» DataSet API (//ci.apache.org/projects/flink/flink-docs-master/apis/batch/index.html)
Transformations (//ci.apache.org/projects/flink/flink-docs-master/apis/batch/dataset_transformations.html)
Zipping Elements (//ci.apache.org/projects/flink/flink-docs-master/apis/batch/zip_elements_guide.html)
Fault Tolerance (//ci.apache.org/projects/flink/flink-docs-master/apis/batch/fault_tolerance.html)
Iterations (//ci.apache.org/projects/flink/flink-docs-master/apis/batch/iterations.html)
Connectors (//ci.apache.org/projects/flink/flink-docs-master/apis/batch/connectors.html)
Python API (//ci.apache.org/projects/flink/flink-docs-master/apis/batch/python.html)
Examples (//ci.apache.org/projects/flink/flink-docs-master/apis/batch/examples.html)
Libraries (//ci.apache.org/projects/flink/flink-docs-master/apis/batch/libs/index.html)
Gelly (//ci.apache.org/projects/flink/flink-docs-master/apis/batch/libs/gelly.html)
Machine Learning (//ci.apache.org/projects/flink/flink-docs-master/apis/batch/libs/ml/index.html)
Table API and SQL (//ci.apache.org/projects/flink/flink-docs-master/apis/batch/libs/table.html)
Hadoop Compatibility (//ci.apache.org/projects/flink/flink-docs-master/apis/batch/hadoop_compatibility.html)

Important: Maven artifacts which depend on Scala are now suffixed with the Scala major version, e.g. "2.10" or "2.11". Please consult the migration guide on the project Wiki (https://cwiki.apache.org/confluence/display/FLINK/Maven+artifact+names+suffixed+with+Scala+version).

Flink DataSet API Programming Guide

DataSet programs in Flink are regular programs that implement transformations on data sets (e.g., filtering, mapping, joining, grouping). The data sets are initially created from certain sources (e.g., by reading files, or from local collections). Results are returned via sinks, which may for example write the data to (distributed) files, or to standard output (for example the command line terminal). Flink programs run in a variety of contexts, standalone, or embedded in other programs. The execution can happen in a local JVM, or on clusters of many machines.

Please see basic concepts (//ci.apache.org/projects/flink/flink-docs-master/apis/common/index.html) for an introduction to the basic concepts of the Flink API.

In order to create your own Flink DataSet program, we encourage you to start with the anatomy of a Flink Program (//ci.apache.org/projects/flink/flink-docs-master/apis/common/index.html#anatomy-of-a-flink-program) and gradually add your own transformations. The remaining sections act as references for additional operations and advanced features.

Example Program
DataSet Transformations
Data Sources
Read Compressed Files
Data Sinks
Iteration Operators
Operating on data objects in functions
Object-Reuse Disabled (DEFAULT)
Object-Reuse Enabled
Debugging
Local Execution Environment
Collection Data Sources and Sinks
Semantic Annotations
Broadcast Variables
Passing Parameters to Functions

Example Program

The following program is a complete, working example of WordCount. You can copy & paste the code to run it locally. You only have to include the correct Flink's library into your project (see Section Linking with Flink) and specify the imports. Then you are ready to go!

Java **Scala**

```
public class WordCountExample {
    public static void main(String[] args) throws Exception {
        final ExecutionEnvironment env = ExecutionEnvironment.getExecutionEnvironment();

        DataSet<String> text = env.fromElements(
            "Who's there?",
            "I think I hear them. Stand, ho! Who's there?");

        DataSet<Tuple2<String, Integer>> wordCounts = text
            .flatMap(new LineSplitter())
            .groupBy(0)
            .sum(1);

        wordCounts.print();
    }

    public static class LineSplitter implements FlatMapFunction<String, Tuple2<String, In
teger>> {
        @Override
        public void flatMap(String line, Collector<Tuple2<String, Integer>> out) {
            for (String word : line.split(" ")) {
                out.collect(new Tuple2<String, Integer>(word, 1));
            }
        }
    }
}
```

Back to top

DataSet Transformations

Data transformations transform one or more DataSets into a new DataSet. Programs can combine multiple transformations into sophisticated assemblies.

This section gives a brief overview of the available transformations. The transformations documentation (dataset_transformations.html) has a full description of all transformations with examples.

Java **Scala**

Transformation	Description

Map	Takes one element and produces one element.
	
	`data.map(new MapFunction<String, Integer>() {`
	`  public Integer map(String value) { return Integer.parseInt(value);`
	`}`
	`});`

FlatMap	Takes one element and produces zero, one, or more elements. `data.flatMap(new FlatMapFunction<String, String>() { public void flatMap(String value, Collector<String> out) { for (String s : value.split(" ")) { out.collect(s); } } });`
MapPartition	Transforms a parallel partition in a single function call. The function get the partition as an `Iterable` stream and can produce an arbitrary number of result values. The number of elements in each partition depends on the degree-of-parallelism and previous operations. `data.mapPartition(new MapPartitionFunction<String, Long>() { public void mapPartition(Iterable<String> values, Collector<Long> out) { long c = 0; for (String s : values) { c++; } out.collect(c); } });`
Filter	Evaluates a boolean function for each element and retains those for which the function returns true. **IMPORTANT:** The system assumes that the function does not modify the elements on which the predicate is applied. Violating this assumption can lead to incorrect results. `data.filter(new FilterFunction<Integer>() { public boolean filter(Integer value) { return value > 1000; } });`
Reduce	Combines a group of elements into a single element by repeatedly combining two elements into one. Reduce may be applied on a full data set, or on a grouped data set. `data.reduce(new ReduceFunction<Integer> { public Integer reduce(Integer a, Integer b) { return a + b; } });`

ReduceGroup	Combines a group of elements into one or more elements. ReduceGroup may be applied on a full data set, or on a grouped data set. `data.reduceGroup(new GroupReduceFunction<Integer, Integer> { public void reduce(Iterable<Integer> values, Collector<Integer> out) { int prefixSum = 0; for (Integer i : values) { prefixSum += i; out.collect(prefixSum); } } });`

Aggregate	Aggregates a group of values into a single value. Aggregation functions can be thought of as built-in reduce functions. Aggregate may be applied on a full data set, or on a grouped data set.
	
	`Dataset<Tuple3<Integer, String, Double>> input = // [...] DataSet<Tuple3<Integer, String, Double>> output = input.aggregate(SUM, 0).and(MIN, 2);`
	
	You can also use short-hand syntax for minimum, maximum, and sum aggregations.
	
	`Dataset<Tuple3<Integer, String, Double>> input = // [...] DataSet<Tuple3<Integer, String, Double>> output = input.sum(0).andMin(2);`

Distinct	Returns the distinct elements of a data set. It removes the duplicate entries from the input DataSet, with respect to all fields of the elements, or a subset of fields. `data.distinct();`
Join	Joins two data sets by creating all pairs of elements that are equal on their keys. Optionally uses a JoinFunction to turn the pair of elements into a single element, or a FlatJoinFunction to turn the pair of elements into arbitrarily many (including none) elements. See the keys section to learn how to define join keys. `result = input1.join(input2) .where(0) // key of the first input (tuple field 0) .equalTo(1); // key of the second input (tuple field 1)` You can specify the way that the runtime executes the join via *Join Hints*. The hints describe whether the join happens through partitioning or broadcasting, and

whether it uses a sort-based or a hash-based algorithm. Please refer to the Transformations Guide (dataset_transformations.html#join-algorithm-hints) for a list of possible hints and an example. If no hint is specified, the system will try to make an estimate of the input sizes and pick a the best strategy according to those estimates.

```
// This executes a join by broadcasting the first data set
// using a hash table for the broadcasted data
result = input1.join(input2, JoinHint.BROADCAST_HASH_FIRST)
               .where(0).equalTo(1);
```

Note that the join transformation works only for equi-joins. Other join types need to be expressed using OuterJoin or CoGroup.

OuterJoin

Performs a left, right, or full outer join on two data sets. Outer joins are similar to regular (inner) joins and create all pairs of elements that are equal on their keys. In addition, records of the "outer" side (left, right, or both in case of full) are preserved if no matching key is found in the other side. Matching pairs of elements (or one element and a `null` value for the other input) are given to a JoinFunction to turn the pair of elements into a single element, or to a FlatJoinFunction to turn the pair of elements into arbitrarily many (including none) elements. See the keys section to learn how to define join keys.

```
input1.leftOuterJoin(input2) // rightOuterJoin or fullOuterJoin for
right or full outer joins
      .where(0)              // key of the first input (tuple field
0)
      .equalTo(1)            // key of the second input (tuple field
1)
      .with(new JoinFunction<String, String, String>() {
          public String join(String v1, String v2) {
             // NOTE:
             // - v2 might be null for leftOuterJoin
             // - v1 might be null for rightOuterJoin
             // - v1 OR v2 might be null for fullOuterJoin
          }
      });
```

CoGroup

	The two-dimensional variant of the reduce operation. Groups each input on one or more fields and then joins the groups. The transformation function is called per pair of groups. See the keys section to learn how to define coGroup keys.
	
	`data1.coGroup(data2)`
	`.where(0)`
	`.equalTo(1)`
	`.with(new CoGroupFunction<String, String, String>() {`
	`public void coGroup(Iterable<String> in1, Iterable<String> in2, Collector<String> out) {`
	`out.collect(...);`
	`}`
	`});`

Cross	Builds the Cartesian product (cross product) of two inputs, creating all pairs of elements. Optionally uses a CrossFunction to turn the pair of elements into a single element
	
	`DataSet<Integer> data1 = // [...]`
	`DataSet<String> data2 = // [...]`
	`DataSet<Tuple2<Integer, String>> result = data1.cross(data2);`
	
	Note: Cross is potentially a **very** compute-intensive operation which can challenge even large compute clusters! It is adviced to hint the system with the DataSet sizes by using *crossWithTiny()* and *crossWithHuge()*.

Union	Produces the union of two data sets. This operation happens implicitly if more than one data set is used for a specific function input.
	
	`DataSet<String> data1 = // [...]`
	`DataSet<String> data2 = // [...]`
	`DataSet<String> result = data1.union(data2);`

Rebalance	Evenly rebalances the parallel partitions of a data set to eliminate data skew. Only Map-like transformations may follow a rebalance transformation.
	
	`DataSet<String> in = // [...]`
	`DataSet<String> result = in.rebalance()`
	`.map(new Mapper());`

Hash-Partition	Hash-partitions a data set on a given key. Keys can be specified as position keys, expression keys, and key selector functions.

```
DataSet<Tuple2<String,Integer>> in = // [...]
DataSet<Integer> result = in.partitionByHash(0)
                            .mapPartition(new PartitionMapper());
```

Range-Partition

Range-partitions a data set on a given key. Keys can be specified as position keys, expression keys, and key selector functions.

```
DataSet<Tuple2<String,Integer>> in = // [...]
DataSet<Integer> result = in.partitionByRange(0)
                            .mapPartition(new PartitionMapper());
```

Custom Partitioning

Manually specify a partitioning over the data.
Note: This method works only on single field keys.

```
DataSet<Tuple2<String,Integer>> in = // [...]
DataSet<Integer> result = in.partitionCustom(Partitioner<K> partitio
ner, key)
```

Sort Partition

Locally sorts all partitions of a data set on a specified field in a specified order. Fields can be specified as tuple positions or field expressions. Sorting on multiple fields is done by chaining sortPartition() calls.

```
DataSet<Tuple2<String,Integer>> in = // [...]
DataSet<Integer> result = in.sortPartition(1, Order.ASCENDING)
                            .mapPartition(new PartitionMapper());
```

First-n

Returns the first n (arbitrary) elements of a data set. First-n can be applied on a regular data set, a grouped data set, or a grouped-sorted data set. Grouping keys can be specified as key-selector functions or field position keys.

```
DataSet<Tuple2<String,Integer>> in = // [...]
// regular data set
DataSet<Tuple2<String,Integer>> result1 = in.first(3);
// grouped data set
DataSet<Tuple2<String,Integer>> result2 = in.groupBy(0)
                                            .first(3);
// grouped-sorted data set
DataSet<Tuple2<String,Integer>> result3 = in.groupBy(0)
                                            .sortGroup(1, Order.ASCE
NDING)
                                            .first(3);
```

The following transformations are available on data sets of Tuples:

Transformation	Description

Project	Selects a subset of fields from the tuples
	
	`DataSet<Tuple3<Integer, Double, String>> in = // [...]`
	`DataSet<Tuple2<String, Integer>> out = in.project(2,0);`

The parallelism of a transformation can be defined by `setParallelism(int)` while `name(String)` assigns a custom name to a transformation which is helpful for debugging. The same is possible for Data Sources and Data Sinks.

`withParameters(Configuration)` passes Configuration objects, which can be accessed from the `open()` method inside the user function.

Back to top

Data Sources

Java **Scala**

Data sources create the initial data sets, such as from files or from Java collections. The general mechanism of creating data sets is abstracted behind an InputFormat (https://github.com/apache/flink/blob/master//flink-core/src/main/java/org/apache/flink/api/common/io/InputFormat.java). Flink comes with several built-in formats to create data sets from common file formats. Many of them have shortcut methods on the *ExecutionEnvironment*.

File-based:

- `readTextFile(path)` / `TextInputFormat` - Reads files line wise and returns them as Strings.
- `readTextFileWithValue(path)` / `TextValueInputFormat` - Reads files line wise and returns them as StringValues. StringValues are mutable strings.
- `readCsvFile(path)` / `CsvInputFormat` - Parses files of comma (or another char) delimited fields. Returns a DataSet of tuples or POJOs. Supports the basic java types and their Value counterparts as field types.
- `readFileOfPrimitives(path, Class)` / `PrimitiveInputFormat` - Parses files of new-line (or another char sequence) delimited primitive data types such as `String` or `Integer`.
- `readFileOfPrimitives(path, delimiter, Class)` / `PrimitiveInputFormat` - Parses files of new-line (or another char sequence) delimited primitive data types such as `String` or `Integer` using the given delimiter.

- `readHadoopFile(FileInputFormat, Key, Value, path)` / `FileInputFormat` - Creates a JobConf and reads file from the specified path with the specified FileInputFormat, Key class and Value class and returns them as Tuple2<Key, Value>.
- `readSequenceFile(Key, Value, path)` / `SequenceFileInputFormat` - Creates a JobConf and reads file from the specified path with type SequenceFileInputFormat, Key class and Value class and returns them as Tuple2<Key, Value>.

Collection-based:

- `fromCollection(Collection)` - Creates a data set from the Java Java.util.Collection. All elements in the collection must be of the same type.
- `fromCollection(Iterator, Class)` - Creates a data set from an iterator. The class specifies the data type of the elements returned by the iterator.
- `fromElements(T ...)` - Creates a data set from the given sequence of objects. All objects must be of the same type.
- `fromParallelCollection(SplittableIterator, Class)` - Creates a data set from an iterator, in parallel. The class specifies the data type of the elements returned by the iterator.
- `generateSequence(from, to)` - Generates the sequence of numbers in the given interval, in parallel.

Generic:

- `readFile(inputFormat, path)` / `FileInputFormat` - Accepts a file input format.
- `createInput(inputFormat)` / `InputFormat` - Accepts a generic input format.

Examples

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

// read text file from local files system
DataSet<String> localLines = env.readTextFile("file:///path/to/my/textfile");

// read text file from a HDFS running at nnHost:nnPort
DataSet<String> hdfsLines = env.readTextFile("hdfs://nnHost:nnPort/path/to/my/textfile");

// read a CSV file with three fields
DataSet<Tuple3<Integer, String, Double>> csvInput = env.readCsvFile("hdfs:///the/CSV/fil
e")
                               .types(Integer.class, String.class, Double.class);

// read a CSV file with five fields, taking only two of them
DataSet<Tuple2<String, Double>> csvInput = env.readCsvFile("hdfs:///the/CSV/file")
                               .includeFields("10010")  // take the first and the fourth
field
                               .types(String.class, Double.class);

// read a CSV file with three fields into a POJO (Person.class) with corresponding fields
DataSet<Person>> csvInput = env.readCsvFile("hdfs:///the/CSV/file")
                         .pojoType(Person.class, "name", "age", "zipcode");


// read a file from the specified path of type TextInputFormat
DataSet<Tuple2<LongWritable, Text>> tuples =
 env.readHadoopFile(new TextInputFormat(), LongWritable.class, Text.class, "hdfs://nnHos
t:nnPort/path/to/file");

// read a file from the specified path of type SequenceFileInputFormat
DataSet<Tuple2<IntWritable, Text>> tuples =
 env.readSequenceFile(IntWritable.class, Text.class, "hdfs://nnHost:nnPort/path/to/fil
e");

// creates a set from some given elements
DataSet<String> value = env.fromElements("Foo", "bar", "foobar", "fubar");

// generate a number sequence
DataSet<Long> numbers = env.generateSequence(1, 10000000);

// Read data from a relational database using the JDBC input format
DataSet<Tuple2<String, Integer> dbData =
    env.createInput(
      // create and configure input format
      JDBCInputFormat.buildJDBCInputFormat()
                     .setDrivername("org.apache.derby.jdbc.EmbeddedDriver")
                     .setDBUrl("jdbc:derby:memory:persons")
                     .setQuery("select name, age from persons")
                     .finish(),
      // specify type information for DataSet
      new TupleTypeInfo(Tuple2.class, STRING_TYPE_INFO, INT_TYPE_INFO)
    );
```

```
// Note: Flink's program compiler needs to infer the data types of the data items which a
re returned
// by an InputFormat. If this information cannot be automatically inferred, it is necessa
ry to
// manually provide the type information as shown in the examples above.
```

Configuring CSV Parsing

Flink offers a number of configuration options for CSV parsing:

- `types(Class ... types)` specifies the types of the fields to parse. **It is mandatory to configure the types of the parsed fields.** In case of the type class Boolean.class, “True” (case-insensitive), “False” (case-insensitive), “1” and “0” are treated as booleans.
- `lineDelimiter(String del)` specifies the delimiter of individual records. The default line delimiter is the new-line character `'\n'`.
- `fieldDelimiter(String del)` specifies the delimiter that separates fields of a record. The default field delimiter is the comma character `','`.
- `includeFields(boolean ... flag)`, `includeFields(String mask)`, or `includeFields(long bitMask)` defines which fields to read from the input file (and which to ignore). By default the first *n* fields (as defined by the number of types in the `types()` call) are parsed.
- `parseQuotedStrings(char quoteChar)` enables quoted string parsing. Strings are parsed as quoted strings if the first character of the string field is the quote character (leading or tailing whitespaces are *not* trimmed). Field delimiters within quoted strings are ignored. Quoted string parsing fails if the last character of a quoted string field is not the quote character or if the quote character appears at some point which is not the start or the end of the quoted string field (unless the quote character is escaped using ‘'). If quoted string parsing is enabled and the first character of the field is *not* the quoting string, the string is parsed as unquoted string. By default, quoted string parsing is disabled.
- `ignoreComments(String commentPrefix)` specifies a comment prefix. All lines that start with the specified comment prefix are not parsed and ignored. By default, no lines are ignored.
- `ignoreInvalidLines()` enables lenient parsing, i.e., lines that cannot be correctly parsed are ignored. By default, lenient parsing is disabled and invalid lines raise an exception.
- `ignoreFirstLine()` configures the InputFormat to ignore the first line of the input file. By default no line is ignored.

Recursive Traversal of the Input Path Directory

For file-based inputs, when the input path is a directory, nested files are not enumerated by default. Instead, only the files inside the base directory are read, while nested files are ignored. Recursive enumeration of nested files can be enabled through the `recursive.file.enumeration` configuration parameter, like in the following example.

```
// enable recursive enumeration of nested input files
ExecutionEnvironment env = ExecutionEnvironment.getExecutionEnvironment();

// create a configuration object
Configuration parameters = new Configuration();

// set the recursive enumeration parameter
parameters.setBoolean("recursive.file.enumeration", true);

// pass the configuration to the data source
DataSet<String> logs = env.readTextFile("file:///path/with.nested/files")
                          .withParameters(parameters);
```

Read Compressed Files

Flink currently supports transparent decompression of input files if these are marked with an appropriate file extension. In particular, this means that no further configuration of the input formats is necessary and any `FileInputFormat` support the compression, including custom input formats. Please notice that compressed files might not be read in parallel, thus impacting job scalability.

The following table lists the currently supported compression methods.

Compression method	File extensions	Parallelizable
DEFLATE	`.deflate`	no
GZip	`.gz`, `.gzip`	no

Back to top

Data Sinks

Java **Scala**

Data sinks consume DataSets and are used to store or return them. Data sink operations are described using an OutputFormat (https://github.com/apache/flink/blob/master//flink-core/src/main/java/org/apache/flink/api/common/io/OutputFormat.java). Flink comes with a variety of built-in output formats that are encapsulated behind operations on the DataSet:

- `writeAsText()` / `TextOuputFormat` - Writes elements line-wise as Strings. The Strings are obtained by calling the *toString()* method of each element.
- `writeAsFormattedText()` / `TextOutputFormat` - Write elements line-wise as Strings. The Strings are obtained by calling a user-defined *format()* method for each element.

- `writeAsCsv(...)` / `CsvOutputFormat` - Writes tuples as comma-separated value files. Row and field delimiters are configurable. The value for each field comes from the *toString()* method of the objects.
- `print()` / `printToErr()` / `print(String msg)` / `printToErr(String msg)` - Prints the *toString()* value of each element on the standard out / strandard error stream. Optionally, a prefix (msg) can be provided which is prepended to the output. This can help to distinguish between different calls to *print*. If the parallelism is greater than 1, the output will also be prepended with the identifier of the task which produced the output.
- `write()` / `FileOutputFormat` - Method and base class for custom file outputs. Supports custom object-to-bytes conversion.
- `output()`/ `OutputFormat` - Most generic output method, for data sinks that are not file based (such as storing the result in a database).

A DataSet can be input to multiple operations. Programs can write or print a data set and at the same time run additional transformations on them.

Examples

Standard data sink methods:

```
// text data
DataSet<String> textData = // [...]

// write DataSet to a file on the local file system
textData.writeAsText("file:///my/result/on/localFS");

// write DataSet to a file on a HDFS with a namenode running at nnHost:nnPort
textData.writeAsText("hdfs://nnHost:nnPort/my/result/on/localFS");

// write DataSet to a file and overwrite the file if it exists
textData.writeAsText("file:///my/result/on/localFS", WriteMode.OVERWRITE);

// tuples as lines with pipe as the separator "a|b|c"
DataSet<Tuple3<String, Integer, Double>> values = // [...]
values.writeAsCsv("file:///path/to/the/result/file", "\n", "|");

// this writes tuples in the text formatting "(a, b, c)", rather than as CSV lines
values.writeAsText("file:///path/to/the/result/file");

// this wites values as strings using a user-defined TextFormatter object
values.writeAsFormattedText("file:///path/to/the/result/file",
    new TextFormatter<Tuple2<Integer, Integer>>() {
        public String format (Tuple2<Integer, Integer> value) {
            return value.f1 + " - " + value.f0;
        }
    });
```

Using a custom output format:

```
DataSet<Tuple3<String, Integer, Double>> myResult = [...]

// write Tuple DataSet to a relational database
myResult.output(
    // build and configure OutputFormat
    JDBCOutputFormat.buildJDBCOutputFormat()
                    .setDrivername("org.apache.derby.jdbc.EmbeddedDriver")
                    .setDBUrl("jdbc:derby:memory:persons")
                    .setQuery("insert into persons (name, age, height) values (?,?,?)")
                    .finish()
    );
```

Locally Sorted Output

The output of a data sink can be locally sorted on specified fields in specified orders using tuple field positions or field expressions. This works for every output format.

The following examples show how to use this feature:

```
DataSet<Tuple3<Integer, String, Double>> tData = // [...]
DataSet<Tuple2<BookPojo, Double>> pData = // [...]
DataSet<String> sData = // [...]

// sort output on String field in ascending order
tData.sortPartition(1, Order.ASCENDING).print();

// sort output on Double field in descending and Integer field in ascending order
tData.sortPartition(2, Order.DESCENDING).sortPartition(0, Order.ASCENDING).print();

// sort output on the "author" field of nested BookPojo in descending order
pData.sortPartition("f0.author", Order.DESCENDING).writeAsText(...);

// sort output on the full tuple in ascending order
tData.sortPartition("*", Order.ASCENDING).writeAsCsv(...);

// sort atomic type (String) output in descending order
sData.sortPartition("*", Order.DESCENDING).writeAsText(...);
```

Globally sorted output is not supported yet.

Back to top

Iteration Operators

Iterations implement loops in Flink programs. The iteration operators encapsulate a part of the program and execute it repeatedly, feeding back the result of one iteration (the partial solution) into the next iteration. There are two types of iterations in Flink: **BulkIteration** and **DeltaIteration**.

This section provides quick examples on how to use both operators. Check out the Introduction to Iterations (iterations.html) page for a more detailed introduction.

Java Scala

Bulk Iterations

To create a BulkIteration call the `iterate(int)` method of the DataSet the iteration should start at. This will return an `IterativeDataSet`, which can be transformed with the regular operators. The single argument to the iterate call specifies the maximum number of iterations.

To specify the end of an iteration call the `closeWith(DataSet)` method on the `IterativeDataSet` to specify which transformation should be fed back to the next iteration. You can optionally specify a termination criterion with `closeWith(DataSet, DataSet)`, which evaluates the second DataSet and terminates the iteration, if this DataSet is empty. If no termination criterion is specified, the iteration terminates after the given maximum number iterations.

The following example iteratively estimates the number Pi. The goal is to count the number of random points, which fall into the unit circle. In each iteration, a random point is picked. If this point lies inside the unit circle, we increment the count. Pi is then estimated as the resulting count divided by the number of iterations multiplied by 4.

```
final ExecutionEnvironment env = ExecutionEnvironment.getExecutionEnvironment();

// Create initial IterativeDataSet
IterativeDataSet<Integer> initial = env.fromElements(0).iterate(10000);

DataSet<Integer> iteration = initial.map(new MapFunction<Integer, Integer>() {
    @Override
    public Integer map(Integer i) throws Exception {
        double x = Math.random();
        double y = Math.random();

        return i + ((x * x + y * y < 1) ? 1 : 0);
    }
});

// Iteratively transform the IterativeDataSet
DataSet<Integer> count = initial.closeWith(iteration);

count.map(new MapFunction<Integer, Double>() {
    @Override
    public Double map(Integer count) throws Exception {
        return count / (double) 10000 * 4;
    }
}).print();

env.execute("Iterative Pi Example");
```

You can also check out the K-Means example (https://github.com/apache/flink/blob/master//flink-examples/flink-examples-batch/src/main/java/org/apache/flink/examples/java/clustering/KMeans.java), which uses a BulkIteration to cluster a set of unlabeled points.

Delta Iterations

Delta iterations exploit the fact that certain algorithms do not change every data point of the solution in each iteration.

In addition to the partial solution that is fed back (called workset) in every iteration, delta iterations maintain state across iterations (called solution set), which can be updated through deltas. The result of the iterative computation is the state after the last iteration. Please refer to the Introduction to Iterations (iterations.html) for an overview of the basic principle of delta iterations.

Defining a DeltaIteration is similar to defining a BulkIteration. For delta iterations, two data sets form the input to each iteration (workset and solution set), and two data sets are produced as the result (new workset, solution set delta) in each iteration.

To create a DeltaIteration call the `iterateDelta(DataSet, int, int)` (or `iterateDelta(DataSet, int, int[])` respectively). This method is called on the initial solution set. The arguments are the initial delta set, the maximum number of iterations and the key positions. The returned `DeltaIteration` object gives you access to the DataSets representing the workset and solution set via the methods `iteration.getWorkset()` and `iteration.getSolutionSet()`.

Below is an example for the syntax of a delta iteration

```
// read the initial data sets
DataSet<Tuple2<Long, Double>> initialSolutionSet = // [...]

DataSet<Tuple2<Long, Double>> initialDeltaSet = // [...]

int maxIterations = 100;
int keyPosition = 0;

DeltaIteration<Tuple2<Long, Double>, Tuple2<Long, Double>> iteration = initialSolutionSet
    .iterateDelta(initialDeltaSet, maxIterations, keyPosition);

DataSet<Tuple2<Long, Double>> candidateUpdates = iteration.getWorkset()
    .groupBy(1)
    .reduceGroup(new ComputeCandidateChanges());

DataSet<Tuple2<Long, Double>> deltas = candidateUpdates
    .join(iteration.getSolutionSet())
    .where(0)
    .equalTo(0)
    .with(new CompareChangesToCurrent());

DataSet<Tuple2<Long, Double>> nextWorkset = deltas
    .filter(new FilterByThreshold());

iteration.closeWith(deltas, nextWorkset)
        .writeAsCsv(outputPath);
```

Back to top

Operating on data objects in functions

Flink's runtime exchanges data with user functions in form of Java objects. Functions receive input objects from the runtime as method parameters and return output objects as result. Because these objects are accessed by user functions and runtime code, it is very important to understand and follow the rules about how the user code may access, i.e., read and modify, these objects.

User functions receive objects from Flink's runtime either as regular method parameters (like a `MapFunction`) or through an `Iterable` parameter (like a `GroupReduceFunction`). We refer to objects that the runtime passes to a user function as *input objects*. User functions can emit objects to the Flink runtime either as a method return value (like a `MapFunction`) or through a `Collector` (like a `FlatMapFunction`). We refer to objects which have been emitted by the user function to the runtime as *output objects*.

Flink's DataSet API features two modes that differ in how Flink's runtime creates or reuses input objects. This behavior affects the guarantees and constraints for how user functions may interact with input and output objects. The following sections define these rules and give coding guidelines to write safe user function code.

Object-Reuse Disabled (DEFAULT)

By default, Flink operates in object-reuse disabled mode. This mode ensures that functions always receive new input objects within a function call. The object-reuse disabled mode gives better guarantees and is safer to use. However, it comes with a certain processing overhead and might cause higher Java garbage collection activity. The following table explains how user functions may access input and output objects in object-reuse disabled mode.

Operation	Guarantees and Restrictions
Reading Input Objects	Within a method call it is guaranteed that the value of an input object does not change. This includes objects served by an Iterable. For example it is safe to collect input objects served by an Iterable in a List or Map. Note that objects may be modified after the method call is left. It is **not safe** to remember objects across function calls.
Modifying Input Objects	You may modify input objects.
Emitting Input Objects	You may emit input objects. The value of an input object may have changed after it was emitted. It is **not safe** to read an input object after it was emitted.
Reading Output Objects	An object that was given to a Collector or returned as method result might have changed its value. It is **not safe** to read an output object.
Modifying Output	You may modify an object after it was emitted and emit it again.

Objects	

Coding guidelines for the object-reuse disabled (default) mode:

- Do not remember and read input objects across method calls.
- Do not read objects after you emitted them.

Object-Reuse Enabled

In object-reuse enabled mode, Flink's runtime minimizes the number of object instantiations. This can improve the performance and can reduce the Java garbage collection pressure. The object-reuse enabled mode is activated by calling `ExecutionConfig.enableObjectReuse()`. The following table explains how user functions may access input and output objects in object-reuse enabled mode.

Operation	Guarantees and Restrictions
Reading input objects received as regular method parameters	Input objects received as regular method arguments are not modified within a function call. Objects may be modified after method call is left. It is **not safe** to remember objects across function calls.
Reading input objects received from an Iterable parameter	Input objects received from an Iterable are only valid until the next() method is called. An Iterable or Iterator may serve the same object instance multiple times. It is **not safe** to remember input objects received from an Iterable, e.g., by putting them in a List or Map.
Modifying Input Objects	You **must not** modify input objects, except for input objects of MapFunction, FlatMapFunction, MapPartitionFunction, GroupReduceFunction, GroupCombineFunction, CoGroupFunction, and InputFormat.next(reuse).
Emitting Input Objects	You **must not** emit input objects, except for input objects of MapFunction, FlatMapFunction, MapPartitionFunction, GroupReduceFunction, GroupCombineFunction, CoGroupFunction, and InputFormat.next(reuse).
Reading Output Objects	An object that was given to a Collector or returned as method result might have changed its value. It is **not safe** to read an output object.
Modifying Output Objects	You may modify an output object and emit it again.

Coding guidelines for object-reuse enabled:

- Do not remember input objects received from an `Iterable`.
- Do not remember and read input objects across method calls.
- Do not modify or emit input objects, except for input objects of `MapFunction`, `FlatMapFunction`,

`MapPartitionFunction`, `GroupReduceFunction`, `GroupCombineFunction`, `CoGroupFunction`, and `InputFormat.next(reuse)`.

- To reduce object instantiations, you can always emit a dedicated output object which is repeatedly modified but never read.

Back to top

Debugging

Before running a data analysis program on a large data set in a distributed cluster, it is a good idea to make sure that the implemented algorithm works as desired. Hence, implementing data analysis programs is usually an incremental process of checking results, debugging, and improving.

Flink provides a few nice features to significantly ease the development process of data analysis programs by supporting local debugging from within an IDE, injection of test data, and collection of result data. This section give some hints how to ease the development of Flink programs.

Local Execution Environment

A `LocalEnvironment` starts a Flink system within the same JVM process it was created in. If you start the LocalEnvironement from an IDE, you can set breakpoints in your code and easily debug your program.

A LocalEnvironment is created and used as follows:

Java **Scala**

```
final ExecutionEnvironment env = ExecutionEnvironment.createLocalEnvironment();

DataSet<String> lines = env.readTextFile(pathToTextFile);
// build your program

env.execute();
```

Collection Data Sources and Sinks

Providing input for an analysis program and checking its output is cumbersome when done by creating input files and reading output files. Flink features special data sources and sinks which are backed by Java collections to ease testing. Once a program has been tested, the sources and sinks can be easily replaced by sources and sinks that read from / write to external data stores such as HDFS.

Collection data sources can be used as follows:

Java **Scala**

```
final ExecutionEnvironment env = ExecutionEnvironment.createLocalEnvironment();

// Create a DataSet from a list of elements
DataSet<Integer> myInts = env.fromElements(1, 2, 3, 4, 5);

// Create a DataSet from any Java collection
List<Tuple2<String, Integer>> data = ...
DataSet<Tuple2<String, Integer>> myTuples = env.fromCollection(data);

// Create a DataSet from an Iterator
Iterator<Long> longIt = ...
DataSet<Long> myLongs = env.fromCollection(longIt, Long.class);
```

A collection data sink is specified as follows:

```
DataSet<Tuple2<String, Integer>> myResult = ...

List<Tuple2<String, Integer>> outData = new ArrayList<Tuple2<String, Integer>>();
myResult.output(new LocalCollectionOutputFormat(outData));
```

Note: Currently, the collection data sink is restricted to local execution, as a debugging tool.

Note: Currently, the collection data source requires that data types and iterators implement `Serializable`. Furthermore, collection data sources can not be executed in parallel (parallelism = 1).

Back to top

Semantic Annotations

Semantic annotations can be used to give Flink hints about the behavior of a function. They tell the system which fields of a function's input the function reads and evaluates and which fields it unmodified forwards from its input to its output. Semantic annotations are a powerful means to speed up execution, because they allow the system to reason about reusing sort orders or partitions across multiple operations. Using semantic annotations may eventually save the program from unnecessary data shuffling or unnecessary sorts and significantly improve the performance of a program.

Note: The use of semantic annotations is optional. However, it is absolutely crucial to be conservative when providing semantic annotations! Incorrect semantic annotations will cause Flink to make incorrect assumptions about your program and might eventually lead to incorrect results. If the behavior of an operator is not clearly predictable, no annotation should be provided. Please read the documentation carefully.

The following semantic annotations are currently supported.

Forwarded Fields Annotation

Forwarded fields information declares input fields which are unmodified forwarded by a function to the same position or to another position in the output. This information is used by the optimizer to infer whether a data property such as sorting or partitioning is preserved by a function. For functions that

operate on groups of input elements such as `GroupReduce`, `GroupCombine`, `CoGroup`, and `MapPartition`, all fields that are defined as forwarded fields must always be jointly forwarded from the same input element. The forwarded fields of each element that is emitted by a group-wise function may originate from a different element of the function's input group.

Field forward information is specified using field expressions. Fields that are forwarded to the same position in the output can be specified by their position. The specified position must be valid for the input and output data type and have the same type. For example the String `"f2"` declares that the third field of a Java input tuple is always equal to the third field in the output tuple.

Fields which are unmodified forwarded to another position in the output are declared by specifying the source field in the input and the target field in the output as field expressions. The String `"f0->f2"` denotes that the first field of the Java input tuple is unchanged copied to the third field of the Java output tuple. The wildcard expression `*` can be used to refer to a whole input or output type, i.e., `"f0->*"` denotes that the output of a function is always equal to the first field of its Java input tuple.

Multiple forwarded fields can be declared in a single String by separating them with semicolons as `"f0; f2->f1; f3->f2"` or in separate Strings `"f0", "f2->f1", "f3->f2"`. When specifying forwarded fields it is not required that all forwarded fields are declared, but all declarations must be correct.

Forwarded field information can be declared by attaching Java annotations on function class definitions or by passing them as operator arguments after invoking a function on a DataSet as shown below.

Function Class Annotations

- `@ForwardedFields` for single input functions such as Map and Reduce.
- `@ForwardedFieldsFirst` for the first input of a functions with two inputs such as Join and CoGroup.
- `@ForwardedFieldsSecond` for the second input of a functions with two inputs such as Join and CoGroup.

Operator Arguments

- `data.map(myMapFnc).withForwardedFields()` for single input function such as Map and Reduce.
- `data1.join(data2).where().equalTo().with(myJoinFnc).withForwardFieldsFirst()` for the first input of a function with two inputs such as Join and CoGroup.
- `data1.join(data2).where().equalTo().with(myJoinFnc).withForwardFieldsSecond()` for the second input of a function with two inputs such as Join and CoGroup.

Please note that it is not possible to overwrite field forward information which was specified as a class annotation by operator arguments.

Example

The following example shows how to declare forwarded field information using a function class annotation:

Java **Scala**

```
@ForwardedFields("f0->f2")
public class MyMap implements
              MapFunction<Tuple2<Integer, Integer>, Tuple3<String, Integer, Integer>> {
  @Override
  public Tuple3<String, Integer, Integer> map(Tuple2<Integer, Integer> val) {
    return new Tuple3<String, Integer, Integer>("foo", val.f1 / 2, val.f0);
  }
}
```

Non-Forwarded Fields

Non-forwarded fields information declares all fields which are not preserved on the same position in a function's output. The values of all other fields are considered to be preserved at the same position in the output. Hence, non-forwarded fields information is inverse to forwarded fields information. Non-forwarded field information for group-wise operators such as `GroupReduce`, `GroupCombine`, `CoGroup`, and `MapPartition` must fulfill the same requirements as for forwarded field information.

IMPORTANT: The specification of non-forwarded fields information is optional. However if used, **ALL!** non-forwarded fields must be specified, because all other fields are considered to be forwarded in place. It is safe to declare a forwarded field as non-forwarded.

Non-forwarded fields are specified as a list of field expressions. The list can be either given as a single String with field expressions separated by semicolons or as multiple Strings. For example both `"f1; f3"` and `"f1", "f3"` declare that the second and fourth field of a Java tuple are not preserved in place and all other fields are preserved in place. Non-forwarded field information can only be specified for functions which have identical input and output types.

Non-forwarded field information is specified as function class annotations using the following annotations:

- `@NonForwardedFields` for single input functions such as Map and Reduce.
- `@NonForwardedFieldsFirst` for the first input of a function with two inputs such as Join and CoGroup.
- `@NonForwardedFieldsSecond` for the second input of a function with two inputs such as Join and CoGroup.

Example

The following example shows how to declare non-forwarded field information:

Java **Scala**

```
@NonForwardedFields("f1") // second field is not forwarded
public class MyMap implements
              MapFunction<Tuple2<Integer, Integer>, Tuple2<Integer, Integer>> {
  @Override
  public Tuple2<Integer, Integer> map(Tuple2<Integer, Integer> val) {
    return new Tuple2<Integer, Integer>(val.f0, val.f1 / 2);
  }
}
```

Read Fields

Read fields information declares all fields that are accessed and evaluated by a function, i.e., all fields that are used by the function to compute its result. For example, fields which are evaluated in conditional statements or used for computations must be marked as read when specifying read fields information. Fields which are only unmodified forwarded to the output without evaluating their values or fields which are not accessed at all are not considered to be read.

IMPORTANT: The specification of read fields information is optional. However if used, **ALL!** read fields must be specified. It is safe to declare a non-read field as read.

Read fields are specified as a list of field expressions. The list can be either given as a single String with field expressions separated by semicolons or as multiple Strings. For example both `"f1; f3"` and `"f1", "f3"` declare that the second and fourth field of a Java tuple are read and evaluated by the function.

Read field information is specified as function class annotations using the following annotations:

- `@ReadFields` for single input functions such as Map and Reduce.
- `@ReadFieldsFirst` for the first input of a function with two inputs such as Join and CoGroup.
- `@ReadFieldsSecond` for the second input of a function with two inputs such as Join and CoGroup.

Example

The following example shows how to declare read field information:

Java **Scala**

```
@ReadFields("f0; f3") // f0 and f3 are read and evaluated by the function.
public class MyMap implements
              MapFunction<Tuple4<Integer, Integer, Integer, Integer>,
                          Tuple2<Integer, Integer>> {
  @Override
  public Tuple2<Integer, Integer> map(Tuple4<Integer, Integer, Integer, Integer> val) {
    if(val.f0 == 42) {
      return new Tuple2<Integer, Integer>(val.f0, val.f1);
    } else {
      return new Tuple2<Integer, Integer>(val.f3+10, val.f1);
    }
  }
}
```

Back to top

Broadcast Variables

Broadcast variables allow you to make a data set available to all parallel instances of an operation, in addition to the regular input of the operation. This is useful for auxiliary data sets, or data-dependent parameterization. The data set will then be accessible at the operator as a Collection.

- **Broadcast**: broadcast sets are registered by name via `withBroadcastSet(DataSet, String)`, and
- **Access**: accessible via `getRuntimeContext().getBroadcastVariable(String)` at the target operator.

Java **Scala**

```
// 1. The DataSet to be broadcasted
DataSet<Integer> toBroadcast = env.fromElements(1, 2, 3);

DataSet<String> data = env.fromElements("a", "b");

data.map(new RichMapFunction<String, String>() {
    @Override
    public void open(Configuration parameters) throws Exception {
      // 3. Access the broadcasted DataSet as a Collection
      Collection<Integer> broadcastSet = getRuntimeContext().getBroadcastVariable("broadc
astSetName");
    }


    @Override
    public String map(String value) throws Exception {
        ...
    }
}).withBroadcastSet(toBroadcast, "broadcastSetName"); // 2. Broadcast the DataSet
```

Make sure that the names (`broadcastSetName` in the previous example) match when registering and accessing broadcasted data sets. For a complete example program, have a look at K-Means Algorithm (https://github.com/apache/flink/blob/master//flink-examples/flink-examples-batch/src/main/java/org/apache/flink/examples/java/clustering/KMeans.java#L96).

Note: As the content of broadcast variables is kept in-memory on each node, it should not become too large. For simpler things like scalar values you can simply make parameters part of the closure of a function, or use the `withParameters(...)` method to pass in a configuration.

Back to top

Passing Parameters to Functions

Parameters can be passed to functions using either the constructor or the `withParameters(Configuration)` method. The parameters are serialized as part of the function object and shipped to all parallel task instances.

Check also the best practices guide on how to pass command line arguments to functions (//ci.apache.org/projects/flink/flink-docs-master/apis/best_practices.html#parsing-command-line-arguments-and-passing-them-around-in-your-flink-application).

Via Constructor

Java **Scala**

```
DataSet<Integer> toFilter = env.fromElements(1, 2, 3);

toFilter.filter(new MyFilter(2));

private static class MyFilter implements FilterFunction<Integer> {

  private final int limit;

  public MyFilter(int limit) {
    this.limit = limit;
  }

  @Override
  public boolean filter(Integer value) throws Exception {
    return value > limit;
  }
}
```

Via `withParameters(Configuration)`

This method takes a Configuration object as an argument, which will be passed to the rich function's `open()` method. The Configuration object is a Map from String keys to different value types.

Java **Scala**

```
DataSet<Integer> toFilter = env.fromElements(1, 2, 3);

Configuration config = new Configuration();
config.setInteger("limit", 2);

toFilter.filter(new RichFilterFunction<Integer>() {
    private int limit;

    @Override
    public void open(Configuration parameters) throws Exception {
      limit = parameters.getInteger("limit", 0);
    }

    @Override
    public boolean filter(Integer value) throws Exception {
      return value > limit;
    }
}).withParameters(config);
```

Globally via the `ExecutionConfig`

Flink also allows to pass custom configuration values to the `ExecutionConfig` interface of the environment. Since the execution config is accessible in all (rich) user functions, the custom configuration will be available globally in all functions.

Setting a custom global configuration

Java **Scala**

```
Configuration conf = new Configuration();
conf.setString("mykey","myvalue");
final ExecutionEnvironment env = ExecutionEnvironment.getExecutionEnvironment();
env.getConfig().setGlobalJobParameters(conf);
```

Please note that you can also pass a custom class extending the `ExecutionConfig.GlobalJobParameters` class as the global job parameters to the execution config. The interface allows to implement the `Map<String, String> toMap()` method which will in turn show the values from the configuration in the web frontend.

Accessing values from the global configuration

Objects in the global job parameters are accessible in many places in the system. All user functions implementing a `Rich*Function` interface have access through the runtime context.

```
public static final class Tokenizer extends RichFlatMapFunction<String, Tuple2<String, In
teger>> {

    private String mykey;
    @Override
    public void open(Configuration parameters) throws Exception {
      super.open(parameters);
      ExecutionConfig.GlobalJobParameters globalParams = getRuntimeContext().getExecution
Config().getGlobalJobParameters();
      Configuration globConf = (Configuration) globalParams;
      mykey = globConf.getString("mykey", null);
    }
    // ... more here ...
```

Back to top