**Apache Avro™ 1.8.2 Documentation**

* [Introduction](http://avro.apache.org/docs/current/#intro)
* [Schemas](http://avro.apache.org/docs/current/#schemas)
* [Comparison with other systems](http://avro.apache.org/docs/current/#compare)

**Introduction**

Apache Avro™ is a data serialization system（me：数据序列化）.

Avro provides:

* Rich data structures.
* A compact, fast, binary data format.
* A container file, to store persistent data.
* Remote procedure call (RPC)（me:avor支持rpc远程调用的数据交互）.
* Simple integration（整合） with dynamic languages（me：与动态语言简单的整合，即与像PHP、python等动态语言的简单整合）. Code generation is not required to read or write data files nor to use or implement RPC protocols（使用avro代码生成功能不需要如何写，也不需要实现RPC协议）. Code generation as an optional optimization, only worth implementing for statically typed languages（代码生成功能只是一个可选的功能，只对静态类型的语言如java/c++/c#等有价值）.

**Schemas**

Avro relies on *schemas(avro依赖于schema)*. When Avro data is read, the schema used when writing it is always present. This permits each datum(数据) to be written with no per-value overheads（日常费用，开销，即写数据/序列化的成本）, making serialization both fast and small. This also facilitates（促进，推进） use with dynamic, scripting languages, since data, together with its schema, is fully self-describing（完全自描述地）.

When Avro data is stored in a file, its schema is stored with it, so that files may be processed later by any program. If the program reading the data expects a different schema this can be easily resolved, since both schemas are present.

Schmea很重要，无论读和写数据（即序列化和反序列化数据都参照schema来操作），写数据的时候会同时将schema一同写入文件等序列化载体以便被之后的程序进一步处理，读取的时候需要提供schema。

When Avro is used in RPC, the client and server exchange schemas in the connection handshake. (This can be optimized so that, for most calls, no schemas are actually transmitted.) Since both client and server both have the other's full schema, correspondence between same named fields, missing fields, extra fields, etc. can all be easily resolved.

当avro被用于RPC时，client和server在handshake连接阶段会交换schemas（这个过程是被优化的，因为对于大多数的调用，实际上并没有真是的schema传递）。因为client和server都持有了另一方的完全的schema，对于同样的字段名，丢失了字段名以及额外的字段名等等都是能够被很容易的处理的。。

Avro schemas are defined with [JSON](http://www.json.org/) . This facilitates implementation in languages that already have JSON libraries.

Avro是由json定义的

**Comparison with other systems**

Avro provides functionality similar to systems such as [Thrift](http://thrift.apache.org/), [Protocol Buffers](http://code.google.com/p/protobuf/), etc. Avro differs from these systems in the following fundamental aspects（在以下几个方面avro不同于其他的系统）.

* *Dynamic typing（动态类型）*: Avro does not require that code be generated（avro并不需要生成代码）. Data is always accompanied（相伴） by a schema that permits full processing（完全处理） of that data without code generation, static datatypes, etc. This facilitates（促进了） construction of generic data-processing systems and languages.
* *Untagged data（没有标志位数据）*: Since the schema is present when data is read, considerably less type information need be encoded with data, resulting in smaller serialization size（由于在读取数据的时候需要提供schema，需要相当小的类型信息（如类信息）被一同编码进数据（即将schema一同编码如数据），所以序列化后的数据很小）.
* *No manually-assigned field IDs（不用手动生成id字段，序列化类需要id字段）*: When a schema changes, both the old and new schema are always present when processing data, so differences may be resolved symbolically, using field names（当schema改变时，旧的和新的schema会被处理以便得出一个统一的解决）.

*Apache Avro, Avro, Apache, and the Avro and Apache logos are trademarks of The Apache Software Foundation.*

# Apache Avro™ 1.8.2 Getting Started (Java)

* [Download](http://avro.apache.org/docs/current/gettingstartedjava.html#download_install)
* [Defining a schema](http://avro.apache.org/docs/current/gettingstartedjava.html#Defining+a+schema)
* [Serializing and deserializing with code generation](http://avro.apache.org/docs/current/gettingstartedjava.html#Serializing+and+deserializing+with+code+generation)
  + [Compiling the schema](http://avro.apache.org/docs/current/gettingstartedjava.html#Compiling+the+schema)
  + [Creating Users](http://avro.apache.org/docs/current/gettingstartedjava.html#Creating+Users)
  + [Serializing](http://avro.apache.org/docs/current/gettingstartedjava.html#Serializing)
  + [Deserializing](http://avro.apache.org/docs/current/gettingstartedjava.html#Deserializing)
  + [Compiling and running the example code](http://avro.apache.org/docs/current/gettingstartedjava.html#Compiling+and+running+the+example+code)
* [Serializing and deserializing without code generation](http://avro.apache.org/docs/current/gettingstartedjava.html#Serializing+and+deserializing+without+code+generation)
  + [Creating users](http://avro.apache.org/docs/current/gettingstartedjava.html#Creating+users)
  + [Serializing](http://avro.apache.org/docs/current/gettingstartedjava.html#Serializing-N101DD)
  + [Deserializing](http://avro.apache.org/docs/current/gettingstartedjava.html#Deserializing-N10206)
  + [Compiling and running the example code](http://avro.apache.org/docs/current/gettingstartedjava.html#Compiling+and+running+the+example+code-N10248)

This is a short guide for getting started with Apache Avro™ using Java. This guide only covers using Avro for data serialization; see Patrick Hunt's [Avro RPC Quick Start](https://github.com/phunt/avro-rpc-quickstart) for a good introduction to using Avro for RPC.

## Download

Avro implementations（avro的其他语言实现） for C, C++, C#, Java, PHP, Python, and Ruby can be downloaded from the [Apache Avro™ Releases](http://avro.apache.org/releases.html) page. This guide uses Avro 1.8.2, the latest version at the time of writing. For the examples in this guide, download avro-1.8.2.jar and avro-tools-1.8.2.jar. The Avro Java implementation also depends on the [Jackson](http://jackson.codehaus.org/) JSON library（使用了JSON库jackon）. From the Jackson [download page](http://wiki.fasterxml.com/JacksonDownload), download the core-asl and mapper-asl jars. Add avro-1.8.2.jar and the Jackson jars to your project's classpath (avro-tools will be used for code generation(avro-tools是被用来自动代码生成的，如果不需要此功能则可以不加载此jar包).

Alternatively, if you are using Maven, add the following dependency to your POM:

<dependency>

<groupId>org.apache.avro</groupId>

<artifactId>avro</artifactId>

<version>1.8.2</version>

</dependency>

As well as the Avro Maven plugin (for performing code generation):

<plugin>

<groupId>org.apache.avro</groupId>

<artifactId>avro-maven-plugin</artifactId>

<version>1.8.2</version>

<executions>

<execution>

<phase>generate-sources</phase>

<goals>

<goal>schema</goal>

</goals>

<configuration>

<sourceDirectory>${project.basedir}/src/main/avro/</sourceDirectory>

<outputDirectory>${project.basedir}/src/main/java/</outputDirectory>

</configuration>

</execution>

</executions>

</plugin>

<plugin>

<groupId>org.apache.maven.plugins</groupId>

<artifactId>maven-compiler-plugin</artifactId>

<configuration>

<source>1.6</source>

<target>1.6</target>

</configuration>

</plugin>

You may also build the required Avro jars from source. Building Avro is beyond the scope of this guide; see the [Build Documentation](https://cwiki.apache.org/AVRO/Build+Documentation) page in the wiki for more information.

## Defining a schema

Avro schemas are defined using JSON. Schemas are composed of [primitive types](http://avro.apache.org/docs/current/spec.html#schema_primitive) (null, boolean, int, long, float, double, bytes, and string) and [complex types](http://avro.apache.org/docs/current/spec.html#schema_complex) (record, enum, array, map, union, and fixed). You can learn more about Avro schemas and types from the specification, but for now let's start with a simple schema example, user.avsc:

{"namespace": "example.avro",

"type": "record",

"name": "User",

"fields": [

{"name": "name", "type": "string"},

{"name": "favorite\_number", "type": ["int", "null"]},

{"name": "favorite\_color", "type": ["string", "null"]}

]

}

This schema defines a record representing a hypothetical user. (Note that a schema file can only contain a single schema definition.) At minimum, a record definition must include its type ("type": "record"), a name ("name": "User"), and fields, in this case name, favorite\_number, and favorite\_color. We also define a namespace ("namespace": "example.avro"), which together with the name attribute defines the "full name" of the schema (example.avro.User in this case).

Fields are defined via an array of objects, each of which defines a name and type (other attributes are optional, see the [record specification](http://avro.apache.org/docs/current/spec.html#schema_record) for more details). The type attribute of a field is another schema object, which can be either a primitive or complex type. For example, the name field of our User schema is the primitive type string, whereas the favorite\_number and favorite\_color fields are both unions, represented by JSON arrays. unions are a complex type that can be any of the types listed in the array; e.g., favorite\_number can either be an int or null, essentially making it an optional field.

## Serializing and deserializing with code generation

### Compiling the schema

Code generation allows us to automatically create classes based on our previously-defined schema. Once we have defined the relevant classes, there is no need to use the schema directly in our programs. We use the avro-tools jar to generate code as follows:

java -jar /path/to/avro-tools-1.8.2.jar compile schema <schema file> <destination>

This will generate the appropriate source files in a package based on the schema's namespace in the provided destination folder. For instance, to generate a User class in package example.avro from the schema defined above, run

java -jar /path/to/avro-tools-1.8.2.jar compile schema user.avsc .

Note that if you using the Avro Maven plugin, there is no need to manually invoke the schema compiler; the plugin automatically performs code generation on any .avsc files present in the configured source directory.

### Creating Users

Now that we've completed the code generation, let's create some Users, serialize them to a data file on disk, and then read back the file and deserialize the User objects.

First let's create some Users and set their fields.

User user1 = new User();

user1.setName("Alyssa");

user1.setFavoriteNumber(256);

// Leave favorite color null

// Alternate constructor

User user2 = new User("Ben", 7, "red");

// Construct via builder

User user3 = User.newBuilder()

.setName("Charlie")

.setFavoriteColor("blue")

.setFavoriteNumber(null)

.build();

As shown in this example, Avro objects can be created either by invoking a constructor directly or by using a builder. Unlike constructors, builders will automatically set any default values specified in the schema. Additionally, builders validate the data as it set, whereas objects constructed directly will not cause an error until the object is serialized. However, using constructors directly generally offers better performance, as builders create a copy of the datastructure before it is written.

Note that we do not set user1's favorite color. Since that record is of type ["string", "null"], we can either set it to a string or leave it null; it is essentially optional. Similarly, we set user3's favorite number to null (using a builder requires setting all fields, even if they are null).

### Serializing

Now let's serialize our Users to disk.

// Serialize user1, user2 and user3 to disk

DatumWriter<User> userDatumWriter = new SpecificDatumWriter<User>(User.class);

DataFileWriter<User> dataFileWriter = new DataFileWriter<User>(userDatumWriter);

dataFileWriter.create(user1.getSchema(), new File("users.avro"));

dataFileWriter.append(user1);

dataFileWriter.append(user2);

dataFileWriter.append(user3);

dataFileWriter.close();

We create a DatumWriter, which converts Java objects into an in-memory serialized format. The SpecificDatumWriter class is used with generated classes and extracts the schema from the specified generated type.

Next we create a DataFileWriter, which writes the serialized records, as well as the schema, to the file specified in the dataFileWriter.create call. We write our users to the file via calls to the dataFileWriter.append method. When we are done writing, we close the data file.

### Deserializing

Finally, let's deserialize the data file we just created.

// Deserialize Users from disk

DatumReader<User> userDatumReader = new SpecificDatumReader<User>(User.class);

DataFileReader<User> dataFileReader = new DataFileReader<User>(file, userDatumReader);

User user = null;

while (dataFileReader.hasNext()) {

// Reuse user object by passing it to next(). This saves us from

// allocating and garbage collecting many objects for files with

// many items.

user = dataFileReader.next(user);

System.out.println(user);

}

This snippet will output:

{"name": "Alyssa", "favorite\_number": 256, "favorite\_color": null}

{"name": "Ben", "favorite\_number": 7, "favorite\_color": "red"}

{"name": "Charlie", "favorite\_number": null, "favorite\_color": "blue"}

Deserializing is very similar to serializing. We create a SpecificDatumReader, analogous to the SpecificDatumWriter we used in serialization, which converts in-memory serialized items into instances of our generated class, in this case User. We pass theDatumReader and the previously created File to a DataFileReader, analogous to the DataFileWriter, which reads the data file on disk.

Next we use the DataFileReader to iterate through the serialized Users and print the deserialized object to stdout. Note how we perform the iteration: we create a single User object which we store the current deserialized user in, and pass this record object to every call of dataFileReader.next. This is a performance optimization that allows the DataFileReader to reuse the same User object rather than allocating a new User for every iteration, which can be very expensive in terms of object allocation and garbage collection if we deserialize a large data file. While this technique is the standard way to iterate through a data file, it's also possible to use for (User user : dataFileReader) if performance is not a concern.

### Compiling and running the example code

This example code is included as a Maven project in the examples/java-example directory in the Avro docs. From this directory, execute the following commands to build and run the example:

$ mvn compile # includes code generation via Avro Maven plugin

$ mvn -q exec:java -Dexec.mainClass=example.SpecificMain

## Serializing and deserializing without code generation

Data in Avro is always stored with its corresponding schema, meaning we can always read a serialized item regardless of whether we know the schema ahead of time. This allows us to perform serialization and deserialization without code generation.

Let's go over the same example as in the previous section, but without using code generation: we'll create some users, serialize them to a data file on disk, and then read back the file and deserialize the users objects.

### Creating users

First, we use a Parser to read our schema definition and create a Schema object.

Schema schema = new Schema.Parser().parse(new File("user.avsc"));

Using this schema, let's create some users.

GenericRecord user1 = new GenericData.Record(schema);

user1.put("name", "Alyssa");

user1.put("favorite\_number", 256);

// Leave favorite color null

GenericRecord user2 = new GenericData.Record(schema);

user2.put("name", "Ben");

user2.put("favorite\_number", 7);

user2.put("favorite\_color", "red");

Since we're not using code generation, we use GenericRecords to represent users. GenericRecord uses the schema to verify that we only specify valid fields. If we try to set a non-existent field (e.g., user1.put("favorite\_animal", "cat")), we'll get anAvroRuntimeException when we run the program.

Note that we do not set user1's favorite color. Since that record is of type ["string", "null"], we can either set it to a string or leave it null; it is essentially optional.

### Serializing

Now that we've created our user objects, serializing and deserializing them is almost identical to the example above which uses code generation. The main difference is that we use generic instead of specific readers and writers.

First we'll serialize our users to a data file on disk.

// Serialize user1 and user2 to disk

File file = new File("users.avro");

DatumWriter<GenericRecord> datumWriter = new GenericDatumWriter<GenericRecord>(schema);

DataFileWriter<GenericRecord> dataFileWriter = new DataFileWriter<GenericRecord>(datumWriter);

dataFileWriter.create(schema, file);

dataFileWriter.append(user1);

dataFileWriter.append(user2);

dataFileWriter.close();

We create a DatumWriter, which converts Java objects into an in-memory serialized format. Since we are not using code generation, we create a GenericDatumWriter. It requires the schema both to determine how to write the GenericRecords and to verify that all non-nullable fields are present.

As in the code generation example, we also create a DataFileWriter, which writes the serialized records, as well as the schema, to the file specified in the dataFileWriter.create call. We write our users to the file via calls to the dataFileWriter.appendmethod. When we are done writing, we close the data file.

### Deserializing

Finally, we'll deserialize the data file we just created.

// Deserialize users from disk

DatumReader<GenericRecord> datumReader = new GenericDatumReader<GenericRecord>(schema);

DataFileReader<GenericRecord> dataFileReader = new DataFileReader<GenericRecord>(file, datumReader);

GenericRecord user = null;

while (dataFileReader.hasNext()) {

// Reuse user object by passing it to next(). This saves us from

// allocating and garbage collecting many objects for files with

// many items.

user = dataFileReader.next(user);

System.out.println(user);

This outputs:

{"name": "Alyssa", "favorite\_number": 256, "favorite\_color": null}

{"name": "Ben", "favorite\_number": 7, "favorite\_color": "red"}

Deserializing is very similar to serializing. We create a GenericDatumReader, analogous to the GenericDatumWriter we used in serialization, which converts in-memory serialized items into GenericRecords. We pass the DatumReader and the previously created File to a DataFileReader, analogous to the DataFileWriter, which reads the data file on disk.

Next, we use the DataFileReader to iterate through the serialized users and print the deserialized object to stdout. Note how we perform the iteration: we create a single GenericRecord object which we store the current deserialized user in, and pass this record object to every call of dataFileReader.next. This is a performance optimization that allows the DataFileReader to reuse the same record object rather than allocating a new GenericRecord for every iteration, which can be very expensive in terms of object allocation and garbage collection if we deserialize a large data file. While this technique is the standard way to iterate through a data file, it's also possible to use for (GenericRecord user : dataFileReader) if performance is not a concern.

### Compiling and running the example code

This example code is included as a Maven project in the examples/java-example directory in the Avro docs. From this directory, execute the following commands to build and run the example:

$ mvn compile

$ mvn -q exec:java -Dexec.mainClass=example.GenericMain