java.nio.channels

Class FileChannel

java.lang.Object
    java.nio.channels.spi.AbstractInterruptibleChannel
        java.nio.channels.FileChannel

All Implemented Interfaces:

Closeable, AutoCloseable, ByteChannel, Channel, GatheringByteChannel, InterruptibleChannel, ReadableByteChannel, ScatteringByteChannel, SeekableByteChannel, WritableByteChannel

```
public abstract class FileChannel
extends AbstractInterruptibleChannel
implements SeekableByteChannel, GatheringByteChannel, ScatteringByteChannel
```

A channel for reading, writing, mapping, and manipulating a file.

A file channel is a `SeekableByteChannel` that is connected to a file. It has a current *position* within its file which can be both *`queried`* and *`modified`*. The file itself contains a variable-length sequence of bytes that can be read and written and whose current *`size`* can be queried. The size of the file increases when bytes are written beyond its current size; the size of the file decreases when it is *truncated*. The file may also have some associated *metadata* such as access permissions, content type, and last-modification time; this class does not define methods for metadata access.

In addition to the familiar read, write, and close operations of byte channels, this class defines the following file-specific operations:

- Bytes may be `read` or *`written`* at an absolute position in a file in a way that does not affect the channel's current position.
- A region of a file may be *`mapped`* directly into memory; for large files this is often much more efficient than invoking the usual `read` or `write` methods.
- Updates made to a file may be *`forced out`* to the underlying storage device, ensuring that data are not lost in the event of a system crash.
- Bytes can be transferred from a file *`to some other channel`*, and *`vice versa`*, in a way that can be optimized by many operating systems into a very fast transfer directly to or from the filesystem cache.
- A region of a file may be *`locked`* against access by other programs.

File channels are safe for use by multiple concurrent threads. The `close` method may be invoked at any time, as specified by the `Channel` interface. Only one operation that involves the channel's position or can change its file's size may be in progress at any given time; attempts to initiate a second such operation while the first is still in progress will block until the first operation completes. Other operations, in particular those that take an explicit position, may proceed concurrently; whether they in fact do so is dependent upon the underlying implementation and is therefore unspecified.

The view of a file provided by an instance of this class is guaranteed to be consistent with other views of the same file provided by other instances in the same program. The view provided by an instance of this class may or may not, however, be consistent with the views seen by other concurrently-running programs due to caching performed by the underlying operating system and delays induced by network-filesystem protocols. This is true regardless of the language in which these other programs are written, and whether they are running on the same machine or on some other machine. The exact nature of any such inconsistencies are system-dependent and are therefore unspecified.

A file channel is created by invoking one of the `open` methods defined by this class. A file channel can also be obtained from an existing `FileInputStream`, `FileOutputStream`, or `RandomAccessFile` object by invoking that object's `getChannel` method, which returns a file channel that is connected to the same underlying file. Where the file channel is obtained from an existing stream or random access file then the state of the file channel is intimately connected to that of the object whose `getChannel` method returned the channel. Changing the channel's position, whether explicitly or by reading or writing bytes, will change the file position of the originating object, and vice versa. Changing the file's length via the file channel will change the length seen via the originating object, and vice versa. Changing the file's content by writing bytes will change the content seen by the originating object, and vice versa.

At various points this class specifies that an instance that is "open for reading," "open for writing," or "open for reading and writing" is required. A channel obtained via the `getChannel` method of a `FileInputStream` instance will be open for reading. A channel obtained via the `getChannel` method of a `FileOutputStream` instance will be open for writing. Finally, a channel obtained via the `getChannel` method of a `RandomAccessFile` instance will be open for reading if the instance was created with mode `"r"` and will be open for reading and writing if the instance was created with mode `"rw"`.

A file channel that is open for writing may be in *append mode*, for example if it was obtained from a file-output stream that was created by invoking the `FileOutputStream(File,boolean)` constructor and passing `true` for the second parameter. In this mode each invocation of a relative write operation first advances the position to the end of the file and then writes the requested data. Whether the advancement of the position and the writing of the data are done in a single atomic operation is system-dependent and therefore unspecified.

Since:

1.4

See Also:

`FileInputStream.getChannel()`, `FileOutputStream.getChannel()`, `RandomAccessFile.getChannel()`

Nested Class Summary

Nested Classes

Modifier and Type	Class and Description
`static class`	**`FileChannel.MapMode`** A typesafe enumeration for file-mapping modes.

Constructor Summary

Constructors

Modifier	Constructor and Description
`protected`	**`FileChannel`**`()` Initializes a new instance of this class.

Method Summary

Methods

Modifier and Type	Method and Description
`abstract void`	**`force`**`(boolean metaData)` Forces any updates to this channel's file to be written to the storage device that contains it.

`FileLock`	`lock()` Acquires an exclusive lock on this channel's file.
`abstract FileLock`	`lock(long position, long size, boolean shared)` Acquires a lock on the given region of this channel's file.
`abstract MappedByteBuffer`	`map(FileChannel.MapMode mode, long position, long size)` Maps a region of this channel's file directly into memory.
`static FileChannel`	`open(Path path, OpenOption... options)` Opens or creates a file, returning a file channel to access the file.
`static FileChannel`	`open(Path path, Set<? extends OpenOption> options, FileAttribute<?>... attrs)` Opens or creates a file, returning a file channel to access the file.
`abstract long`	`position()` Returns this channel's file position.
`abstract FileChannel`	`position(long newPosition)` Sets this channel's file position.
`abstract int`	`read(ByteBuffer dst)` Reads a sequence of bytes from this channel into the given buffer.
`long`	`read(ByteBuffer[] dsts)` Reads a sequence of bytes from this channel into the given buffers.
`abstract long`	`read(ByteBuffer[] dsts, int offset, int length)` Reads a sequence of bytes from this channel into a subsequence of the given buffers.
`abstract int`	`read(ByteBuffer dst, long position)` Reads a sequence of bytes from this channel into the given buffer, starting at the given file position.
`abstract long`	`size()` Returns the current size of this channel's file.
`abstract long`	`transferFrom(ReadableByteChannel src, long position, long count)` Transfers bytes into this channel's file from the given readable byte channel.
`abstract long`	`transferTo(long position, long count, WritableByteChannel target)` Transfers bytes from this channel's file to the given writable byte channel.
`abstract FileChannel`	`truncate(long size)` Truncates this channel's file to the given size.
`FileLock`	`tryLock()` Attempts to acquire an exclusive lock on this channel's file.
`abstract FileLock`	`tryLock(long position, long size, boolean shared)` Attempts to acquire a lock on the given region of this channel's file.
`abstract int`	`write(ByteBuffer src)` Writes a sequence of bytes to this channel from the given buffer.
`long`	`write(ByteBuffer[] srcs)` Writes a sequence of bytes to this channel from the given buffers.
`abstract long`	`write(ByteBuffer[] srcs, int offset, int length)` Writes a sequence of bytes to this channel from a subsequence of the given buffers.
`abstract int`	`write(ByteBuffer src, long position)` Writes a sequence of bytes to this channel from the given buffer, starting at the given file position.

Methods inherited from class java.nio.channels.spi.AbstractInterruptibleChannel

`begin, close, end, implCloseChannel, isOpen`

Methods inherited from class java.lang.Object

`clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait`

Methods inherited from interface java.nio.channels.Channel

`close, isOpen`

Constructor Detail

FileChannel

```
protected FileChannel()
```

Initializes a new instance of this class.

Method Detail

open

```
public static FileChannel open(Path path,
               Set<? extends OpenOption> options,
               FileAttribute<?>... attrs)
                        throws IOException
```

Opens or creates a file, returning a file channel to access the file.

The `options` parameter determines how the file is opened. The `READ` and `WRITE` options determine if the file should be opened for reading and/or writing. If neither option (or the `APPEND` option) is contained in the array then the file is opened for reading. By default reading or writing commences at the beginning of the file.

In the addition to `READ` and `WRITE`, the following options may be present:

Option	Description
`APPEND`	If this option is present then the file is opened for writing and each invocation of the channel's `write` method first advances the position to the end of the file and then writes the requested data. Whether the advancement of the position and the writing of the data are done in a single atomic operation is system-dependent and therefore unspecified. This option may not be used in conjunction with the `READ` or `TRUNCATE_EXISTING` options.
`TRUNCATE_EXISTING`	If this option is present then the existing file is truncated to a size of 0 bytes. This option is ignored when the file is opened only for reading.
`CREATE_NEW`	If this option is present then a new file is created, failing if the file already

	exists. When creating a file the check for the existence of the file and the creation of the file if it does not exist is atomic with respect to other file system operations. This option is ignored when the file is opened only for reading.
`CREATE`	If this option is present then an existing file is opened if it exists, otherwise a new file is created. When creating a file the check for the existence of the file and the creation of the file if it does not exist is atomic with respect to other file system operations. This option is ignored if the `CREATE_NEW` option is also present or the file is opened only for reading.
`DELETE_ON_CLOSE`	When this option is present then the implementation makes a *best effort* attempt to delete the file when closed by the the `close` method. If the `close` method is not invoked then a *best effort* attempt is made to delete the file when the Java virtual machine terminates.
`SPARSE`	When creating a new file this option is a *hint* that the new file will be sparse. This option is ignored when not creating a new file.
`SYNC`	Requires that every update to the file's content or metadata be written synchronously to the underlying storage device. (see Synchronized I/O file integrity).
`DSYNC`	Requires that every update to the file's content be written synchronously to the underlying storage device. (see Synchronized I/O file integrity).

An implementation may also support additional options.

The `attrs` parameter is an optional array of file `file-attributes` to set atomically when creating the file.

The new channel is created by invoking the `newFileChannel` method on the provider that created the `Path`.

Parameters:

`path` - The path of the file to open or create

`options` - Options specifying how the file is opened

`attrs` - An optional list of file attributes to set atomically when creating the file

Returns:

A new file channel

Throws:

`IllegalArgumentException` - If the set contains an invalid combination of options

`UnsupportedOperationException` - If the `path` is associated with a provider that does not support creating file channels, or an unsupported open option is specified, or the array contains an attribute that cannot be set atomically when creating the file

`IOException` - If an I/O error occurs

`SecurityException` - If a security manager is installed and it denies an unspecified permission required by the implementation. In the case of the default provider, the `SecurityManager.checkRead(String)` method is invoked to check read access if the file is opened for reading. The `SecurityManager.checkWrite(String)` method is invoked to check write access if the file is opened for writing

Since:

1.7

open

```
public static FileChannel open(Path path,
               OpenOption... options)
                        throws IOException
```

Opens or creates a file, returning a file channel to access the file.

An invocation of this method behaves in exactly the same way as the invocation

```
    fc.open(file, opts, new FileAttribute<?>[0]);
```

where `opts` is a set of the options specified in the `options` array.

Parameters:

`path` - The path of the file to open or create

`options` - Options specifying how the file is opened

Returns:

A new file channel

Throws:

`IllegalArgumentException` - If the set contains an invalid combination of options

`UnsupportedOperationException` - If the `path` is associated with a provider that does not support creating file channels, or an unsupported open option is specified

`IOException` - If an I/O error occurs

`SecurityException` - If a security manager is installed and it denies an unspecified permission required by the implementation. In the case of the default provider, the `SecurityManager.checkRead(String)` method is invoked to check read access if the file is opened for reading. The `SecurityManager.checkWrite(String)` method is invoked to check write access if the file is opened for writing

Since:

1.7

read

```
public abstract int read(ByteBuffer dst)
                  throws IOException
```

Reads a sequence of bytes from this channel into the given buffer.

Bytes are read starting at this channel's current file position, and then the file position is updated with the number of bytes actually read. Otherwise this method behaves exactly as specified in the `ReadableByteChannel` interface.

Specified by:

`read` in interface `ReadableByteChannel`

Specified by:

`read` in interface `SeekableByteChannel`

Parameters:

`dst` - The buffer into which bytes are to be transferred

Returns:

The number of bytes read, possibly zero, or `-1` if the channel has reached end-of-stream

Throws:

`ClosedChannelException` - If this channel is closed

`AsynchronousCloseException` - If another thread closes this channel while the read operation is in progress

`ClosedByInterruptException` - If another thread interrupts the current thread while the read operation is in progress, thereby closing the channel and setting the current thread's interrupt status

`IOException` - If some other I/O error occurs

read

```
public abstract long read(ByteBuffer[] dsts,
        int offset,
        int length)
                   throws IOException
```

Reads a sequence of bytes from this channel into a subsequence of the given buffers.

Bytes are read starting at this channel's current file position, and then the file position is updated with the number of bytes actually read. Otherwise this method behaves exactly as specified in the `ScatteringByteChannel` interface.

Specified by:

`read` in interface `ScatteringByteChannel`

Parameters:

`dsts` - The buffers into which bytes are to be transferred

`offset` - The offset within the buffer array of the first buffer into which bytes are to be transferred; must be non-negative and no larger than `dsts.length`

`length` - The maximum number of buffers to be accessed; must be non-negative and no larger than `dsts.length` - `offset`

Returns:

The number of bytes read, possibly zero, or `-1` if the channel has reached end-of-stream

Throws:

`ClosedChannelException` - If this channel is closed

`AsynchronousCloseException` - If another thread closes this channel while the read operation is in progress

`ClosedByInterruptException` - If another thread interrupts the current thread while the read operation is in progress, thereby closing the channel and setting the current thread's interrupt status

`IOException` - If some other I/O error occurs

read

```
public final long read(ByteBuffer[] dsts)
                throws IOException
```

Reads a sequence of bytes from this channel into the given buffers.

Bytes are read starting at this channel's current file position, and then the file position is updated with the number of bytes actually read. Otherwise this method behaves exactly as specified in the `ScatteringByteChannel` interface.

Specified by:

`read` in interface `ScatteringByteChannel`

Parameters:

`dsts` - The buffers into which bytes are to be transferred

Returns:

The number of bytes read, possibly zero, or `-1` if the channel has reached end-of-stream

Throws:

`ClosedChannelException` - If this channel is closed

`AsynchronousCloseException` - If another thread closes this channel while the read operation is in progress

`ClosedByInterruptException` - If another thread interrupts the current thread while the read operation is in progress, thereby closing the channel and setting the current thread's interrupt status

`IOException` - If some other I/O error occurs

write

```
public abstract int write(ByteBuffer src)
                   throws IOException
```

Writes a sequence of bytes to this channel from the given buffer.

Bytes are written starting at this channel's current file position unless the channel is in append mode, in which case the position is first advanced to the end of the file. The file is grown, if necessary, to accommodate the written bytes, and then the file position is updated with the number of bytes actually written. Otherwise this method behaves exactly as specified by the `WritableByteChannel` interface.

Specified by:

`write` in interface `SeekableByteChannel`

Specified by:

`write` in interface `WritableByteChannel`

Parameters:

`src` - The buffer from which bytes are to be retrieved

Returns:

The number of bytes written, possibly zero

Throws:

`ClosedChannelException` - If this channel is closed

`AsynchronousCloseException` - If another thread closes this channel while the write operation is in progress

`ClosedByInterruptException` - If another thread interrupts the current thread while the write operation is in progress, thereby closing the channel and setting the current thread's interrupt status

`IOException` - If some other I/O error occurs

write

```
public abstract long write(ByteBuffer[] srcs,
         int offset,
         int length)
                  throws IOException
```

Writes a sequence of bytes to this channel from a subsequence of the given buffers.

Bytes are written starting at this channel's current file position unless the channel is in append mode, in which case the position is first advanced to the end of the file. The file is grown, if necessary, to accommodate the written bytes, and then the file position is updated with the number of bytes actually written. Otherwise this method behaves exactly as specified in the `GatheringByteChannel` interface.

Specified by:

`write` in interface `GatheringByteChannel`

Parameters:

`srcs` - The buffers from which bytes are to be retrieved

`offset` - The offset within the buffer array of the first buffer from which bytes are to be retrieved; must be non-negative and no larger than `srcs.length`

`length` - The maximum number of buffers to be accessed; must be non-negative and no larger than `srcs.length - offset`

Returns:

The number of bytes written, possibly zero

Throws:

`ClosedChannelException` - If this channel is closed

`AsynchronousCloseException` - If another thread closes this channel while the write operation is in progress

`ClosedByInterruptException` - If another thread interrupts the current thread while the write operation is in progress, thereby closing the channel and setting the current thread's interrupt status

`IOException` - If some other I/O error occurs

write

```
public final long write(ByteBuffer[] srcs)
                 throws IOException
```

Writes a sequence of bytes to this channel from the given buffers.

Bytes are written starting at this channel's current file position unless the channel is in append mode, in which case the position is first advanced to the end of the file. The file is grown, if necessary, to accommodate the written bytes, and then the file position is updated with the number of bytes actually written. Otherwise this method behaves exactly as specified in the `GatheringByteChannel` interface.

Specified by:

`write` in interface `GatheringByteChannel`

Parameters:

`srcs` - The buffers from which bytes are to be retrieved

Returns:

The number of bytes written, possibly zero

Throws:

`ClosedChannelException` - If this channel is closed

`AsynchronousCloseException` - If another thread closes this channel while the write operation is in progress

`ClosedByInterruptException` - If another thread interrupts the current thread while the write operation is in progress, thereby closing the channel and setting the current thread's interrupt status

`IOException` - If some other I/O error occurs

position

```
public abstract long position()
                       throws IOException
```

Returns this channel's file position.

Specified by:

`position` in interface `SeekableByteChannel`

Returns:

This channel's file position, a non-negative integer counting the number of bytes from the beginning of the file to the current position

Throws:

`ClosedChannelException` - If this channel is closed

`IOException` - If some other I/O error occurs

position

```
public abstract FileChannel position(long newPosition)
                              throws IOException
```

Sets this channel's file position.

Setting the position to a value that is greater than the file's current size is legal but does not change the size of the file. A later attempt to read bytes at such a position will immediately return an end-of-file indication. A later attempt to write bytes at such a position will cause the file to be grown to accommodate the new bytes; the values of any bytes between the previous end-of-file and the newly-written bytes are unspecified.

Specified by:

`position` in interface `SeekableByteChannel`

Parameters:

`newPosition` - The new position, a non-negative integer counting the number of bytes from the beginning of the file

Returns:

This file channel

Throws:

`ClosedChannelException` - If this channel is closed

`IllegalArgumentException` - If the new position is negative

`IOException` - If some other I/O error occurs

size

```
public abstract long size()
                   throws IOException
```

Returns the current size of this channel's file.

Specified by:

`size` in interface `SeekableByteChannel`

Returns:

The current size of this channel's file, measured in bytes

Throws:

`ClosedChannelException` - If this channel is closed

`IOException` - If some other I/O error occurs

truncate

```
public abstract FileChannel truncate(long size)
                              throws IOException
```

Truncates this channel's file to the given size.

If the given size is less than the file's current size then the file is truncated, discarding any bytes beyond the new end of the file. If the given size is greater than or equal to the file's current size then the file is not modified. In either case, if this channel's file position is greater than the given size then it is set to that size.

Specified by:

`truncate` in interface `SeekableByteChannel`

Parameters:

`size` - The new size, a non-negative byte count

Returns:

This file channel

Throws:

`NonWritableChannelException` - If this channel was not opened for writing

`ClosedChannelException` - If this channel is closed

`IllegalArgumentException` - If the new size is negative

`IOException` - If some other I/O error occurs

force

```
public abstract void force(boolean metaData)
                    throws IOException
```

Forces any updates to this channel's file to be written to the storage device that contains it.

If this channel's file resides on a local storage device then when this method returns it is guaranteed that all changes made to the file since this channel was created, or since this method was last invoked, will have been written to that device. This is useful for ensuring that critical information is not lost in the event of a system crash.

If the file does not reside on a local device then no such guarantee is made.

The `metaData` parameter can be used to limit the number of I/O operations that this method is required to perform. Passing `false` for this parameter indicates that only updates to the file's content need be written to storage; passing `true` indicates that updates to both the file's content and metadata must be written, which generally requires at least one more I/O operation. Whether this parameter actually has any effect is dependent upon the underlying operating system and is therefore unspecified.

Invoking this method may cause an I/O operation to occur even if the channel was only opened for reading. Some operating systems, for example, maintain a last-access time as part of a file's metadata, and this time is updated whenever the file is read. Whether or not this is actually done is system-dependent and is therefore unspecified.

This method is only guaranteed to force changes that were made to this channel's file via the methods defined in this class. It may or may not force changes that were made by modifying the content of a *`mapped byte buffer`* obtained by invoking the `map` method. Invoking the `force` method of the mapped byte buffer will force changes made to the buffer's content to be written.

Parameters:

`metaData` - If `true` then this method is required to force changes to both the file's content and metadata to be written to storage; otherwise, it need only force content changes to be written

Throws:

`ClosedChannelException` - If this channel is closed

`IOException` - If some other I/O error occurs

transferTo

```
public abstract long transferTo(long position,
              long count,
              WritableByteChannel target)
                       throws IOException
```

Transfers bytes from this channel's file to the given writable byte channel.

An attempt is made to read up to `count` bytes starting at the given `position` in this channel's file and write them to the target channel. An invocation of this method may or may not transfer all of the requested bytes; whether or not it does so depends upon the natures and states of the channels. Fewer than the requested number of bytes are transferred if this channel's file contains fewer than `count` bytes starting at the given `position`, or if the target channel is non-blocking and it has fewer than `count` bytes free in its output buffer.

This method does not modify this channel's position. If the given position is greater than the file's current size then no bytes are transferred. If the target channel has a position then bytes are written starting at that position and then the position is incremented by the number of bytes written.

This method is potentially much more efficient than a simple loop that reads from this channel and writes to the target channel. Many operating systems can transfer bytes directly from the filesystem cache to the target channel without actually copying them.

Parameters:

`position` - The position within the file at which the transfer is to begin; must be non-negative

`count` - The maximum number of bytes to be transferred; must be non-negative

`target` - The target channel

Returns:

The number of bytes, possibly zero, that were actually transferred

Throws:

`IllegalArgumentException` - If the preconditions on the parameters do not hold

`NonReadableChannelException` - If this channel was not opened for reading

`NonWritableChannelException` - If the target channel was not opened for writing

`ClosedChannelException` - If either this channel or the target channel is closed

`AsynchronousCloseException` - If another thread closes either channel while the transfer is in progress

`ClosedByInterruptException` - If another thread interrupts the current thread while the transfer is in progress, thereby closing both channels and setting the current thread's interrupt status

`IOException` - If some other I/O error occurs

transferFrom

```
public abstract long transferFrom(ReadableByteChannel src,
                long position,
                long count)
                         throws IOException
```

Transfers bytes into this channel's file from the given readable byte channel.

An attempt is made to read up to `count` bytes from the source channel and write them to this channel's file starting at the given `position`. An invocation of this method may or may not transfer all of the requested bytes; whether or not it does so depends upon the natures and states of the channels. Fewer than the requested number of bytes will be transferred if the source channel has fewer than `count` bytes remaining, or if the source channel is non-blocking and has fewer than `count` bytes immediately available in its input buffer.

This method does not modify this channel's position. If the given position is greater than the file's current size then no bytes are transferred. If the source channel has a position then bytes are read starting at that position and then the position is incremented by the number of bytes read.

This method is potentially much more efficient than a simple loop that reads from the source channel and writes to this channel. Many operating systems can transfer bytes directly from the source channel into the filesystem cache without actually copying them.

Parameters:

`src` - The source channel

`position` - The position within the file at which the transfer is to begin; must be non-negative

`count` - The maximum number of bytes to be transferred; must be non-negative

Returns:

The number of bytes, possibly zero, that were actually transferred

Throws:

`IllegalArgumentException` - If the preconditions on the parameters do not hold

`NonReadableChannelException` - If the source channel was not opened for reading

`NonWritableChannelException` - If this channel was not opened for writing

`ClosedChannelException` - If either this channel or the source channel is closed

`AsynchronousCloseException` - If another thread closes either channel while the transfer is in progress

`ClosedByInterruptException` - If another thread interrupts the current thread while the transfer is in progress, thereby closing both channels and setting the current thread's interrupt status

`IOException` - If some other I/O error occurs

read

```
public abstract int read(ByteBuffer dst,
       long position)
                throws IOException
```

Reads a sequence of bytes from this channel into the given buffer, starting at the given file position.

This method works in the same manner as the `read(ByteBuffer)` method, except that bytes are read starting at the given file position rather than at the channel's current position. This method does not modify this channel's position. If the given position is greater than the file's current size then no bytes are read.

Parameters:

`dst` - The buffer into which bytes are to be transferred

`position` - The file position at which the transfer is to begin; must be non-negative

Returns:

The number of bytes read, possibly zero, or `-1` if the given position is greater than or equal to the file's current size

Throws:

`IllegalArgumentException` - If the position is negative

`NonReadableChannelException` - If this channel was not opened for reading

`ClosedChannelException` - If this channel is closed

`AsynchronousCloseException` - If another thread closes this channel while the read operation is in progress

`ClosedByInterruptException` - If another thread interrupts the current thread while the read operation is in progress, thereby closing the channel and setting the current thread's interrupt status

`IOException` - If some other I/O error occurs

write

```
public abstract int write(ByteBuffer src,
        long position)
                 throws IOException
```

Writes a sequence of bytes to this channel from the given buffer, starting at the given file position.

This method works in the same manner as the `write(ByteBuffer)` method, except that bytes are written starting at the given file position rather than at the channel's current position. This method does not modify this channel's position. If the given position is greater than the file's current size then the file will be grown to accommodate the new bytes; the values of any bytes between the previous end-of-file and the newly-written bytes are unspecified.

Parameters:

`src` - The buffer from which bytes are to be transferred

`position` - The file position at which the transfer is to begin; must be non-negative

Returns:

The number of bytes written, possibly zero

Throws:

`IllegalArgumentException` - If the position is negative

`NonWritableChannelException` - If this channel was not opened for writing

`ClosedChannelException` - If this channel is closed

`AsynchronousCloseException` - If another thread closes this channel while the write operation is in progress

`ClosedByInterruptException` - If another thread interrupts the current thread while the write operation is in progress, thereby closing the channel and setting the current thread's interrupt status

`IOException` - If some other I/O error occurs

map

```
public abstract MappedByteBuffer map(FileChannel.MapMode mode,
                   long position,
                   long size)
                            throws IOException
```

Maps a region of this channel's file directly into memory.

A region of a file may be mapped into memory in one of three modes:

- *Read-only:* Any attempt to modify the resulting buffer will cause a `ReadOnlyBufferException` to be thrown. (`MapMode.READ_ONLY`)

- *Read/write:* Changes made to the resulting buffer will eventually be propagated to the file; they may or may not be made visible to other programs that have mapped the same file. (`MapMode.READ_WRITE`)

- *Private:* Changes made to the resulting buffer will not be propagated to the file and will not be visible to other programs that have mapped the same file; instead, they will cause private copies of the modified portions of the buffer to be created. (`MapMode.PRIVATE`)

For a read-only mapping, this channel must have been opened for reading; for a read/write or private mapping, this channel must have been opened for both reading and writing.

The *`mapped byte buffer`* returned by this method will have a position of zero and a limit and capacity of `size`; its mark will be undefined. The buffer and the mapping that it represents will remain valid until the buffer itself is garbage-collected.

A mapping, once established, is not dependent upon the file channel that was used to create it. Closing the channel, in particular, has no effect upon the validity of the mapping.

Many of the details of memory-mapped files are inherently dependent upon the underlying operating system and are therefore unspecified. The behavior of this method when the requested region is not completely contained within this channel's file is unspecified. Whether changes made to the content or size of the underlying file, by this program or another, are propagated to the buffer is unspecified. The rate at which changes to the buffer are propagated to the file is unspecified.

For most operating systems, mapping a file into memory is more expensive than reading or writing a few tens of kilobytes of data via the usual `read` and `write` methods. From the standpoint of performance it is generally only worth mapping relatively large files into memory.

Parameters:

`mode` - One of the constants `READ_ONLY`, `READ_WRITE`, or `PRIVATE` defined in the `FileChannel.MapMode` class, according to whether the file is to be mapped read-only, read/write, or privately (copy-on-write), respectively `position` - The position within the file at which the mapped region is to start; must be non-negative

`size` - The size of the region to be mapped; must be non-negative and no greater than `Integer.MAX_VALUE`

Returns:

The mapped byte buffer

Throws:

`NonReadableChannelException` - If the `mode` is `READ_ONLY` but this channel was not opened for reading

`NonWritableChannelException` - If the `mode` is `READ_WRITE` or `PRIVATE` but this channel was not opened for both reading and writing

`IllegalArgumentException` - If the preconditions on the parameters do not hold

`IOException` - If some other I/O error occurs

See Also:

`FileChannel.MapMode`, `MappedByteBuffer`

lock

```
public abstract FileLock lock(long position,
            long size,
            boolean shared)
                     throws IOException
```

Acquires a lock on the given region of this channel's file.

An invocation of this method will block until the region can be locked, this channel is closed, or the invoking thread is interrupted, whichever comes first.

If this channel is closed by another thread during an invocation of this method then an `AsynchronousCloseException` will be thrown.

If the invoking thread is interrupted while waiting to acquire the lock then its interrupt status will be set and a `FileLockInterruptionException` will be thrown. If the invoker's interrupt status is set when this method is invoked then that exception will be thrown immediately; the thread's interrupt status will not be changed.

The region specified by the `position` and `size` parameters need not be contained within, or even overlap, the actual underlying file. Lock regions are fixed in size; if a locked region initially contains the end of the file and the file grows beyond the region then the new portion of the file will not be covered by the lock. If a file is expected to grow in size and a lock on the entire file is required then a region starting at zero, and no smaller than the expected maximum size of the file, should be locked. The zero-argument `lock()` method simply locks a region of size `Long.MAX_VALUE`.

Some operating systems do not support shared locks, in which case a request for a shared lock is automatically converted into a request for an exclusive lock. Whether the newly-acquired lock is shared or exclusive may be tested by invoking the resulting lock object's `isShared` method.

File locks are held on behalf of the entire Java virtual machine. They are not suitable for controlling access to a file by multiple threads within the same virtual machine.

Parameters:

`position` - The position at which the locked region is to start; must be non-negative

`size` - The size of the locked region; must be non-negative, and the sum `position` + `size` must be non-negative

`shared` - `true` to request a shared lock, in which case this channel must be open for reading (and possibly writing); `false` to request an exclusive lock, in which case this channel must be open for writing (and possibly reading)

Returns:

A lock object representing the newly-acquired lock

Throws:

`IllegalArgumentException` - If the preconditions on the parameters do not hold

`ClosedChannelException` - If this channel is closed

`AsynchronousCloseException` - If another thread closes this channel while the invoking thread is blocked in this method

`FileLockInterruptionException` - If the invoking thread is interrupted while blocked in this method

`OverlappingFileLockException` - If a lock that overlaps the requested region is already held by this Java virtual machine, or if another thread is already blocked in this method and is attempting to lock an overlapping region

`NonReadableChannelException` - If `shared` is `true` this channel was not opened for reading

`NonWritableChannelException` - If `shared` is `false` but this channel was not opened for writing

`IOException` - If some other I/O error occurs

See Also:

`lock()`, `tryLock()`, `tryLock(long,long,boolean)`

lock

```
public final FileLock lock()
                    throws IOException
```

Acquires an exclusive lock on this channel's file.

An invocation of this method of the form `fc.lock()` behaves in exactly the same way as the invocation

```
    fc.lock(0L, Long.MAX_VALUE, false)
```

Returns:

A lock object representing the newly-acquired lock

Throws:

`ClosedChannelException` - If this channel is closed

`AsynchronousCloseException` - If another thread closes this channel while the invoking thread is blocked in this method

`FileLockInterruptionException` - If the invoking thread is interrupted while blocked in this method

`OverlappingFileLockException` - If a lock that overlaps the requested region is already held by this Java virtual machine, or if another thread is already blocked in this method and is attempting to lock an overlapping region of the same file

`NonWritableChannelException` - If this channel was not opened for writing

`IOException` - If some other I/O error occurs

See Also:

`lock(long,long,boolean)`, `tryLock()`, `tryLock(long,long,boolean)`

tryLock

```
public abstract FileLock tryLock(long position,
               long size,
               boolean shared)
                        throws IOException
```

Attempts to acquire a lock on the given region of this channel's file.

This method does not block. An invocation always returns immediately, either having acquired a lock on the requested region or having failed to do so. If it fails to acquire a lock because an overlapping lock is held by another program then it returns `null`. If it fails to acquire a lock for any other reason then an appropriate exception is thrown.

The region specified by the `position` and `size` parameters need not be contained within, or even overlap, the actual underlying file. Lock regions are fixed in size; if a locked region initially contains the end of the file and the file grows beyond the region then the new portion of the file will not be covered by the lock. If a file is expected to grow in size and a lock on the entire file is required then a region starting at zero, and no smaller than the expected maximum size of the file, should be locked. The zero-argument `tryLock()` method simply locks a region of size `Long.MAX_VALUE`.

Some operating systems do not support shared locks, in which case a request for a shared lock is automatically converted into a request for an exclusive lock. Whether the newly-acquired lock is shared or exclusive may be tested by invoking the resulting lock object's `isShared` method.

File locks are held on behalf of the entire Java virtual machine. They are not suitable for controlling access to a file by multiple threads within the same virtual machine.

Parameters:

`position` - The position at which the locked region is to start; must be non-negative

`size` - The size of the locked region; must be non-negative, and the sum `position` + `size` must be non-negative

`shared` - `true` to request a shared lock, `false` to request an exclusive lock

Returns:

A lock object representing the newly-acquired lock, or `null` if the lock could not be acquired because another program holds an overlapping lock

Throws:

`IllegalArgumentException` - If the preconditions on the parameters do not hold

`ClosedChannelException` - If this channel is closed

`OverlappingFileLockException` - If a lock that overlaps the requested region is already held by this Java virtual machine, or if another thread is already blocked in this method and is attempting to lock an overlapping region of the same file

`IOException` - If some other I/O error occurs

See Also:

`lock()`, `lock(long,long,boolean)`, `tryLock()`

tryLock

```
public final FileLock tryLock()
                       throws IOException
```

Attempts to acquire an exclusive lock on this channel's file.

An invocation of this method of the form `fc.tryLock()` behaves in exactly the same way as the invocation

```
fc.tryLock(0L, Long.MAX_VALUE, false)
```

Returns:

A lock object representing the newly-acquired lock, or `null` if the lock could not be acquired because another program holds an overlapping lock

Throws:

`ClosedChannelException` - If this channel is closed

`OverlappingFileLockException` - If a lock that overlaps the requested region is already held by this Java virtual machine, or if another thread is already blocked in this method and is attempting to lock an overlapping region

`IOException` - If some other I/O error occurs

See Also:

`lock()`, `lock(long,long,boolean)`, `tryLock(long,long,boolean)`