## The ZIP package

The ZIP package provides features not found in java.util.zip:

* Support for encodings other than UTF-8 for filenames and comments. Starting with Java7 this is supported by java.util.zip as well.
* Access to internal and external attributes (which are used to store Unix permission by some zip implementations).
* Structured support for extra fields.

In addition to the information stored in ArchiveEntry a ZipArchiveEntry stores internal and external attributes as well as extra fields which may contain information like Unix permissions, information about the platform they've been created on, their last modification time and an optional comment.

### ZipArchiveInputStream vs ZipFile

ZIP archives store a archive entries in sequence and contain a registry of all entries at the very end of the archive. It is acceptable for an archive to contain several entries of the same name and have the registry (called the central directory) decide which entry is actually to be used (if any).

In addition the ZIP format stores certain information only inside the central directory but not together with the entry itself, this is:

* internal and external attributes
* different or additional extra fields

This means the ZIP format cannot really be parsed correctly while reading a non-seekable stream, which is what ZipArchiveInputStream is forced to do. As a result ZipArchiveInputStream

* may return entries that are not part of the central directory at all and shouldn't be considered part of the archive.
* may return several entries with the same name.
* will not return internal or external attributes.
* may return incomplete extra field data.
* may return unknown sizes and CRC values for entries until the next entry has been reached if the archive uses the data descriptor feature (see below).
* can not skip over bytes that occur before the real zip stream. This means self-extracting zips as they are created by some tools can not be read using ZipArchiveInputStream at all. This also applies to Chrome extension archives, for example.

ZipArchiveInputStream shares these limitations with java.util.zip.ZipInputStream.

ZipFile is able to read the central directory first and provide correct and complete information on any ZIP archive.

ZIP archives know a feature called the data descriptor which is a way to store an entry's length after the entry's data. This can only work reliably if the size information can be taken from the central directory or the data itself can signal it is complete, which is true for data that is compressed using the DEFLATED compression algorithm.

ZipFile has access to the central directory and can extract entries using the data descriptor reliably. The same is true for ZipArchiveInputStream as long as the entry is DEFLATED. For STORED entries ZipArchiveInputStream can try to read ahead until it finds the next entry, but this approach is not safe and has to be enabled by a constructor argument explicitly. For example it will completely fail if the stored entry is a ZIP archive itself. Starting with Compress 1.19 ZipArchiveInputStream will perform a few sanity for STORED entries with data descriptors and throw an exception if they fail.

If possible, you should **always** prefer ZipFile over ZipArchiveInputStream.

ZipFile requires a SeekableByteChannel that will be obtained transparently when reading from a file. The class org.apache.commons.compress.utils.SeekableInMemoryByteChannel allows you to read from an in-memory archive.

### ZipArchiveOutputStream

ZipArchiveOutputStream has three constructors, one of them uses a File argument, one a SeekableByteChannel and the last uses an OutputStream. The File version will try to use SeekableByteChannel and fall back to using a FileOutputStream internally if that fails.

If ZipArchiveOutputStream can use SeekableByteChannel it can employ some optimizations that lead to smaller archives. It also makes it possible to add uncompressed (setMethod used with STORED) entries of unknown size when calling putArchiveEntry - this is not allowed if ZipArchiveOutputStream has to use an OutputStream.

If you know you are writing to a file, you should always prefer the File- or SeekableByteChannel-arg constructors. The class org.apache.commons.compress.utils.SeekableInMemoryByteChannel allows you to write to an in-memory archive.

### Extra Fields

Inside a ZIP archive, additional data can be attached to each entry. The java.util.zip.ZipEntry class provides access to this via the get/setExtra methods as arrays of bytes.

Actually the extra data is supposed to be more structured than that and Compress' ZIP package provides access to the structured data as ZipExtraField instances. Only a subset of all defined extra field formats is supported by the package, any other extra field will be stored as UnrecognizedExtraField.

Prior to version 1.1 of this library trying to read an archive with extra fields that didn't follow the recommended structure for those fields would cause Compress to throw an exception. Starting with version 1.1 these extra fields will now be read as UnparseableExtraFieldData.

Prior to version 1.19 of this library trying to read an archive with extra fields that Compress expects to understand but that used a different content than expected would cause Compress to throw an exception. Starting with version 1.19 these extra fields will now be read as UnrecognizedExtraField. Using ZipArchiveEntry.getExtraFields(ExtraFieldParsingBehavior) you have a more fine grained control over the parser.

### Encoding

Traditionally the ZIP archive format uses CodePage 437 as encoding for file name, which is not sufficient for many international character sets.

Over time different archivers have chosen different ways to work around the limitation - the java.util.zip packages simply uses UTF-8 as its encoding for example.

Ant has been offering the encoding attribute of the zip and unzip task as a way to explicitly specify the encoding to use (or expect) since Ant 1.4. It defaults to the platform's default encoding for zip and UTF-8 for jar and other jar-like tasks (war, ear, ...) as well as the unzip family of tasks.

More recent versions of the ZIP specification introduce something called the "language encoding flag" which can be used to signal that a file name has been encoded using UTF-8. All ZIP-archives written by Compress will set this flag, if the encoding has been set to UTF-8. Our interoperability tests with existing archivers didn't show any ill effects (in fact, most archivers ignore the flag to date), but you can turn off the "language encoding flag" by setting the attribute useLanguageEncodingFlag to false on the ZipArchiveOutputStream if you should encounter problems.

The ZipFile and ZipArchiveInputStream classes will recognize the language encoding flag and ignore the encoding set in the constructor if it has been found.

The InfoZIP developers have introduced new ZIP extra fields that can be used to add an additional UTF-8 encoded file name to the entry's metadata. Most archivers ignore these extra fields. ZipArchiveOutputStream supports an option createUnicodeExtraFields which makes it write these extra fields either for all entries ("always") or only those whose name cannot be encoded using the specified encoding (not-encodeable), it defaults to "never" since the extra fields create bigger archives.

The fallbackToUTF8 attribute of ZipArchiveOutputStream can be used to create archives that use the specified encoding in the majority of cases but UTF-8 and the language encoding flag for filenames that cannot be encoded using the specified encoding.

The ZipFile and ZipArchiveInputStream classes recognize the Unicode extra fields by default and read the file name information from them, unless you set the constructor parameter scanForUnicodeExtraFields to false.

#### Recommendations for Interoperability

The optimal setting of flags depends on the archivers you expect as consumers/producers of the ZIP archives. Below are some test results which may be superseded with later versions of each tool.

* The java.util.zip package used by the jar executable or to read jars from your CLASSPATH reads and writes UTF-8 names, it doesn't set or recognize any flags or Unicode extra fields.
* Starting with Java7 java.util.zip writes UTF-8 by default and uses the language encoding flag. It is possible to specify a different encoding when reading/writing ZIPs via new constructors. The package now recognizes the language encoding flag when reading and ignores the Unicode extra fields.
* 7Zip writes CodePage 437 by default but uses UTF-8 and the language encoding flag when writing entries that cannot be encoded as CodePage 437 (similar to the zip task with fallbacktoUTF8 set to true). It recognizes the language encoding flag when reading and ignores the Unicode extra fields.
* WinZIP writes CodePage 437 and uses Unicode extra fields by default. It recognizes the Unicode extra field and the language encoding flag when reading.
* Windows' "compressed folder" feature doesn't recognize any flag or extra field and creates archives using the platforms default encoding - and expects archives to be in that encoding when reading them.
* InfoZIP based tools can recognize and write both, it is a compile time option and depends on the platform so your mileage may vary.
* PKWARE zip tools recognize both and prefer the language encoding flag. They create archives using CodePage 437 if possible and UTF-8 plus the language encoding flag for file names that cannot be encoded as CodePage 437.

So, what to do?

If you are creating jars, then java.util.zip is your main consumer. We recommend you set the encoding to UTF-8 and keep the language encoding flag enabled. The flag won't help or hurt java.util.zip prior to Java7 but archivers that support it will show the correct file names.

For maximum interop it is probably best to set the encoding to UTF-8, enable the language encoding flag and create Unicode extra fields when writing ZIPs. Such archives should be extracted correctly by java.util.zip, 7Zip, WinZIP, PKWARE tools and most likely InfoZIP tools. They will be unusable with Windows' "compressed folders" feature and bigger than archives without the Unicode extra fields, though.

If Windows' "compressed folders" is your primary consumer, then your best option is to explicitly set the encoding to the target platform. You may want to enable creation of Unicode extra fields so the tools that support them will extract the file names correctly.

### Encryption and Alternative Compression Algorithms

In most cases entries of an archive are not encrypted and are either not compressed at all or use the DEFLATE algorithm, Commons Compress' ZIP archiver will handle them just fine. As of version 1.7, Commons Compress can also decompress entries compressed with the legacy SHRINK and IMPLODE algorithms of PKZIP 1.x. Version 1.11 of Commons Compress adds read-only support for BZIP2. Version 1.16 adds read-only support for DEFLATE64 - also known as "enhanced DEFLATE".

The ZIP specification allows for various other compression algorithms and also supports several different ways of encrypting archive contents. Neither of those methods is currently supported by Commons Compress and any such entry can not be extracted by the archiving code.

ZipFile's and ZipArchiveInputStream's canReadEntryData methods will return false for encrypted entries or entries using an unsupported encryption mechanism. Using this method it is possible to at least detect and skip the entries that can not be extracted.

### Zip64 Support

The traditional ZIP format is limited to archive sizes of four gibibyte (actually 232 - 1 bytes ≈ 4.3 GB) and 65635 entries, where each individual entry is limited to four gibibyte as well. These limits seemed excessive in the 1980s.

Version 4.5 of the ZIP specification introduced the so called "Zip64 extensions" to push those limitations for compressed or uncompressed sizes of up to 16 exbibyte (actually 264 - 1 bytes ≈ 18.5 EB, i.e 18.5 x 1018 bytes) in archives that themselves can take up to 16 exbibyte containing more than 18 x 1018 entries.

Apache Commons Compress 1.2 and earlier do not support Zip64 extensions at all.

Starting with Apache Commons Compress 1.3 ZipArchiveInputStream and ZipFile transparently support Zip64 extensions. By default ZipArchiveOutputStream supports them transparently as well (i.e. it adds Zip64 extensions if needed and doesn't use them for entries/archives that don't need them) if the compressed and uncompressed sizes of the entry are known when putArchiveEntry is called or ZipArchiveOutputStream uses SeekableByteChannel (see [above](https://commons.apache.org/proper/commons-compress/zip.html#ZipArchiveOutputStream)). If only the uncompressed size is known ZipArchiveOutputStream will assume the compressed size will not be bigger than the uncompressed size.

ZipArchiveOutputStream's setUseZip64 can be used to control the behavior. Zip64Mode.AsNeeded is the default behavior described in the previous paragraph.

If ZipArchiveOutputStream is writing to a non-seekable stream it has to decide whether to use Zip64 extensions or not before it starts wrtiting the entry data. This means that if the size of the entry is unknown when putArchiveEntry is called it doesn't have anything to base the decision on. By default it will not use Zip64 extensions in order to create archives that can be extracted by older archivers (it will later throw an exception in closeEntry if it detects Zip64 extensions had been needed). It is possible to instruct ZipArchiveOutputStream to always create Zip64 extensions by using the setUseZip64 with an argument of Zip64Mode.Always; use this if you are writing entries of unknown size to a stream and expect some of them to be too big to fit into the traditional limits.

Zip64Mode.Always creates archives that use Zip64 extensions for all entries, even those that don't require them. Such archives will be slightly bigger than archives created with one of the other modes and not be readable by unarchivers that don't support Zip64 extensions.

Zip64Mode.Never will not use any Zip64 extensions at all and may lead to a Zip64RequiredException to be thrown if ZipArchiveOutputStream detects that one of the format's limits is exceeded. Archives created in this mode will be readable by all unarchivers; they may be slightly smaller than archives created with SeekableByteChannel in Zip64Mode.AsNeeded mode if some of the entries had unknown sizes.

The java.util.zip package and the jar command of Java5 and earlier can not read Zip64 extensions and will fail if the archive contains any. So if you intend to create archives that Java5 can consume you must set the mode to Zip64Mode.Never

#### Known Limitations

Some of the theoretical limits of the format are not reached because Apache Commons Compress' own API (ArchiveEntry's size information uses a long) or its usage of Java collections or SeekableByteChannel internally. The table below shows the theoretical limits supported by Apache Commons Compress. In practice it is very likely that you'd run out of memory or your file system won't allow files that big long before you reach either limit.

#### Known Interoperability Problems

The java.util.zip package of OpenJDK7 supports Zip 64 extensions but its ZipInputStream and ZipFile classes will be unable to extract archives created with Commons Compress 1.3's ZipArchiveOutputStream if the archive contains entries that use the data descriptor, are smaller than 4 GiB and have Zip 64 extensions enabled. I.e. the classes in OpenJDK currently only support archives that use Zip 64 extensions only when they are actually needed. These classes are used to load JAR files and are the base for the jar command line utility as well.

### Consuming Archives Completely

Prior to version 1.5 ZipArchiveInputStream would return null from getNextEntry or getNextZipEntry as soon as the first central directory header of the archive was found, leaving the whole central directory itself unread inside the stream. Starting with version 1.5 ZipArchiveInputStream will try to read the archive up to and including the "end of central directory" record effectively consuming the archive completely.

### Symbolic Links

Starting with Compress 1.5 ZipArchiveEntry recognizes Unix Symbolic Link entries written by InfoZIP's zip.

The ZipFile class contains a convenience method to read the link name of an entry. Basically all it does is read the contents of the entry and convert it to a string using the given file name encoding of the archive.

### Parallel zip creation

Starting with Compress 1.10 there is now built-in support for parallel creation of zip archives

Multiple threads can write to their own ScatterZipOutputStream instance that is backed to file or to some user-implemented form of storage (implementing ScatterGatherBackingStore).

When the threads finish, they can join these streams together to a complete zip file using the writeTo method that will write a single ScatterOutputStream to a target ZipArchiveOutputStream.

To assist this process, clients can use ParallelScatterZipCreator that will handle threads pools and correct memory model consistency so the client can avoid these issues.

Until version 1.18, there was no guarantee of order of the entries when writing a Zip file with ParallelScatterZipCreator. In consequence, when writing well-formed Zip files this way, it was usually necessary to keep a separate ScatterZipOutputStream that received all directories and wrote this to the target ZipArchiveOutputStream before the ones created through ParallelScatterZipCreator. This was the responsibility of the client.

Starting with version 1.19, entries order is kept, then this specific handling of directories is not necessary any more.

See the examples section for a code sample demonstrating how to make a zip file.