# Apache Beam：一个开源的统一的分布式数据处理编程库

# <http://shiyanjun.cn/archives/1567.html>

# Apache Beam Programming Guide

The **Beam Programming Guide** is intended for Beam users who want to use the Beam SDKs to create data processing pipelines. It provides guidance for using the Beam SDK classes to build and test your pipeline. It is not intended as an exhaustive reference, but as a language-agnostic, high-level guide to programmatically building your Beam pipeline. As the programming guide is filled out, the text will include code samples in multiple languages to help illustrate how to implement Beam concepts in your pipelines.

**Adapt for:**

* Java SDK
* Python SDK

## Contents

* [Overview](https://beam.apache.org/documentation/programming-guide/#overview)
* [Creating the Pipeline](https://beam.apache.org/documentation/programming-guide/#pipeline)
* [Working with PCollections](https://beam.apache.org/documentation/programming-guide/#pcollection)
  + [Creating a PCollection](https://beam.apache.org/documentation/programming-guide/#pccreate)
  + [PCollection Characteristics](https://beam.apache.org/documentation/programming-guide/#pccharacteristics)
    - [Element Type](https://beam.apache.org/documentation/programming-guide/#pcelementtype)
    - [Immutability](https://beam.apache.org/documentation/programming-guide/#pcimmutability)
    - [Random Access](https://beam.apache.org/documentation/programming-guide/#pcrandomaccess)
    - [Size and Boundedness](https://beam.apache.org/documentation/programming-guide/#pcsizebound)
    - [Element Timestamps](https://beam.apache.org/documentation/programming-guide/#pctimestamps)
* [Applying Transforms](https://beam.apache.org/documentation/programming-guide/#transforms)
  + [Using ParDo](https://beam.apache.org/documentation/programming-guide/#transforms-pardo)
  + [Using GroupByKey](https://beam.apache.org/documentation/programming-guide/#transforms-gbk)
  + [Using Combine](https://beam.apache.org/documentation/programming-guide/#transforms-combine)
  + [Using Flatten and Partition](https://beam.apache.org/documentation/programming-guide/#transforms-flatten-partition)
  + [General Requirements for Writing User Code for Beam Transforms](https://beam.apache.org/documentation/programming-guide/#transforms-usercodereqs)
  + [Side Inputs and Side Outputs](https://beam.apache.org/documentation/programming-guide/#transforms-sideio)
* [Composite Transforms](https://beam.apache.org/documentation/programming-guide/#transforms-composite)
* [Pipeline I/O](https://beam.apache.org/documentation/programming-guide/#io)
* [Running the Pipeline](https://beam.apache.org/documentation/programming-guide/#running)
* [Data Encoding and Type Safety](https://beam.apache.org/documentation/programming-guide/#coders)
* [Working with Windowing](https://beam.apache.org/documentation/programming-guide/#windowing)
* [Working with Triggers](https://beam.apache.org/documentation/programming-guide/#triggers)

## Overview

To use Beam, you need to first create a driver program using the classes in one of the Beam SDKs. Your driver program defines your pipeline, including all of the inputs, transforms, and outputs; it also sets execution options for your pipeline (typically passed in using command-line options). These include the Pipeline Runner, which, in turn, determines what back-end your pipeline will run on.

The Beam SDKs provide a number of abstractions that simplify the mechanics of large-scale distributed data processing. The same Beam abstractions work with both batch and streaming data sources. When you create your Beam pipeline, you can think about your data processing task in terms of these abstractions. They include:

* Pipeline: A Pipeline encapsulates your entire data processing task, from start to finish. This includes reading input data, transforming that data, and writing output data. All Beam driver programs must create a Pipeline. When you create the Pipeline, you must also specify the execution options that tell the Pipeline where and how to run.
* PCollection: A PCollection represents a distributed data set that your Beam pipeline operates on. The data set can be bounded, meaning it comes from a fixed source like a file, or unbounded, meaning it comes from a continuously updating source via a subscription or other mechanism. Your pipeline typically creates an initial PCollection by reading data from an external data source, but you can also create a PCollection from in-memory data within your driver program. From there, PCollections are the inputs and outputs for each step in your pipeline.
* Transform: A Transform represents a data processing operation, or a step, in your pipeline. Every Transform takes one or more PCollection objects as input, perfroms a processing function that you provide on the elements of that PCollection, and produces one or more output PCollection objects.
* I/O Source and Sink: Beam provides Source and Sink APIs to represent reading and writing data, respectively. Source encapsulates the code necessary to read data into your Beam pipeline from some external source, such as cloud file storage or a subscription to a streaming data source. Sink likewise encapsulates the code necessary to write the elements of a PCollection to an external data sink.

A typical Beam driver program works as follows:

* Create a Pipeline object and set the pipeline execution options, including the Pipeline Runner.
* Create an initial PCollection for pipeline data, either using the Source API to read data from an external source, or using a Create transform to build a PCollection from in-memory data.
* Apply **Transforms** to each PCollection. Transforms can change, filter, group, analyze, or otherwise process the elements in a PCollection. A transform creates a new output PCollection without consuming the input collection. A typical pipeline applies subsequent transforms to the each new output PCollection in turn until processing is complete.
* Output the final, transformed PCollection(s), typically using the Sink API to write data to an external source.
* **Run** the pipeline using the designated Pipeline Runner.

When you run your Beam driver program, the Pipeline Runner that you designate constructs a **workflow graph** of your pipeline based on the PCollection objects you’ve created and transforms that you’ve applied. That graph is then executed using the appropriate distributed processing back-end, becoming an asynchronous “job” (or equivalent) on that back-end.

## Creating the pipeline

The Pipeline abstraction encapsulates all the data and steps in your data processing task. Your Beam driver program typically starts by constructing a [Pipeline](https://beam.apache.org/documentation/sdks/javadoc/0.6.0/index.html?org/apache/beam/sdk/Pipeline.html) object, and then using that object as the basis for creating the pipeline’s data sets as PCollections and its operations as Transforms.

To use Beam, your driver program must first create an instance of the Beam SDK class Pipeline (typically in the main() function). When you create your Pipeline, you’ll also need to set some **configuration options**. You can set your pipeline’s configuration options programatically, but it’s often easier to set the options ahead of time (or read them from the command line) and pass them to the Pipeline object when you create the object.

The pipeline configuration options determine, among other things, the PipelineRunner that determines where the pipeline gets executed: locally, or using a distributed back-end of your choice. Depending on where your pipeline gets executed and what your specifed Runner requires, the options can also help you specify other aspects of execution.

To set your pipeline’s configuration options and create the pipeline, create an object of type [PipelineOptions](https://beam.apache.org/documentation/sdks/javadoc/0.6.0/index.html?org/apache/beam/sdk/options/PipelineOptions.html) and pass it to Pipeline.Create(). The most common way to do this is by parsing arguments from the command-line:

* Java
* Python

public static void main(String[] args) {

// Will parse the arguments passed into the application and construct a PipelineOptions

// Note that --help will print registered options, and --help=PipelineOptionsClassName

// will print out usage for the specific class.

PipelineOptions options =

PipelineOptionsFactory.fromArgs(args).create();

Pipeline p = Pipeline.create(options);

The Beam SDKs contain various subclasses of PipelineOptions that correspond to different Runners. For example, DirectPipelineOptions contains options for the Direct (local) pipeline runner, while DataflowPipelineOptions contains options for using the runner for Google Cloud Dataflow. You can also define your own custom PipelineOptions by creating an interface that extends the Beam SDKs’ PipelineOptions class.

## Working with PCollections

The [PCollection](https://beam.apache.org/documentation/sdks/javadoc/0.6.0/index.html?org/apache/beam/sdk/values/PCollection.html) abstraction represents a potentially distributed, multi-element data set. You can think of a PCollection as “pipeline” data; Beam transforms use PCollection objects as inputs and outputs. As such, if you want to work with data in your pipeline, it must be in the form of a PCollection.

After you’ve created your Pipeline, you’ll need to begin by creating at least one PCollection in some form. The PCollection you create serves as the input for the first operation in your pipeline.

### Creating a PCollection

You create a PCollection by either reading data from an external source using Beam’s [Source API](https://beam.apache.org/documentation/programming-guide/#io), or you can create a PCollection of data stored in an in-memory collection class in your driver program. The former is typically how a production pipeline would ingest data; Beam’s Source APIs contain adapters to help you read from external sources like large cloud-based files, databases, or subscription services. The latter is primarily useful for testing and debugging purposes.

#### Reading from an external source

To read from an external source, you use one of the [Beam-provided I/O adapters](https://beam.apache.org/documentation/programming-guide/#io). The adapters vary in their exact usage, but all of them from some external data source and return a PCollection whose elements represent the data records in that source.

Each data source adapter has a Read transform; to read, you must apply that transform to the Pipeline object itself. TextIO.Read, for example, reads from an external text file and returns a PCollection whose elements are of type String, each String represents one line from the text file. Here’s how you would apply TextIO.Read to your Pipeline to create a PCollection:

* Java
* Python

public static void main(String[] args) {

// Create the pipeline.

PipelineOptions options =

PipelineOptionsFactory.fromArgs(args).create();

Pipeline p = Pipeline.create(options);

PCollection<String> lines = p.apply(

"ReadMyFile", TextIO.Read.from("protocol://path/to/some/inputData.txt"));

}

See the [section on I/O](https://beam.apache.org/documentation/programming-guide/#io) to learn more about how to read from the various data sources supported by the Beam SDK.

#### Creating a PCollection from in-memory data

To create a PCollection from an in-memory Java Collection, you use the Beam-provided Create transform. Much like a data adapter’s Read, you apply Create directly to your Pipeline object itself.

As parameters, Create accepts the Java Collection and a Coder object. The Coder specifies how the elements in the Collection should be [encoded](https://beam.apache.org/documentation/programming-guide/#pcelementtype).

The following example code shows how to create a PCollection from an in-memory List:

* Java
* Python

public static void main(String[] args) {

// Create a Java Collection, in this case a List of Strings.

static final List<String> LINES = Arrays.asList(

"To be, or not to be: that is the question: ",

"Whether 'tis nobler in the mind to suffer ",

"The slings and arrows of outrageous fortune, ",

"Or to take arms against a sea of troubles, ");

// Create the pipeline.

PipelineOptions options =

PipelineOptionsFactory.fromArgs(args).create();

Pipeline p = Pipeline.create(options);

// Apply Create, passing the list and the coder, to create the PCollection.

p.apply(Create.of(LINES)).setCoder(StringUtf8Coder.of())

}

### PCollection characteristics

A PCollection is owned by the specific Pipeline object for which it is created; multiple pipelines cannot share a PCollection. In some respects, a PCollection functions like a collection class. However, a PCollection can differ in a few key ways:

#### Element type

The elements of a PCollection may be of any type, but must all be of the same type. However, to support distributed processing, Beam needs to be able to encode each individual element as a byte string (so elements can be passed around to distributed workers). The Beam SDKs provide a data encoding mechanism that includes built-in encoding for commonly-used types as well as support for specifying custom encodings as needed.

#### Immutability

A PCollection is immutable. Once created, you cannot add, remove, or change individual elements. A Beam Transform might process each element of a PCollection and generate new pipeline data (as a new PCollection), but it does not consume or modify the original input collection.

#### Random access

A PCollection does not support random access to individual elements. Instead, Beam Transforms consider every element in a PCollection individually.

#### Size and boundedness

A PCollection is a large, immutable “bag” of elements. There is no upper limit on how many elements a PCollection can contain; any given PCollection might fit in memory on a single machine, or it might represent a very large distributed data set backed by a persistent data store.

A PCollection can be either **bounded** or **unbounded** in size. A **bounded** PCollection represents a data set of a known, fixed size, while an **unbounded** PCollection represents a data set of unlimited size. Whether a PCollection is bounded or unbounded depends on the source of the data set that it represents. Reading from a batch data source, such as a file or a database, creates a bounded PCollection. Reading from a streaming or continously-updating data source, such as Pub/Sub or Kafka, creates an unbounded PCollection (unless you explicitly tell it not to).

The bounded (or unbounded) nature of your PCollection affects how Beam processes your data. A bounded PCollection can be processed using a batch job, which might read the entire data set once, and perform processing in a job of finite length. An unbounded PCollection must be processed using a streaming job that runs continuously, as the entire collection can never be available for processing at any one time.

When performing an operation that groups elements in an unbounded PCollection, Beam requires a concept called **Windowing** to divide a continuously updating data set into logical windows of finite size. Beam processes each window as a bundle, and processing continues as the data set is generated. These logical windows are determined by some characteristic associated with a data element, such as a **timestamp**.

#### Element timestamps

Each element in a PCollection has an associated intrinsic **timestamp**. The timestamp for each element is initially assigned by the [Source](https://beam.apache.org/documentation/programming-guide/#io) that creates the PCollection. Sources that create an unbounded PCollection often assign each new element a timestamp that corresponds to when the element was read or added.

**Note**: Sources that create a bounded PCollection for a fixed data set also automatically assign timestamps, but the most common behavior is to assign every element the same timestamp (Long.MIN\_VALUE).

Timestamps are useful for a PCollection that contains elements with an inherent notion of time. If your pipeline is reading a stream of events, like Tweets or other social media messages, each element might use the time the event was posted as the element timestamp.

You can manually assign timestamps to the elements of a PCollection if the source doesn’t do it for you. You’ll want to do this if the elements have an inherent timestamp, but the timestamp is somewhere in the structure of the element itself (such as a “time” field in a server log entry). Beam has [Transforms](https://beam.apache.org/documentation/programming-guide/#transforms) that take a PCollection as input and output an identical PCollection with timestamps attached; see [Assigning Timestamps](https://beam.apache.org/documentation/programming-guide/#windowing) for more information on how to do so.

## Applying transforms

In the Beam SDKs, **transforms** are the operations in your pipeline. A transform takes a PCollection (or more than one PCollection) as input, performs an operation that you specify on each element in that collection, and produces a new output PCollection. To invoke a transform, you must **apply** it to the input PCollection.

In Beam SDK each transform has a generic apply method . Invoking multiple Beam transforms is similar to method chaining, but with one slight difference: You apply the transform to the input PCollection, passing the transform itself as an argument, and the operation returns the output PCollection. This takes the general form:

* Java
* Python

[Output PCollection] = [Input PCollection].apply([Transform])

Because Beam uses a generic apply method for PCollection, you can both chain transforms sequentially and also apply transforms that contain other transforms nested within (called **composite transforms** in the Beam SDKs).

How you apply your pipeline’s transforms determines the structure of your pipeline. The best way to think of your pipeline is as a directed acyclic graph, where the nodes are PCollections and the edges are transforms. For example, you can chain transforms to create a sequential pipeline, like this one:

* Java
* Python

[Final Output PCollection] = [Initial Input PCollection].apply([First Transform])

.apply([Second Transform])

.apply([Third Transform])

The resulting workflow graph of the above pipeline looks like this:

[Sequential Graph Graphic]

However, note that a transform does not consume or otherwise alter the input collection–remember that a PCollection is immutable by definition. This means that you can apply multiple transforms to the same input PCollection to create a branching pipeline, like so:

* Java
* Python

[Output PCollection 1] = [Input PCollection].apply([Transform 1])

[Output PCollection 2] = [Input PCollection].apply([Transform 2])

The resulting workflow graph from the branching pipeline above looks like this:

[Branching Graph Graphic]

You can also build your own composite transforms that nest multiple sub-steps inside a single, larger transform. Composite transforms are particularly useful for building a reusable sequence of simple steps that get used in a lot of different places.

### Transforms in the Beam SDK

The transforms in the Beam SDKs provide a generic **processing framework**, where you provide processing logic in the form of a function object (colloquially referred to as “user code”). The user code gets applied to the elements of the input PCollection. Instances of your user code might then be executed in parallel by many different workers across a cluster, depending on the pipeline runner and back-end that you choose to execute your Beam pipeline. The user code running on each worker generates the output elements that are ultimately added to the final output PCollection that the transform produces.

### Core Beam transforms

Beam provides the following transforms, each of which represents a different processing paradigm:

* ParDo
* GroupByKey
* Combine
* Flatten and Partition

#### ParDo

ParDo is a Beam transform for generic parallel processing. The ParDo processing paradigm is similar to the “Map” phase of a Map/Shuffle/Reduce-style algorithm: a ParDo transform considers each element in the input PCollection, performs some processing function (your user code) on that element, and emits zero, one, or multiple elements to an output PCollection.

ParDo is useful for a variety of common data processing operations, including:

* **Filtering a data set.** You can use ParDo to consider each element in a PCollection and either output that element to a new collection, or discard it.
* **Formatting or type-converting each element in a data set.** If your input PCollection contains elements that are of a different type or format than you want, you can use ParDo to perform a conversion on each element and output the result to a new PCollection.
* **Extracting parts of each element in a data set.** If you have a PCollection of records with multiple fields, for example, you can use a ParDo to parse out just the fields you want to consider into a new PCollection.
* **Performing computations on each element in a data set.** You can use ParDo to perform simple or complex computations on every element, or certain elements, of a PCollection and output the results as a new PCollection.

In such roles, ParDo is a common intermediate step in a pipeline. You might use it to extract certain fields from a set of raw input records, or convert raw input into a different format; you might also use ParDo to convert processed data into a format suitable for output, like database table rows or printable strings.

When you apply a ParDo transform, you’ll need to provide user code in the form of a DoFn object. DoFn is a Beam SDK class that defines a distributed processing function.

When you create a subclass of DoFn, note that your subclass should adhere to the [General Requirements for Writing User Code for Beam Transforms](https://beam.apache.org/documentation/programming-guide/#transforms-usercodereqs).

##### Applying ParDo

Like all Beam transforms, you apply ParDo by calling the apply method on the input PCollection and passing ParDo as an argument, as shown in the following example code:

* Java
* Python

// The input PCollection of Strings.

PCollection<String> words = ...;

// The DoFn to perform on each element in the input PCollection.

static class ComputeWordLengthFn extends DoFn<String, Integer> { ... }

// Apply a ParDo to the PCollection "words" to compute lengths for each word.

PCollection<Integer> wordLengths = words.apply(

ParDo

.of(new ComputeWordLengthFn())); // The DoFn to perform on each element, which

// we define above.

In the example, our input PCollection contains String values. We apply a ParDo transform that specifies a function (ComputeWordLengthFn) to compute the length of each string, and outputs the result to a new PCollection of Integer values that stores the length of each word.

##### Creating a DoFn

The DoFn object that you pass to ParDo contains the processing logic that gets applied to the elements in the input collection. When you use Beam, often the most important pieces of code you’ll write are these DoFns–they’re what define your pipeline’s exact data processing tasks.

**Note:** When you create your DoFn, be mindful of the [General Requirements for Writing User Code for Beam Transforms](https://beam.apache.org/documentation/programming-guide/#transforms-usercodereqs) and ensure that your code follows them.

A DoFn processes one element at a time from the input PCollection. When you create a subclass of DoFn, you’ll need to provide type parameters that match the types of the input and output elements. If your DoFn processes incoming String elements and produces Integer elements for the output collection (like our previous example, ComputeWordLengthFn), your class declaration would look like this:

static class ComputeWordLengthFn extends DoFn<String, Integer> { ... }

Inside your DoFn subclass, you’ll write a method annotated with @ProcessElement where you provide the actual processing logic. You don’t need to manually extract the elements from the input collection; the Beam SDKs handle that for you. Your @ProcessElement method should accept an object of type ProcessContext. The ProcessContext object gives you access to an input element and a method for emitting an output element:

static class ComputeWordLengthFn extends DoFn<String, Integer> {

@ProcessElement

public void processElement(ProcessContext c) {

// Get the input element from ProcessContext.

String word = c.element();

// Use ProcessContext.output to emit the output element.

c.output(word.length());

}

}

**Note:** If the elements in your input PCollection are key/value pairs, you can access the key or value by using ProcessContext.element().getKey() or ProcessContext.element().getValue(), respectively.

A given DoFn instance generally gets invoked one or more times to process some arbitrary bundle of elements. However, Beam doesn’t guarantee an exact number of invocations; it may be invoked multiple times on a given worker node to account for failures and retries. As such, you can cache information across multiple calls to your processing method, but if you do so, make sure the implementation **does not depend on the number of invocations**.

In your processing method, you’ll also need to meet some immutability requirements to ensure that Beam and the processing back-end can safely serialize and cache the values in your pipeline. Your method should meet the following requirements:

* You should not in any way modify an element returned by ProcessContext.element() or ProcessContext.sideInput() (the incoming elements from the input collection).
* Once you output a value using ProcessContext.output() or ProcessContext.sideOutput(), you should not modify that value in any way.

##### Lightweight DoFns and other abstractions

If your function is relatively straightforward, you can simplify your use of ParDo by providing a lightweight DoFn in-line, as an anonymous inner class instance.

Here’s the previous example, ParDo with ComputeLengthWordsFn, with the DoFn specified as an anonymous inner class instance:

* Java
* Python

// The input PCollection.

PCollection<String> words = ...;

// Apply a ParDo with an anonymous DoFn to the PCollection words.

// Save the result as the PCollection wordLengths.

PCollection<Integer> wordLengths = words.apply(

"ComputeWordLengths", // the transform name

ParDo.of(new DoFn<String, Integer>() { // a DoFn as an anonymous inner class instance

@ProcessElement

public void processElement(ProcessContext c) {

c.output(c.element().length());

}

}));

If your ParDo performs a one-to-one mapping of input elements to output elements–that is, for each input element, it applies a function that produces exactly one output element, you can use the higher-level MapElements transform. MapElements can accept an anonymous Java 8 lambda function for additional brevity.

Here’s the previous example using MapElements:

* Java
* Python

// The input PCollection.

PCollection<String> words = ...;

// Apply a MapElements with an anonymous lambda function to the PCollection words.

// Save the result as the PCollection wordLengths.

PCollection<Integer> wordLengths = words.apply(

MapElements.via((String word) -> word.length())

.withOutputType(new TypeDescriptor<Integer>() {});

**Note:** You can use Java 8 lambda functions with several other Beam transforms, including Filter, FlatMapElements, and Partition.

#### Using GroupByKey

GroupByKey is a Beam transform for processing collections of key/value pairs. It’s a parallel reduction operation, analogous to the Shuffle phase of a Map/Shuffle/Reduce-style algorithm. The input to GroupByKey is a collection of key/value pairs that represents a multimap, where the collection contains multiple pairs that have the same key, but different values. Given such a collection, you use GroupByKey to collect all of the values associated with each unique key.

GroupByKey is a good way to aggregate data that has something in common. For example, if you have a collection that stores records of customer orders, you might want to group together all the orders from the same postal code (wherein the “key” of the key/value pair is the postal code field, and the “value” is the remainder of the record).

Let’s examine the mechanics of GroupByKey with a simple example case, where our data set consists of words from a text file and the line number on which they appear. We want to group together all the line numbers (values) that share the same word (key), letting us see all the places in the text where a particular word appears.

Our input is a PCollection of key/value pairs where each word is a key, and the value is a line number in the file where the word appears. Here’s a list of the key/value pairs in the input collection:

cat, 1

dog, 5

and, 1

jump, 3

tree, 2

cat, 5

dog, 2

and, 2

cat, 9

and, 6

...

GroupByKey gathers up all the values with the same key and outputs a new pair consisting of the unique key and a collection of all of the values that were associated with that key in the input collection. If we apply GroupByKey to our input collection above, the output collection would look like this:

cat, [1,5,9]

dog, [5,2]

and, [1,2,6]

jump, [3]

tree, [2]

...

Thus, GroupByKey represents a transform from a multimap (multiple keys to individual values) to a uni-map (unique keys to collections of values).

**A Note on Key/Value Pairs:** Beam represents key/value pairs slightly differently depending on the language and SDK you’re using. In the Beam SDK for Java, you represent a key/value pair with an object of type KV<K, V>. In Python, you represent key/value pairs with 2-tuples.

#### Using Combine

[Combine](https://beam.apache.org/documentation/sdks/javadoc/0.6.0/index.html?org/apache/beam/sdk/transforms/Combine.html) is a Beam transform for combining collections of elements or values in your data. Combine has variants that work on entire PCollections, and some that combine the values for each key in PCollections of key/value pairs.

When you apply a Combine transform, you must provide the function that contains the logic for combining the elements or values. The combining function should be commutative and associative, as the function is not necessarily invoked exactly once on all values with a given key. Because the input data (including the value collection) may be distributed across multiple workers, the combining function might be called multiple times to perform partial combining on subsets of the value collection. The Beam SDK also provides some pre-built combine functions for common numeric combination operations such as sum, min, and max.

Simple combine operations, such as sums, can usually be implemented as a simple function. More complex combination operations might require you to create a subclass of CombineFn that has an accumulation type distinct from the input/output type.

##### ****Simple combinations using simple functions****

The following example code shows a simple combine function.

* Java
* Python

// Sum a collection of Integer values. The function SumInts implements the interface SerializableFunction.

public static class SumInts implements SerializableFunction<Iterable<Integer>, Integer> {

@Override

public Integer apply(Iterable<Integer> input) {

int sum = 0;

for (int item : input) {

sum += item;

}

return sum;

}

}

##### ****Advanced combinations using CombineFn****

For more complex combine functions, you can define a subclass of CombineFn. You should use CombineFn if the combine function requires a more sophisticated accumulator, must perform additional pre- or post-processing, might change the output type, or takes the key into account.

A general combining operation consists of four operations. When you create a subclass of CombineFn, you must provide four operations by overriding the corresponding methods:

1. **Create Accumulator** creates a new “local” accumulator. In the example case, taking a mean average, a local accumulator tracks the running sum of values (the numerator value for our final average division) and the number of values summed so far (the denominator value). It may be called any number of times in a distributed fashion.
2. **Add Input** adds an input element to an accumulator, returning the accumulator value. In our example, it would update the sum and increment the count. It may also be invoked in parallel.
3. **Merge Accumulators** merges several accumulators into a single accumulator; this is how data in multiple accumulators is combined before the final calculation. In the case of the mean average computation, the accumulators representing each portion of the division are merged together. It may be called again on its outputs any number of times.
4. **Extract Output** performs the final computation. In the case of computing a mean average, this means dividing the combined sum of all the values by the number of values summed. It is called once on the final, merged accumulator.

The following example code shows how to define a CombineFn that computes a mean average:

* Java
* Python

public class AverageFn extends CombineFn<Integer, AverageFn.Accum, Double> {

public static class Accum {

int sum = 0;

int count = 0;

}

@Override

public Accum createAccumulator() { return new Accum(); }

@Override

public Accum addInput(Accum accum, Integer input) {

accum.sum += input;

accum.count++;

return accum;

}

@Override

public Accum mergeAccumulators(Iterable<Accum> accums) {

Accum merged = createAccumulator();

for (Accum accum : accums) {

merged.sum += accum.sum;

merged.count += accum.count;

}

return merged;

}

@Override

public Double extractOutput(Accum accum) {

return ((double) accum.sum) / accum.count;

}

}

If you are combining a PCollection of key-value pairs, [per-key combining](https://beam.apache.org/documentation/programming-guide/#transforms-combine-per-key) is often enough. If you need the combining strategy to change based on the key (for example, MIN for some users and MAX for other users), you can define a KeyedCombineFn to access the key within the combining strategy.

##### ****Combining a PCollection into a single value****

Use the global combine to transform all of the elements in a given PCollection into a single value, represented in your pipeline as a new PCollection containing one element. The following example code shows how to apply the Beam provided sum combine function to produce a single sum value for a PCollection of integers.

* Java
* Python

// Sum.SumIntegerFn() combines the elements in the input PCollection.

// The resulting PCollection, called sum, contains one value: the sum of all the elements in the input PCollection.

PCollection<Integer> pc = ...;

PCollection<Integer> sum = pc.apply(

Combine.globally(new Sum.SumIntegerFn()));

##### Global windowing:

If your input PCollection uses the default global windowing, the default behavior is to return a PCollection containing one item. That item’s value comes from the accumulator in the combine function that you specified when applying Combine. For example, the Beam provided sum combine function returns a zero value (the sum of an empty input), while the min combine function returns a maximal or infinite value.

To have Combine instead return an empty PCollection if the input is empty, specify .withoutDefaults when you apply your Combine transform, as in the following code example:

* Java
* Python

PCollection<Integer> pc = ...;

PCollection<Integer> sum = pc.apply(

Combine.globally(new Sum.SumIntegerFn()).withoutDefaults());

##### Non-global windowing:

If your PCollection uses any non-global windowing function, Beam does not provide the default behavior. You must specify one of the following options when applying Combine:

* Specify .withoutDefaults, where windows that are empty in the input PCollection will likewise be empty in the output collection.
* Specify .asSingletonView, in which the output is immediately converted to a PCollectionView, which will provide a default value for each empty window when used as a side input. You’ll generally only need to use this option if the result of your pipeline’s Combine is to be used as a side input later in the pipeline.

##### ****Combining values in a key-grouped collection****

After creating a key-grouped collection (for example, by using a GroupByKey transform) a common pattern is to combine the collection of values associated with each key into a single, merged value. Drawing on the previous example from GroupByKey, a key-grouped PCollection called groupedWords looks like this:

cat, [1,5,9]

dog, [5,2]

and, [1,2,6]

jump, [3]

tree, [2]

...

In the above PCollection, each element has a string key (for example, “cat”) and an iterable of integers for its value (in the first element, containing [1, 5, 9]). If our pipeline’s next processing step combines the values (rather than considering them individually), you can combine the iterable of integers to create a single, merged value to be paired with each key. This pattern of a GroupByKey followed by merging the collection of values is equivalent to Beam’s Combine PerKey transform. The combine function you supply to Combine PerKey must be an associative reduction function or a subclass of CombineFn.

* Java
* Python

// PCollection is grouped by key and the Double values associated with each key are combined into a Double.

PCollection<KV<String, Double>> salesRecords = ...;

PCollection<KV<String, Double>> totalSalesPerPerson =

salesRecords.apply(Combine.<String, Double, Double>perKey(

new Sum.SumDoubleFn()));

// The combined value is of a different type than the original collection of values per key.

// PCollection has keys of type String and values of type Integer, and the combined value is a Double.

PCollection<KV<String, Integer>> playerAccuracy = ...;

PCollection<KV<String, Double>> avgAccuracyPerPlayer =

playerAccuracy.apply(Combine.<String, Integer, Double>perKey(

new MeanInts())));

#### Using Flatten and Partition

[Flatten](https://beam.apache.org/documentation/sdks/javadoc/0.6.0/index.html?org/apache/beam/sdk/transforms/Flatten.html) and [Partition](https://beam.apache.org/documentation/sdks/javadoc/0.6.0/index.html?org/apache/beam/sdk/transforms/Partition.html) are Beam transforms for PCollection objects that store the same data type. Flatten merges multiple PCollection objects into a single logical PCollection, and Partition splits a single PCollection into a fixed number of smaller collections.

##### ****Flatten****

The following example shows how to apply a Flatten transform to merge multiple PCollection objects.

* Java
* Python

// Flatten takes a PCollectionList of PCollection objects of a given type.

// Returns a single PCollection that contains all of the elements in the PCollection objects in that list.

PCollection<String> pc1 = ...;

PCollection<String> pc2 = ...;

PCollection<String> pc3 = ...;

PCollectionList<String> collections = PCollectionList.of(pc1).and(pc2).and(pc3);

PCollection<String> merged = collections.apply(Flatten.<String>pCollections());

##### Data encoding in merged collections:

By default, the coder for the output PCollection is the same as the coder for the first PCollection in the input PCollectionList. However, the input PCollection objects can each use different coders, as long as they all contain the same data type in your chosen language.

##### Merging windowed collections:

When using Flatten to merge PCollection objects that have a windowing strategy applied, all of the PCollection objects you want to merge must use a compatible windowing strategy and window sizing. For example, all the collections you’re merging must all use (hypothetically) identical 5-minute fixed windows or 4-minute sliding windows starting every 30 seconds.

If your pipeline attempts to use Flatten to merge PCollection objects with incompatible windows, Beam generates an IllegalStateException error when your pipeline is constructed.

##### ****Partition****

Partition divides the elements of a PCollection according to a partitioning function that you provide. The partitioning function contains the logic that determines how to split up the elements of the input PCollection into each resulting partition PCollection. The number of partitions must be determined at graph construction time. You can, for example, pass the number of partitions as a command-line option at runtime (which will then be used to build your pipeline graph), but you cannot determine the number of partitions in mid-pipeline (based on data calculated after your pipeline graph is constructed, for instance).

The following example divides a PCollection into percentile groups.

* Java
* Python

// Provide an int value with the desired number of result partitions, and a PartitionFn that represents the partitioning function.

// In this example, we define the PartitionFn in-line.

// Returns a PCollectionList containing each of the resulting partitions as individual PCollection objects.

PCollection<Student> students = ...;

// Split students up into 10 partitions, by percentile:

PCollectionList<Student> studentsByPercentile =

students.apply(Partition.of(10, new PartitionFn<Student>() {

public int partitionFor(Student student, int numPartitions) {

return student.getPercentile() // 0..99

\* numPartitions / 100;

}}));

// You can extract each partition from the PCollectionList using the get method, as follows:

PCollection<Student> fortiethPercentile = studentsByPercentile.get(4);

#### General Requirements for writing user code for Beam transforms

When you build user code for a Beam transform, you should keep in mind the distributed nature of execution. For example, there might be many copies of your function running on a lot of different machines in parallel, and those copies function independently, without communicating or sharing state with any of the other copies. Depending on the Pipeline Runner and processing back-end you choose for your pipeline, each copy of your user code function may be retried or run multiple times. As such, you should be cautious about including things like state dependency in your user code.

In general, your user code must fulfill at least these requirements:

* Your function object must be **serializable**.
* Your function object must be **thread-compatible**, and be aware that the Beam SDKs are not thread-safe.

In addition, it’s recommended that you make your function object **idempotent**.

**Note:** These requirements apply to subclasses of DoFn (a function object used with the [ParDo](https://beam.apache.org/documentation/programming-guide/#transforms-pardo) transform), CombineFn (a function object used with the [Combine](https://beam.apache.org/documentation/programming-guide/#transforms-combine) transform), and WindowFn (a function object used with the [Window](https://beam.apache.org/documentation/programming-guide/#windowing) transform).

##### Serializability

Any function object you provide to a transform must be **fully serializable**. This is because a copy of the function needs to be serialized and transmitted to a remote worker in your processing cluster. The base classes for user code, such as DoFn, CombineFn, and WindowFn, already implement Serializable; however, your subclass must not add any non-serializable members.

Some other serializability factors you should keep in mind are:

* Transient fields in your function object are not transmitted to worker instances, because they are not automatically serialized.
* Avoid loading a field with a large amount of data before serialization.
* Individual instances of your function object cannot share data.
* Mutating a function object after it gets applied will have no effect.
* Take care when declaring your function object inline by using an anonymous inner class instance. In a non-static context, your inner class instance will implicitly contain a pointer to the enclosing class and that class’ state. That enclosing class will also be serialized, and thus the same considerations that apply to the function object itself also apply to this outer class.

##### Thread-compatibility

Your function object should be thread-compatible. Each instance of your function object is accessed by a single thread on a worker instance, unless you explicitly create your own threads. Note, however, that **the Beam SDKs are not thread-safe**. If you create your own threads in your user code, you must provide your own synchronization. Note that static members in your function object are not passed to worker instances and that multiple instances of your function may be accessed from different threads.

##### Idempotence

It’s recommended that you make your function object idempotent–that is, that it can be repeated or retried as often as necessary without causing unintended side effects. The Beam model provides no guarantees as to the number of times your user code might be invoked or retried; as such, keeping your function object idempotent keeps your pipeline’s output deterministic, and your transforms’ behavior more predictable and easier to debug.

#### Side Inputs and Side Outputs

##### ****Side inputs****

In addition to the main input PCollection, you can provide additional inputs to a ParDo transform in the form of side inputs. A side input is an additional input that your DoFn can access each time it processes an element in the input PCollection. When you specify a side input, you create a view of some other data that can be read from within the ParDo transform’s DoFn while procesing each element.

Side inputs are useful if your ParDo needs to inject additional data when processing each element in the input PCollection, but the additional data needs to be determined at runtime (and not hard-coded). Such values might be determined by the input data, or depend on a different branch of your pipeline.

##### Passing side inputs to ParDo:

* Java
* Python

// Pass side inputs to your ParDo transform by invoking .withSideInputs.

// Inside your DoFn, access the side input by using the method DoFn.ProcessContext.sideInput.

// The input PCollection to ParDo.

PCollection<String> words = ...;

// A PCollection of word lengths that we'll combine into a single value.

PCollection<Integer> wordLengths = ...; // Singleton PCollection

// Create a singleton PCollectionView from wordLengths using Combine.globally and View.asSingleton.

final PCollectionView<Integer> maxWordLengthCutOffView =

wordLengths.apply(Combine.globally(new Max.MaxIntFn()).asSingletonView());

// Apply a ParDo that takes maxWordLengthCutOffView as a side input.

PCollection<String> wordsBelowCutOff =

words.apply(ParDo.withSideInputs(maxWordLengthCutOffView)

.of(new DoFn<String, String>() {

public void processElement(ProcessContext c) {

String word = c.element();

// In our DoFn, access the side input.

int lengthCutOff = c.sideInput(maxWordLengthCutOffView);

if (word.length() <= lengthCutOff) {

c.output(word);

}

}}));

##### Side inputs and windowing:

A windowed PCollection may be infinite and thus cannot be compressed into a single value (or single collection class). When you create a PCollectionView of a windowed PCollection, the PCollectionView represents a single entity per window (one singleton per window, one list per window, etc.).

Beam uses the window(s) for the main input element to look up the appropriate window for the side input element. Beam projects the main input element’s window into the side input’s window set, and then uses the side input from the resulting window. If the main input and side inputs have identical windows, the projection provides the exact corresponding window. However, if the inputs have different windows, Beam uses the projection to choose the most appropriate side input window.

For example, if the main input is windowed using fixed-time windows of one minute, and the side input is windowed using fixed-time windows of one hour, Beam projects the main input window against the side input window set and selects the side input value from the appropriate hour-long side input window.

If the main input element exists in more than one window, then processElement gets called multiple times, once for each window. Each call to processElement projects the “current” window for the main input element, and thus might provide a different view of the side input each time.

If the side input has multiple trigger firings, Beam uses the value from the latest trigger firing. This is particularly useful if you use a side input with a single global window and specify a trigger.

##### ****Side outputs****

While ParDo always produces a main output PCollection (as the return value from apply), you can also have your ParDo produce any number of additional output PCollections. If you choose to have multiple outputs, your ParDo returns all of the output PCollections (including the main output) bundled together.

##### Tags for side outputs:

* Java
* Python

// To emit elements to a side output PCollection, create a TupleTag object to identify each collection that your ParDo produces.

// For example, if your ParDo produces three output PCollections (the main output and two side outputs), you must create three TupleTags.

// The following example code shows how to create TupleTags for a ParDo with a main output and two side outputs:

// Input PCollection to our ParDo.

PCollection<String> words = ...;

// The ParDo will filter words whose length is below a cutoff and add them to

// the main ouput PCollection<String>.

// If a word is above the cutoff, the ParDo will add the word length to a side output

// PCollection<Integer>.

// If a word starts with the string "MARKER", the ParDo will add that word to a different

// side output PCollection<String>.

final int wordLengthCutOff = 10;

// Create the TupleTags for the main and side outputs.

// Main output.

final TupleTag<String> wordsBelowCutOffTag =

new TupleTag<String>(){};

// Word lengths side output.

final TupleTag<Integer> wordLengthsAboveCutOffTag =

new TupleTag<Integer>(){};

// "MARKER" words side output.

final TupleTag<String> markedWordsTag =

new TupleTag<String>(){};

// Passing Output Tags to ParDo:

// After you specify the TupleTags for each of your ParDo outputs, pass the tags to your ParDo by invoking .withOutputTags.

// You pass the tag for the main output first, and then the tags for any side outputs in a TupleTagList.

// Building on our previous example, we pass the three TupleTags (one for the main output and two for the side outputs) to our ParDo.

// Note that all of the outputs (including the main output PCollection) are bundled into the returned PCollectionTuple.

PCollectionTuple results =

words.apply(

ParDo

// Specify the tag for the main output, wordsBelowCutoffTag.

.withOutputTags(wordsBelowCutOffTag,

// Specify the tags for the two side outputs as a TupleTagList.

TupleTagList.of(wordLengthsAboveCutOffTag)

.and(markedWordsTag))

.of(new DoFn<String, String>() {

// DoFn continues here.

...

}

##### Emitting to side outputs in your DoFn:

* Java
* Python

// Inside your ParDo's DoFn, you can emit an element to a side output by using the method ProcessContext.sideOutput.

// Pass the appropriate TupleTag for the target side output collection when you call ProcessContext.sideOutput.

// After your ParDo, extract the resulting main and side output PCollections from the returned PCollectionTuple.

// Based on the previous example, this shows the DoFn emitting to the main and side outputs.

.of(new DoFn<String, String>() {

public void processElement(ProcessContext c) {

String word = c.element();

if (word.length() <= wordLengthCutOff) {

// Emit this short word to the main output.

c.output(word);

} else {

// Emit this long word's length to a side output.

c.sideOutput(wordLengthsAboveCutOffTag, word.length());

}

if (word.startsWith("MARKER")) {

// Emit this word to a different side output.

c.sideOutput(markedWordsTag, word);

}

}}));

## Composite Transforms

**Note:** This section is in progress ([BEAM-1452](https://issues.apache.org/jira/browse/BEAM-1452)).

## Pipeline I/O

When you create a pipeline, you often need to read data from some external source, such as a file in external data sink or a database. Likewise, you may want your pipeline to output its result data to a similar external data sink. Beam provides read and write transforms for a number of common data storage types. If you want your pipeline to read from or write to a data storage format that isn’t supported by the built-in transforms, you can implement your own read and write transforms.

A guide that covers how to implement your own Beam IO transforms is in progress ([BEAM-1025](https://issues.apache.org/jira/browse/BEAM-1025)).

### Reading input data

Read transforms read data from an external source and return a PCollection representation of the data for use by your pipeline. You can use a read transform at any point while constructing your pipeline to create a new PCollection, though it will be most common at the start of your pipeline.

#### Using a read transform:

* Java
* Python

PCollection<String> lines = p.apply(TextIO.Read.from("gs://some/inputData.txt"));

### Writing output data

Write transforms write the data in a PCollection to an external data source. You will most often use write transforms at the end of your pipeline to output your pipeline’s final results. However, you can use a write transform to output a PCollection’s data at any point in your pipeline.

#### Using a Write transform:

* Java
* Python

output.apply(TextIO.Write.to("gs://some/outputData"));

### File-based input and output data

#### Reading from multiple locations:

Many read transforms support reading from multiple input files matching a glob operator you provide. Note that glob operators are filesystem-specific and obey filesystem-specific consistency models. The following TextIO example uses a glob operator (\*) to read all matching input files that have prefix “input-“ and the suffix “.csv” in the given location:

* Java
* Python

p.apply(“ReadFromText”,

TextIO.Read.from("protocol://my\_bucket/path/to/input-\*.csv");

To read data from disparate sources into a single PCollection, read each one independently and then use the [Flatten](https://beam.apache.org/documentation/programming-guide/#transforms-flatten-partition) transform to create a single PCollection.

#### Writing to multiple output files:

For file-based output data, write transforms write to multiple output files by default. When you pass an output file name to a write transform, the file name is used as the prefix for all output files that the write transform produces. You can append a suffix to each output file by specifying a suffix.

The following write transform example writes multiple output files to a location. Each file has the prefix “numbers”, a numeric tag, and the suffix “.csv”.

* Java
* Python

records.apply("WriteToText",

TextIO.Write.to("protocol://my\_bucket/path/to/numbers")

.withSuffix(".csv"));

### Beam-provided I/O APIs

See the language specific source code directories for the Beam supported I/O APIs. Specific documentation for each of these I/O sources will be added in the future. ([BEAM-1054](https://issues.apache.org/jira/browse/BEAM-1054))

|  |  |  |  |
| --- | --- | --- | --- |
| **Language** | **File-based** | **Messaging** | **Database** |
| Java | [AvroIO](https://github.com/apache/beam/blob/master/sdks/java/core/src/main/java/org/apache/beam/sdk/io/AvroIO.java)  [HDFS](https://github.com/apache/beam/tree/master/sdks/java/io/hdfs)  [TextIO](https://github.com/apache/beam/blob/master/sdks/java/core/src/main/java/org/apache/beam/sdk/io/TextIO.java)  [XML](https://github.com/apache/beam/blob/master/sdks/java/core/src/main/java/org/apache/beam/sdk/io/) | [JMS](https://github.com/apache/beam/tree/master/sdks/java/io/jms)  [Kafka](https://github.com/apache/beam/tree/master/sdks/java/io/kafka)  [Kinesis](https://github.com/apache/beam/tree/master/sdks/java/io/kinesis)  [Google Cloud PubSub](https://github.com/apache/beam/blob/master/sdks/java/core/src/main/java/org/apache/beam/sdk/io) | [Apache HBase](https://github.com/apache/beam/tree/master/sdks/java/io/hbase)  [MongoDB](https://github.com/apache/beam/tree/master/sdks/java/io/mongodb)  [JDBC](https://github.com/apache/beam/tree/master/sdks/java/io/jdbc)  [Google BigQuery](https://github.com/apache/beam/tree/master/sdks/java/io/google-cloud-platform/src/main/java/org/apache/beam/sdk/io/gcp/bigquery)  [Google Cloud Bigtable](https://github.com/apache/beam/tree/master/sdks/java/io/google-cloud-platform/src/main/java/org/apache/beam/sdk/io/gcp/bigtable)  [Google Cloud Datastore](https://github.com/apache/beam/tree/master/sdks/java/io/google-cloud-platform/src/main/java/org/apache/beam/sdk/io/gcp/datastore) |
| Python | [avroio](https://github.com/apache/beam/blob/master/sdks/python/apache_beam/io/avroio.py)  [textio](https://github.com/apache/beam/blob/master/sdks/python/apache_beam/io/textio.py) |  | [Google BigQuery](https://github.com/apache/beam/blob/master/sdks/python/apache_beam/io/gcp/bigquery.py)  [Google Cloud Datastore](https://github.com/apache/beam/tree/master/sdks/python/apache_beam/io/gcp/datastore) |

## Running the pipeline

To run your pipeline, use the run method. The program you create sends a specification for your pipeline to a pipeline runner, which then constructs and runs the actual series of pipeline operations. Pipelines are executed asynchronously by default.

* Java
* Python

pipeline.run();

For blocking execution, append the waitUntilFinish method:

* Java
* Python

pipeline.run().waitUntilFinish();

## Data encoding and type safety

When you create or output pipeline data, you’ll need to specify how the elements in your PCollections are encoded and decoded to and from byte strings. Byte strings are used for intermediate storage as well reading from sources and writing to sinks. The Beam SDKs use objects called coders to describe how the elements of a given PCollection should be encoded and decoded.

### Using coders

You typically need to specify a coder when reading data into your pipeline from an external source (or creating pipeline data from local data), and also when you output pipeline data to an external sink.

In the Beam SDK for Java, the type Coder provides the methods required for encoding and decoding data. The SDK for Java provides a number of Coder subclasses that work with a variety of standard Java types, such as Integer, Long, Double, StringUtf8 and more. You can find all of the available Coder subclasses in the [Coder package](https://github.com/apache/beam/tree/master/sdks/java/core/src/main/java/org/apache/beam/sdk/coders).

When you read data into a pipeline, the coder indicates how to interpret the input data into a language-specific type, such as integer or string. Likewise, the coder indicates how the language-specific types in your pipeline should be written into byte strings for an output data sink, or to materialize intermediate data in your pipeline.

The Beam SDKs set a coder for every PCollection in a pipeline, including those generated as output from a transform. Most of the time, the Beam SDKs can automatically infer the correct coder for an output PCollection.

Note that coders do not necessarily have a 1:1 relationship with types. For example, the Integer type can have multiple valid coders, and input and output data can use different Integer coders. A transform might have Integer-typed input data that uses BigEndianIntegerCoder, and Integer-typed output data that uses VarIntCoder.

You can explicitly set a Coder when inputting or outputting a PCollection. You set the Coder by calling the method .withCoder when you apply your pipeline’s read or write transform.

Typically, you set the Coder when the coder for a PCollection cannot be automatically inferred, or when you want to use a different coder than your pipeline’s default. The following example code reads a set of numbers from a text file, and sets a Coder of type TextualIntegerCoder for the resulting PCollection:

* Java
* Python

PCollection<Integer> numbers =

p.begin()

.apply(TextIO.Read.named("ReadNumbers")

.from("gs://my\_bucket/path/to/numbers-\*.txt")

.withCoder(TextualIntegerCoder.of()));

You can set the coder for an existing PCollection by using the method PCollection.setCoder. Note that you cannot call setCoder on a PCollection that has been finalized (e.g. by calling .apply on it).

You can get the coder for an existing PCollection by using the method getCoder. This method will fail with anIllegalStateException if a coder has not been set and cannot be inferred for the given PCollection.

### Coder inference and default coders

The Beam SDKs require a coder for every PCollection in your pipeline. Most of the time, however, you do not need to explicitly specify a coder, such as for an intermediate PCollection produced by a transform in the middle of your pipeline. In such cases, the Beam SDKs can infer an appropriate coder from the inputs and outputs of the transform used to produce the PCollection.

Each pipeline object has a CoderRegistry. The CoderRegistry represents a mapping of Java types to the default coders that the pipeline should use for PCollections of each type.

By default, the Beam SDK for Java automatically infers the Coder for the elements of an output PCollection using the type parameter from the transform’s function object, such as DoFn. In the case of ParDo, for example, a DoFn<Integer, String>function object accepts an input element of type Integer and produces an output element of type String. In such a case, the SDK for Java will automatically infer the default Coder for the output PCollection<String> (in the default pipeline CoderRegistry, this is StringUtf8Coder).

NOTE: If you create your PCollection from in-memory data by using the Create transform, you cannot rely on coder inference and default coders. Create does not have access to any typing information for its arguments, and may not be able to infer a coder if the argument list contains a value whose exact run-time class doesn’t have a default coder registered.

When using Create, the simplest way to ensure that you have the correct coder is by invoking withCoder when you apply the Create transform.

#### Default coders and the CoderRegistry

Each Pipeline object has a CoderRegistry object, which maps language types to the default coder the pipeline should use for those types. You can use the CoderRegistry yourself to look up the default coder for a given type, or to register a new default coder for a given type.

CoderRegistry contains a default mapping of coders to standard Java types for any pipeline you create using the Beam SDK for Java . The following table shows the standard mapping:

| **Java Type** | **Default Coder** |
| --- | --- |
| Double | DoubleCoder |
| Instant | InstantCoder |
| Integer | VarIntCoder |
| Iterable | IterableCoder |
| KV | KvCoder |
| List | ListCoder |
| Map | MapCoder |
| Long | VarLongCoder |
| String | StringUtf8Coder |
| TableRow | TableRowJsonCoder |
| Void | VoidCoder |
| byte[ ] | ByteArrayCoder |
| TimestampedValue | TimestampedValueCoder |

##### Looking up a default coder

You can use the method CoderRegistry.getDefaultCoder to determine the default Coder for a Java type. You can access the CoderRegistry for a given pipeline by using the method Pipeline.getCoderRegistry. This allows you to determine (or set) the default Coder for a Java type on a per-pipeline basis: i.e. “for this pipeline, verify that Integer values are encoded using BigEndianIntegerCoder.”

##### Setting the default coder for a type

To set the default Coder for a Java type for a particular pipeline, you obtain and modify the pipeline’s CoderRegistry. You use the method Pipeline.getCoderRegistry to get the CoderRegistry object, and then use the method CoderRegistry.registerCoder to register a new Coder for the target type.

The following example code demonstrates how to set a default Coder, in this case BigEndianIntegerCoder, for Integer values for a pipeline.

* Java
* Python

PipelineOptions options = PipelineOptionsFactory.create();

Pipeline p = Pipeline.create(options);

CoderRegistry cr = p.getCoderRegistry();

cr.registerCoder(Integer.class, BigEndianIntegerCoder.class);

##### Annotating a custom data type with a default coder

If your pipeline program defines a custom data type, you can use the @DefaultCoder annotation to specify the coder to use with that type. For example, let’s say you have a custom data type for which you want to use SerializableCoder. You can use the @DefaultCoder annotation as follows:

@DefaultCoder(AvroCoder.class)

public class MyCustomDataType {

...

}

If you’ve created a custom coder to match your data type, and you want to use the @DefaultCoder annotation, your coder class must implement a static Coder.of(Class<T>) factory method.

public class MyCustomCoder implements Coder {

public static Coder<T> of(Class<T> clazz) {...}

...

}

@DefaultCoder(MyCustomCoder.class)

public class MyCustomDataType {

...

}

**Note:** This guide is still in progress. There is an open issue to finish the guide ([BEAM-193](https://issues.apache.org/jira/browse/BEAM-193))