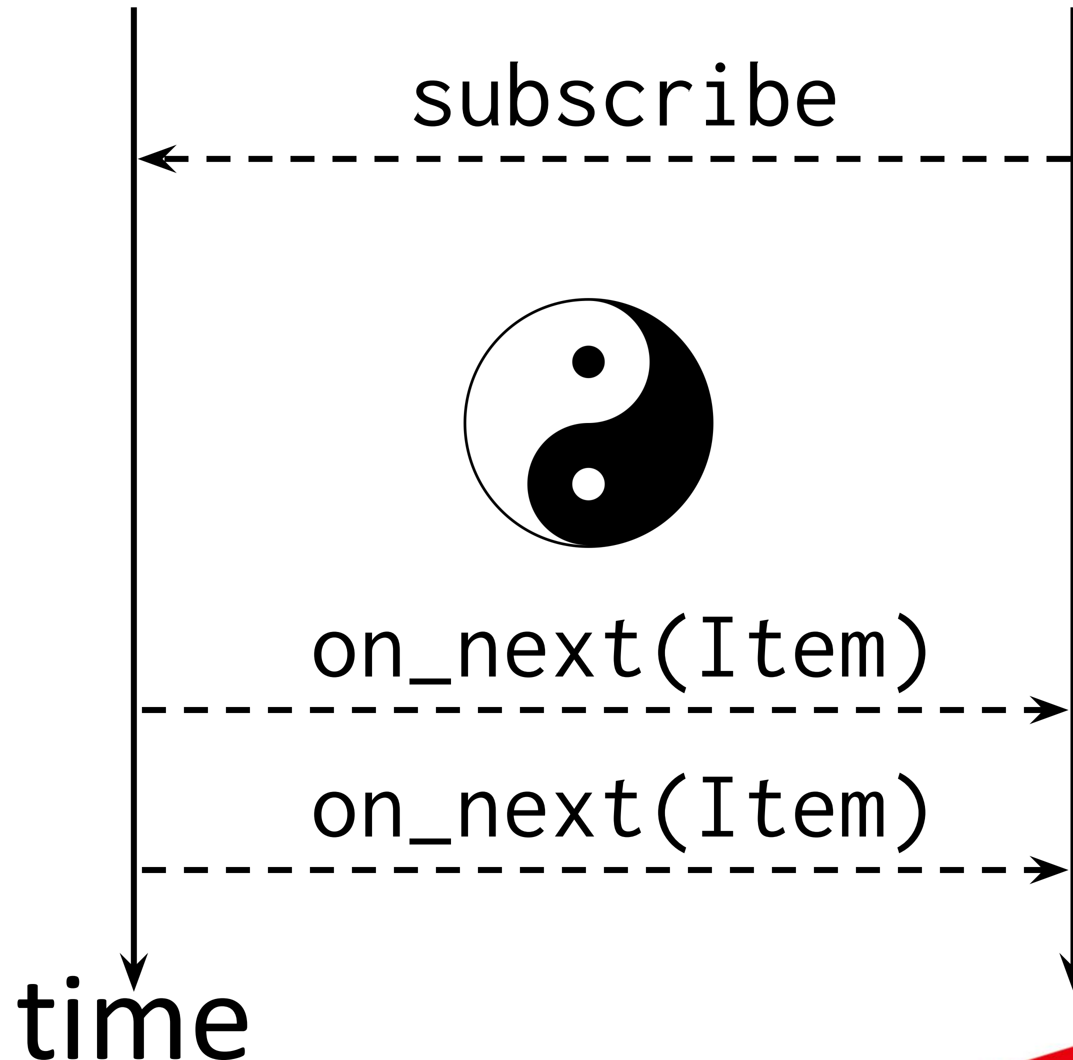
- There is no “universal” reactive solution
- ReactiveX is one among many
  - Roots in .NET (Reactive Extensions)
  - Erik Meijner



GANG OF FOUR'S OBSERVER PATTERN



PUSH-BASED FRAMEWORK



Centro Universitario de la Defensa Zaragoza

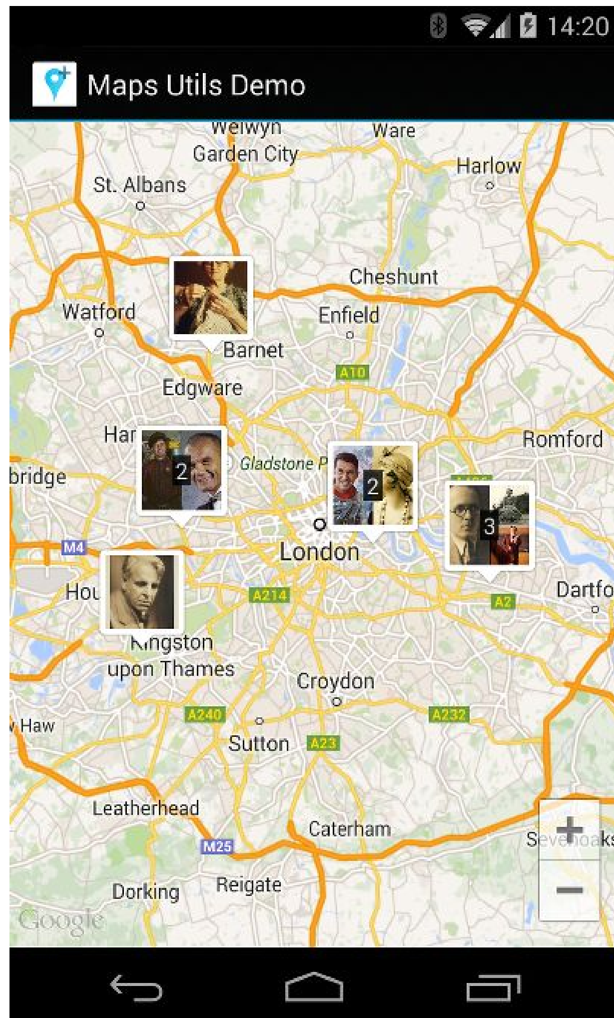
RELATED TECHNOLOGIES

- Callbacks
- Futures/Promises (polling)
  - `asyncProcess.isDone()`
  - `asyncProcess.getResult()`
  - Can become unwieldy when futures depend on other futures
- Ada AI12-0197-1 (generator functions)

```
yield Datum; -- “Queues” a ready datum
…
for X of Datum_Generator do
   -- Blocks until data available
```

Centro Universitario de la Defensa Zaragoza

Main selling point: composability



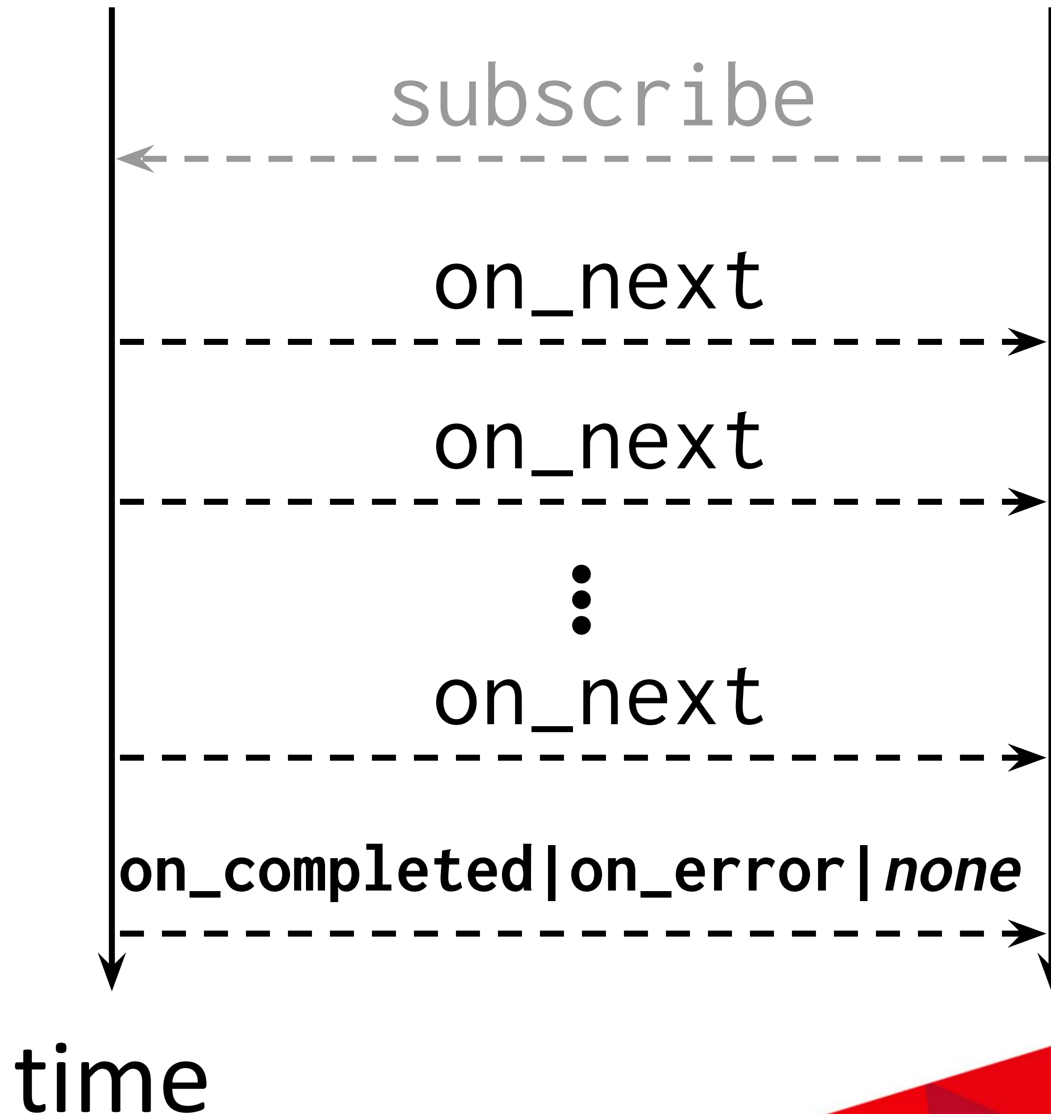

```
Observable
    .from(Map.dragEvent)
    .inBackground()
    .do(Map.whoIsNear(event))
    .doConcurrently
        (Users.getPicture(id))
    .do(crop(image))
    .inGUIthread()
    .do(updateMarker(image))
    .subscribe()
```

Rx COMPOSABILITY

- Advantages:
  - Imperative-like sequences
    - NOT executed when declared
    - Executed for every emitted item (event)
  - Abstract-away concerns like:
    - Low-level tasking
    - Synchronization
    - Thread-safety
    - Concurrent data structures
    - [Non-]Blocking I/O
    - Callback interactions

Centro Universitario de la Defensa Zaragoza

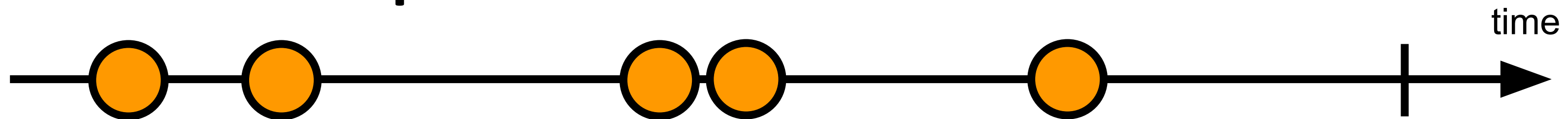
Rx CONTRACT



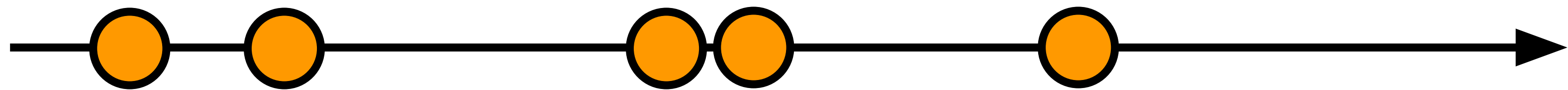
Rx CONTRACT / MARBLE DIAGRAMS

```
On_Next* (On_Completed|On_Error)?
```

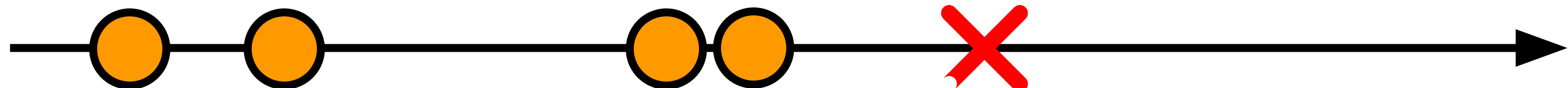
- Finite sequence:



  Infinite sequence:



- Failed sequence:



In mutual exclusion

RxAda DESIGN CONCERNS

- Ease of use (library clients)
  - Number of instantiations
    - Shallow learning curve
  - Boilerplate in creation of sequences
    - Preserve top-bottom order of actions when possible
- Ease of development
  - Reasonable (?) complexity
    - New features
    - Ongoing maintenance
    - Attract contributors
  - Avoid definite/indefinite multiplicity
    - Traits-based implementation

RxAda CONTRACT

On_Next* (On_Completed|On_Error)?

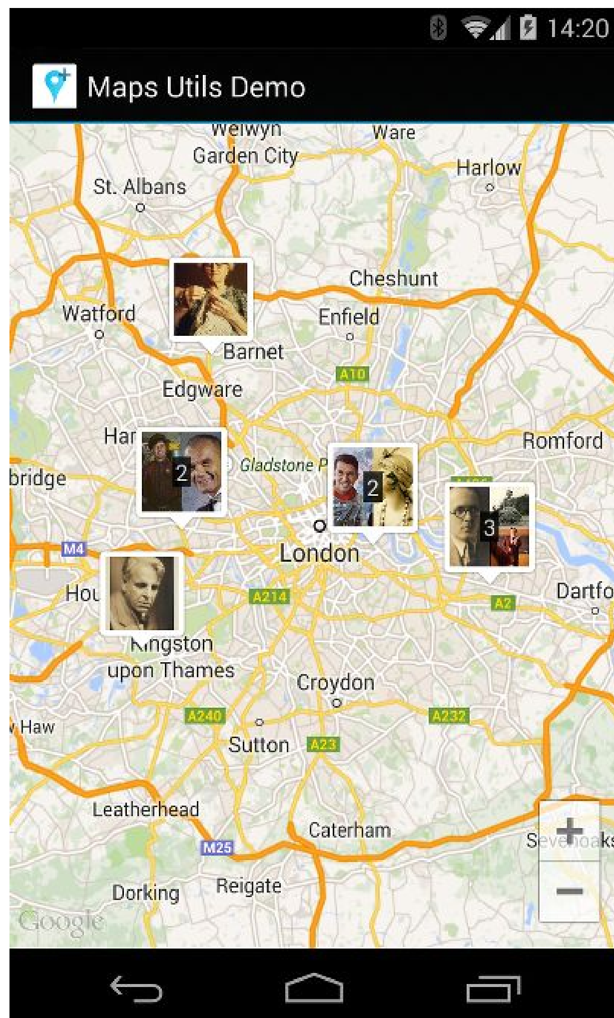
```
generic
    type T (<>) is private; -- T is the type to be received by the Observer
package Rx.Contracts is

    type Observer is interface; --  Someone interested in receiving data

    procedure On_Next     (This  : in out Observer; V : T) is abstract;
    procedure On_Complete (This  : in out Observer) is abstract;
    procedure On_Error    (This  : in out Observer; Error : Errors.Occurrence) is abstract;
     -- Errors encapsulate an Exception_Occurrence

    type Observable is interface; --  An emitter of data to which an observer can subscribe

    procedure Subscribe (Producer : in out Observable;
                         Consumer : in out Observer'Class) is abstract;

end Rx.Contracts;
```

CLASSWIDE CONTRACT ALTERNATIVE (DISCARDED)

```
package Rx.Contracts is -- No longer generic

   type Rx_Item is interface;

   type Observer is interface; --  Someone interested in receiving data
   procedure On_Next (This : in out Observer; V : Rx_Item'Class) is abstract;
```

- Pros 😊:
  - No user instantiations
  - Dot notation available for all operators
- Cons ☹:
  - User view conversions
    - Runtime checks only
    - Code pollution with "downcasts"
  - Or some kind of marshalling
    - Would require instantiations anyway

COMPOSABILITY

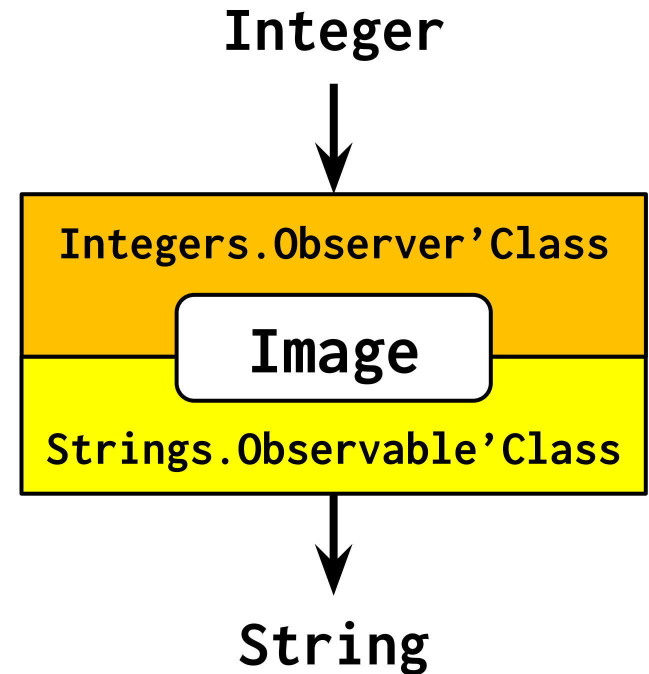


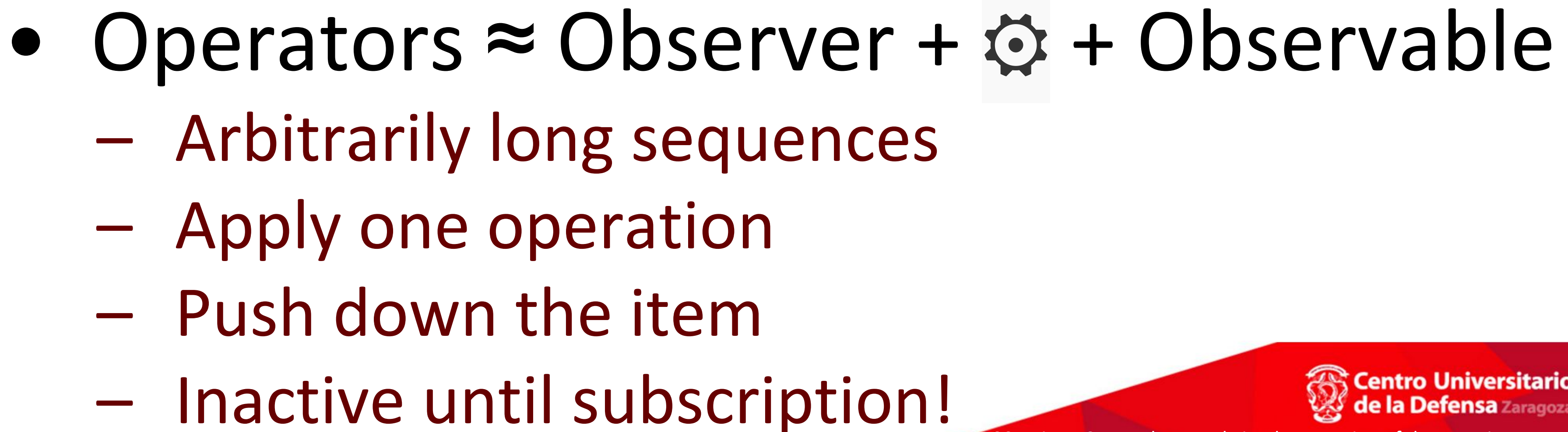

```
Observable
    .from(Map.dragEvent)
    .inBackground()
    .do(Map.whoIsNear(event))
    .doConcurrently
        (Users.getPicture(id))
    .do(crop(image))
    .inGUIthread()
    .do(updateMarker(image))
    .subscribe()
```

COMPOSITION: OPERATORS

- What is that between
  - Observable
  - …
  - …
  - …
  - Observer

?





- Operators ≈ Observer + ⚙ + Observable
  - Arbitrarily long sequences
  - Apply one operation
  - Push down the item
  - Inactive until subscription!

PREDEFINED OPERATORS (subset)

- Creating
  - Just, From
  - Interval, Range
- Transforming
  - Map, FlatMap
  - Buffer, Window
  - Scan
- Filtering
  - Filter, Last, Skip
  - Debounce, Sample
  - Distinct
- Combining
  - Merge, Zip
- Tasking
  - Interval, Timer
  - ObserveOn
  - SubscribeOn
  - Delay, Timeout
- Logical
  - TakeUntil
- Mathematical
  - Reduce

OPERATOR CONCATENATION (JAVA / C++)

- Java: dot notation
  - Chains an operator
  - Returns another observable

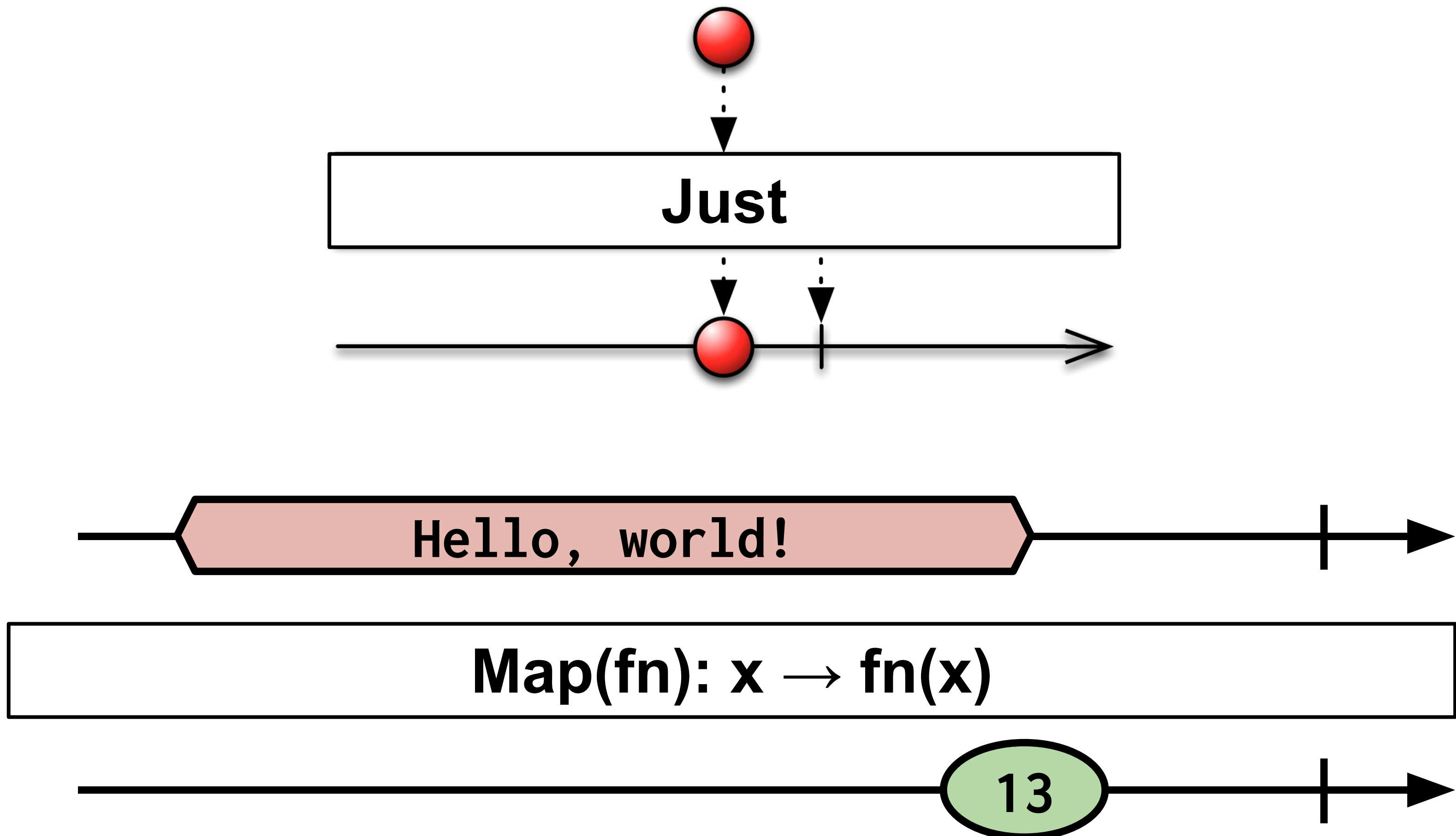
```
public class Observable<T> {
    // T is the type emitted by this observable

    public final Observable<java.lang.Integer> count()
    // The result is another Observable that emits integers
```

- C++: pipe operator “|”
- Ada: “&”

Centro Universitario de la Defensa Zaragoza

TOY EXAMPLE

```
Observable
        .just("Hello, world!")
        .map(str -> str.length())
        .subscribe(System.out::println);
```



RxAda DESIGN

- Ada syntax:
  - No anonymous/lambdas → no inline code
    - 😊 We cannot have boilerplate in the sequence
    - 😊 ☹ We must move logic elsewhere (a-la std containers)

```
declare
   S : Rx.Subscriptions.Subscription;
begin
   S := -- We can't ignore the resulting subscription in Ada
     Just ("Hello, world!") &
     Map (Length'Access) &
     Subscribe (Put_Line'Access);
```

  - But also…

```
package Strings  is new Rx.Indefinites (String);
package Integers is new Rx.Definites (Integer);
package Str_To_Int is new Rx.Operators
   (Strings.Observables, Integers.Observables);
```

Ada DOT NOTATION

- First preference, but unusable:
  - Freezing rules
    - All subprograms at top package level
    - But transforming operators require a second type
      - ⊘ In a child package
        - » Dot notation no longer available
      - ⊘ Instantiate pairs of types
        - » Circular dependencies
- Settled on using the “&” function
  - Inspired by C++ and comp.lang.ada discussions
  - ☹ Requires “use” for every type mapping
    - Same_Type ⟶ Same_Type (e.g., Filter)
    - From_Type ⟶ Into_Type (e.g., Map)

FINAL EXAMPLE

- Every second,
  - take the corresponding integer (1, 2, …),
  - Hash its string representation (in bg task)
  - Print it (in IO task)

```
S : constant Subscription :=
      Interval (First => 1, Period => 1.0)
      &
      Observe_On (Schedulers.Computation)
      &
      Map (Image'Access)
      &
      Map (String_Hash'Access)
      &
      Observe_On (Schedulers.IO)
      &
      Subscribe (Put_Line'Access);
```

FINAL EXAMPLE: REQUISITES

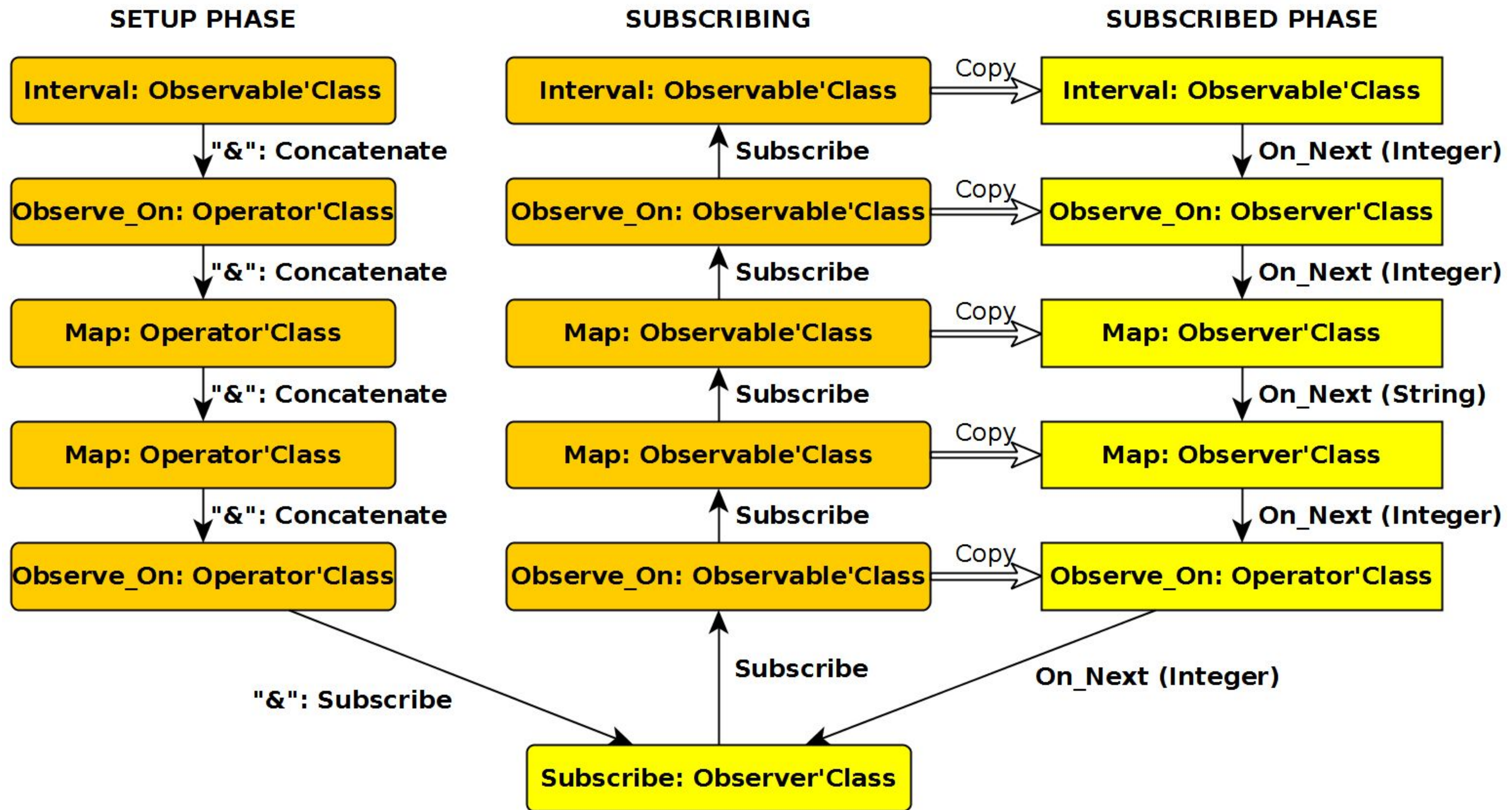
```ada
use Rx.Std;
use Rx.Std.Integers;
use Rx.Std.Integer_To_String;
use Rx.Std.String_To_Integer;

type Hashes is mod 2 ** 32;
function Modular_Hash is
   new System.String_Hash.Hash (Character, Rx_String, Hashes);

function String_Hash (S : String) return Rx_Integer is
   (Rx_Integer (Modular_Hash (S)));

function Image (I : Rx_Integer) return String is
   (Rx_Integer'Image (I));
```

FINAL EXAMPLE

- Every second,
  - take the corresponding integer (1, 2, …),
  - Hash its string representation (in bg)
  - Print it (in IO task)

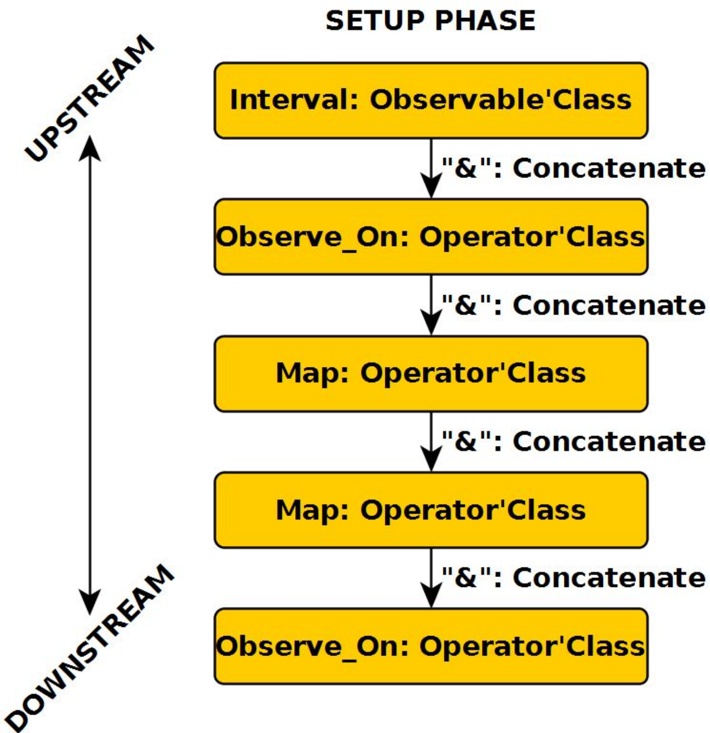
```
S : constant Subscription :=
      Interval (First => 1, Period => 1.0)
      &
      Observe_On (Schedulers.Computation)
      &
      Map (Image'Access)
      &
      Map (String_Hash'Access)
      &
      Observe_On (Schedulers.IO)
      &
      Subscribe (Put_Line'Access);
```

WHAT IS HAPPENING?

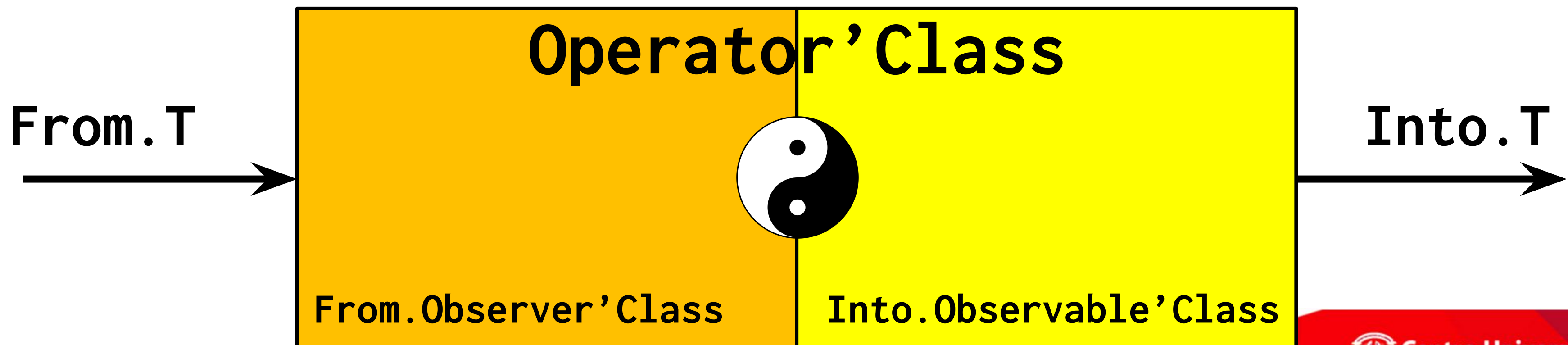


Centro Universitario de la Defensa Zaragoza

SETUP PHASE / OPERATOR ANATOMY

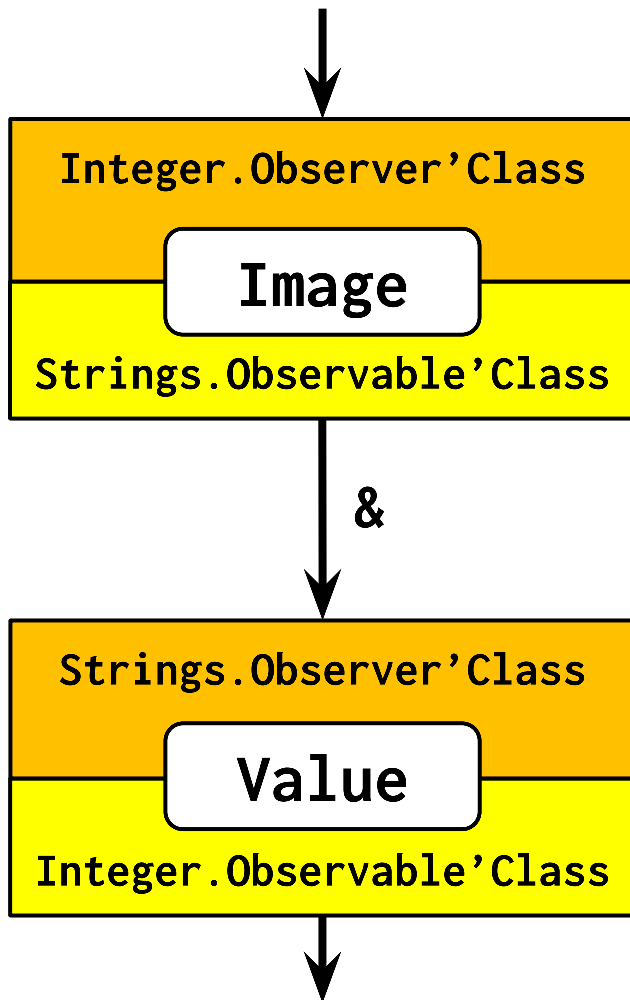



```ada
generic
   with package From is new Rx.Impl.Typed (<>);
   with package Into is new Rx.Impl.Typed (<>);
package Rx.Impl.Transformers with Preelaborate is

   type Operator is new
     Into.Contracts.Observable and
     From.Contracts.Observer
   with private;

   function "&" (Producer : From.Observable'Class;
                 Consumer : Operator'Class)
                 return       Into.Observable'Class;
```



COMPILE-TIME OPERATOR CONSISTENCY

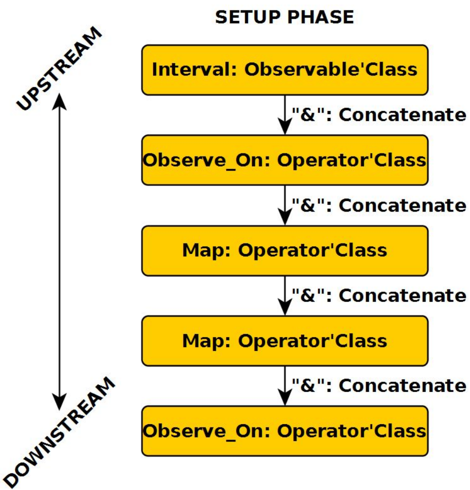
```
function "&" (Producer : From.Observable'Class;
              Consumer : Operator'Class)
              return     Into.Observable'Class;
```



"&" returns the Observable view of the same Operator

Distinct "&" for every [From.Observer + Into.Observable] combination

SUBSCRIPTION ACTIVATION




```
constant Subscription :=
  Interval (First => 1, Period => 1.0)
  &
  Observe_On (Schedulers.Computation)
  &
  Map (Image'Access)
  &
  Map (String_Hash'Access)
  &
  Observe_On (Schedulers.IO)
  &
  Subscribe (Put_Line'Access);
```

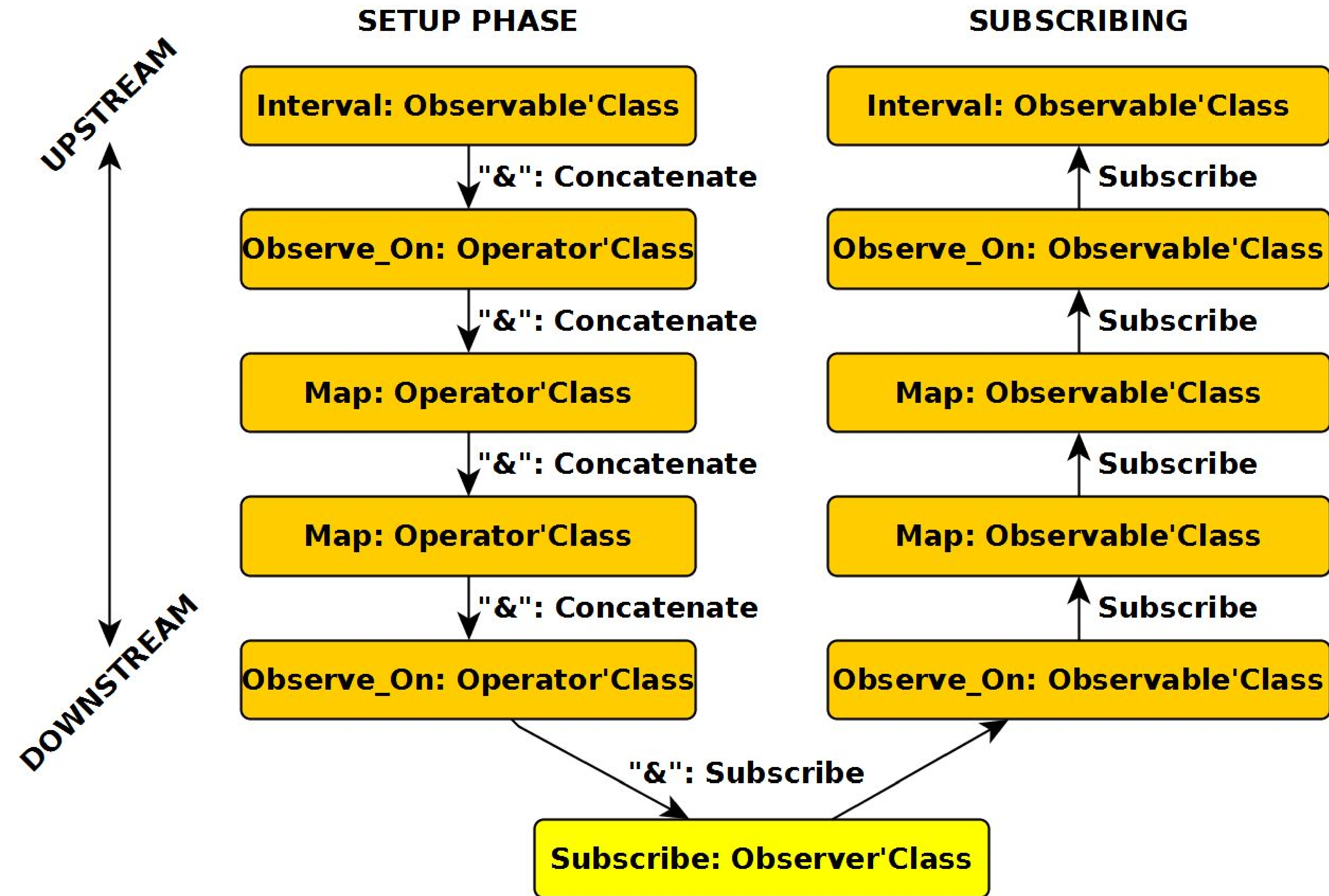
calls to Concatenate

returns an Operator

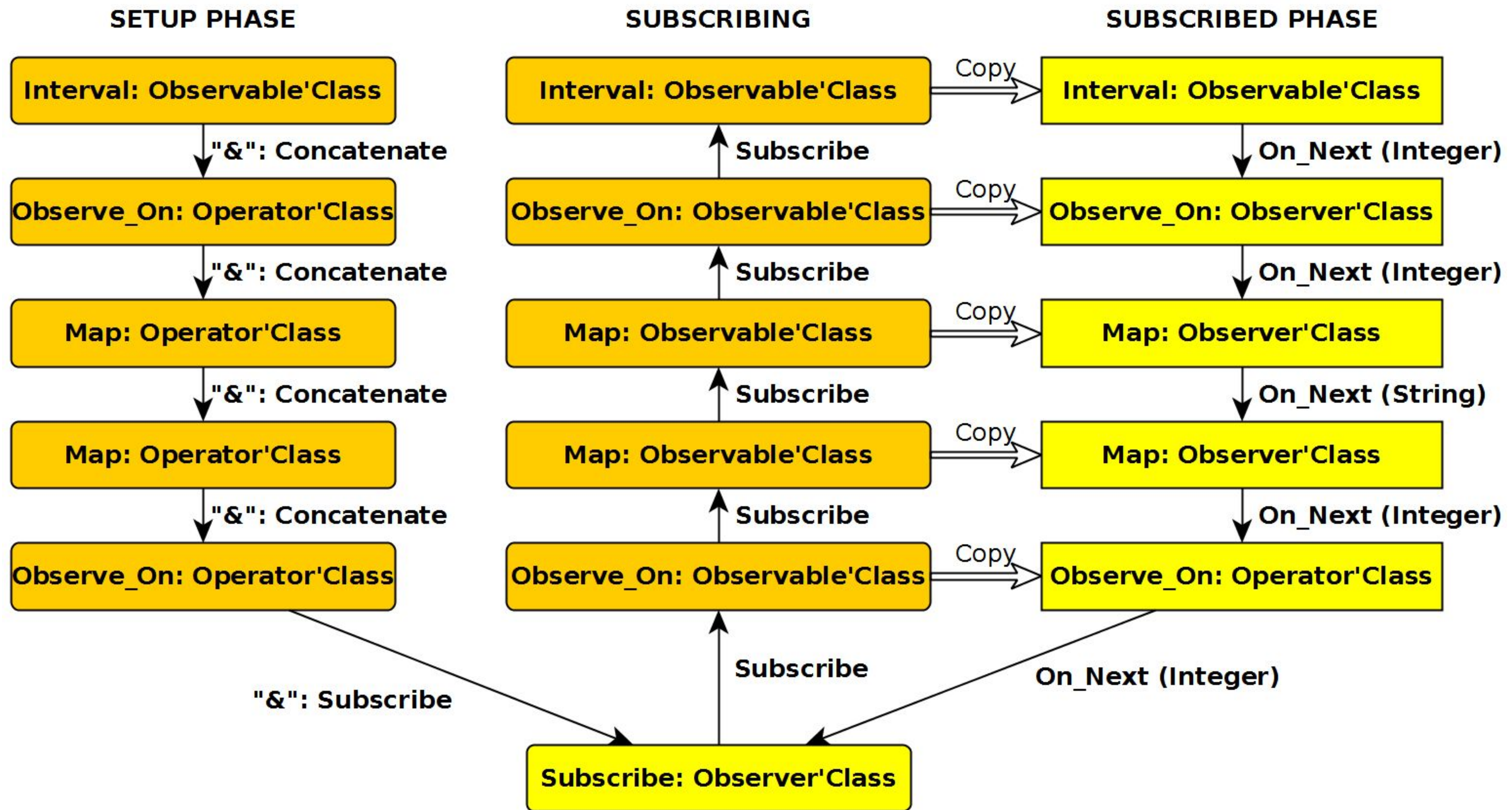
calls to Subscribe

returns an Observer

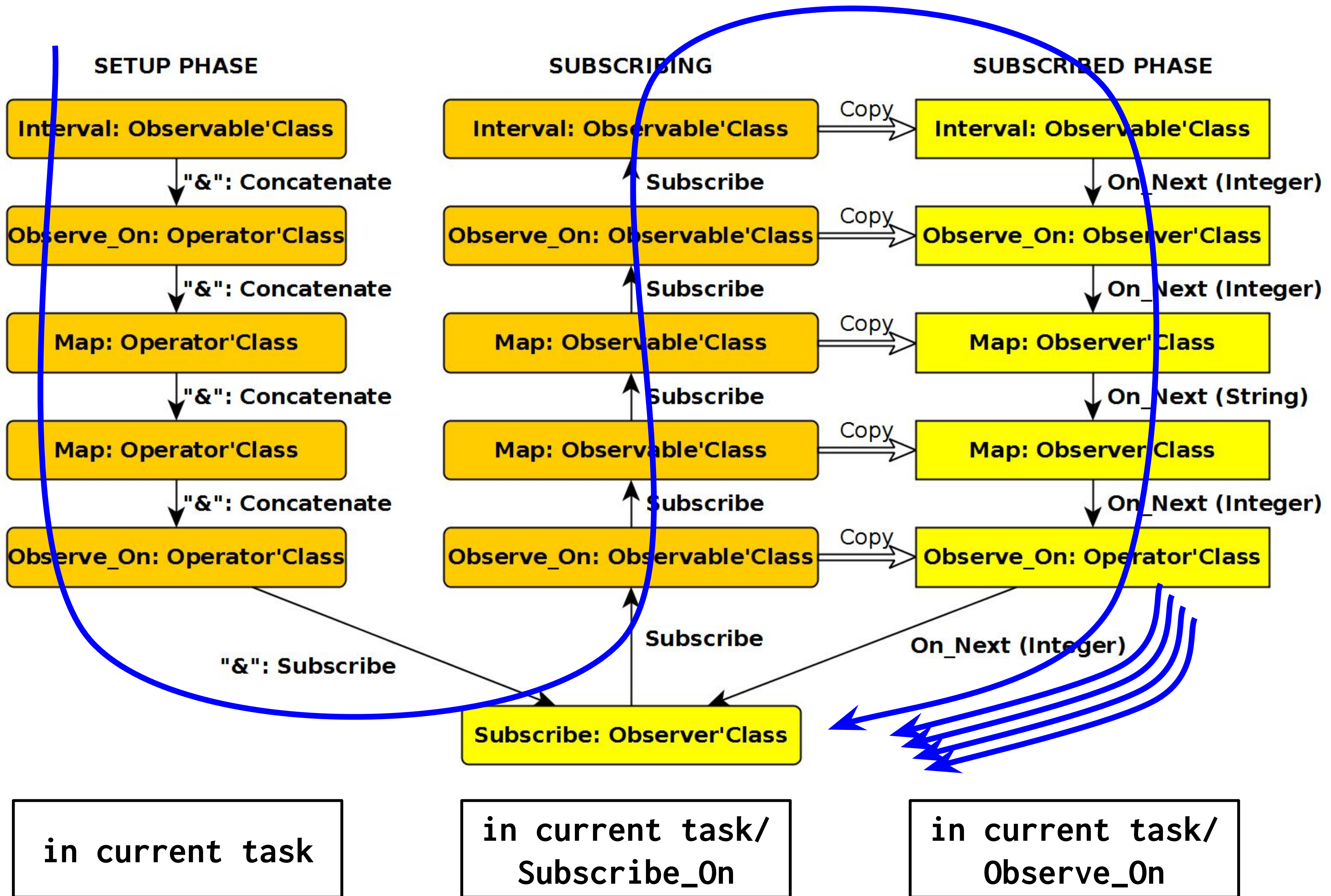
SUBSCRIPTION ACTIVATION



WHAT IS HAPPENING?



WHAT IS HAPPENING?



TASKING

```
type Scheduler is private;

function IO return Scheduler;
--  Thread pool that can grow unboundedly

function Computation return Scheduler;
--  Thread pool with as many threads as CPUs

function New_Thread return Scheduler;

package Pools is

    type Pool is tagged limited private;

    function Create (Max_Size : Positive;
                     Name     : String := "") return Pool;

    function Get_Next (This : in out Pool) return Scheduler;
    --  Round-robin use of threads

    function Get_Idle (This : in out Pool) return Scheduler;
    --  Returns a currently idle thread in the pool, if available

end Pools;
```

Subscribe_On
Observe_On
Op. param.

N-TO-1 PRODUCERS/CONSUMER

```
declare
   S : Rx.Subscriptions.Subscription;
begin
   S :=
      Producer1 &
      Merge(Producer2) &
      Merge(Producer3) &
      Subscribe;
end;
```

1-TO-N PRODUCER/CONSUMERS

```
Observable
    .range(1,1000)
    .flatMap(i -> Observable
                    .just(i)
                    .subscribeOn(Schedulers.computation())
                    .map(i2 -> intenseCalculation(i2)))
    .subscribe()
```

Concurrent flows

Main flow

Concurrent subscriptions

1-TO-N PRODUCER/CONSUMERS

```
Observable
    .range(1,1000)
    .flatMap(i -> Observable
                    .just(i)
                    .subscribeOn(Schedulers.computation())
                    .map(i2 -> intenseCalculation(i2)))
    .subscribe()
```

Main flow

Concurrent flows

```
function Single_Processor(W : Workload) -- At library level
   return Workloads.Observable'Class
is
   use Workloads;
begin
   return
      Just (W) &
      SubscribeOn(Schedulers.Computation) &
      Map (Intense_Calculation'Access);
end Single_Processor;

declare
   S : Rx.Subscriptions.Subscription;
begin
   S :=
      Whatever &
      Flat_Map(Single_Processor'Access) &
      Subscribe;
end;
```

LESSONS LEARNED

- The good:
  - Personal goal achieved
  - Reasonable good-looking, working library
  - Most complexity under wraps (for clients)

- The not-so-good:
  - Number of instances needed
  - Code size, compilation time
  - “use” for all types involved

FUTURE STEPS

- More operators
  - Some unimplemented
  - Obscure variants
- Fix for GNAT GPL 2017
  - Compilation currently broken
  - Check fixes to bugs in GPL 2016
- Use in real/testbed project (!)
- Evaluate performance
  - Impact of copy semantics

THANKS FOR YOUR ATTENTION

https://bitbucket.org/amosteo/rxada/

amosteo@unizar.es

@mosteobotic

Centro Universitario de la Defensa Zaragoza

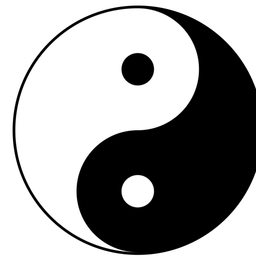
cud.unizar.es

Academia General Militar · Ctra. Huesca s/n · 50090 Zaragoza · 976 739 500

Rx OBSERVABLE / OBSERVER duality

```
Observable
 .subscribe
```

```
Observer
.on_next
```



Something that can be observed

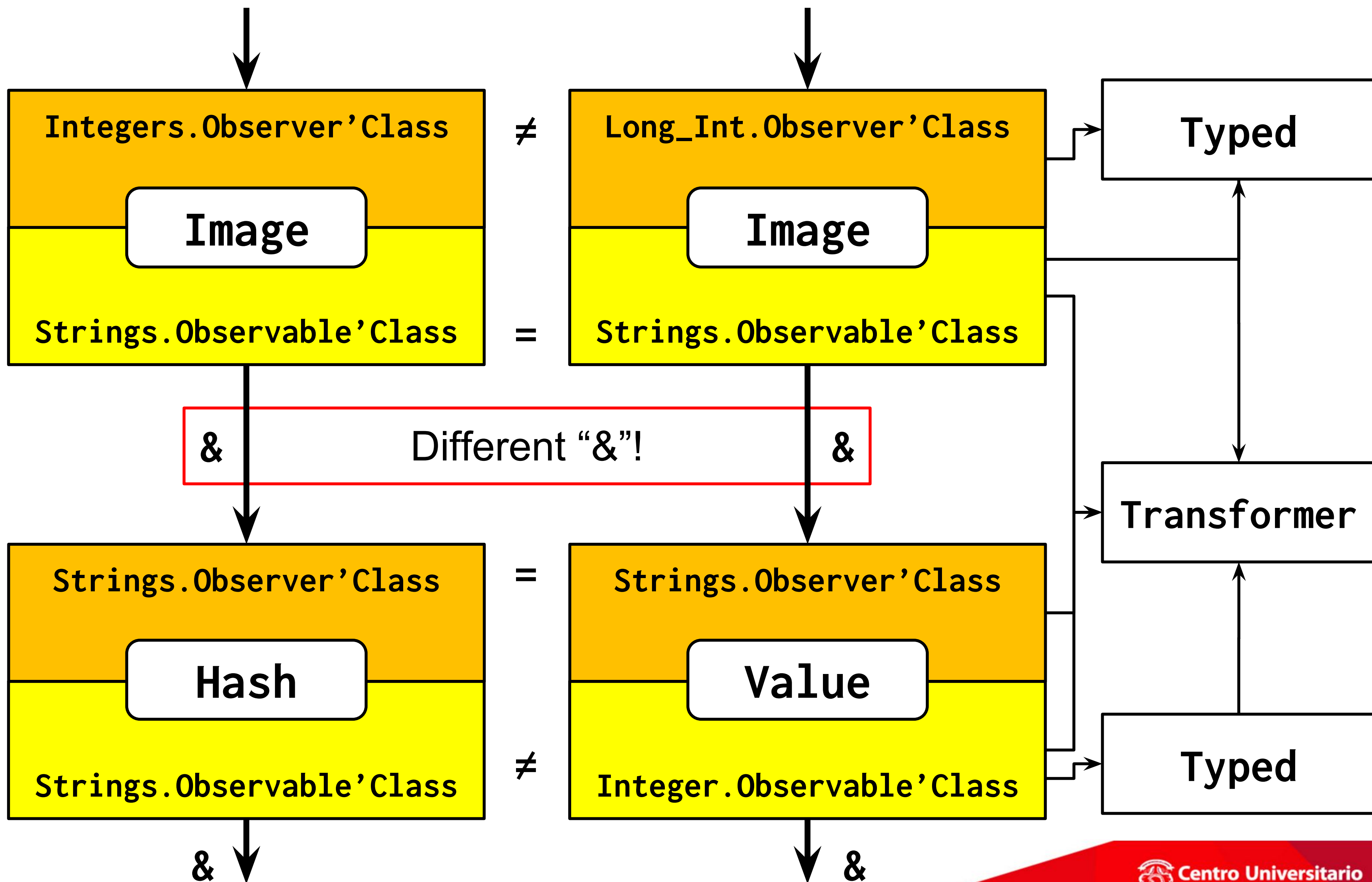
Something that observes an observable

TRAITS-BASED IMPLEMENTATION

```
generic
   type T (<>) is private; -- The user-facing type
   type D      is private; -- Definite type for storage of T
   with function To_Definite   (V : T) return D is <>;
   with function To_Indefinite (V : D) return T is <>;
package Rx.Traits.Types with Preelaborate is
```

```
generic
   with package Type_Traits is new Rx.Traits.Types (<>);
package Rx.Impl.Typed with Preelaborate is

   package Contracts is new Rx.Contracts      (Type_Traits.T);
   package Actions   is new Rx.Actions.Typed (Type_Traits.T);
```

COMPILE-TIME OPERATOR CONSISTENCY (II)



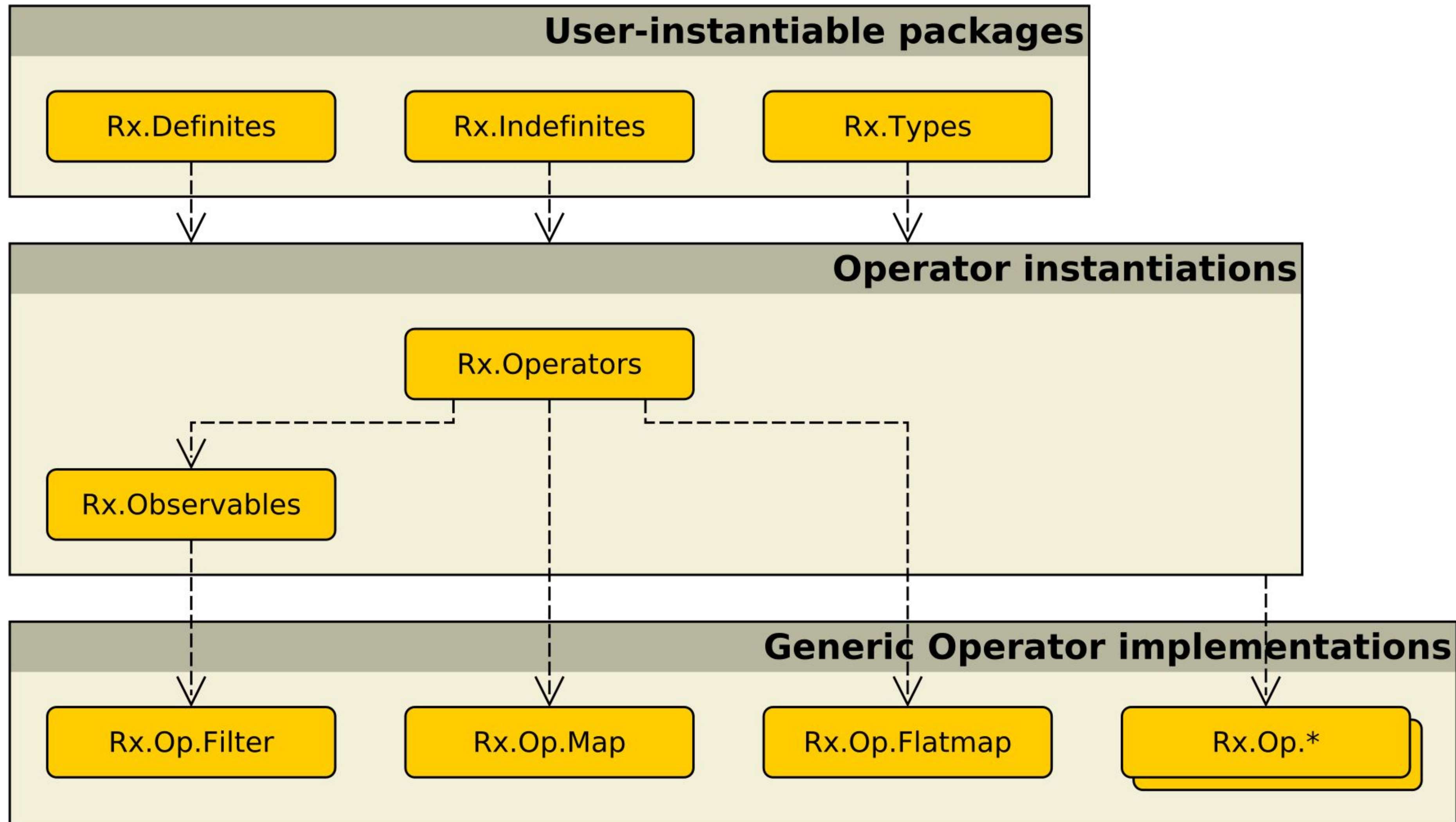
Ada GENERICS

- No inline instantiation (nor inference)

```
public class Observable<T> {
   // T is the type emitted by this observable

   public final <R> Observable<R>
       map(Func1<? super T,? extends R> func);
   // R is the result of func, and also
   // the type emitted by the observable returned by Map
```

- Ada: Explicit instantiations for
  - Each user type
  - Each type conversion between user types
    - With distinct source/result
    - Integer ⟶ String
    - String ⟶ Integer

USER PACKAGES



Centro Universitario de la Defensa Zaragoza

LANGUAGE SUPPORT FOR BOILERPLATE (Java)

Pre-Java 8 (anonymous classes)

```
Observable
        .just("Hello, world!")
        .map(new Func1<String, Integer>() {
            @Override
            public Integer call(String str) {
                return str.length();
            }
        })
        .subscribe(new Action1<Integer>() {
            @Override
            public void call(Integer o) {
                System.out.println(o);
            }
        });
```

Post-Java 8 (inference/lambda/references)

```
Observable
        .just("Hello, world!")
        .map(str -> str.length())
        .subscribe(System.out::println);
```

Centro Universitario de la Defensa Zaragoza

OPERATOR TESTS

- Many operators require parameters that are subprograms
  - Accesses to library level subprograms
  - Tagged types allow parameterization without generics

```
-- Test limit operator
Subs :=
  Integers.From ((1, 3, 2)) &
  Limit (2) &
  Subscribe (Checker'(Do_Count => True, Ok_Count => 2,
                      Do_First => True, Ok_First => 1,
                      Do_Last  => True, Ok_Last  => 3));
```

- BTW: Rx is lazy about tasks

"RUNNABLE" PARAMETERS

- Many operators require parameters that are subprograms
  - Accesses to library level subprograms
  - Tagged types allow parameterization without generics

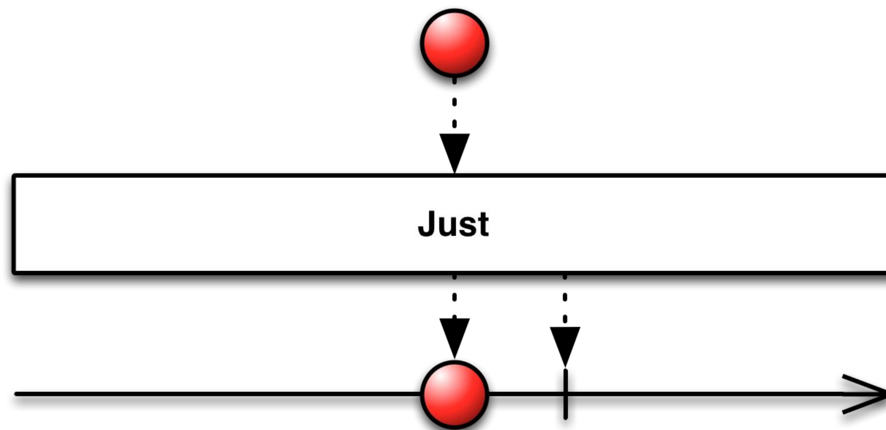
```
package Rx.Actions with Preelaborate is

   type Filter0 is access function return Boolean;

   type TFilter0 is interface;
   function Check (Filter : in out TFilter0)
                   return          Boolean is abstract;

   function Wrap (Check : Filter0) return TFilter0'Class;
```

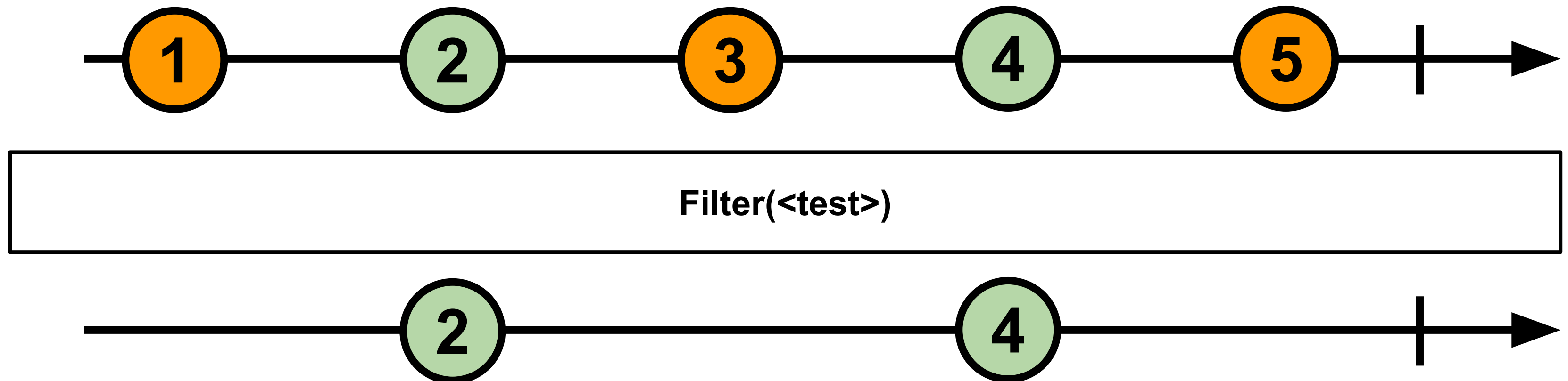
OBSERVABLE EXAMPLE FROM GENERATOR

```
procedure Subscribe
  (Producer : in out Observable;
   Consumer : in out Observer'Class) is
begin
   for Item of Producer.Generator loop
      Consumer.On_Next(Item);
   end loop;
   Consumer.On_Completed;
exception
   when E : others =>
      Consumer.On_Error
         (Errors.From_Exception(E));
end On_Subscribe;
```

Centro Universitario de la Defensa Zaragoza

JUST / FILTER






```
Observable.just(1, 2, 3, 4, 5)
          .filter(i -> (i % 2) == 0)
          .subscribe(System.out::println);
```