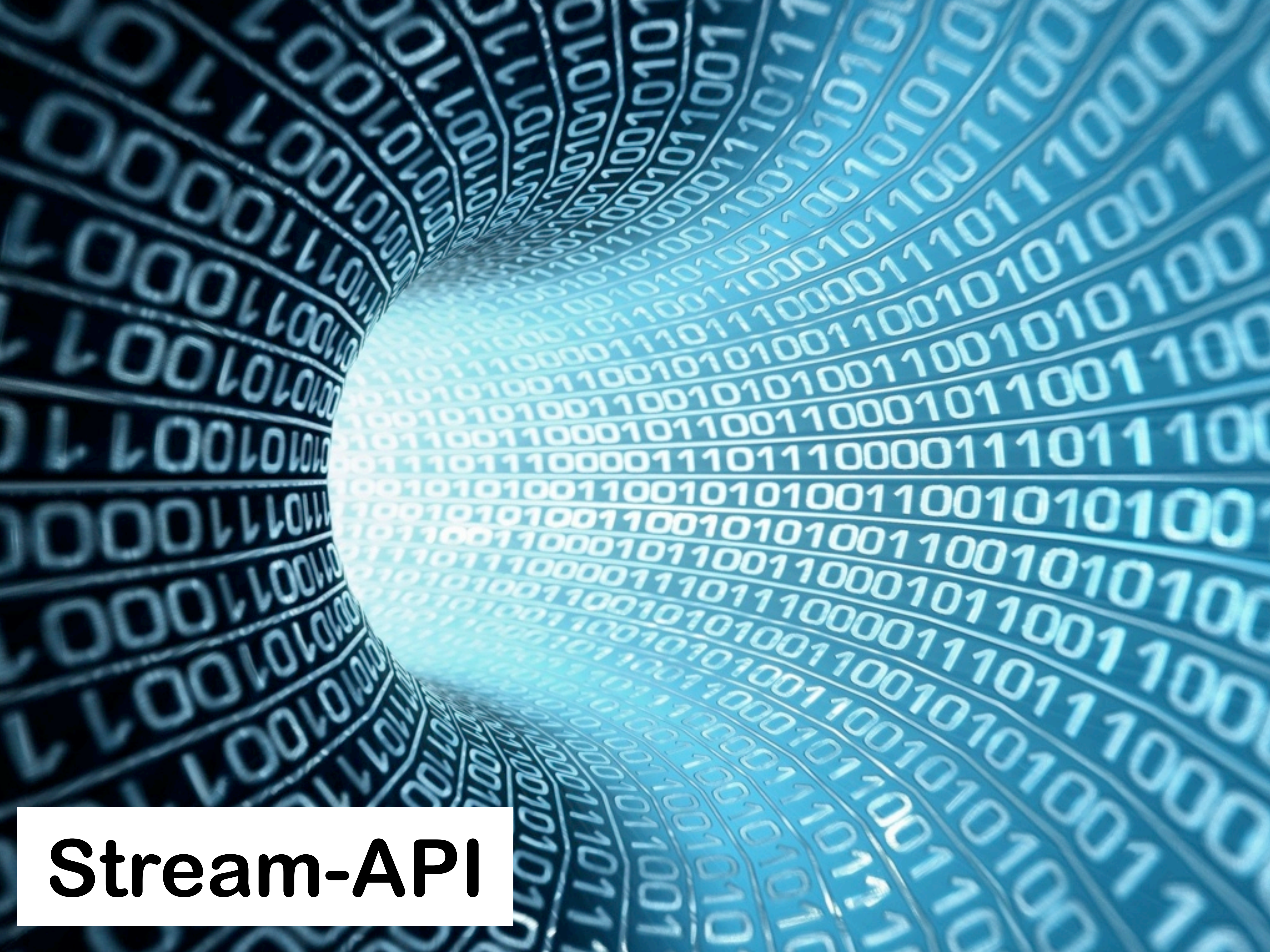


Stream-API

What I‘ll talk about...

- Needs and Objectives, „Philosophy“
- The *stream-api* (derived from PG-542)
  - Concepts
  - Example

Needs and Objectives

- Clean and easy-to-use API
  - Understandable by the Physics guys :-)
  - Easy to extend+integrate, small foot-print
  - Quick and standalone debugging capabilities
  - Support for multi-threading
  - Easy to document (e.g. your extensions)
  - *deployable on (multiple) servers (support for distributed data processing)*

What do we want to do?

- Read high-volume data streams
  - Define+Execute Data-Stream-Processes
  - Implement/Add our own operators/processors
  - Evaluate processors (memory,speed,accuracy)

- Following the ideas of
  - data flow (vertical view)
  - anytime services (horizontal view)

Naming Conventions

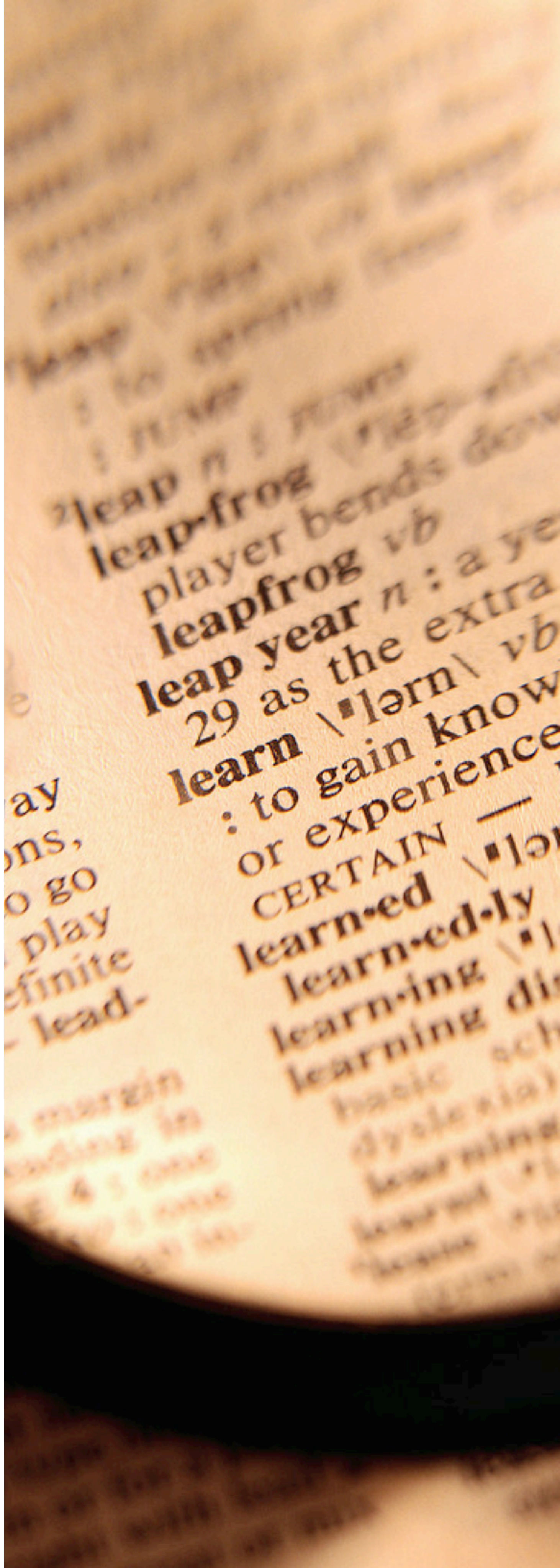
Naming Conventions

- **Data Item = Event? = Instance = Example**
  A single item of data (e.g. vector) that is „atomic“

- **Data Stream**
  A (possibly unbound) sequence of data items

- **Processor = Operator**
  Passive element that can be executed and will process a single data item

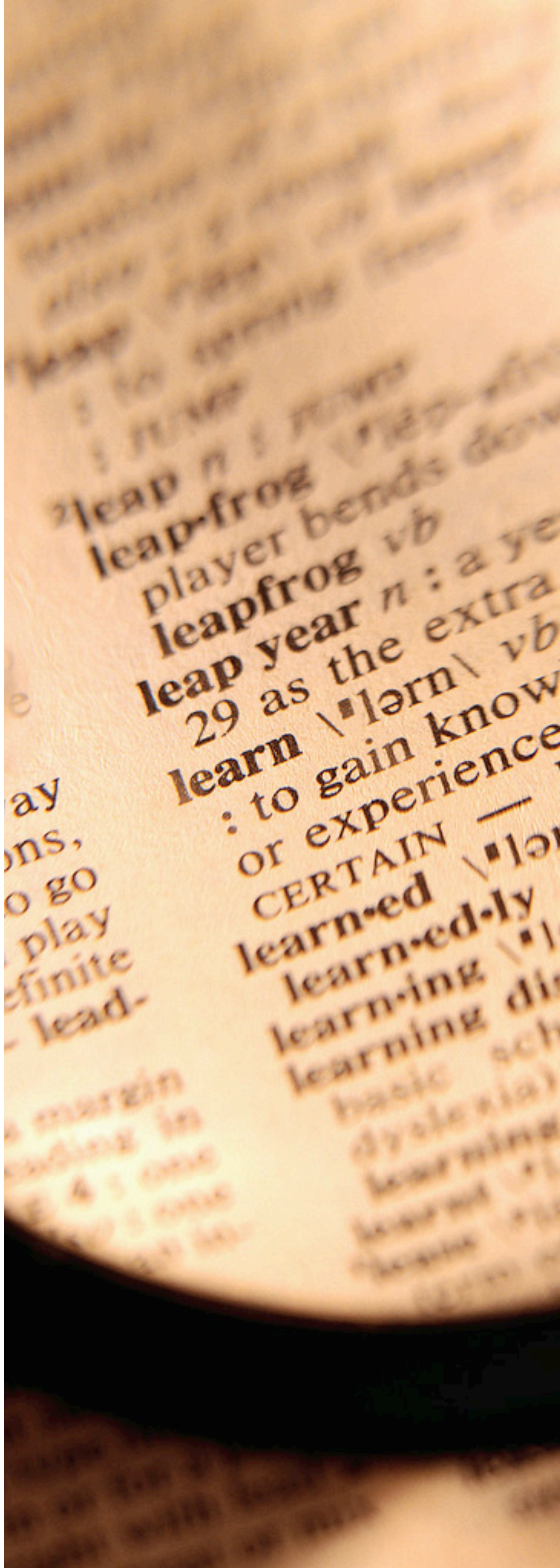
- **Process = ?**
  Active element (thread) that will read from some input (queue/stream) and execute processors



Naming Conventions

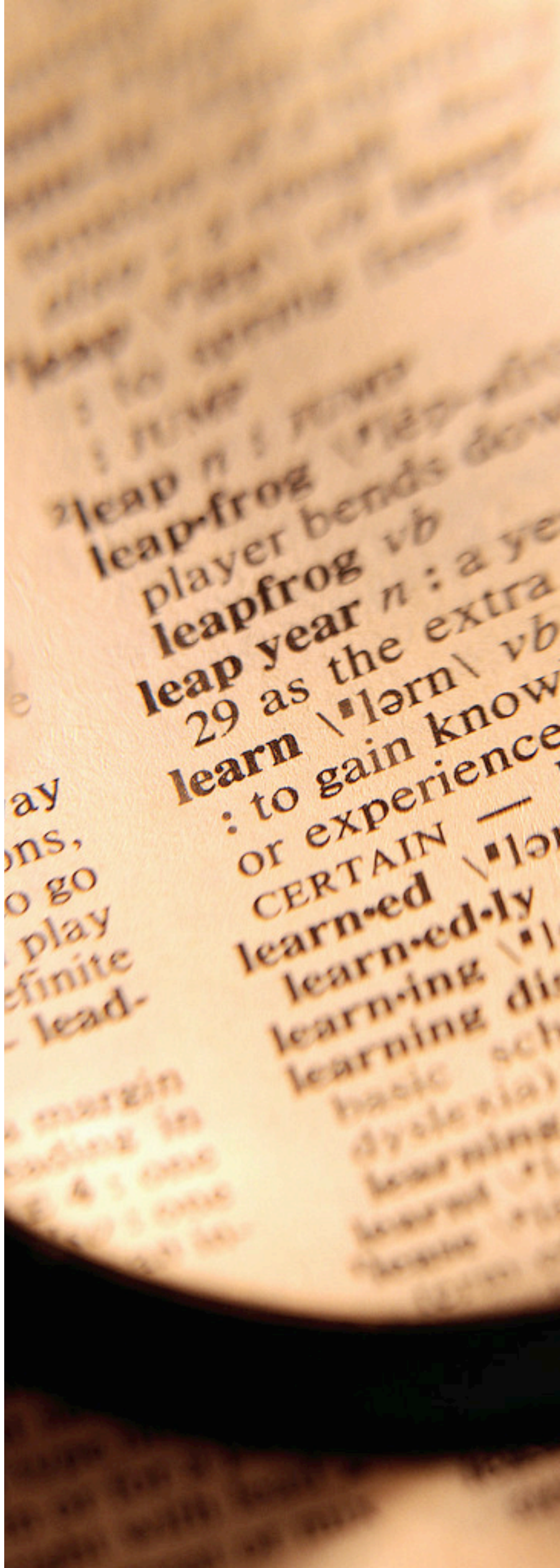
- **Service = Model = ?**
  Element that provides some functionality in a **thread-safe** manner (e.g. return copy of prediction-model)
  A processor CAN be a Service/Model/?
  Streams => Anytime Paradigm!!

- **Container = Runtime**
  An environment that contains multiple streams, processes and monitors



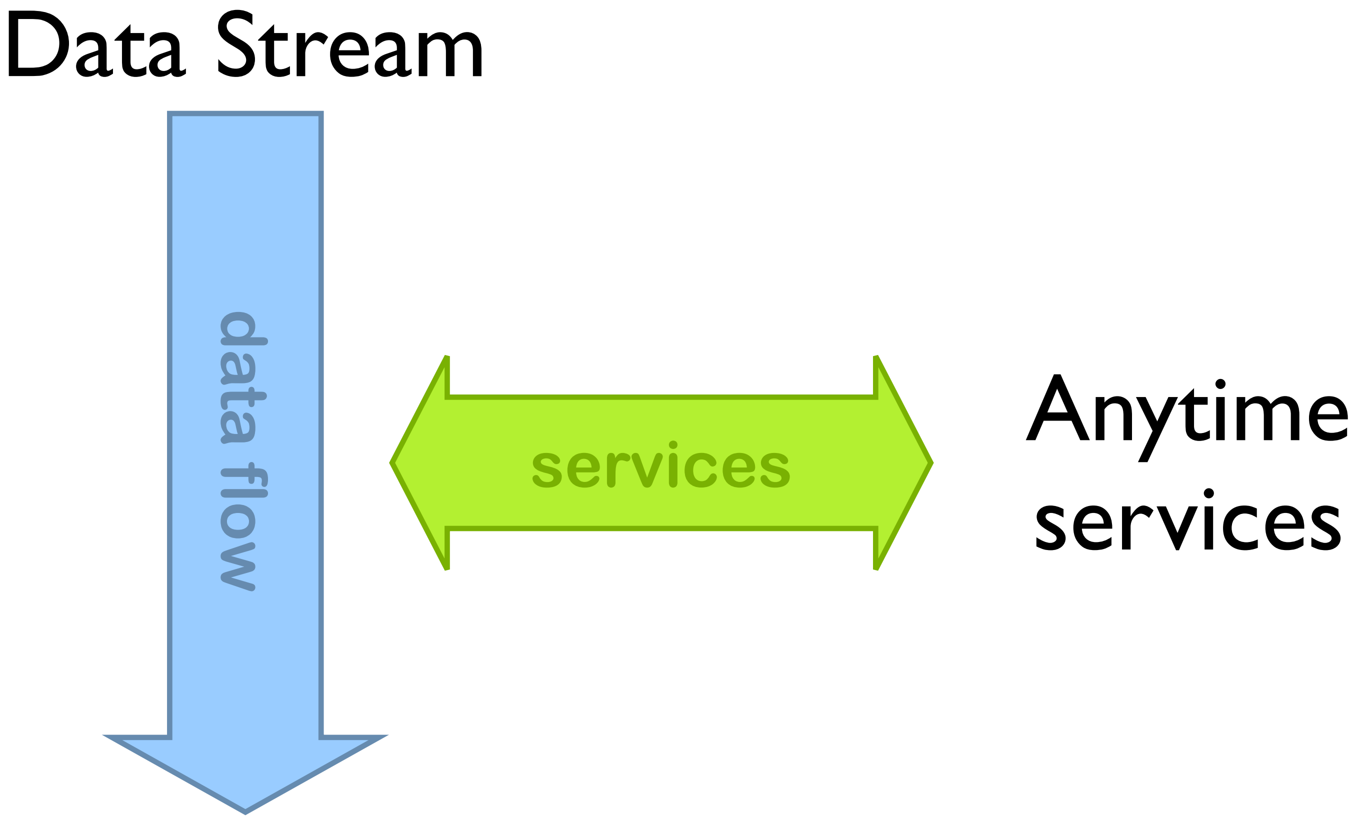
Anytime Services

- Stream/Online Algorithms provide services that can be queried at anytime
  - prediction services (class, outliers)
  - summaries (quantiles, top-k elements)
  - cluster mappings/clusterings



Data Stream Processing

- Two views of data stream processing:
  - **data flow**
  - **anytime service**

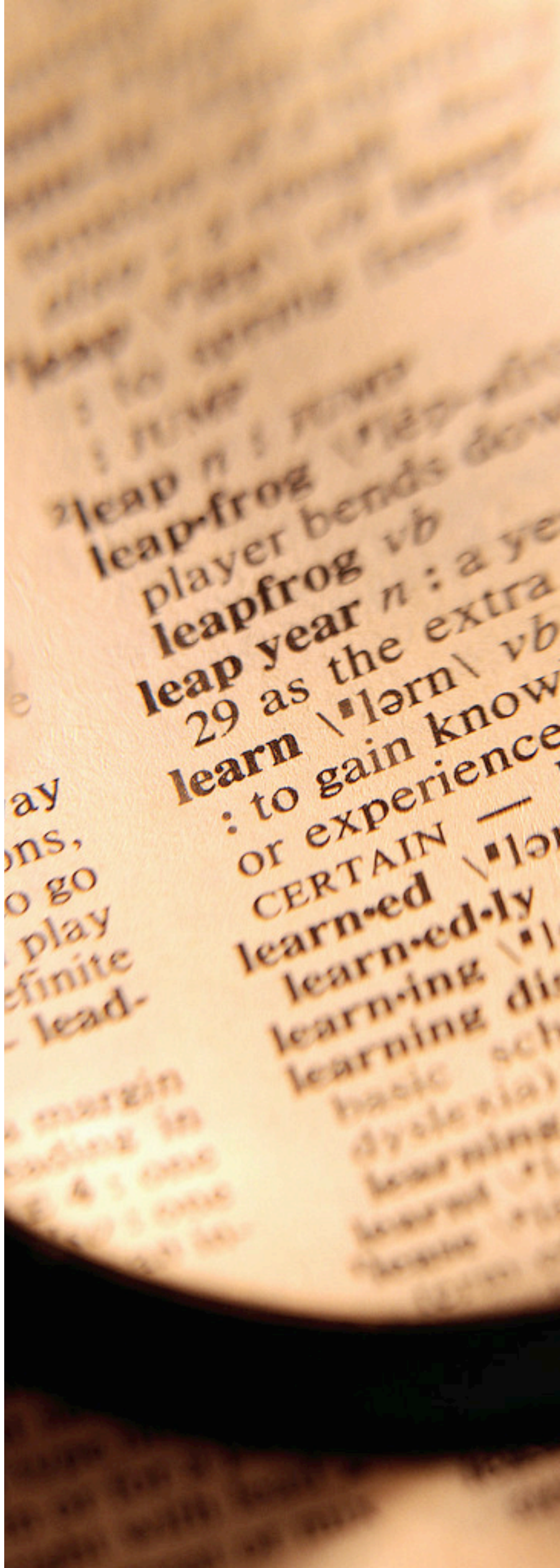


Data Stream

data flow

services

Anytime services



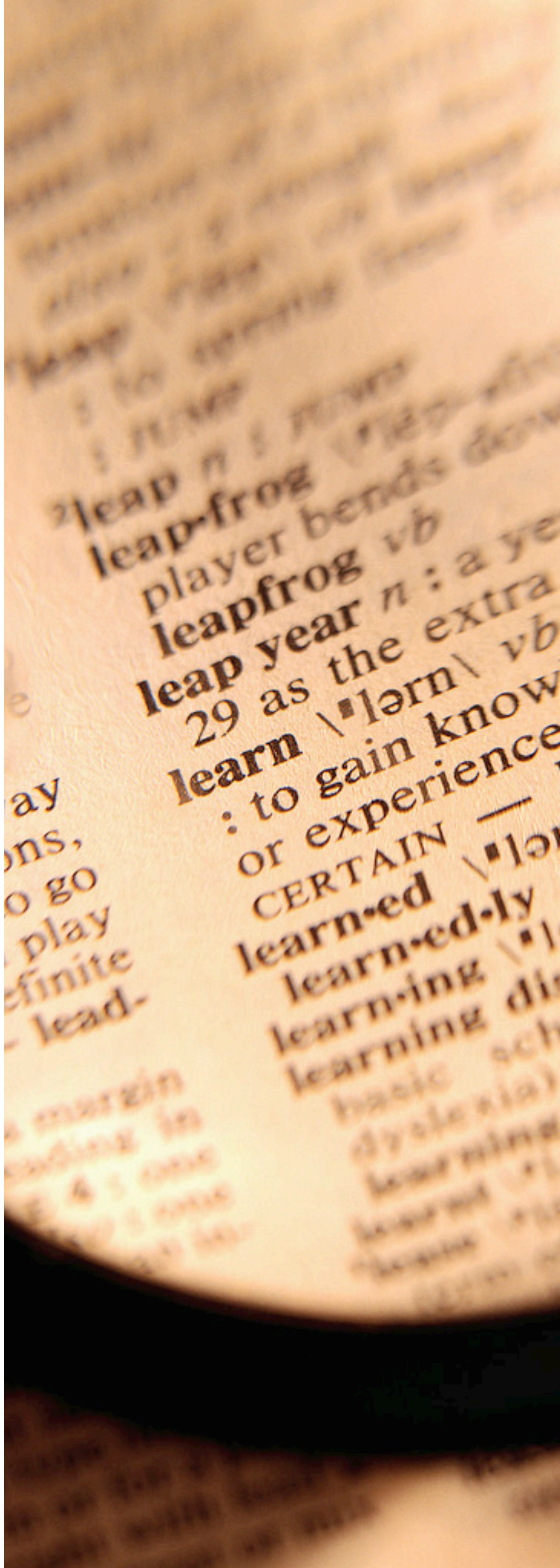
What do we want to do?

```
<Container id=“box“>

  <Stream id=“ds“ url=“file:/golf.csv“
          class=“stream.io.CsvStream“ />

  <Process input=“ds“>
     <Preprocessing />
     <Skip condition=“@label == null“ />
     <NaiveBayes id=“NaiveBayes“ />
  </Process>

<Container>
```



The „streams“ Project

- Split into three basic modules
- **stream-api**
  - provides basic interfaces and classes
- **stream-core**
  - includes streams (I/O), parsers, simple basic processors
- **stream-runtime**
  - Execution environment for stream experiments

The stream-api

- Derived from project group pg-542
- Available as open-source Maven project
    - building automatically downloads all libs
    - can be deployed to repository for everybody
    - follows **convention-over-configuration**
    - Most (all?) conventions can be customized by custom implementations
- Inspired by Maven, Tomcat, SOA,...

The stream-api - data flow

- **Data flow**
  - Data flow is provided by queues/streams
  - Processes typically sequential
- **Anytime services**
  - Provided by naming service
  - process elements register to naming service
  - online algorithms follow anytime paradigm
  - control flow is orthogonal to data flow

stream.data

- Data items - what is processed in a stream?
    - A simple hashmap called „Data“ (interface)

```
/*
 * A single Data item
 */
public interface Data
  extends Map<String,Serializable>,
          Serializable
{
    public static long serialVer...
}
```

stream.data

- **Why a Map?**
  - Available in any language
  - Concept understood by any „programmer“
  - Simple.
- **What is stored in a Map?**
  - java.lang.Double
  - java.lang.String
  - stream.data.tree.TreeNode (for SQL-trees)
  - Your serializable object

stream.io

- The stream-api provides some I/O classes for data-streams:
  - stream.io.CsvStream
  - stream.io.ArffStream
  - stream.io.SvmLightStream
  - stream.io.LineStream
  - stream.io.AccessLogStream
  - ...

stream.io

- stream.io.LineReader is more than reading lines
- It include a parser-generator for a simple grammar
  - For example the following format

```
<Stream class=“stream.io.LineStream“
    format=“%(IP) [%(DATE)] %(URL)“ />
```

will parse the data shown below and automatically set the attributes IP, DATE and URL

```
12.3.4.1 [2012/03/01 13:03:14] /index..
12.3.4.1 [2012/03/01 13:03:15] /image..
12.3.4.1 [2012/03/01 13:03:15] /style..
```

Conventions

- How do I store stuff in a map?
- Pick a name (CONVENTION !!!), the put it in:

```
{
    Data item = new DataImpl();
    item.put( key, „My String“ );
    item.put( key, new MyObject());
}
```

Conventions

- Map allows use of Python/Jython/JavaScript...

```
<JavaScript>
   data.put( “answer“, 42 );
</JavaScript>
```

- This in turn *might* ease rapid-prototyping for Physicists :-)

Conventions

- Pick your key-names with a convention in mind:
  - Each (key,value) pair is an (attribute,value) :-)
  - Golf data set:

```
outlook = rainy
temperature = mild
humidity = high
play = no
```

- What about special „attributes“?
  - I call them „annotations“, because they annotate the data
  - Should not be used by learners (convention)

Conventions

- Annotation keys start with an „@“
  - same as in Java‘s annotations
  - prefix determines the attribute role
- Labeled golf data:

```
outlook = rainy
temperature = mild
humidity = high
@label = no
```

Conventions

- This allows multiple labels:

```
outlook = rainy
temperature = mild
humidity = high
@label:umbrella = no
@label:play = yes
```

- Other annotations possible

```
...
@label:play = yes
@prediction:NB = no
@error:NB = 1.0
@outlier = true
```

Conventions

- But my attribute is already called „@something“!
  - The basic data structure is a Hashmap

```
// remove the attribute
value = data.remove(„@something“);

// put it back with a new name
data.put( „_at_something“, value );
```

Processing Data

- So lets start processing some data
- Simply write a Processor:

```
public class MyProcessor
   implements stream.Processor
{

  public Data process( Data item ){
     // do your work...
     return item;
  }
}
```

I need Parameters!!!

- Again, CONVENTIONS are your best friend:

```
public class MyProcessor
   implements stream.Processor
{
   ...
   Double lambda;

   public void setLambda( Double d ){
       lambda = d;
   }

   public Double getLambda(){
       return lambda;
   }
}
```

Parameters (Bean Convent.)

- Parameters from XML are automatically injected into the processors *before* init(..)

```
package my.package;

public class MyProcessor
   implements stream.Processor
{
   public void setLambda(Double d){..}
}
```

```
<my.package.MyProcessor
              lambda=“10.4“ />
```

Processing Data

- ConditionedProcessor provides flexible expressions for conditioned processing

```
package my.package;

public class MyProcessor
   extends stream.ConditionedProcessor
{
   ...
}
```

```
<my.package.MyProcessor
   condition="%{data.@label} = yes"
   lambda="10.4" />
```

Anytime Services

Anytime Services

- Data processors executed in data flow order...

- Processors (e.g. Learners) can provide *anytime services*
- Implemented as custom Interface

```
package stream;

public interface Service
   extends Remote
{
}
```

Anytime Services

- A simple counter service that provides the number of events processed

```
public interface CountService
   extends stream.Service
{
   public Long getNumberOfItems();
}
```

A simple Counter

- A processor that counts elements

```
public class MyCounter
  implements stream.Processor,
           CountService
{
  Long count = 0L;

  public Long getNumberOfItems(){
      return count;
  }

  public Data process(Data item){
      count++;
      return item;
  }
}
```

Using the Service

- A simple processor that **uses** the count-service

```
public class PrintCount
  implements stream.Processor,
{
 CountService counter;

 public void setCounter(CountService s){
   counter = s;
 }

 public Data process(Data item){
   ..println(counter.getNumberOfItems());
   return item;
 }
}
```

Setting it up

```
</Container>
  <Stream id="input" class="stream.io.CsvStream"
     url="http://kirmes.cs.../multi-golf.csv.gz" />

  <Process input="input">
     <my.package.MyCounter id="cnt">
     <my.package.PrintCount counter-ref="cnt">
  </Process>
</Container>
```

Setting it up

```
</Container>
  <Stream id="input" class="stream.io.CsvStream"
     url="http://kirmes.cs.../multi-golf.csv.gz" />

  <Process input="input">
     <my.package.MyCounter id="cnt">
     <my.package.PrintCount counter-ref="cnt">
  </Process>
</Container>
```

1. `counter-ref=“cnt“`
2. `lookup( cnt ) => CountService`
3. `setCounter( CountService )`

stream.runtime

- The stream-api provides a runtime environment to create processors/streams from XML

```
java -cp stream-runtime.jar:mylib.jar \
    stream.run my-processes.xml
```

- Automatically creates your processors, streams, sets parameters (e.g. setLambda(..))
- Starts all processes and waits until all have finished (e.g. completed processing their stream)

```
<Container>
 <Stream id="input" class="stream.io.CsvStream"
         url="http://kirmes.cs.../multi-golf.csv.gz" />

 <Process input="input">

   <!-- Renames ‘play‘ to ‘@label‘    -->
   <MapKeys from="play" to="@label" />

   <!-- use NaiveBayes Model for prediction -->
   <Prediction ref="NaiveBayes" />
   <NaiveBayes id="NaiveBayes" />

   <!--
     Adds @error:NaiveBayes by checking @label=@prediction:NaiveBayes
     -->
   <PredictionError learner="NaiveBayes" />

   <Average keys="@error:NaiveBayes" />

   <PrintData />
 </Process>
</Container>
```

How do I document my stuff?

- As simple as possible - use Markdown
- You code: `my/package/MyClass.java`
- Your doc: `my/package/MyClass.md`

```
CSVStream
=========

This data stream source reads simple comma
separated values from a file/url. Each line is
split using a separator (regular expression).

Lines starting with a hash character (`#`) are
regarded to be headers which define the names
of the columns.
```

The current stream-api 1.0

- The current state of the *stream-api* is
  - a multi-threaded runtime environment (XML)
  - several stream I/O classes (more to come)
  - some pre-processors (easy to implement)
  - local naming service
  - simply include it as maven dependency
- Work in progress:
  - Several learners being adapted from pg542
  - multi-server environment (remote naming)

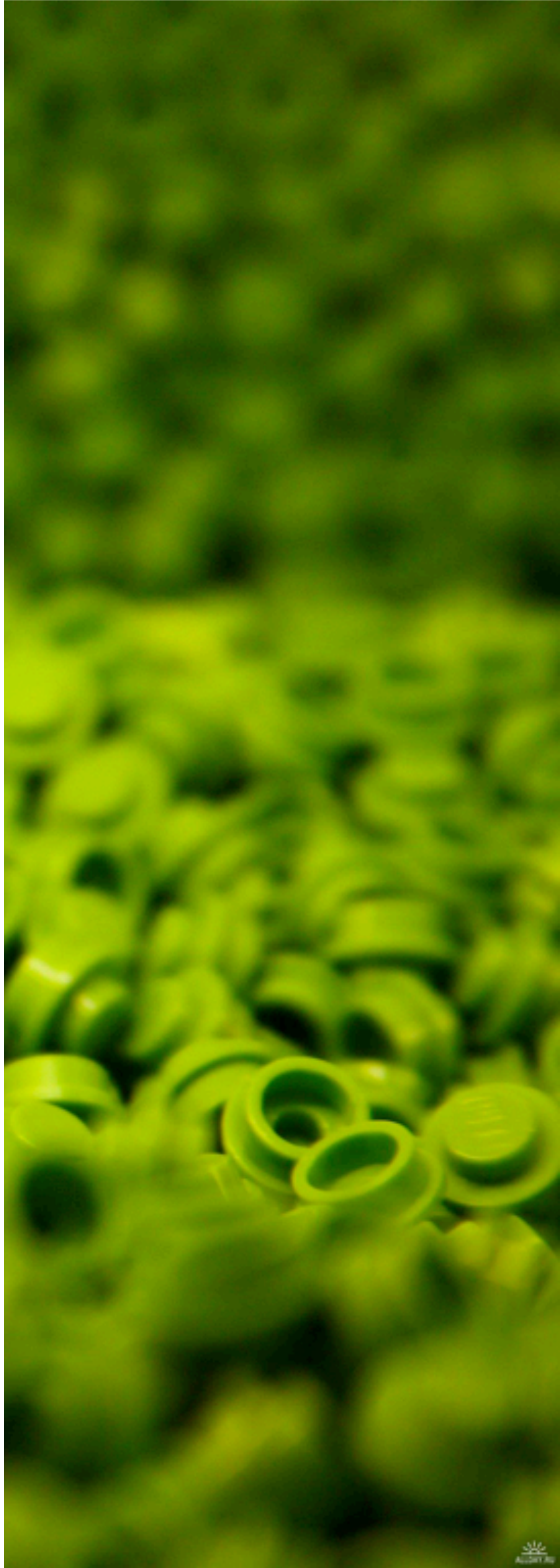
Fachprojekt auf bitbucket.org

Fachprojekt - bitbucket.org

- Maven-Projekt mit Beispiel-Code

`https://bitbucket.org/cbockermann/fachprojekt`

- Enthält einen CounterService, der für eine Menge von Keys (Attributen) die Elemente zählt
- XML in **`src/main/resources/example.xml`**
- Start-Klasse mit main-Methode (**`example.ExampleRun`**)



Fachprojekt - bitbucket.org

- Bauen des Fachprojektes mit Maven

```
# git clone https://...

# cd fachprojekt
# mvn assembly:assembly
```

- Starten eines XML files

```
# java -cp target/Fachprojekt.jar \
      file:test.xml
```