# DataStream API

## Basics

Apache Flink® Training

dataArtisans

Flink v1.3 – 19.06.2017

# DataStream API

- Stream Processing
- Java and Scala
- All examples here in Java for Flink 1.3
- Documentation available at

  flink.apache.org

# DataStream API by Example

# Window WordCount: main Method

```
public static void main(String[] args) throws Exception {
    // set up the execution environment
    final StreamExecutionEnvironment env =
          StreamExecutionEnvironment.getExecutionEnvironment();

    // configure event time
    env.setStreamTimeCharacteristic(TimeCharacteristic.EventTime);

    DataStream<Tuple2<String, Integer>> counts = env
            // read stream of words from socket
            .socketTextStream("localhost", 9999)
            // split up the lines in tuples containing: (word,1)
            .flatMap(new Splitter())
            // key stream by the tuple field "0"
            .keyBy(0)
            // compute counts every 5 minutes
            .timeWindow(Time.minutes(5))
            //sum up tuple field "1"
            .sum(1);

    // print result in command line
    counts.print();
    // execute program
    env.execute("Socket WordCount Example");
}
```

# Stream Execution Environment

```
public static void main(String[] args) throws Exception {
    // set up the execution environment
    final StreamExecutionEnvironment env =
         StreamExecutionEnvironment.getExecutionEnvironment();

    // configure event time
    env.setStreamTimeCharacteristic(TimeCharacteristic.EventTime);

    DataStream<Tuple2<String, Integer>> counts = env
            // read stream of words from socket
            .socketTextStream("localhost", 9999)
            // split up the lines in tuples containing: (word,1)
            .flatMap(new Splitter())
            // key stream by the tuple field "0"
            .keyBy(0)
            // compute counts every 5 minutes
            .timeWindow(Time.minutes(5))
            //sum up tuple field "1"
            .sum(1);

    // print result in command line
    counts.print();
    // execute program
    env.execute("Socket WordCount Example");
}
```

# Data Sources

```
public static void main(String[] args) throws Exception {
    // set up the execution environment
    final StreamExecutionEnvironment env =
          StreamExecutionEnvironment.getExecutionEnvironment();

    // configure event time
    env.setStreamTimeCharacteristic(TimeCharacteristic.EventTime);

    DataStream<Tuple2<String, Integer>> counts = env
            // read stream of words from socket
            .socketTextStream("localhost", 9999)
            // split up the lines in tuples containing: (word,1)
            .flatMap(new Splitter())
            // key stream by the tuple field "0"
            .keyBy(0)
            // compute counts every 5 minutes
            .timeWindow(Time.minutes(5))
            //sum up tuple field "1"
            .sum(1);

    // print result in command line
    counts.print();
    // execute program
    env.execute("Socket WordCount Example");
}
```

# Data types

```
public static void main(String[] args) throws Exception {
    // set up the execution environment
    final StreamExecutionEnvironment env =
          StreamExecutionEnvironment.getExecutionEnvironment();

    // configure event time
    env.setStreamTimeCharacteristic(TimeCharacteristic.EventTime);

    DataStream<Tuple2<String, Integer>> counts = env
            // read stream of words from socket
            .socketTextStream("localhost", 9999)
            // split up the lines in tuples containing: (word,1)
            .flatMap(new Splitter())
            // key stream by the tuple field "0"
            .keyBy(0)
            // compute counts every 5 minutes
            .timeWindow(Time.minutes(5))
            //sum up tuple field "1"
            .sum(1);

    // print result in command line
    counts.print();
    // execute program
    env.execute("Socket WordCount Example");
}
```

# Transformations

```
public static void main(String[] args) throws Exception {
    // set up the execution environment
    final StreamExecutionEnvironment env =
          StreamExecutionEnvironment.getExecutionEnvironment();

    // configure event time
    env.setStreamTimeCharacteristic(TimeCharacteristic.EventTime);

    DataStream<Tuple2<String, Integer>> counts = env
            // read stream of words from socket
            .socketTextStream("localhost", 9999)
            // split up the lines in tuples containing: (word,1)
            .flatMap(new Splitter())
            // key stream by the tuple field "0"
            .keyBy(0)
            // compute counts every 5 minutes
            .timeWindow(Time.minutes(5))
            //sum up tuple field "1"
            .sum(1);

    // print result in command line
    counts.print();
    // execute program
    env.execute("Socket WordCount Example");
}
```

# User functions

```
public static void main(String[] args) throws Exception {
    // set up the execution environment
    final StreamExecutionEnvironment env =
          StreamExecutionEnvironment.getExecutionEnvironment();

    // configure event time
    env.setStreamTimeCharacteristic(TimeCharacteristic.EventTime);

    DataStream<Tuple2<String, Integer>> counts = env
            // read stream of words from socket
            .socketTextStream("localhost", 9999)
            // split up the lines in tuples containing: (word,1)
            .flatMap(new Splitter())
            // key stream by the tuple field "0"
            .keyBy(0)
            // compute counts every 5 minutes
            .timeWindow(Time.minutes(5))
            //sum up tuple field "1"
            .sum(1);

    // print result in command line
    counts.print();
    // execute program
    env.execute("Socket WordCount Example");
}
```

# DataSinks

```
public static void main(String[] args) throws Exception {
    // set up the execution environment
    final StreamExecutionEnvironment env =
          StreamExecutionEnvironment.getExecutionEnvironment();

    // configure event time
    env.setStreamTimeCharacteristic(TimeCharacteristic.EventTime);

    DataStream<Tuple2<String, Integer>> counts = env
            // read stream of words from socket
            .socketTextStream("localhost", 9999)
            // split up the lines in tuples containing: (word,1)
            .flatMap(new Splitter())
            // key stream by the tuple field "0"
            .keyBy(0)
            // compute counts every 5 minutes
            .timeWindow(Time.minutes(5))
            //sum up tuple field "1"
            .sum(1);

    // print result in command line
    counts.print();
    // execute program
    env.execute("Socket WordCount Example");
}
```

# Execute!

```
public static void main(String[] args) throws Exception {
    // set up the execution environment
    final StreamExecutionEnvironment env =
          StreamExecutionEnvironment.getExecutionEnvironment();

    // configure event time
    env.setStreamTimeCharacteristic(TimeCharacteristic.EventTime);

    DataStream<Tuple2<String, Integer>> counts = env
            // read stream of words from socket
            .socketTextStream("localhost", 9999)
            // split up the lines in tuples containing: (word,1)
            .flatMap(new Splitter())
            // key stream by the tuple field "0"
            .keyBy(0)
            // compute counts every 5 minutes
            .timeWindow(Time.minutes(5))
            //sum up tuple field "1"
            .sum(1);

    // print result in command line
    counts.print();
    // execute program
    env.execute("Socket WordCount Example");
}
```

# Window WordCount: FlatMap

```
public static class Splitter
  implements FlatMapFunction<String, Tuple2<String, Integer>> {

  @Override
  public void flatMap(String value,
                      Collector<Tuple2<String, Integer>> out)
    throws Exception {
        // normalize and split the line
        String[] tokens = value.toLowerCase().split("\\W+");

        // emit the pairs
        for (String token : tokens) {
            if (token.length() > 0) {
                out.collect(
                   new Tuple2<String, Integer>(token, 1));
            }
        }
    }
}
```

# WordCount: Interface

```
public static class Splitter
  implements FlatMapFunction<String, Tuple2<String, Integer>> {

  @Override
  public void flatMap(String value,
                      Collector<Tuple2<String, Integer>> out)
    throws Exception {
        // normalize and split the line
        String[] tokens = value.toLowerCase().split("\\W+");

        // emit the pairs
        for (String token : tokens) {
            if (token.length() > 0) {
                out.collect(
                   new Tuple2<String, Integer>(token, 1));
            }
        }
    }
}
```

# WordCount: Types

```
public static class Splitter
  implements FlatMapFunction<String, Tuple2<String, Integer>> {

  @Override
  public void flatMap(String value,
                      Collector<Tuple2<String, Integer>> out)
    throws Exception {
        // normalize and split the line
        String[] tokens = value.toLowerCase().split("\\W+");

        // emit the pairs
        for (String token : tokens) {
            if (token.length() > 0) {
                out.collect(
                   new Tuple2<String, Integer>(token, 1));
            }
        }
    }
}
```

# WordCount: Collector

```
public static class Splitter
  implements FlatMapFunction<String, Tuple2<String, Integer>> {

  @Override
  public void flatMap(String value,
                      Collector<Tuple2<String, Integer>> out)
    throws Exception {
        // normalize and split the line
        String[] tokens = value.toLowerCase().split("\\W+");

        // emit the pairs
        for (String token : tokens) {
            if (token.length() > 0) {
                out.collect(
                   new Tuple2<String, Integer>(token, 1));
            }
        }
    }
}
```

What kind of data can Flink handle?

# DataStream API: Data Types

# Data Types

- Flink aims to be able to process data of any type
- DataSet and DataStream APIs share the same type system
- Basic Types
  - String, Long, Integer, Boolean, …
  - Arrays
- Composite Types
  - Tuples
  - POJOs
  - Scala Case Classes

# Tuples

- Easiest and most efficient way to encapsulate data
- Scala: use default Scala tuples (1 to 22 fields)
- Java: Tuple1 up to Tuple25

```
Tuple2<String, String> person =
  new Tuple2<>("Max", "Mustermann”);

Tuple3<String, String, Integer> person =
  new Tuple3<>("Max", "Mustermann", 42);

Tuple4<String, String, Integer, Boolean> person =
  new Tuple4<>("Max", "Mustermann", 42, true);

// zero based index!
String firstName = person.f0;
String secondName = person.f1;
Integer age = person.f2;
Boolean fired = person.f3;
```

# POJOs

- Any Java class that
  - Has an empty default constructor
  - Has publicly accessible fields
    - public field or default getter & setter

```
public class Person {
  public int id;
  public String name;
  public Person() {};
  public Person(int id, String name) {…};
}

DataStream<Person> p =
  env.fromElements(new Person(1, "Bob"));
```

# Case Classes (Scala)

- Scala case classes are natively supported

```
case class Person(id: Int, name: String)

d: DataStream[Person] =
      env.fromElements(Person(1, "Bob"))
```

# DataStream API: Operators

# Transformations: map & flatMap

```
DataStream<Integer> integers = env.fromElements(1, 2, 3, 4);

// Regular Map - Takes one element and produces one element
DataStream<Integer> doubleIntegers =
  integers.map(new MapFunction<Integer, Integer>() {
    @Override
    public Integer map(Integer value) {
      return value * 2;
    }
  });

doubleIntegers.print();
> 2, 4, 6, 8

// Flat Map - Takes one element and produces zero, one, or more elements
DataStream<Integer> doubleIntegers2 =
  integers.flatMap(new FlatMapFunction<Integer, Integer>() {
    @Override
    public void flatMap(Integer value, Collector<Integer> out) {
      out.collect(value * 2);
    }
  });

doubleIntegers2.print();
> 2, 4, 6, 8
```

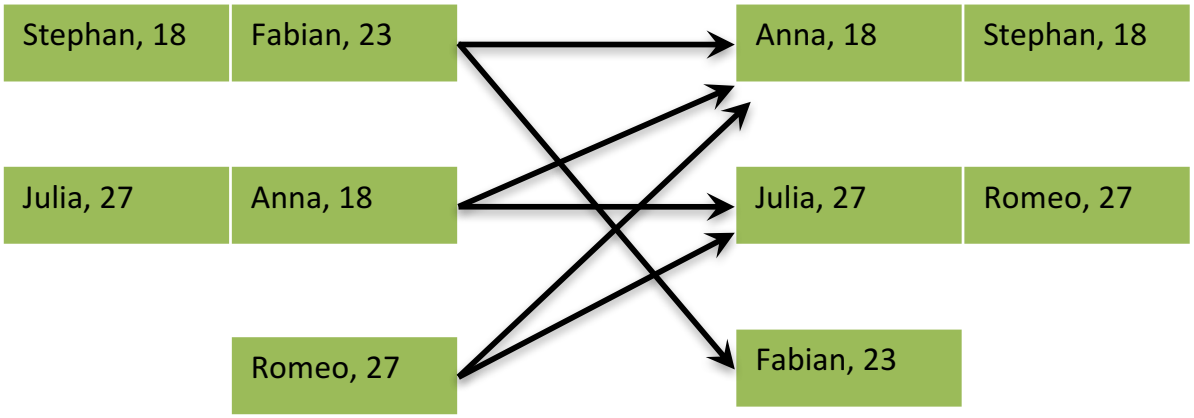
# Transformations: Filter

```
// The DataStream
DataStream<Integer> integers = env.fromElements(1, 2, 3, 4);

DataStream<Integer> filtered =
  integers.filter(new FilterFunction<Integer>() {
    @Override
    public boolean filter(Integer value) {
      return value != 3;
    }
  });

filtered.print();
> 1, 2, 4
```

# Transformations: KeyBy

- A DataStream can be organized by a key
  - Partitions the data, i.e., all elements with the same key are processed by the same operator
  - Certain operators are key-aware
  - Operator state can be partitioned by key

```
// (name, age) of passengers
DataStream<Tuple2<String, Integer>> passengers = …

// key by second field (age)
DataStream<Tuple2<String, Integer>> grouped = passengers.keyBy(1)
```



# Reduce (conceptually)

```
public Integer reduce(Integer a, Integer b) {
    return a + b;
}
```

[1, 2, 3, 4] ➔ reduce() means: (((1 + 2) + 3) + 4) = 10

# Reduce on a Stream

- Can only be used with keyed or windowed streams
- Example with `reduce` on a KeyedStream

```
// Produce running sums of the even and odd integers.

List<Tuple2<String, Integer>>
  data = new ArrayList<Tuple2<String, Integer>>();

data.add(new Tuple2<>("odd", 1));
data.add(new Tuple2<>("even", 2));
data.add(new Tuple2<>("odd", 3));
data.add(new Tuple2<>("even", 4));

DataStream<Tuple2<String, Integer>>
  tuples = env.fromCollection(data);

KeyedStream<Tuple2<String, Integer>, Tuple>
  odd_and_evens = tuples.keyBy(0);
```

# Reduce on a KeyedStream

```
DataStream<Tuple2<String, Integer>> sums =
  odd_and_evens.reduce(new ReduceFunction<Tuple2<String, Integer>>() {
    @Override
    public Tuple2<String, Integer> reduce(
      Tuple2<String, Integer> t1,
      Tuple2<String, Integer> t2) throws Exception {
          return new Tuple2<>(t1.f0, t1.f1 + t2.f1);
      }
  });

sums.print();
env.execute();

3> (odd,1)
3> (odd,4)
4> (even,2)
4> (even,6)
```

# Specifying Keys

# Keyed Streams

- keyBy() partitions a DataStream
  - a key is extracted from each element
- Basis of operating in parallel
- Data types used as keys must have valid implementations of `hashCode()` and `equals()`, which rules out arrays (for example)
- Composite types can be used as keys
  - all the fields must be key types
  - nested fields can also be used as keys

# Keys for Tuples

- Define keys by field position

```
DataStream<Tuple3<Integer, String, Double>> d = …

// key stream by String field
d.keyBy(1);
```

- Or field names

```
// key stream by Double field
d.keyBy("f2");
```

# Keys for POJOs

- Define keys by field name

```
DataStream<Person> d = …

// key stream by “name” field
d.keyBy("name");
```

# Keys for Case Classes (Scala)

- Define keys by field name

```
case class Person(id: Int, name: String)
d: DataStream[Person] = …

// key stream by field “name”
d.keyBy("name")
```

# Working With Multiple Streams

# Connected Streams

- Connect two streams to correlate them with each other
- Apply functions on connected streams to share state
- Typical use case is to use one stream for control and another for data

```
DataStream<String> control = …
DataStream<String> data = …

DataStream<String> result = control
  .connect(data)
  .flatMap(new MyCoFlatMap());
```

# FlatMap on Connected Streams

```
private static final class MyCoFlatMap
  implements CoFlatMapFunction<String, String, String> {
    HashSet blacklist = new HashSet();

    @Override
    public void flatMap1(String control_value, Collector<String> out) {
        blacklist.add(control_value);
        out.collect("listed " + control_value);
    }

    @Override
    public void flatMap2(String data_value, Collector<String> out) {
        if (blacklist.contains(data_value)) {
            out.collect("skipped " + data_value);
        } else {
            out.collect("passed " + data_value);
        }
    }
}
```

# FlatMap on Connected Streams

```
StreamExecutionEnvironment env =
  StreamExecutionEnvironment.getExecutionEnvironment();

DataStream<String> control =
    env.fromElements("DROP", "IGNORE");

DataStream<String> data =
    env.fromElements("data", "DROP", "artisans", "IGNORE");

DataStream<String> result = control
  .broadcast()
  .connect(data)
  .flatMap(new MyCoFlatMap());

result.print();
env.execute();
```

# FlatMap on Connected Streams

```
control = env.fromElements("DROP", "IGNORE");
data = env.fromElements("data", "DROP", "artisans", "IGNORE");

...

env.execute();

> listed DROP
> listed IGNORE
> passed data
> skipped DROP
> passed artisans
> skipped IGNORE
```

# Broadcast

```
DataStream<String> result = control
   .broadcast()
   .connect(data)
   .flatMap(new MyCoFlatMap());
```

- Events are replicated to all downstream operators
- This is not a magical, managed, replicated state solution
- And you have to consider the race condition implications

# Can also use Map on Connected Streams

```
DataStream<String> strings = …
DataStream<Integer> ints = …


ints.connect(strings)
    .map(new CoMapFunction<Integer, String, Boolean>() {
        @Override
        public Boolean map1 (Integer value) {
            return true;
        }
        @Override
        public Boolean map2 (String value) {
            return false;
        }
    });
```

# Rich Functions

# Rich Functions

- Function interfaces have only one method
  - Single abstract method (SAM)
  - Support for Java8 lambda functions

- There is a “Rich” variant of each function type
  - `RichFlatMapFunction`, …
  - Additional methods
    - `open(Configuration c)`
    - `close()`
    - `getRuntimeContext()`

# Rich Functions & RuntimeContext

- RuntimeContext has useful methods
  - `getIndexOfThisSubtask()`
  - `getNumberOfParallelSubtasks()`
  - `getExecutionConfig()`

- RuntimeContext also provides access to partitioned state (discussed later)
  - `getState()`

# Wrap-up

# Some tips

- Use `env.fromElements(..)` or `env.fromCollection(..)` to quickly create a DataStream to experiment with

- Use `print()` to print a DataStream

- Lazy execution can make debugging tricky, but you can use breakpoints in your IDE