**CHAPTER 5**

**The DataStream API (v1.7)**

This chapter introduces the basics of Flink’s DataStream API. We show the structure and components of a typical Flink streaming application, discuss Flink’s type systems and the supported data types, and present data and partitioning transformations. Window operators, time-based transformations, stateful operators, and connectors are discussed in the next chapters. After reading this chapter, you will know how to implement a stream processing application with basic functionality. Our code exam‐ ples use Scala for conciseness, but the Java API is mostly analogous (exceptions or special cases will be pointed out). We also provide complete example applications implemented in Java and Scala in our [GitHub repositories](https://github.com/streaming-with-flink/).

# Hello, Flink!

Let’s start with a simple example to get a first impression of what it is like to write streaming applications with the DataStream API. We will use this example to show‐ case the basic structure of a Flink program and introduce some important features of the DataStream API. Our example application ingests a stream of temperature meas‐ urements from multiple sensors.

First, let’s have a look at the data type we will be using to represent sensor readings:

**case class SensorReading**( id**: String**,

timestamp**: Long**, temperature**: Double**)

The program in [Example 5-1](#_bookmark0) converts the temperatures from Fahrenheit to Celsius and computes the average temperature every 5 seconds for each sensor.

*Example 5-1. Compute the average temperature every 5 seconds for a stream of sensors*

*// Scala object that defines the DataStream program in the main() method.*

**object AverageSensorReadings** {

*// main() defines and executes the DataStream program*

**def** main(args**: Array**[**String**]) {

*// set up the streaming execution environment*

**val** env **= StreamExecutionEnvironment**.getExecutionEnvironment

*// use event time for the application*

env.setStreamTimeCharacteristic(**TimeCharacteristic**.**EventTime**)

*// create a DataStream[SensorReading] from a stream source*

**val** sensorData**: DataStream**[**SensorReading**] **=** env

*// ingest sensor readings with a SensorSource SourceFunction*

.addSource(**new SensorSource**)

*// assign timestamps and watermarks (required for event time)*

.assignTimestampsAndWatermarks(**new SensorTimeAssigner**)

**val** avgTemp**: DataStream**[**SensorReading**] **=** sensorData

*// convert Fahrenheit to Celsius with an inline lambda function*

.map( r **=>** {

**val** celsius **=** (r.temperature - 32) \* (5.0 / 9.0)

**SensorReading**(r.id, r.timestamp, celsius)

} )

*// organize readings by sensor id*

.keyBy(**\_**.id)

*// group readings in 5 second tumbling windows*

.timeWindow(**Time**.seconds(5))

*// compute average temperature using a user-defined function*

.apply(**new TemperatureAverager**)

*// print result stream to standard out*

avgTemp.print()

*// execute application*

env.execute("Compute average sensor temperature")

}

}

You have probably already noticed that Flink programs are defined and submitted for execution in regular Scala or Java methods. Most commonly, this is done in a static

main method. In our example, we define the AverageSensorReadings object and include most of the application logic inside main().

To structure a typical Flink streaming application:

1. Set up the execution environment.
2. Read one or more streams from data sources.
3. Apply streaming transformations to implement the application logic.
4. Optionally output the result to one or more data sinks.
5. Execute the program.

We now look at these parts in detail.

## Set Up the Execution Environment

The first thing a Flink application needs to do is set up its *execution environment*. The execution environment determines whether the program is running on a local machine or on a cluster. In the DataStream API, the execution environment of an

application is represented by the StreamExecutionEnvironment. In our example, we retrieve the execution environment by calling the static getExecutionEnvironment() method. This method returns a local or remote environment, depending on the con‐

text in which the method is invoked. If the method is invoked from a submission cli‐ ent with a connection to a remote cluster, a remote execution environment is returned. Otherwise, it returns a local environment.

It is also possible to explicitly create local or remote execution environments as fol‐ lows:

*// create a local stream execution environment*

**val** localEnv**: StreamExecutionEnvironment.createLocalEnvironment**()

*// create a remote stream execution environment*

**val** remoteEnv **= StreamExecutionEnvironment**.createRemoteEnvironment( "host", *// hostname of JobManager*

1234, *// port of JobManager process*

"path/to/jarFile.jar") *// JAR file to ship to the JobManager*

Next, we use env.setStreamTimeCharacteristic(TimeCharacteristic.EventTime)

to instruct our program to interpret time semantics using event time. The execution

environment offers more configuration options, such as setting the program parallel‐ ism and enabling fault tolerance.

## Read an Input Stream

Once the execution environment has been configured, it is time to do some actual work and start processing streams. The StreamExecutionEnvironment provides methods to create stream sources that ingest data streams into the application. Data

streams can be ingested from sources such as message queues or files, or also be gen‐ erated on the fly.

In our example, we use:

**val** sensorData**: DataStream**[**SensorReading**] **=**

env.addSource(**new SensorSource**)

to connect to the source of the sensor measurements and create an initial DataStream of type SensorReading. Flink supports many data types, which we describe in the next section. Here, we use a Scala case class as the data type that we defined before. A

SensorReading contains the sensor ID, a timestamp denoting when the measurement was taken, and the measured temperature. The assignTimestampsAndWater marks(new SensorTimeAssigner) method assigns the timestamps and watermarks that are required for event time. The implementation details of SensorTimeAssigner

do not concern us right now.

## Apply Transformations

Once we have a DataStream, we can apply a transformation on it. There are different types of transformations. Some transformations can produce a new DataStream, pos‐ sibly of a different type, while other transformations do not modify the records of the

DataStream but reorganize it by partitioning or grouping. The logic of an application is defined by chaining transformations.

In our example, we first apply a map() transformation that converts the temperature of each sensor reading to Celsius. Then, we use the keyBy() transformation to parti‐ tion the sensor readings by their sensor ID. Next, we define a timeWindow() transfor‐ mation, which groups the sensor readings of each sensor ID partition into tumbling

windows of 5 seconds:

**val** avgTemp**: DataStream**[**SensorReading**] **=** sensorData

.map( r **=>** {

**val** celsius **=** (r.temperature - 32) \* (5.0 / 9.0)

**SensorReading**(r.id, r.timestamp, celsius)

} )

.keyBy(**\_**.id)

.timeWindow(**Time**.seconds(5))

.apply(**new TemperatureAverager**)

Window transformations are described in detail in [“Window Operators” on page 122](#_bookmark4). Finally, we apply a user-defined function that computes the average temperature on each window. We discuss the implementation of a user-defined function in a later sec‐ tion of this chapter.

## Output the Result

Streaming applications usually emit their results to some external system, such as Apache Kafka, a filesystem, or a database. Flink provides a well-maintained collection of stream sinks that can be used to write data to different systems. It is also possible to

implement your own streaming sinks. There are also applications that do not emit results but keep them internally to serve them via Flink’s queryable state feature.

In our example, the result is a DataStream[SensorReading] record. Every record contains an average temperature of a sensor over a period of 5 seconds. The result stream is written to the standard output by calling print():

avgTemp.print()



## Execute

Note that the choice of a streaming sink affects the end-to-end con‐ sistency of an application, whether the result of the application is provided with at-least once or exactly-once semantics. The end-to- end consistency of the application depends on the integration of the chosen stream sinks with Flink’s checkpointing algorithm. We will discuss this topic in more detail in [“Application Consistency](#_bookmark0) [Guarantees” on page 184](#_bookmark0).

When the application has been completely defined, it can be executed by call‐ ing StreamExecutionEnvironment.execute(). This is the last call in our example:

env.execute("Compute average sensor temperature")

Flink programs are executed lazily. That is, the API calls that create stream sources and transformations do not immediately trigger any data processing. Instead, the API calls construct an execution plan in the execution environment, which consists of the stream sources created from the environment and all transformations that were tran‐

sitively applied to these sources. Only when execute() is called does the system trig‐

ger the execution of the program.

The constructed plan is translated into a JobGraph and submitted to a JobManager for execution. Depending on the type of execution environment, a JobManager is started as a local thread (local execution environment) or the JobGraph is sent to a remote JobManager. If the JobManager runs remotely, the JobGraph must be shipped together with a JAR file that contains all classes and required dependencies of the application.

# Transformations

In this section we give an overview of the basic transformations of the DataStream API. Time-related operators such as window operators and other specialized trans‐ formations are described in later chapters. A stream transformation is applied on one or more streams and converts them into one or more output streams. Writing a Data‐

Stream API program essentially boils down to combining such transformations to create a dataflow graph that implements the application logic.

Most stream transformations are based on user-defined functions. The functions encapsulate the user application logic and define how the elements of the input stream are transformed into the elements of the output stream. Functions, such as

MapFunction in the following, are defined as classes that implement a

transformation-specific function interface:

**class MyMapFunction extends MapFunction**[**Int**, **Int**] {

**override def** map(value**: Int**)**: Int** = value + 1

}

The function interface defines the transformation method that needs to be imple‐ mented by the user, such as the map() method in the example above.

Most function interfaces are designed as SAM (single abstract method) interfaces and they can be implemented as Java 8 lambda functions. The Scala DataStream API also has built-in support for lambda functions. When presenting the transformations of the DataStream API, we show the interfaces for all function classes, but mostly use lambda functions instead of function classes in code examples for brevity.

The DataStream API provides transformations for the most common data transfor‐ mation operations. If you are familiar with batch data processing APIs, functional programming languages, or SQL you will find the API concepts very easy to grasp. We present the transformations of the DataStream API in four categories:

1. Basic transformations are transformations on individual events.
2. KeyedStream transformations are transformations that are applied to events in the context of a key.
3. Multistream transformations merge multiple streams into one stream or split one stream into multiple streams.
4. Distribution transformations reorganize stream events.

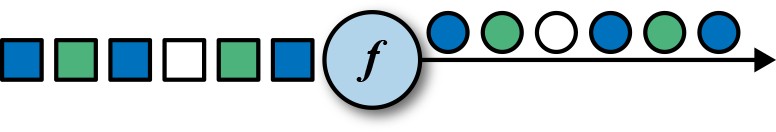
## Basic Transformations

Basic transformations process individual events, meaning that each output record was produced from a single input record. Simple value conversions, splitting of records, or filtering of records are examples of common basic functions. We explain their semantics and show code examples.

### Map

The map transformation is specified by calling the DataStream.map() method and produces a new DataStream. It passes each incoming event to a user-defined mapper

that returns exactly one output event, possibly of a different type. [Figure 5-1](#_bookmark1) shows a map transformation that converts every square into a circle.



*Figure 5-1. A map operation that transforms every square into a circle of the same color*

The MapFunction is typed to the types of the input and output events and can be specified using the MapFunction interface. It defines the map() method that trans‐ forms an input event into exactly one output event:

// T: the type of input elements

// O: the type of output elements MapFunction[T, O]

> map(T): O

The following is a simple mapper that extracts the first field (id) of each SensorRead ing in the input stream:

**val** readings**: DataStream**[**SensorReading**] **=** ...

**val** sensorIds**: DataStream**[**String**] **=** readings.map(**new MyMapFunction**)

**class MyMapFunction extends MapFunction**[**SensorReading**, **String**] {

**override def** map(r**: SensorReading**)**: String** = r.id

}

When using the Scala API or Java 8, the mapper can also be expressed as a lambda function:

**val** readings**: DataStream**[**SensorReading**] **=** ...

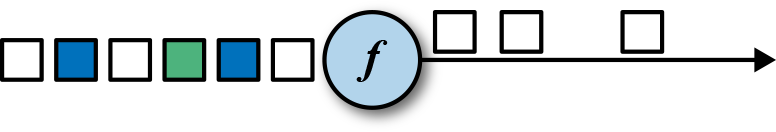
**val** sensorIds**: DataStream**[**String**] **=** readings.map(r **=>** r.id)

### Filter

The filter transformation drops or forwards events of a stream by evaluating a boolean condition on each input event. A return value of true preserves the input event and forwards it to the output, and false results in dropping the event. A filter transformation is specified by calling the DataStream.filter() method and pro‐

duces a new DataStream of the same type as the input DataStream. [Figure 5-2](#_bookmark15) shows

a filter operation that only preserves white squares.



*Figure 5-2. A filter operation that only retains white values*

The boolean condition is implemented as a function either using the FilterFunction interface or a lambda function. The FilterFunction interface is typed to the type of the input stream and defines the filter() method that is called with an input event and returns a boolean:

// T: the type of elements FilterFunction[T]

> filter(T): Boolean

The following example shows a filter that drops all sensor measurements with tem‐ perature below 25°F:

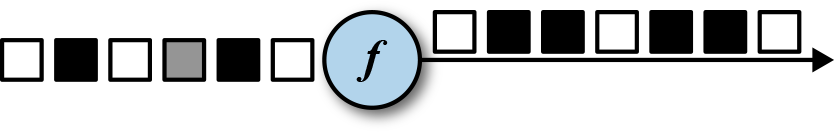
**val** readings**: DataStream**[**SensorReadings**] **=** ...

**val** filteredSensors **=** readings

.filter( r => r.temperature >= 25 )

### FlatMap

The flatMap transformation is similar to map, but it can produce zero, one, or more output events for each incoming event. In fact, the flatMap transformation is a gener‐ alization of filter and map and can be used to implement both those operations. [Figure 5-3](#_bookmark5) shows a flatMap operation that differentiates its output based on the color of the incoming event. If the input is a white square, it outputs the event unmodified. Black squares are duplicated, and gray squares are filtered out.



*Figure 5-3. A flatMap operation that outputs white squares, duplicates black squares, and drops gray squares*

The flatMap transformation applies a function on each incoming event. The corre‐ sponding FlatMapFunction defines the flatMap() method, which may return zero, one, or more events as results by passing them to the Collector object:

// T: the type of input elements

// O: the type of output elements FlatMapFunction[T, O]

> flatMap(T, Collector[O]): Unit

This example shows a flatMap transformation commonly found in data processing tutorials. The function is applied on a stream of sentences, splits each sentence by the space character, and emits each resulting word as an individual record:

**val** sentences**: DataStream**[**String**] **=** ...

**val** words**: DataStream**[**String**] **=** sentences

.flatMap(id **=>** id.split(" "))

## KeyedStream Transformations

A common requirement of many applications is to process groups of events that share a certain property together. The DataStream API features the abstraction of a

KeyedStream, which is a DataStream that has been logically partitioned into disjoint

substreams of events that share the same key.

Stateful transformations that are applied on a KeyedStream read from and write to state in the context of the currently processed event’s key. This means that all events

with the same key access the same state and thus can be processed together.

Note that stateful transformations and keyed aggregates have to be used with care. If the key domain is continuously growing—for example, because the key is a unique transaction ID—you have to clean up state for keys that are no longer active to avoid memory problems. Refer to [“Implementing Stateful Functions” on page](#_bookmark2) [154](#_bookmark2), which discusses stateful functions in detail.

A KeyedStream can be processed using the map, flatMap, and filter transformations that you saw earlier. In the following, we will use a keyBy transformation to convert a DataStream into a KeyedStream and keyed transformations such as rolling aggrega‐

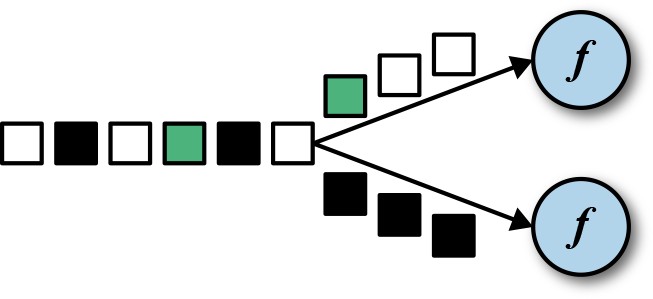
tions and reduce.

### keyBy

The keyBy transformation converts a DataStream into a KeyedStream by specifying a key. Based on the key, the events of the stream are assigned to partitions, so that all

events with the same key are processed by the same task of the subsequent operator. Events with different keys can be processed by the same task, but the keyed state of a task’s function is always accessed in the scope of the current event’s key.

Considering the color of the input event as the key, [Figure 5-4](#_bookmark11) assigns black events to one partition and all other events to another partition.



*Figure 5-4. A keyBy operation that partitions events based on color*

The keyBy() method receives an argument that specifies the key (or keys) to group by and returns a KeyedStream. There are different ways to specify keys. We cover

them in [“Defining Keys and Referencing Fields” on page 102](#_bookmark12). The following code declares the id field as the key of a stream of SensorReading records:

**val** readings**: DataStream**[**SensorReading**] **=** ...

**val** keyed**: KeyedStream**[**SensorReading**, **String**] **=** readings

.keyBy(r **=>** r.id)

The lambda function r => r.id extracts the id field of a sensor reading r.

### Rolling aggregations

Rolling aggregation transformations are applied on a KeyedStream and produce a DataStream of aggregates, such as sum, minimum, and maximum. A rolling aggre‐ gate operator keeps an aggregated value for every observed key. For each incoming

event, the operator updates the corresponding aggregate value and emits an event with the updated value. A rolling aggregation does not require a user-defined func‐ tion but receives an argument that specifies on which field the aggregate is computed. The DataStream API provides the following rolling aggregation methods:

sum()

A rolling sum of the input stream on the specified field.

min()

A rolling minimum of the input stream on the specified field.

max()

A rolling maximum of the input stream on the specified field.

minBy()

A rolling minimum of the input stream that returns the event with the lowest

value observed so far.

maxBy()

A rolling maximum of the input stream that returns the event with the highest

value observed so far.

It is not possible to combine multiple rolling aggregation methods—only a single roll‐ ing aggregate can be computed at a time.

Consider the following example of keying a stream of Tuple3[Int, Int, Int] on the first field and computing a rolling sum on the second field:

**val** inputStream**: DataStream**[(**Int**, **Int**, **Int**)] **=** env.fromElements( (1, 2, 2), (2, 3, 1), (2, 2, 4), (1, 5, 3))

**val** resultStream**: DataStream**[(**Int**, **Int**, **Int**)] **=** inputStream

.keyBy(0) *// key on first field of the tuple*

.sum(1) *// sum the second field of the tuple in place*

In this example the tuple input stream is keyed by the first field and the rolling sum is computed on the second field. The output of the example is (1,2,2) followed by (1,7,2) for the key “1” and (2,3,1) followed by (2,5,1) for the key “2.” The first

field is the common key, the second field is the sum, and the third field is not defined.

#### Only Use Rolling Aggregations on Bounded Key Domains

The rolling aggregate operator keeps a state for every key that is processed. Since this state is never cleaned up, you should only apply a rolling aggregations operator on a stream with a bounded key domain.

### Reduce

The reduce transformation is a generalization of the rolling aggregation. It applies a ReduceFunction on a KeyedStream, which combines each incoming event with the current reduced value, and produces a DataStream. A reduce transformation does

not change the type of the stream. The type of the output stream is the same as the

type of the input stream.

The function can be specified with a class that implements the ReduceFunction inter‐ face. ReduceFunction defines the reduce() method, which takes two input events and returns an event of the same type:

// T: the element type ReduceFunction[T]

> reduce(T, T): T

In the example below, the stream is keyed by language and the result is a continuously updated list of words per language:

**val** inputStream**: DataStream**[(**String**, **List**[**String**])] **=** env.fromElements( ("en", **List**("tea")), ("fr", **List**("vin")), ("en", **List**("cake")))

**val** resultStream**: DataStream**[(**String**, **List**[**String**])] **=** inputStream

.keyBy(0)

.reduce((x, y) **=>** (x.\_1, x.\_2 ::: y.\_2))

The lambda reduce function forwards the first tuple field (the key field) and concate‐ nates the List[String] values of the second tuple field.

#### Only Use Rolling Reduce on Bounded Key Domains

The rolling reduce operator keeps a state for every key that is pro‐ cessed. Since this state is never cleaned up, you should only apply a rolling reduce operator on a stream with a bounded key domain.

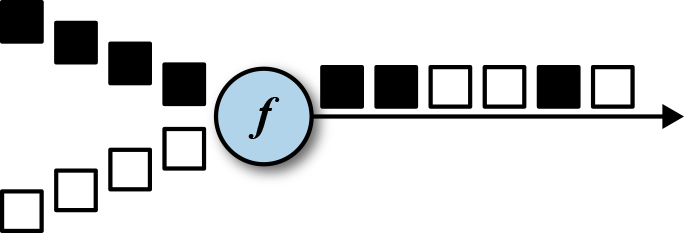
## Multistream Transformations

Many applications ingest multiple streams that need to be jointly processed or split a stream in order to apply different logic to different substreams. In the following, we discuss the DataStream API transformations that process multiple input streams or emit multiple output streams.

### Union

The DataStream.union() method merges two or more DataStreams of the same type and produces a new DataStream of the same type. Subsequent transformations pro‐ cess the elements of all input streams. [Figure 5-5](#_bookmark6) shows a union operation that

merges black and gray events into a single output stream.



*Figure 5-5. A union operation that merges two input streams into one*

The events are merged in a FIFO fashion—the operator does not produce a specific order of events. Moreover, the union operator does not perform duplication elimina‐ tion. Every input event is emitted to the next operator.

The following shows how to union three streams of type SensorReading into a single stream:

**val** parisStream**: DataStream**[**SensorReading**] **=** ... **val** tokyoStream**: DataStream**[**SensorReading**] **=** ... **val** rioStream**: DataStream**[**SensorReading**] **=** ...

**val** allCities**: DataStream**[**SensorReading**] **=** parisStream

.union(tokyoStream, rioStream)

### Connect, coMap, and coFlatMap

Combining events of two streams is a very common requirement in stream process‐ ing. Consider an application that monitors a forest area and outputs an alert when‐ ever there is a high risk of fire. The application receives the stream of temperature sensor readings you have seen previously and an additional stream of smoke level measurements. When the temperature is over a given threshold and the smoke level is high, the application emits a fire alert.

The DataStream API provides the connect transformation to support such use cases.[1](#_bookmark20) The DataStream.connect() method receives a DataStream and returns a Connected Streams object, which represents the two connected streams:

*// first stream*

**val** first**: DataStream**[**Int**] **=** ...

*// second stream*

**val** second**: DataStream**[**String**] **=** ...

*// connect streams*

**val** connected**: ConnectedStreams**[**Int**, **String**] **=** first.connect(second)

The ConnectedStreams object provides map() and flatMap() methods that expect a

CoMapFunction and CoFlatMapFunction as argument respectively.[2](#_bookmark19)

Both functions are typed on the types of the first and second input stream and on the type of the output stream and define two methods—one for each input. map1() and flatMap1() are called to process an event of the first input and map2() and flat Map2() are invoked to process an event of the second input:

// IN1: the type of the first input stream

// IN2: the type of the second input stream

// OUT: the type of the output elements CoMapFunction[IN1, IN2, OUT]

> map1(IN1): OUT

> map2(IN2): OUT

// IN1: the type of the first input stream

// IN2: the type of the second input stream

// OUT: the type of the output elements CoFlatMapFunction[IN1, IN2, OUT]

> flatMap1(IN1, Collector[OUT]): Unit

> flatMap2(IN2, Collector[OUT]): Unit

#### A Function Cannot Choose Which ConnectedStreams to Read

It is not possible to control the order in which the methods of a CoMapFunction or CoFlatMapFunction are called. Instead, a method is called as soon as an event has arrived via the corre‐

sponding input.

Joint processing of two streams usually requires that events of both streams are deter‐ ministically routed based on some condition to be processed by the same parallel

instance of an operator. By default, connect() does not establish a relationship

1. Flink features dedicated operators for time-based stream joins, which are discussed in [Chapter 6](#_bookmark3). The connect transformation and the cofunctions discussed in this section are more generic.
2. You can also apply a CoProcessFunction to ConnectedStreams. We discuss CoProcessFunction in [Chapter 6](#_bookmark3).

between the events of both streams so events of both streams are randomly assigned to operator instances. This behavior yields nondeterministic results and is usually

undesirable. In order to achieve deterministic transformations on ConnectedStreams, connect() can be combined with keyBy() or broadcast(). We first show the keyBy() case:

**val** one**: DataStream**[(**Int**, **Long**)] **=** ...

**val** two**: DataStream**[(**Int**, **String**)] **=** ...

*// keyBy two connected streams*

**val** keyedConnect1**: ConnectedStreams**[(**Int**, **Long**), (**Int**, **String**)] **=** one

.connect(two)

.keyBy(0, 0) *// key both input streams on first attribute*

*// alternative: connect two keyed streams*

**val** keyedConnect2**: ConnectedStreams**[(**Int**, **Long**), (**Int**, **String**)] **=** one.keyBy(0)

.connect(two.keyBy(0)

Regardless of whether you keyBy() ConnectedStreams or you connect() two Keyed Streams, the connect() transformation will route all events from both streams with the same key to the same operator instance. Note that the keys of both streams should

refer to the same class of entities, just like a join predicate in a SQL query. An opera‐ tor that is applied on a connected and keyed stream has access to keyed state.[3](#_bookmark18)

The next example shows how to connect a (nonkeyed) DataStream with a broadcas‐ ted stream:

**val** first**: DataStream**[(**Int**, **Long**)] **=** ...

**val** second**: DataStream**[(**Int**, **String**)] **=** ...

*// connect streams with broadcast*

**val** keyedConnect**: ConnectedStreams**[(**Int**, **Long**), (**Int**, **String**)] **=** first

*// broadcast second input stream*

.connect(second.broadcast())

All events of the broadcasted stream are replicated and sent to all parallel operator instances of the subsequent processing function. The events of the nonbroadcasted stream are simply forwarded. Hence, the elements of both input streams can be jointly processed.

1. See [Chapter 8](#_bookmark3) for details on keyed state.



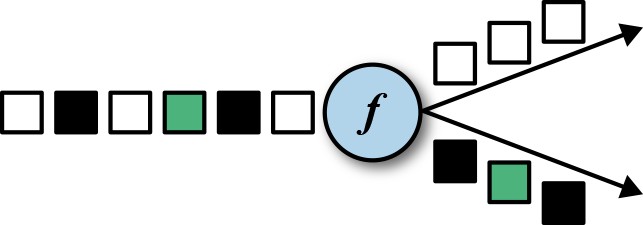
### Split and select

You can use broadcast state to connect a keyed and a broadcast stream. Broadcast state is an improved version of the broadcast()- connect() transformation. It also supports connecting a keyed and

a broadcasted stream and storing the broadcasted events in man‐

aged state. This allows you to implement operators that are dynam‐ ically configured via a data stream (e.g., to add or remove filtering rules or update machine-learning models). The broadcast state is discussed in detail in [“Using Connected Broadcast State” on page](#_bookmark16) [160](#_bookmark16).

Split is the inverse transformation to the union transformation. It divides an input stream into two or more output streams of the same type as the input stream. Each incoming event can be routed to zero, one, or more output streams. Hence, split can also be used to filter or replicate events. [Figure 5-6](#_bookmark7) shows a split operator that routes all white events into a separate stream than the rest.



*Figure 5-6. A split operation that splits the input stream into a stream of white events and a stream of others*

The DataStream.split() method receives an OutputSelector that defines how stream elements are assigned to named outputs. The OutputSelector defines the select() method that is called for each input event and returns a java.lang.Itera ble[String]. The String values that are returned for a record specify the output

streams to which the record is routed.

// IN: the type of the split elements OutputSelector[IN]

> select(IN): Iterable[String]

The DataStream.split() method returns a SplitStream, which provides a select() method to select one or more streams from the SplitStream by specify‐ ing the output names.

[Example 5-2](#_bookmark0) splits a stream of numbers into a stream of large numbers and a stream of small numbers.

*Example 5-2. Split a tuple stream into a stream with large numbers and a stream with small numbers.*

**val** inputStream**: DataStream**[(**Int**, **String**)] **=** ...

**val** splitted**: SplitStream**[(**Int**, **String**)] **=** inputStream

.split(t **=> if** (t.\_1 > 1000) **Seq**("large") **else Seq**("small"))

**val** large**: DataStream**[(**Int**, **String**)] **=** splitted.select("large")

**val** small**: DataStream**[(**Int**, **String**)] **=** splitted.select("small")

**val** all**: DataStream**[(**Int**, **String**)] **=** splitted.select("small", "large")

One restriction of the split transformation is that all outgoing streams are of the same type as the input type. In [“Emitting to Side](#_bookmark8) [Outputs” on page 119](#_bookmark8), we present the side-output feature of the process functions, which can emit multiple streams of different types from a function.

## Distribution Transformations

Partitioning transformations correspond to the data exchange strategies we intro‐ duced in [“Data Exchange Strategies” on page 19](#_bookmark0). These operations define how events are assigned to tasks. When building applications with the DataStream API the sys‐ tem automatically chooses data partitioning strategies and routes data to the correct destination depending on the operation semantics and the configured parallelism. Sometimes it is necessary or desirable to control the partitioning strategies at the application level or define custom partitioners. For instance, if we know that the load

of the parallel partitions of a DataStream is skewed, we might want to rebalance the data to evenly distribute the computation load of subsequent operators. Alternatively,

the application logic might require that all tasks of an operation receive the same data or that events be distributed following a custom strategy. In this section, we

present DataStream methods that enable users to control partitioning strategies or

define their own.

Note that keyBy() is different from the distribution transforma‐ tions discussed in this section. All transformations in this section produce a DataStream whereas keyBy() results in a KeyedStream, on which transformation with access to keyed state can be applied.

*Random*

The random data exchange strategy is implemented by the DataStream.shuf fle() method. The method distributes records randomly according to a uniform distribution to the parallel tasks of the following operator.

*Round-Robin*

The rebalance() method partitions the input stream so that events are evenly distributed to successor tasks in a round-robin fashion. [Figure 5-7](#_bookmark9) illustrates the

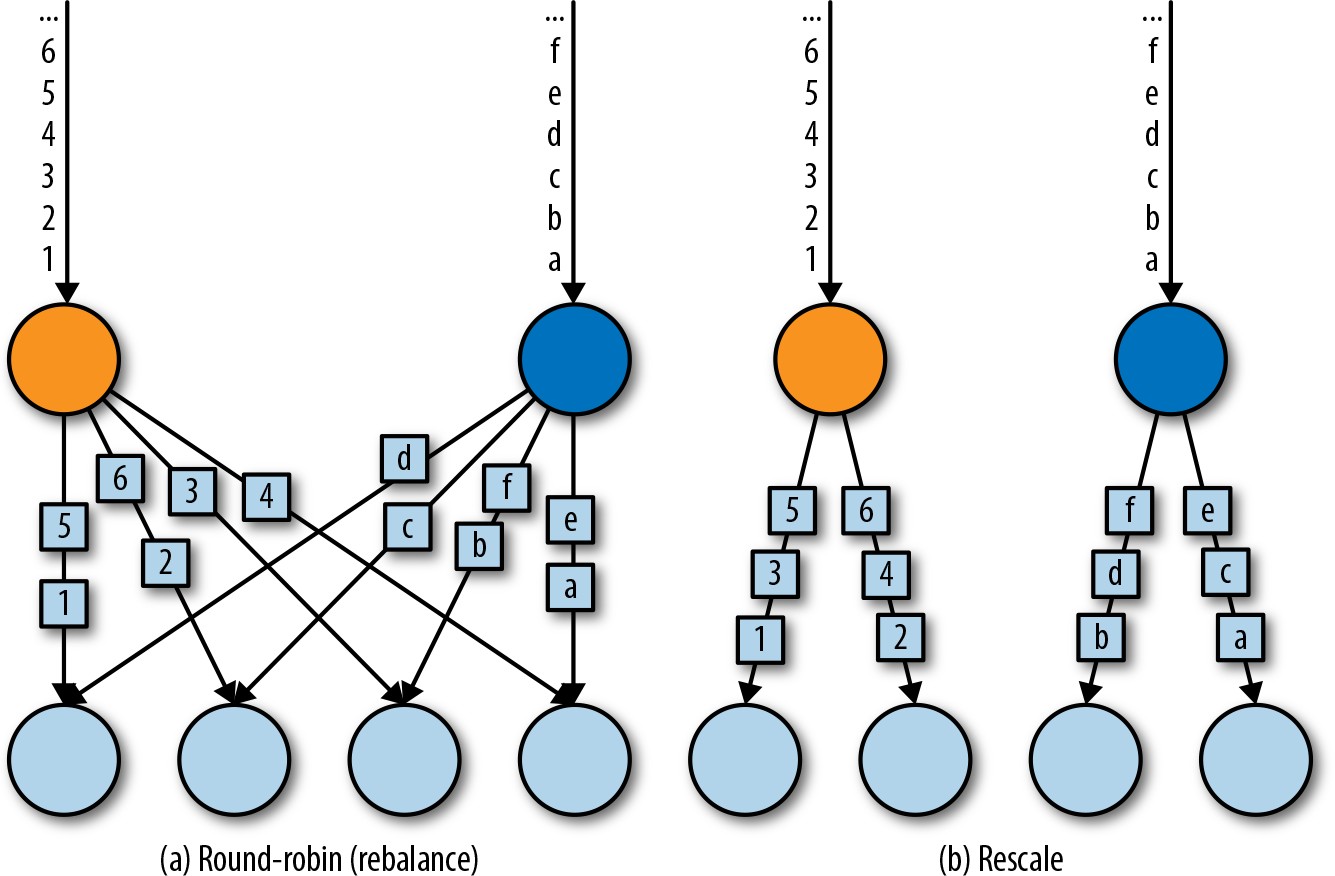
round-robin distribution transformation.

*Rescale*

The rescale() method also distributes events in a round-robin fashion, but only to a subset of successor tasks. In essence, the rescale partitioning strategy offers a

way to perform a lightweight load rebalance when the number of sender and receiver tasks is not the same. The rescale transformation is more efficient if the number of receiver tasks is a multitude of the number of sender tasks or vice versa.

The fundamental difference between rebalance() and rescale() lies in the way task connections are formed. While rebalance() will create communication channels between all sending tasks to all receiving tasks, rescale() will only cre‐ ate channels from each task to some of the tasks of the downstream operator. The

connection pattern of the rescale distribution transformation is shown in [Figure 5-7](#_bookmark9).

*Figure 5-7. Rebalance and rescale transformations Broadcast*

The broadcast() method replicates the input data stream so that all events are sent to all parallel tasks of the downstream operator.

*Global*

The global() method sends all events of the input data stream to the first paral‐ lel task of the downstream operator. This partitioning strategy must be used with

care, as routing all events to the same task might impact application perfor‐ mance.

*Custom*

When none of the predefined partitioning strategies is suitable, you can define your own by using the partitionCustom() method. This method receives a Par titioner object that implements the partitioning logic and the field or key posi‐

tion on which the stream is to be partitioned. The following example partitions a

stream of integers so that all negative numbers are sent to the first task and all other numbers are sent to a random task:

**val** numbers**: DataStream**[(**Int**)] **=** ... numbers.partitionCustom(myPartitioner, 0)

**object myPartitioner extends Partitioner**[**Int**] {

**val** r **=** scala.util.**Random**

**override def** partition(key**: Int**, numPartitions**: Int**)**: Int** = {

**if** (key < 0) 0 **else** r.nextInt(numPartitions)

}

}

# Setting the Parallelism

Flink applications are executed in parallel in a distributed environment such as a cluster of machines. When a DataStream program is submitted to the JobManager for execution, the system creates a dataflow graph and prepares the operators for execu‐ tion. Each operator is parallelized into one or multiple tasks. Each task will process a subset of the operator’s input stream. The number of parallel tasks of an operator is called the parallelism of the operator. It determines how much the operator’s process‐ ing effort can be distributed and also how much data it can process.

The parallelism of an operator can be controlled at the level of the execution environ‐ ment or per individual operator. By default, the parallelism of all operators of an application is set as the parallelism of the application’s execution environment. The parallelism of the environment (and thus also the default parallelism of all operators) is automatically initialized based on the context in which the application is started. If the application runs in a local execution environment the parallelism is set to match the number of CPU cores. When submitting an application to a running Flink cluster, the environment parallelism is set to the default parallelism of the cluster unless it is explicitly specified via the submission client (see [“Running and Managing Streaming](#_bookmark14) [Applications” on page 245](#_bookmark14) for more details).

In general, it is a good idea to define the parallelism of your operators relative to the default parallelism of the environment. This allows you to easily scale the application by adjusting its parallelism via the submission client. You can access the default paral‐ lelism of the environment as shown in the following example:

**val** env**: StreamExecutionEnvironment.getExecutionEnvironment**

*// get default parallelism as configured in the cluster config or*

*// explicitly specified via the submission client.*

**val** defaultP **=** env.env.getParallelism

You can also override the default parallelism of the environment, but you will no longer be able to control the parallelism of your application via the submission client:

**val** env**: StreamExecutionEnvironment.getExecutionEnvironment**

*// set parallelism of the environment*

env.setParallelism(32)

The default parallelism of an operator can be overridden by specifying it explicitly. In the following example, the source operator will be executed with the default parallel‐ ism of the environment, the map transformation has double as many tasks as the source, and the sink operation will always be executed by two parallel tasks:

**val** env **= StreamExecutionEnvironment**.getExecutionEnvironment

*// get default parallelism*

**val** defaultP **=** env.getParallelism

*// the source runs with the default parallelism*

**val** result**:** = env.addSource(**new CustomSource**)

*// the map parallelism is set to double the default parallelism*

.map(**new MyMapper**).setParallelism(defaultP \* 2)

*// the print sink parallelism is fixed to 2*

.print().setParallelism(2)

When you submit the application via a submission client and specify the parallelism to be 16, the source will run with a parallelism of 16, the mapper will run with 32 tasks, and the sink will run with 2 tasks. If you run the application in a local environ‐ ment—or example, from your IDE—on a machine with 8 cores, the source task will run with 8 tasks, the mapper with 16 tasks, and the sink with 2 tasks.

# Types

Flink DataStream applications process streams of events that are represented as data objects. Functions are called with data objects and emit data objects. Internally, Flink needs to be able to handle these objects. They need to be serialized and deserialized to ship them over the network or write them into or read them from state backends, checkpoints, and savepoints. In order to do this efficiently, Flink requires detailed knowledge of the types of data the application processes. Flink uses the concept of

type information to represent data types and generate specific serializers, deserializ‐ ers, and comparators for every data type.

Flink also features a type extraction system that analyzes the input and return types of functions to automatically obtain type information and hence serializers and deserial‐ izers. However, in certain situations, such as lambda functions or generic types, it is necessary to explicitly provide type information to make an application work or improve its performance.

In this section, we discuss the types supported by Flink, how to create type informa‐ tion for a data type, and how to help Flink’s type system with hints if it cannot auto‐ matically infer the return type of a function.

## Supported Data Types

Flink supports all common data types that are available in Java and Scala. The most widely used types can be grouped into the following categories:

* + Primitives
  + Java and Scala tuples
  + Scala case classes
  + POJOs, including classes generated by Apache Avro
  + Some special types

Types that are not specially handled are treated as generic types and serialized using the [Kryo serialization framework](https://github.com/EsotericSoftware/kryo).

#### Only Use Kryo as a Fallback Solution

Note that you should avoid using Kryo if possible. Since Kryo is a general-purpose serializer it is usually not very efficient. Flink pro‐ vides configuration options to improve the efficiency by preregis‐ tering classes to Kryo. Moreover, Kryo does not provide a good migration path to evolve data types.

Let’s look at each type category.

*Primitives*

All Java and Scala primitive types, such as Int (or Integer for Java), String, and Double, are supported. Here is an example that processes a stream of Long values and increments each element:

**val** numbers**: DataStream**[**Long**] **=** env.fromElements(1L, 2L, 3L, 4L) numbers.map( n **=>** n + 1)

*Java and Scala tuples*

Tuples are composite data types that consist of a fixed number of typed fields.

The Scala DataStream API uses regular Scala tuples. The following example fil‐ ters a DataStream of tuples with two fields:

*// DataStream of Tuple2[String, Integer] for Person(name, age)* **val** persons**: DataStream**[(**String**, **Integer**)] **=** env.fromElements( ("Adam", 17),

("Sarah", 23))

*// filter for persons of age > 18*

persons.filter(p **=>** p.\_2 > 18)

Flink provides efficient implementations of Java tuples. Flink’s Java tuples can have up to 25 fields, with each length is implemented as a separate class—Tuple1, Tuple2, up to Tuple25. The tuple classes are strongly typed.

We can rewrite the filtering example in the Java DataStream API as follows:

*// DataStream of Tuple2<String, Integer> for Person(name, age)* DataStream<Tuple2<String, Integer>> persons = env.fromElements( Tuple2.of("Adam", 17),

Tuple2.of("Sarah", 23));

*// filter for persons of age > 18*

persons.filter(p -> p.f1 > 18);

Tuple fields can be accessed by the name of their public fields—f0, f1, f2, etc., as shown earlier—or by position using the getField(int pos) method, where indexes start at 0:

Tuple2<String, Integer> personTuple = Tuple2.of("Alex", "42"); Integer age = personTuple.getField(1); *// age = 42*

In contrast to their Scala counterparts, Flink’s Java tuples are mutable, so the val‐ ues of fields can be reassigned. Functions can reuse Java tuples in order to reduce the pressure on the garbage collector. The following example shows how to update a field of a Java tuple:

personTuple.f1 = 42; *// set the 2nd field to 42*

personTuple.setField(43, 1); *// set the 2nd field to 43*

*Scala case classes*

Flink supports Scala case classes. Case class fields are accessed by name. In the following, we define a case class Person with two fields: name and age. As for the tuples, we filter the DataStream by age:

**case class Person**(name**: String**, age**: Int**)

**val** persons**: DataStream**[**Person**] **=** env.fromElements(

**Person**("Adam", 17),

**Person**("Sarah", 23))

*// filter for persons with age > 18*

persons.filter(p **=>** p.age > 18)

*POJOs*

Flink analyzes each type that does not fall into any category and checks to see if it can be identified and handled as a POJO type. Flink accepts a class as a POJO if it satisfies the following conditions:

* + - It is a public class.
    - It has a public constructor without any arguments—a default constructor.
    - All fields are public or accessible through getters and setters. The getter and setter functions must follow the default naming scheme, which is Y getX() and setX(Y x) for a field x of type Y.
    - All fields have types that are supported by Flink.

For example, the following Java class will be identified as a POJO by Flink:

**public class Person** {

*// both fields are public* **public** String name; **public int** age;

*// default constructor is present*

**public** Person() {}

**public** Person(String name, **int** age) {

**this**.name = name;

**this**.age = age;

}

}

DataStream<Person> persons = env.fromElements(

**new** Person("Alex", 42),

**new** Person("Wendy", 23));

Avro-generated classes are automatically identified by Flink and handled as POJOs.

*Arrays, Lists, Maps, Enums, and other special types*

Flink supports several special-purpose types, such as primitive and object Array types; Java’s ArrayList, HashMap, and Enum types; and Hadoop Writable types. Moreover, it provides type information for Scala’s Either, Option, and Try types, and Flink’s Java version of the Either type.

## Creating Type Information for Data Types

The central class in Flink’s type system is TypeInformation. It provides the system with the necessary information it needs to generate serialiazers and comparators. For

instance, when you join or group by some key, TypeInformation allows Flink to per‐ form the semantic check of whether the fields used as keys are valid.

When an application is submitted for execution, Flink’s type system tries to automati‐ cally derive the TypeInformation for every data type that is processed by the frame‐ work. A so-called type extractor analyzes the generic types and return types of all functions to obtain the respective TypeInformation objects. Hence, you might use

Flink for a while without ever needing to worry about TypeInformation for your data

types. However, sometimes the type extractor fails or you might want to define your

own types and tell Flink how to handle them efficiently. In such cases, you need to generate a TypeInformation for a specific data type.

Flink provides two utility classes for Java and Scala with static methods to generate a TypeInformation. For Java, the helper class is org.apache.flink.api.common.type info.Types, and it is used as shown in the following examples:

*// TypeInformation for primitive types*

TypeInformation<Integer> intType = Types.INT;

*// TypeInformation for Java Tuples* TypeInformation<Tuple2<Long, String>> tupleType = Types.TUPLE(Types.LONG, Types.STRING);

*// TypeInformation for POJOs*

TypeInformation<Person> personType = Types.POJO(Person.class);

TypeInformation’s helper class is org.apache.flink.api.scala.typeutils.Types

for the Scala API, and it is used as shown in the following:

*// TypeInformation for primitive types*

**val** stringType**: TypeInformation**[**String**] **= Types**.**STRING**

*// TypeInformation for Scala Tuples*

**val** tupleType**: TypeInformation**[(**Int**, **Long**)] **= Types**.**TUPLE**[(**Int**, **Long**)]

*// TypeInformation for case classes*

**val** caseClassType**: TypeInformation**[**Person**] **= Types**.**CASE\_CLASS**[**Person**]

#### Type Information in the Scala API

In the Scala API, Flink uses Scala compiler macros that generate TypeInformation objects for all data types at compile time. To access the createTypeInformation macro function, make sure to

always add the following import statement to your Scala applica‐

tion:

import org.apache.flink.streaming.api.scala.\_

## Explicitly Providing Type Information

In most cases, Flink can automatically infer types and generate the correct TypeInfor mation. Flink’s type extractor leverages reflection and analyzes function signatures and subclass information to derive the correct output type of a user-defined function.

Sometimes, though, the necessary information cannot be extracted (e.g., because of Java erasing generic type information). Moreover, in some cases Flink might not

choose the TypeInformation that generates the most efficient serializers and deserial‐ izers. Hence, you might need to explicitly provide TypeInformation objects to Flink for some of the data types used in your application.

There are two ways to provide TypeInformation. First, you can extend a function class to explicitly provide the TypeInformation of its return type by implementing the ResultTypeQueryable interface. The following example shows a MapFunction that provides its return type:

**class Tuple2ToPersonMapper extends MapFunction**[(**String**, **Int**), **Person**] **with ResultTypeQueryable**[**Person**] {

**override def** map(v**:** (**String**, **Int**))**: Person** = **Person**(v.\_1, v.\_2)

*// provide the TypeInformation for the output data type*

**override def** getProducedType**: TypeInformation**[**Person**] **= Types**.**CASE\_CLASS**[**Person**]

}

In the Java DataStream API, you can also use the returns() method to explicitly specify the return type of an operator when defining the dataflow as shown in the fol‐

lowing:

DataStream<Tuple2<String, Integer>> tuples = ... DataStream<Person> persons = tuples

.map(t -> **new** Person(t.f0, t.f1))

*// provide TypeInformation for the map lambda function's return type*

.returns(Types.POJO(Person.class));

# Defining Keys and Referencing Fields

Some of the transformations you saw in the previous section require a key specifica‐ tion or field reference on the input stream type. In Flink, keys are not predefined in the input types like in systems that work with key-value pairs. Instead, keys are defined as functions over the input data. Therefore, it is not necessary to define data types to hold keys and values, which avoids a lot of boilerplate code.

In the following, we discuss different methods to reference fields and define keys on data types.

## Field Positions

If the data type is a tuple, keys can be defined by simply using the field position of the corresponding tuple element. The following example keys the input stream by the second field of the input tuple:

**val** input**: DataStream**[(**Int**, **String**, **Long**)] **=** ...

**val** keyed **=** input.keyBy(1)

Composite keys consisting of more than one tuple field can also be defined. In this case, the positions are provided as a list, one after the other. We can key the input stream by the second and third fields as follows:

**val** keyed2 **=** input.keyBy(1, 2)

## Field Expressions

Another way to define keys and select fields is by using String-based field expres‐ sions. Field expressions work for tuples, POJOs, and case classes. They also support

the selection of nested fields.

In the introductory example of this chapter, we defined the following case class:

**case class SensorReading**( id**: String**,

timestamp**: Long**, temperature**: Double**)

To key the stream by sensor ID we can pass the field name id to the keyBy() func‐ tion:

**val** sensorStream**: DataStream**[**SensorReading**] **=** ...

**val** keyedSensors **=** sensorStream.keyBy("id")

POJO or case class fields are selected by their field name like in the above example. Tuple fields are referenced either by their field name (1-offset for Scala tuples, 0-offset for Java tuples) or by their 0-offset field index:

**val** input**: DataStream**[(**Int**, **String**, **Long**)] **=** ... **val** keyed1 **=** input.keyBy("2") *// key by 3rd field* **val** keyed2 **=** input.keyBy("\_1") *// key by 1st field*

DataStream<Tuple3<Integer, String, Long>> javaInput = ... javaInput.keyBy("f2") *// key Java tuple by 3rd field*

Nested fields in POJOs and tuples are selected by denoting the nesting level with a "." (period character). Consider the following case classes:

**case class Address**( address**: String**, zip**: String** country**: String**)

**case class Person**( name**: String**,

birthday**:** (**Int**, **Int**, **Int**), *// year, month, day*

address**: Address**)

If we want to reference a person’s ZIP code, we can use a field expression:

**val** persons**: DataStream**[**Person**] **=** ... persons.keyBy("address.zip") *// key by nested POJO field*

It is also possible to nest expressions on mixed types. The following expression accesses the field of a tuple nested in a POJO:

persons.keyBy("birthday.\_1") *// key by field of nested tuple*

A full data type can be selected using the wildcard field expression "\_" (underscore character):

persons.keyBy("birthday.\_") *// key by all fields of nested tuple*

## Key Selectors

A third option to specify keys are KeySelector functions. A KeySelector function extracts a key from an input event:

// T: the type of input elements

// KEY: the type of the key KeySelector[IN, KEY]

> getKey(IN): KEY

The introductory example actually uses a simple KeySelector function in the

keyBy() method:

**val** sensorData**: DataStream**[**SensorReading**] **=** ...

**val** byId**: KeyedStream**[**SensorReading**, **String**] **=** sensorData

.keyBy(r **=>** r.id)

A KeySelector function receives an input item and returns a key. The key does not necessarily have to be a field of the input event but can be derived using arbitrary computations. In the following, the KeySelector function returns the maximum of the tuple fields as the key:

**val** input **: DataStream**[(**Int**, **Int**)] **=** ...

**val** keyedStream **=** input.keyBy(value **=>** math.max(value.\_1, value.\_2))

Compared to field positions and field expressions, an advantage of KeySelector functions is that the resulting key is strongly typed due to the generic types of the KeySelector class.

# Implementing Functions

You’ve seen user-defined functions in action in the code examples of this chapter so far. In this section, we explain the different ways in which you can define and para‐ metrize functions in the DataStream API in more detail.

## Function Classes

Flink exposes all interfaces for user-defined functions, such as MapFunction, Filter Function, and ProcessFunction, as interfaces or abstract classes.

A function is implemented by implementing the interface or extending the abstract class. In the following example, we implement a FilterFunction that filters for strings that contain the word "flink":

**class FlinkFilter extends FilterFunction**[**String**] { **override def** filter(value**: String**)**: Boolean** = { value.contains("flink")

}

}

An instance of the function class can then be passed as an argument to the filter transformation:

**val** flinkTweets **=** tweets.filter(**new FlinkFilter**)

Functions can also be implemented as anonymous classes:

**val** flinkTweets **=** tweets.filter(

**new RichFilterFunction**[**String**] {

**override def** filter(value**: String**)**: Boolean** = { value.contains("flink")

}

})

Functions can receive parameters through their constructor. We can parametrize the above example and pass the String "flink" as a parameter to the KeywordFilter constructor as shown below:

**val** tweets**: DataStream**[**String**] **=** ???

**val** flinkTweets **=** tweets.filter(**new KeywordFilter**("flink"))

**class KeywordFilter**(keyWord**: String**) **extends FilterFunction**[**String**] { **override def** filter(value**: String**)**: Boolean** = { value.contains(keyWord)

}

}

When a program is submitted for execution, all function objects are serialized using Java serialization and shipped to all parallel tasks of their corresponding operators. Therefore, all configuration values are preserved after the object is deserialized.

#### Functions Must Be Java Serializable

Flink serializes all function objects with Java serialization to ship them to the worker processes. Everything contained in a user func‐

tion must be Serializable.

If your function requires a nonserializable object instance, you can either implement it as a rich function and initialize the nonserializ‐

able field in the open() method or override the Java serialization

and deserialization methods.

## Lambda Functions

Most DataStream API methods accept lambda functions. Lambda functions are avail‐ able for Scala and Java and offer a simple and concise way to implement application logic when no advanced operations such as accessing state and configuration are required. The following example show a lambda function that filters tweets contain‐ ing the word “flink”:

**val** tweets**: DataStream**[**String**] **=** ...

*// a filter lambda function that checks if tweets contains the word "flink"*

**val** flinkTweets **=** tweets.filter(**\_**.contains("flink"))

## Rich Functions

Oftentimes there is a need to initialize a function before it processes the first record or to retrieve information about the context in which it is executed. The DataStream API provides rich functions that expose more functionality than the regular functions discussed until now.

There are rich versions of all the DataStream API transformation functions, and you can use them in the same places you can use a regular function or lambda function. Rich functions can be parameterized just like regular function classes. The name of a

rich function starts with Rich followed by the transformation name—RichMapFunc tion, RichFlatMapFunction, and so on.

When using a rich function, you can implement two additional methods to the func‐ tion’s lifecycle:

* + The open() method is an initialization method for the rich function. It is called once per task before a transformation method like filter or map is called. open() is typically used for setup work that needs to be done only once. Note that the

Configuration parameter is only used by the DataSet API and not by the Data‐ Stream API. Hence, it should be ignored.

* + The close() method is a finalization method for the function and it is called once per task after the last call of the transformation method. Thus, it is com‐

monly used for cleanup and releasing resources.

In addition, the method getRuntimeContext() provides access to the function’s Run timeContext. The RuntimeContext can be used to retrieve information such as the function’s parallelism, its subtask index, and the name of the task that executes the

function. Further, it includes methods for accessing partitioned state. Stateful stream processing in Flink is discussed in detail in [“Implementing Stateful Functions” on](#_bookmark2)

[page 154](#_bookmark2). The following example code shows how to use the methods of a RichFlat MapFunction. [Example 5-3](#_bookmark10) shows the methods of a RichFLatMapFunction

*Example 5-3. The open() and close() methods of a RichFlatMapFunction.*

**class MyFlatMap extends RichFlatMapFunction**[**Int**, (**Int**, **Int**)] {

**var** subTaskIndex **=** 0

**override def** open(configuration**: Configuration**)**: Unit** = { subTaskIndex **=** getRuntimeContext.getIndexOfThisSubtask

*// do some initialization*

*// e.g., establish a connection to an external system*

}

**override def** flatMap(in**: Int**, out**: Collector**[(**Int**, **Int**)])**: Unit** = {

*// subtasks are 0-indexed*

**if**(in % 2 == subTaskIndex) { out.collect((subTaskIndex, in))

}

*// do some more processing*

}

**override def** close()**: Unit** = {

*// do some cleanup, e.g., close connections to external systems*

}

}

# Including External and Flink Dependencies

Adding external dependencies is a common requirement when implementing Flink applications. There are many popular libraries out there, such as Apache Commons or Google Guava, for various use cases. Moreover, most Flink applications depend on one or more of Flink’s connectors to ingest data from or emit data to external sys‐ tems, like Apache Kafka, filesystems, or Apache Cassandra. Some applications also leverage Flink’s domain-specific libraries, such as the Table API, SQL, or the CEP library. Consequently, most Flink applications do not only depend on Flink’s Data‐ Stream API dependency and the Java SDK but also on additional third-party and Flink-internal dependencies.

**Including External and Flink Dependencies | 107**

When an application is executed, all of its dependencies must be available to the application. By default, only the core API dependencies (DataStream and DataSet APIs) are loaded by a Flink cluster. All other dependencies an application requires must be explicitly provided.

The reason for this is to keep the number of default dependencies low.[4](#_bookmark17) Most connec‐ tors and libraries rely on one or more libraries, which typically have several addi‐ tional transitive dependencies. Often, these include frequently used libraries, such as Apache Commons or Google’s Guava. Many problems originate from incompatibili‐ ties among different versions of the same library that are pulled in from different connectors or directly from the user application.

There are two ways to ensure all dependencies are available to an application when it is executed:

1. Bundle all dependencies into the application JAR file. This yields a self- contained, yet typically quite large, application JAR file.
2. The JAR file of a dependency can be added to the *./lib* folder of a Flink setup. In this case, the dependencies are loaded into the classpath when Flink processes are started. A dependency that is added to the classpath like this is available to (and might interfere with) all applications that run on the Flink setup.

Building a so-called fat JAR file is the preferred way to handle application dependen‐ cies. The Flink Maven archetypes we introduced in [“Bootstrap a Flink Maven Project”](#_bookmark13) [on page 76](#_bookmark13) generate Maven projects that are configured to produce application-fat JARs that include all required dependencies. Dependencies included in the classpath of Flink processes by default are automatically excluded from the JAR file. The *pom.xml* file of a generated Maven project contains comments that explain how to add additional dependencies.

# Summary

In this chapter we introduced the basics of Flink’s DataStream API. We examined the structure of Flink programs and learned how to combine data and partitioning trans‐ formations to build streaming applications. We also looked into supported data types and different ways to specify keys and user-defined functions. If you take a step back and read the introductory example once more, you hopefully have a better idea about what is going on. In [Chapter 6](#_bookmark3), things are going to get even more interesting—we will learn how to enrich our programs with window operators and time semantics.

1. Flink also aims to keep its own external dependencies to a minimum and hides most of them (including tran‐ sitive dependencies) from user applications to prevent version conflicts.