Opencsv Users Guide

# General

Opencsv is an easy-to-use CSV (comma-separated values) parser library for Java. It was developed because all the CSV parsers at the time didn’t have commercial-friendly licenses. Java 7 is currently the minimum supported version.

# Features

Opencsv supports all the basic CSV-type things you’re likely to want to do:

* Arbitrary numbers of values per line.

每行有任意数目的值

* Ignoring commas in quoted elements.

忽略引号中的逗号

* Handling quoted entries with embedded carriage returns (i.e. entries that span multiple lines).

处理带有嵌入式回车符的引用实体

* Configurable separator and quote characters (or use sensible defaults).

可配置的分隔符和引用符号

All of these things can be done reading and writing, using a manifest of malleable methodologies:

* To and from an array of strings.
* To and from annotated beans.
* From a database
* Read all the entries at once, or use an Iterator-style model

# Developer Documentation

Here is an overview of how to use opencsv in your project.

Once you have absorbed the overview of how opencsv works, please consult the well-maintained Javadocs for further details.

## Quick start

This is limited to the easiest, most powerful way of using opencsv to allow you to hit the ground running.

For reading, create a bean to harbor the information you want to read, annotate the bean fields with the opencsv annotations, then do this:

|  |
| --- |
| List<MyBean> beans = new CsvToBeanBuilder(FileReader("yourfile.csv"))  .withType(Visitors.class).build().parse(); |
| 根据Visitors的注解不去相关的信息，转换成MyBean的List |

For writing, create a bean to harbor the information you want to write, annotate the bean fields with the opencsv annotations, then do this:

|  |
| --- |
| // List<MyBean> beans comes from somewhere earlier in your code.  Writer writer = new FileWriter("yourfile.csv");  StatefulBeanToCsv beanToCsv = new StatefulBeanToCsvBuilder(writer).build();  beanToCsv.write(beans);  writer.close(); |

### Even quicker start

Starting with version 4.2, there’s another handy way of reading CSV files that doesn’t even require creating special classes. If your CSV file has headers, you can just initialize a CSVReaderHeaderAware and start reading the values out as a map:

|  |
| --- |
| Map<String, String> values = new CSVReaderHeaderAware(new FileReader("yourfile.csv")).readMap(); |

## Upgrading from 3.x to 4.x

4.0 is a major release because it breaks backward compatibility. What do you get for that? Here is a list of the improvements in opencsv 4.0.

* We have rewritten the bean code to be multi-threaded so that reading from an input directly into beans is significantly faster. Performance benefits depend largely on your data and hardware, but our non-rigorous tests indicate that reading now takes a <em>third</em> of the time it used to.
* We have rewritten the bean code to be multi-threaded so that writing from a list of beans is significantly faster. Performance benefits depend largely on your data and hardware, but our non-rigorous tests indicate that writing now takes <em>half</em> of the time it used to.
* There is a new iterator available for iterating through the input into beans. This iterator is consistent in every way with the behavior of the code that reads all data sets at once into a list of beans. The old iterator did not support all features, like locales and custom converters.
* opencsv now supports internationalization for all error messages it produces. The easiest way to benefit from this is to make certain the default locale is the one you want. Otherwise, look for the withErrorLocale() and setErrorLocale() methods in various classes. Localizations are provided for American English and German. Further submissions are welcome, but with a submission you enter into a life-long contract to provide updates for any new messages for the language(s) you submit. If you break this contract, you forefit your soul.
* Support for national character sets was added to ResultSetHelperService (NClob, NVarchar, NChar, LongNVarchar).

Here are the things you can expect to encounter during an upgrade, and what to do about them.

* Java 7 is now the minimum supported version. Tough noogies.
* Everything that was deprecated has been removed.
* BeanToCsv is no more. Please use StatefulBeanToCsv instead. The quick start guide above gives you an example.
* @CsvBind was replaced with @CsvBindByName. It really is as simple as search and replace.
* ConvertGermanToBooleanRequired was removed. Replace it with @CsvCustomBindByName(converter = ConvertGermanToBoolean.class, required = true).
* In the rare case that you have written your own mapping strategy:
* MappingStrategy now includes a method verifyLineLength(). If you derive your mapping strategy from one of ours, you’re okay. Otherwise, you will have to implement it.
* In the rare case that you used opencsv 3.10, registerBeginningOfRecordForReading() and registerEndOfRecordForReading() were removed from MappingStrategy. They were the result of thought processes worthy of nothing more accomplished than a drunken monkey. I may write that because I wrote the bad code. If you derived your mapping strategy from one of ours, you’re okay. Otherwise, you’ll have to remove these methods.
* findDescriptor no longer includes "throws IntrospectionException" in its method signature. If you had it, you’ll have to get rid of it. If you had it an needed it, you’ll have to rewrite your code.
* There are now requirements for thread-safety imposed on certain methods in every mapping strategy. See the Javadoc for MappingStrategy for details.
* The method setErrorLocale() is now required. If you derive your implementation from one of ours, you’re fine. If not, implement it, or make it a no-op.
* The method setType() is now required. If you derive your implementation from one of ours, you’re fine. If not, implement it, or make it a no-op.
* MappingUtils was really meant to be for internal use, but of course we can’t control that, so let it be said that:
* the class is now named OpencsvUtils, because it encompasses more than mapping, and
* the determineMappingStrategy() method now requires a locale for error messages. Null can be used for the default locale.
* The constructors for BeanFieldDate and BeanFieldPrimitiveType now require a locale for error messages. This is to avoid a proliferation of constructors or setters. These classes probably ought not to be used in your code directly, and probably ought to be final, but we still thought it best to inform you.
* The interface BeanField requires the method setErrorLocale(). Assuming you derive all of your BeanField implementations from AbstractBeanField, this does not affect you.

And we have a new list of things that we have deprecated and plan to remove in 5.0, as well as what you can do about it.

* IterableCSVToBean and IterableCSVToBeanBuilder have both been deprecated. CsvToBean itself is now iterable; use it instead.
* All constructors except the ones with the smallest (often nullary, using defaults for all values) and largest argument lists (which often have only package access) have been deprecated. The constructors in between have grown over the years as opencsv has added features, and they’ve become unwieldy. We encourage all of our users to use the builders we provide instead of the constructors.
* All variants of CsvToBean.parse() except the no-argument variant. Please use the builder we provide.
* MappingStrategy.findDescriptor() will no longer be necessary in 5.0 because the plan is to move to reflection completely and no longer use introspection.

## Core concepts

There are a couple of concepts that most users of opencsv need to understand, and that apply equally to reading and writing.

### Configuration

"CSV" stands for "comma-separated values", but life would be too simple if that were always true. Often the separator is a semicolon. Sometimes the separator character is included in the data for a field itself, so quotation characters are necessary. Those quotation characters could be included in the data also, so an escape character is necessary. All of these configuration options and more are given to the parser or the CSVWriter as necessary. Naturally, it’s easier for you to give them to a builder and the builder passes them on to the right class.

Say you’re using a tab for your separator, you can do something like this:

|  |
| --- |
| CSVReader reader = new CSVReader(new FileReader("yourfile.csv"), '\t'); |

or for reading with annotations:

|  |
| --- |
| CsvToBean csvToBean = new CsvToBeanBuilder(new FileReader("yourfile.csv"))  .withSeparator('\t').build(); |

And if you single-quoted your escaped characters rather than double-quoting them, you can use the three-argument constructor:

|  |
| --- |
| CSVReader reader = new CSVReader(new FileReader("yourfile.csv"), '\t', '\''); |

or for reading with annotations:

|  |
| --- |
| CsvToBean csvToBean = new CsvToBeanBuilder(new FileReader("yourfile.csv"))  .withSeparator('\t').withQuoteChar('\'').build(); |

### Error handling

opencsv uses structured exception handling, including checked and unchecked exceptions. The checked exceptions are typically errors in input data and do not have to impede further parsing. They could occur at any time during normal operation in a production environment. They occur during reading or writing.

The unchecked errors are typically the result of incorrect programming and should not be thrown in a production environment with well-tested code.

opencsv gives you two options for handling the checked exceptions both while reading and while writing. You may either choose to have all exceptions thrown and handle these, or you may choose to have them collected so you can inspect and deal with them after parsing. If you don’t have them collected, the first error in the input file will force a cessation of parsing. The default is to throw exceptions.

To change exception handling, simply use CsvToBeanBuilder.withThrowExceptions() for reading and StatefulBeanToCsvBuilder.withThrowExceptions() for writing, then collect the results after data processing with CsvToBean.getCapturedExceptions() for reading and StatefulBeanToCsv.getCapturedExceptions() for writing.

### Annotations

The most powerful mechanism opencsv has for reading and writing CSV files involves defining beans that the fields of the CSV file can be mapped to and from, and annotating the fields of these beans so opencsv can do the rest. In brief, these annotations are:

* + CsvBindByName: Maps a bean field to a field in the CSV file based on the name of the header for that field in the CSV input.
  + CsvBindByPosition: Maps a bean field to a field in the CSV file based on the numerical position of the field in the CSV input.
  + CsvBindAndSplitByName: Maps a Collection-based bean field to a field in the CSV file based on the name of the header for that field in the CSV input.
  + CsvBindAndSplitByPosition: Maps a Collection-based bean field to a field in the CSV file based on the numerical position of the field in the CSV input.
  + CsvBindAndJoinByName: Maps multiple input columns in the CSV file to one bean field based on the name of the headers for those fields in the CSV input.
  + CsvBindAndJoinByPosition: Maps multiple input columns in the CSV file to one bean field based on the numerical positions of those fields in the CSV input.
  + CsvDate: Must be applied to bean fields of date/time types for automatic conversion to work, and must be used in conjunction with one of the preceding six annotations.
  + CsvNumber: May be applied to bean fields of a type derived from java.lang.Number, and when used must be used in conjunction with one of the first six annotations.
  + CsvCustomBindByName: The same as CsvBindByName, but must provide its own data conversion class.
  + CsvCustomBindByPosition: The same as CsvBindByPosition, but must provide its own data conversion class.

As you can infer, there are two strategies for annotating beans, depending on your input:

* + Annotating by header name
  + Annotating by column position

It is possible to annotate bean fields both with header-based and position-based annotations. If you do, position-based annotations take precedence if the mapping strategy is automatically determined. To use the header-based annotations, you would need to instantiate and pass in a HeaderColumnNameMappingStrategy. When might this be useful? Possibly reading two different sources that provide the same data, but one includes headers and the other doesn’t. Possibly to convert between headerless input and output with headers. Further use cases are left as an exercise for the reader.

Most of the more detailed documentation on using annotations is in the section on reading data. The use of annotations applies equally well to writing data, though; the annotations define a two-way mapping between bean fields and fields in a CSV file. Writing is then simply reading in reverse.

## Reading

Most users of opencsv find themselves needing to read CSV files, and opencsv excels at this. But then, opencsv excels at everything. :)

### Parsing

It’s unlikely that you will need to concern yourself with exactly how parsing works in opencsv, but documentation wouldn’t be documentation if it didn’t cover all of the obscure nooks and crannies. So here we go.

Parsers in opencsv implement the interface ICSVParser. You are free to write your own, if you feel the need to. opencsv itself provides two parsers, detailed in the following sections.

Although opencsv attempts to be simple to use for most use cases, and thus tries not to make the choice of a parser obvious, you are still always free to instantiate whichever parser suits your needs and pass it to the builder or reader you are using.

#### CSVParser

The original, tried and true parser that does fairly well everything you need to do, and does it well. If you don’t tell opencsv otherwise, it uses this parser.

The advantage of the CSVParser is that it’s highly configurable and has the best chance of parsing "non-standard" CSV data. The disadvantage is that while highly configurable it was found that there were RFC4180 data that it could not parse. Thus the RFC4180Parser was created.

#### RFC4180Parser

[RFC4180](https://www.rfc-editor.org/rfc/rfc4180.txt) defines a standard for all of the nitty-gritty questions of just precisely how CSV files are to be formatted, delimited, and escaped. Since opencsv predates RFC4180 by a few days and every effort was made to preserve backwards compatibility, it was necessary to write a new parser for full compliance with RFC4180.

The main difference between between the CSVParser and the RFC4180Parser is that the CSVParser uses an escape character to denote "unprintable" characters while the RFC4180 spec takes all characters between the first and last quote as gospel (with the exception of the double quote which is escaped by a double quote).

两个解析器的主要不同是CSVParser使用转义字符代表不可打印字符，然而RFC4180解析器指定拿双引号之间的字符作为内容，用两个引号来注解引号。

### Reading into an array of strings

At the most basic, you can use opencsv to parse an input and return a String[], thus:

|  |
| --- |
| CSVReader reader = new CSVReader(new FileReader("yourfile.csv"));  String [] nextLine;  while ((nextLine = reader.readNext()) != null) {  // nextLine[] is an array of values from the line  System.out.println(nextLine[0] + nextLine[1] + "etc...");  } |
| 转换成一个数组 |

One step up is reading all lines of the input file at once into a List<String[]>, thus:

|  |
| --- |
| CSVReader reader = new CSVReader(new FileReader("yourfile.csv"));  List<String[]> myEntries = reader.readAll(); |
| 转换成一个数组的list |

The last option for getting at an array of strings is to use an iterator:

|  |
| --- |
| CSVIterator iterator = new CSVIterator(new CSVReader(new FileReader("yourfile.csv")));  for(String[] nextLine : iterator) {  // nextLine[] is an array of values from the line  System.out.println(nextLine[0] + nextLine[1] + "etc...");  } |
| 使用迭代器 |

or:

|  |
| --- |
| CSVReader reader = new CSVReader(new FileReader("yourfile.csv"));  for(String[] nextLine : reader.iterator()) {  // nextLine[] is an array of values from the line  System.out.println(nextLine[0] + nextLine[1] + "etc...");  } |

### Reading into beans 把CSV文件转成Bean对象的List

Arrays of strings are all good and well, but there are simpler, more modern ways of data processing. Specifically, opencsv can read a CSV file directly into a list of beans. Quite often, that’s what we want anyway, to be able to pass the data around and process it as a connected dataset instead of individual fields whose position in an array must be intuited. We shall start with the easiest and most powerful method of reading data into beans, and work our way down to the cogs that offer finer control, for those who have a need for such a thing.

Performance always being one of our top concerns, reading is written to be multi-threaded, which truly speeds the library up by quite a bit. There are two performance choices left in your hands: . Time vs. memory: The classic trade-off. If memory is not a problem, read using CsvToBean.parse(), which will read all beans at once and is multi-threaded. If your memory is limited, use CsvToBean.iterator() and iterate over the input. Only one bean is read at a time, making multi-threading impossible and slowing down reading, but only one object is in memory at a time (assuming you process and release the object for the garbage collector immediately). . Ordered vs. unordered. opencsv preserves the order of the data given to it by default. Maintaining order when using parallel programming requires some extra effort which means extra CPU time. If order does not matter to you, use CsvToBeanBuilder.withOrderedResults(false). The performance benefit is not large, but it is measurable. The ordering or lack thereof applies to data as well as any captured exceptions.

The bean work was begun by Kyle Miller and extended by Tom Squires and Andrew Jones.

#### Annotations 用注解定义bean的Field和CSV列的映射

By simply defining a bean and annotating the fields, opencsv can do all of the rest. When we write "bean", that’s a loose approximation of the requirements. Actually, if you use annotations, opencsv uses reflection (not introspection) on reading, so all you need is a POJO (plain old Java object) that does not have to conform to the Java Bean Specification, but is required to be public and have a public nullary constructor. If getters and setters are present and accessible, they are used. Otherwise, opencsv bypasses access control restrictions to get to member variables.

Writing is unfortunately not quite the same. Writing requires getters.

Besides the basic mapping strategy, there are various mechanisms for processing certain kinds of data.

##### Annotating by header name

CSV files should have header names for all fields in the file, and these can be used to great advantage. By annotating a bean field with the name of the header whose data should be written in the field, opencsv can do all of the matching and copying for you. This also makes you independent of the order in which the headers occur in the file. For data like this:

|  |
| --- |
| firstName,lastName,visitsToWebsite  John,Doe,12  Jane,Doe,23 |

you could create the following bean:

|  |
| --- |
| public class Visitors {  @CsvBindByName  private String firstName;  @CsvBindByName  private String lastName;  @CsvBindByName  private int visitsToWebsite;  // Getters and setters go here.  } |

Here we simply name the fields identically to the header names. After that, reading is a simple job:

|  |
| --- |
| List<Visitors> beans = new CsvToBeanBuilder(FileReader("yourfile.csv"))  .withType(Visitors.class).build().parse(); |

This will give you a list of the two beans as defined in the example input file. Note how type conversions to basic data types (wrapped and unwrapped primitives and Strings) occur automatically.

Input can get more complicated, though, and opencsv gives you the tools to deal with that. Let’s start with the possibility that the header names can’t be mapped to Java field names:

|  |
| --- |
| First name,Last name,1 visit only  John,Doe,true  Jane,Doe,false |

In this case, we have spaces in the names and one header with a number as the initial character. Other problems can be encountered, such as international characters in header names. Additionally, we would like to require that at least the name be mandatory. For this case, our bean doesn’t look much different:

|  |
| --- |
| public class Visitors {  @CsvBindByName(column = "First Name", required = true)  private String firstName;  @CsvBindByName(column = "Last Name", required = true)  private String lastName;  @CsvBindByName(column = "1 visit only")  private boolean onlyOneVisit;  // Getters and setters go here.  } |

The code for reading remains unchanged.

Now let’s say that your data for whatever reason look like this:

|  |
| --- |
| First name,Last name,1 visit only  John middle:Bubba,Doe,true  Jane middle:Rachel,Doe,false |

Someone has included the person’s middle name in the field for the first name. But we really only want the first name. Do we have to write a custom converter? No, friends, there is an easier way:

|  |
| --- |
| @CsvBindByName(column = "First Name", required = true, capture="([^ ]+) .\*")  private String firstName; |

The capture option to all of the binding annotations (except the custom binding annotations, of course) allows you to tell opencsv just what part of the input field should actually be considered significant. opencsv takes the contents of the first capture group. In this example, we take everything up to but not including the first space and discard the rest. Please read the Javadoc for more details and handling of edge cases.

##### Annotating by column position

Not every scribe of CSV files is kind enough to provide header names. This is a no-no, but we’re not here to condemn the authors of poor data exports. Our goal is to provide our users with everything they could possibly need to parse CSV files, no matter how bad, as long as they’re still logically coherent in some way.

To that end, we have also accounted for the possibility that there are no headers, and data must be divined from column position. We will return to our previous input file sans header names:

|  |
| --- |
| John,Doe,12  Jane,Doe,23 |

The bean for these data would be:

|  |
| --- |
| public class Visitors {  @CsvBindByPosition(position = 0)  private String firstName;  @CsvBindByPosition(position = 1)  private String lastName;  @CsvBindByPosition(position = 2)  private int visitsToWebsite;  // Getters and setters go here.  } |

Besides that, the annotations behave the same as their header name counterparts.

##### Locales, dates, numbers

We’ve considered primitives, but we haven’t considered more complex yet common data types. We have also not considered locales other than the default locale or formatting options beyond those provided by a locale. Here we shall do all of this at the same time. Consider this input file:

|  |
| --- |
| username,valid since,annual salary  user1,01.01.2010,100.000€  user2,31.07.2014,50.000€ |

The dates are dd.MM.yyyy, the salaries use a dot as the thousands delimiter, and a currency symbol is in use. For this input we create the following bean:

|  |
| --- |
| public class Employees {  @CsvBindByName(required = true)  private String username;  @CsvBindByName(column = "valid since")  @CsvDate("dd.MM.yyyy")  private Date validSince;  @CsvBindByName(column = "annual salary", locale = "de-DE")  @CsvNumber("#.###¤")  private int salary;  // Getters and setters go here.  } |

The date is handled with the annotation @CsvDate in addition to the mapping annotation. @CsvDate can take a format string, and incidentally handles all common date-type classes. See the Javadocs for more details. The format of the salary, including thousands separator and currency symbol, are dealt with using a combination of the German locale, one of many countries where the thousands separator is a dot, and @CsvNumber.

##### Collection-based bean fields (one-to-many mappings)

CSV files are lists, right? Well, some people like lists within lists. For them, we have the ability to annotate bean fields that are declared to be some type implementing java.util.Collection. When using CsvBindAndSplitByName or CsvBindAndSplitByPosition, one field in the CSV file is taken to be a list of data that are separated by a delimiter of some kind. The input is split along this delimiter and the results are put in a Collection and assigned to the bean field. What kind of Collection? Any kind you want. If opencsv knows it, it instantiates an implementing class for you. If opencsv doesn’t know it, you can educate opencsv. Every reasonable Collection-based interface from the JDK is known, and well as Bag and SortedBag from Apache Commons Collections. Some examples would doubtless illuminate my meaning.

|  |
| --- |
| public class Student {  @CsvBindAndSplitByName(elementType = Float.class)  Collection<Float> testScores;  @CsvBindAndSplitByName(elementType = Double.class, collectionType = LinkedList.class)  List<? extends Number> quizScores;  @CsvBindAndSplitByName(elementType = Date.class, splitOn = ";+", writeDelimiter = ";")  @CsvDate("yyyy-MM-dd")  SortedSet<Date> tardies;  @CsvBindAndSplitByName(elementType= Teacher.class, splitOn = "\\|", converter = TextToTeacher.class)  List<Teacher> teachers;  @CsvBindByName  int studentID;  // Getters and setters go here |

This shows us much of the power of these annotations in a few lines. Let’s take the first field. It is defined to be a Collection of Floats. Note, please, the annotation @CsvBindAndSplitByName (or the equivalent for position) always requires the type of an element of the collection being created. Nothing else is mandatory. In particular, Collection itself has no directly implementing classes, but please note, we didn’t indicate to opencsv which kind of collection we want. opencsv chooses one for us.

The next field is a List of something derived from Number. This is where it becomes apparent why the element type is mandatory — it cannot always be determined. Besides that, in this line we are not satisfied with the List implementation opencsv chooses, so we specify LinkedList with the collectionType parameter to the annotation.

The third field is a SortedSet of dates (when a student was tardy to class). Sorted for convenience, and a set to avoid clerical errors of double entry. For this field we have specified that the string separating elements of this list in the input is one or more semicolons. This string is always interpreted as a regular expression. Interestingly, in case we write these data out to a CSV file later, the elements of the list should be separated with a single semicolon. Perhaps someone is trying to convert the data from a older format or remove redundancies.

The forth field is a list of teachers the student has. This field demonstrates the combination of collection-based fields and custom converters. The converter, which must be derived from AbstractCsvConverter, could look like this:

|  |
| --- |
| public class TextToTeacher extends AbstractCsvConverter {  @Override  public Object convertToRead(String value) {  Teacher t = new Teacher();  String[] split = value.split("\\.", 2);  t.setSalutation(split[0]);  t.setSurname(split[1]);  return t;  }  @Override  public String convertToWrite(Object value) {  Teacher t = (Teacher) value;  return String.format(""%s.%s", t.getSalutation(), t.getSurname());  }  } |

The corresponding data structure would be:

|  |
| --- |
| public class Teacher {  private String salutation;  private String surname;  // Getters and setters go here  } |

The final field is simply for student identification.

The input to be mapped to this bean could look like this:

|  |
| --- |
| studentID,testScores,quizScores,tardies,teachers  1,100.0 97.2 18.9,77 90.3 88.8,,Mr.Stone|Mrs.Mason  2,56.6 97.2 90.0,82.0 79.6 66.9,2017-01-02;2017-03-04;;;2017-03-04;;2017-05-31,Ms.Currie|Mr.Feynman |

The first student has never been tardy, so that list will be empty (but never null). The school secretary accidentally entered a tardy for the second student twice, but this will be eliminated by the SortedSet.

Let’s say you want to tell opencsv which Collection implementation to use, perhaps because you want to make certain it’s one that will perform better for your usage pattern, or perhaps because you want to use one opencsv knows nothing about, like your own implementation. There are two ways of doing this. We already saw one: specify the implementation you want to use in the annotation with the parameter "collectionType". The only stipulations on the implementing class are that it be public and have a nullary constructor. The other way is to declare the type of the bean field using the implementing class rather than the interface implemented, thus:

|  |
| --- |
| public class MySuperDuperIntegerList extends ArrayList<Integer> {  // Do something super duper.  }  public class DataClass {  @CsvBindAndSplitByName(elementType = Integer.class)  MySuperDuperIntegerList myList;  // Getter and setter go here  } |

Here, instead of declaring List<Integer> myList, we used the implementing class. opencsv will respect this and instantiate the class specified. That class can be parameterized, naturally (e.g. MySuperDuperList<Integer>).

All of the other features you know, love, and depend on, such as a field being required, or support for locales, is equally well supported for Collection-based members.

For details on which subinterfaces of Collection opencsv knows and exactly what implementation opencsv uses for those interfaces if you don’t specify one, see the Javadoc for the annotations CsvBindAndSplitByName or CsvBindAndSplitByPosition.

##### MultiValuedMap-based bean fields (many-to-one mappings)

If Collection-based bean fields were there to split one element into many, MultiValuedMap-based bean fields are there to consolidate many elements into one. What if you have the following input?

|  |
| --- |
| Album,Artist,Artist,Artist,Track1,Track2,Track3,Track4  We are the World,Michael Jackson,Lionel Richie,Stevie Wonder,We are the World,We are the World (instrumental),Did this album,Have any other tracks? |

The first difficulty you will encounter is that three columns have the same name. The second difficulty is that the number of tracks in the header might increase over time, but you want them all. Both problems are easily solved, as are all problems in the opencsv-world:

|  |
| --- |
| public class Album {  @CsvBindByName(column = "Album")  private String albumTitle;  @CsvBindAndJoinByName(column = "Artist", elementType = String.class)  private MultiValuedMap<String, String> artists;  @CsvBindAndJoinByName(column = "Track[0-9]+", elementType = String.class, mapType = HashSetValuedHashMap.class, required = true)  private MultiValuedMap<String, String> tracks;  // Getters and setters go here  } |

The first field is unimportant for this illustration.

The second field is a MultiValuedMap that collects all of the values under all of the columns with the name "Album". If you are not familiar with MultiValuedMap, it is a part of Apache Commons Collections. The first parameter is the index, and the second parameter is the value. In the case of CsvBindAndJoinByName, the index should always be a string. The value should be of a type to which the elementType from the annotation is assignable.

Why would we choose to use such a cumbersome data type as a MultiValuedMap to implement this feature? Why not a simple List and everyone is happy? Two reasons: First, someone will want to know what the header was actually named on reading, and second, opencsv needs to know what the header is named when it writes beans to a CSV file. And really, at least for reading, a MultiValuedMap isn’t that cumbersome: Mostly you will want a list of all values, not caring about which header they were under, and that can simply be had by calling values() on the field.

Back to our topic, the second field will be a MultiValuedMap with exactly one key: "Artist". Under this key, there will be a list with up to three entries, in this case "Michael Jackson", "Lionel Richie" and "Stevie Wonder". It only remains to note that the type of the elements being read must always be specified for the same reason it is necessary for Collection-based bean fields.

The third field sums up most of the rest of the features this annotation provides. As you can see, the definition of the column names is a regular expression. Naturally, the "column" attribute of CsvBindAndJoinByName is always interpreted as a regular expression. In this annotation we have also requested a specific implementation of MultiValuedMap, which opencsv will honor. We have decided that this field is mandatory, which in this case means that at least one matching header must be in the input, and every record must have a non-empty value for at least one of the matching columns. Given the input from above, this MultiValuedMap will have four entries, one for each column, and each of these entries will have a list of one element as its value. The elements will be the track titles.

All of the usual features apply: conversion locale, combination with CsvDate, custom converters as with collection-based fields, and specifying your own implementation of MultiValuedMap either through the annotation or by defining the field with the specific implementation (default implementations for the applicable interface are documented in the Javadoc for CsvBindAndJoinByName). The latter being said, if the MultiValuedMap is already present (and possibly contains values), say through the use of a constructor, it will not be overwritten, but rather added to.

What about precedence? To stay with our running example, what if after extending the number of track titles in the input significantly (which would require no changes to the bean), we hire some junior programmer who doesn’t get it, and he adds the following field to the bean:

|  |
| --- |
| @CsvBindByName(column = "Track21")  private String track21; |

What does opencsv do with this? It follows the general computing principle of "specific trumps general": It puts any information found under the header "Track21" into the new field, not the MultiValuedMap. Obviously this doesn’t exist for the sole purpose of creating mistakes; you can use it to your advantage if you want one otherwise matching column to be treated individually.

Since we’re on the topic of precedence, what happens if two regular expressions from CsvBindAndJoinByName match one and the same input header name? Don’t do this. The results are undefined.

While minding the last caveat, it is possible to use this feature to collect everything not otherwise mapped:

|  |
| --- |
| public class Demonstration {  @CsvBindByName(column = "index")  private String index;  @CsvBindAndJoinByName(column = ".\*", elementType = String.class)  private MultiValuedMap<String, String> theRest;  // Getters and setters go here  } |

There is another way one could possibly use this feature: Let’s say you get input of the same information from two different sources, and for reasons that are beyond your control, they have different header names. Perhaps they are in different languages. In one file, the header is:

|  |
| --- |
| studentID,given name,surname |

And in another file, it’s:

|  |
| --- |
| Schueler-ID,Vorname,Nachname |

You really don’t want two beans for the same thing. You can simply do this:

|  |
| --- |
| public class Student {  @CsvBindAndJoinByName(column = "(student|Schueler-)ID")  private MultiValuedMap<String, Integer> id;  @CsvBindAndJoinByName(column = "(given |Vor)name")  private MultiValuedMap<String, String> givenName;  @CsvBindAndJoinByName(column = "(sur|Nach)name")  private MultiValuedMap<String, String> surname;  // Getters and setters go here  } |

The only down side is, you will have to unpack the values with code like:

|  |
| --- |
| bean.getSurname().values().toArray(new String[1])[0]; |

But wait! That’s not all! Using CsvBindAndJoinByPosition we can do the same thing with input that does not include headers. Let’s just say for the sake of argument that our album example from earlier now no longer includes headers, and that the structure grew over time. Perhaps the first version of the CSV file only included one artist, and the other two fields for artist were added at two different points in time after that. The tracks grew over time as well. So now our input looks like this:

|  |
| --- |
| We are the World,Michael Jackson,We are the World,We are the World (instrumental),Lionel Richie,Did this album,Stevie Wonder,Have any other tracks? |

In other words, first the album name, then the first artist, followed by two tracks, then the second artist followed by one more track, then the third artist again followed by one track. The bean for these data would look like this:

|  |
| --- |
| public class Album {  @CsvBindByPosition(position = 0)  private String albumName;  @CsvBindAndJoinByPosition(position = "1,4,6", elementType = String.class)  MultiValuedMap<Integer, String> artists;  @CsvBindAndJoinByPosition(position = "2-3,5,7-", elementType = String.class)  MultiValuedMap<Integer, String> tracks;  // Getters and setters go here  } |

The first thing to notice in this example is that we have used CsvBindAndJoinByPosition, which takes a list of zero-based column numbers and ranges as its most important argument. The list is comma-separated, and can include any number of column indices as well as closed (e.g. "3-5") and half-open (e.g. "-5" or "10-") ranges.

The next thing to notice in this example is that for CsvBindAndJoinByPosition, the index type to MultiValuedMap must be Integer. Values are saved under the index of the column position they were found in.

The last thing to notice is that as long as new column positions are added to the end of the file and these are all new tracks, they will all be placed in the variable "tracks" because the column position definition from the CsvBindAndJoinByPosition annotation defines an open range starting at index 7.

As with a header-based mapping, it is possible to create a mop-up field, if no other fields are mapped with CsvBindAndJoinByPosition, by mapping to a MultiValuedMap using the fully open range expression "-".

Writing with CsvBindAndJoinByName and CsvBindAndJoinByPosition are slightly more complicated. Both include ambiguous information about the source of the data, one in the form of regular expressions, and the other in the form of ranges. Once the data have been read in, there is no way from this information alone to determine which column each header came from. That, as we have already said, is why we use a MultiValuedMap: the index gives us this vital information. That said, it should be obvious that when writing, the MultiValuedMap must be completely filled out for every bean before sending it off to be written. That is, every index that is expected in the output must be present in the map and have at least a null value.

##### Custom converters

Now, we know that input data can get very messy, so we have provided our users with the ability to deal with the messiest of data by allowing you to define your own custom converters. The custom converters here are used at the level of the entire field, not like the custom converters previously covered in collection-based and MultiValuedMap-based bean fields. Every converter must be derived from AbstractBeanField, must be public, and must have a public nullary constructor. For reading, the convert() method must be overridden. opencsv provides two custom converters in the package com.opencsv.bean.customconverter. These can be useful converters themselves, but they also exist for instructive purposes: If you want to write your own custom converter, look at these for examples of how it’s done.

Let’s use two as illustrations. Let’s say we have the following input file:

|  |
| --- |
| cluster,nodes,production  cluster1,node1 node2,wahr  cluster2,node3 node4 node5,falsch |

In this file we have a list of server clusters. The cluster name comes first, followed by a space-delimited list of names of servers in the cluster. The final field indicates whether the cluster is in production use or not, but the truth value uses German. Here is the appropriate bean, using the custom converters opencsv provides:

|  |
| --- |
| public class Cluster {  @CsvBindByName  private String cluster;  @CsvCustomBindByName(converter = ConvertSplitOnWhitespace.class)  private String[] nodes;  @CsvCustomBindByName(converter = ConvertGermanToBoolean.class)  private boolean production;  // Getters and setters go here.  } |

More than that is not necessary. If you need boolean values in other languages, take a gander at the code in ConvertGermanToBoolean; Apache BeanUtils provides a slick way of converting booleans.

The corresponding annotations for custom converters based on column position are also provided.

#### Reading into beans without annotations 不用注解，设定表头

If annotations are anathema to you, you can bypass them with carefully structured data, beans and with somewhat more code. For example, here’s how you can map to a bean based on the field positions in your CSV file:

|  |
| --- |
| ColumnPositionMappingStrategy strat = new ColumnPositionMappingStrategy();  strat.setType(YourOrderBean.class);  String[] columns = new String[] {"name", "orderNumber", "id"}; // the fields to bind to in your bean  strat.setColumnMapping(columns);  CsvToBean csv = new CsvToBean();  List list = csv.parse(strat, yourReader); |

Please note, if you do not use annotations, opencsv uses introspection to access member variables, so your objects will have to be honest-to-God beans.

#### Skipping, filtering and verifying 跳过前面的行

With some input it can be helpful to skip the first few lines. opencsv provides for this need with CsvToBeanBuilder.withSkipLines(), which ultimately is used on the appropriate constructor for CSVReader, if you would prefer to do everything without the use of the builders. This will skip the first few lines of the raw input, not the CSV data, in case some input provides heaven knows what before the first line of CSV data, such as a legal disclaimer or copyright information.

So, for example, you can skip the first two lines by doing:

|  |
| --- |
| CSVReader reader = new CSVReader(new FileReader("yourfile.csv"), '\t', '\'', 2); |

or for reading with annotations:

|  |
| --- |
| CsvToBean csvToBean = new CsvToBeanBuilder(new FileReader("yourfile.csv"))  .withSeparator('\t').withQuoteChar('\'').withSkipLines(2).build(); |

Filtering is different in that it works on CSV records and it applies to the whole input. It can also only be used with a bean mapping strategy. To filter input beans, implement CsvToBeanFilter and pass your implementation to CsvToBeanBuilder.withFilter(), or equivalently if you’re not using the builders, to the appropriate parse() method from CsvToBean or even setFilter().

Verifying is slightly different still. With verifying, a complete finished bean is checked for desirability and consistency. By implementing BeanFilter and passing it to CsvToBeanBuilder.withVerifier(), each bean will be vetted before being returned to the calling code. Beans can be silently filtered if they are simply undesirable data sets, or if the data are inconsistent and this is considered an error for the surrounding logic, CsvConstraintViolationException may be thrown.

## Writing

Less often used, but just as comfortable as reading CSV files is writing them. And believe me, a lot of work went into making writing CSV files as comfortable as possible for you, our users.

There are three methods of writing CSV data: \* Writing from an array of strings \* Writing from a list of beans \* Writing from an SQL ResultSet

### Writing from an array of strings

CSVWriter follows the same semantics as the CSVReader. For example, to write a tab-separated file:

|  |
| --- |
| CSVWriter writer = new CSVWriter(new FileWriter("yourfile.csv"), '\t');  // feed in your array (or convert your data to an array)  String[] entries = "first#second#third".split("#");  writer.writeNext(entries);  writer.close(); |

If you’d prefer to use your own quote characters, you may use the three argument version of the constructor, which takes a quote character (or feel free to pass in CSVWriter.NO\_QUOTE\_CHARACTER).

You can also customize the line terminators used in the generated file (which is handy when you’re exporting from your Linux web application to Windows clients). There is a constructor argument for this purpose.

### Writing from a list of beans

The easiest way to write CSV files will in most cases be StatefulBeanToCsv, which is simplest to create with StatefulBeanToCsvBuilder, and which is thus named because there used to be a BeanToCsv. Thankfully, no more.

|  |
| --- |
| // List<MyBean> beans comes from somewhere earlier in your code.  Writer writer = new FileWriter("yourfile.csv");  StatefulBeanToCsv beanToCsv = new StatefulBeanToCsvBuilder(writer).build();  beanToCsv.write(beans);  writer.close(); |

Notice, please, we did not tell opencsv what kind of bean we are writing or what mapping strategy is to be used. opencsv determines these things automatically. Annotations are not even strictly necessary: if there are no annotations, opencsv assumes you want to write the whole bean using the header name mapping strategy and uses the field names as the column headers. Naturally, the mapping strategy can be dictated, if necessary, through StatefulBeanToCsvBuilder.withMappingStrategy(), or the constructor for StatefulBeanToCsv.

Just as we can use the "capture" option to the binding annotations, if you use annotations on writing, you can use the "format" option to dictate how the field should be formatted if simply writing the bean field value is not enough. Please see the Javadoc for the annotations for details.

Just as in reading into beans, there is a performance trade-off while writing that is left in your hands: ordered vs. unordered data. If the order of the data written to the output and the order of any exceptions captured during processing do not matter to you, use StatefulBeanToCsv.withOrderedResults(false) to obtain slightly better performance.

#### Changing the write order 改变写的顺序

If you do nothing, the order of the columns on writing will be ascending according to position for column index-based mappings, and ascending according to name for header name-based mappings. You can change this order, if you must.

|  |
| --- |
| // List<MyBean> beans comes from somewhere earlier in your code.  Writer writer = new FileWriter("yourfile.csv");  HeaderColumnNameMappingStrategy<MyBean> strategy = new HeaderColumnNameMappingStrategy<>();  strategy.setType(MyBean.class);  strategy.setColumnOrderOnWrite(new MyComparator());  StatefulBeanToCsv beanToCsv = StatefulBeanToCsvBuilder(writer)  .withMappingStrategy(strategy)  .build();  beanToCsv.write(beans);  writer.close(); |

The same method exists for ColumnPositionMappingStrategy. If you wish to use your own ordering, you must instantiate your own mapping strategy and pass it in to StatefulBeanToCsvBuilder.

We expect there will be plenty of people who find using a Comparator uncomfortable, because they have an exact order that they need that has nothing to do with any kind of rule-based ordering. For these people we have included com.opencsv.bean.comparator.LiteralComparator. It is instantiated with an array of strings for header name mapping or integers for column position mapping that define the order desired.

### From a database table

Here’s a nifty little trick for those of you out there who often work directly with databases and want to write the results of a query directly to a CSV file. Sean Sullivan added a neat feature to CSVWriter so you can pass writeAll() a ResultSet from an SQL query.

|  |
| --- |
| java.sql.ResultSet myResultSet = . . .  writer.writeAll(myResultSet, includeHeaders); |

## Nuts and bolts

Now we start to poke around under the hood of opencsv.

### Flow of data through opencsv

We have tried to hide all of the classes and how they work together in opencsv by providing you with builders, since you will rarely need to know all the details of opencsv’s internal workings. But for those blessed few, here is how all of the pieces fit together for reading:

1. You must provide a Reader. This can be any Reader, but a FileReader or StringReader are the most common options.
2. If you wish, you may provide a parser (anything implementing ICSVParser).
3. The Reader can be wrapped in a CSVReader, which is also given the parser, if you have used your own. Otherwise, opencsv creates its own parser and even its own CSVReader. If you are reading into an array of strings, this is where the trail ends.
4. For those reading into beans, a MappingStrategy is the next step.
5. If you want filtering, you can create a CsvToBeanFilter or a BeanVerifier.
6. The MappingStrategy and the Reader or CSVReader and optionally the CsvToBeanFilter or BeanVerifier are passed to a CsvToBean, which uses them to parse input and populate beans.
7. If you have any custom converters, they are called for each bean field as CsvToBean is populating the bean fields.

For writing it’s a little simpler:

1. You must provide a Writer. This can be any Writer, but a FileWriter or a StringWriter are the most common options.
2. The Writer is wrapped in a CSVWriter. This is always done for you.
3. Create a MappingStrategy if you need to. Otherwise opencsv will automatically determine one.
4. Create a StatefulBeanToCsv, give it the MappingStrategy and the Writer.
5. If you have any custom converters, they are called for each bean field as the field is written out to the CSV file.

### Mapping strategies

Opencsv has the concept of a mapping strategy. This is what translates a column from an input file into a field in a bean or vice versa. As we have already implied in the documentation of the annotations, there are two basic mapping strategies: Mapping by header name and mapping by column position. These are incarnated in HeaderColumnNameMappingStrategy and ColumnPositionMappingStrategy respectively. There is one more addendum to the header name mapping strategy: If you need to translate names from the input file to field names and you are not using annotations, you will need to use HeaderColumnNameTranslateMappingStrategy.

If you use annotations and CsvToBeanBuilder (for reading) or StatefulBeanToCsv(Builder) (for writing), an appropriate mapping strategy is automatically determined, and you need worry about nothing else.

Naturally, you can implement your own mapping strategies as you see fit. Your mapping strategy must implement the interface MappingStrategy, but has no other requirement. Feel free to derive a class from the existing implementations for simplicity.

If you have implemented your own mapping strategy, or if you need to override the automatic selection of a mapping strategy, for example if you are reading the same bean with one mapping strategy, but writing it with a different one for conversion purposes, you need to let opencsv know which mapping strategy it must use. For reading, this is accomplished by passing an instance of your mapping strategy to CsvToBeanBuilder.withMappingStrategy(). For writing, pass your strategy to StatefulBeanToCsvBuilder.withMappingStrategy().

# Frequently Asked Questions

## Where can I get it?

Source and binaries are available from [SourceForge](http://sourceforge.net/projects/opencsv/)

## Can I use opencsv in my commercial applications?

Yes. opencsv is available under a commercial-friendly Apache 2.0 license. You are free to include it in your commericial applications without any fee or charge, and you are free to modify it to suit your circumstances. To find out more details of the license, read the [Apache 2.0 license agreement](http://www.apache.org/licenses/LICENSE-2.0)

## Can I get the source? More example code?

You can view the source from the [opencsv source section](http://sourceforge.net/p/opencsv/source/ci/master/tree/). The source section also gives you the URL to the git repository so you can download source code. There is also a sample addressbook CSV reader in the /examples directory. And for extra marks, there’s a JUnit test suite in the /test directory.

## How can I use it in my Maven projects?

Add a dependency element to your pom:

|  |
| --- |
| <dependency>  <groupId>com.opencsv</groupId>  <artifactId>opencsv</artifactId>  <version>4.5</version>  </dependency> |

## Who maintains opencsv?

* opencsv was developed in a couple of hours by Glen Smith but has since passed the torch and moved on to other projects. You can read his [blog](http://blogs.bytecode.com.au/glen) for more info and contact details.
* Scott Conway - co-maintainer of project. Commits too numerous to mention here.
* Andrew Rucker Jones - co-maintainer of project. Expanded on the annotation work done by Tom Squires and put some extra polish on the documentation.
* Sean Sullivan contributed work and was maintainer for a time.
* Kyle Miller contributed the bean binding work.
* Tom Squires has expanded on the bean work done by Kyle Miller to add annotations.
* Maciek Opala contributed alot of his time modernizing opencsv. He moved the repository to git and fixed several issues.
* J.C. Romanda contributed several fixes.

## How do I report issues?

You can report issues on the [support page](http://sourceforge.net/projects/opencsv/support) at Sourceforge. Please post a sample file that demonstrates your issue. For bonus marks, post a patch too. :-)

## What are the "gotchas"?

We maintain a separate page of issues/questions/resolutions on our [sourceforge wiki](https://sourceforge.net/p/opencsv/wiki/FAQ/) to enable us to make changes without a release.