



CHAPTER 14

java.nio and Subpackages

This chapter documents the New I/O API defined by the `java.nio` package and its sub-packages. It covers:

`java.nio`

Defines the `Buffer` class and type-specific subclasses, most notably the `ByteBuffer` class that is often used for I/O in the `java.nio.channels` package.

`java.nio.channels`

Defines the `Channel` abstraction for high-performance I/O and implements channels for file and network I/O. Also allows nonblocking I/O with the `Selector` class.

`java.nio.channels.spi`

The service provider interface for channel and selector implementations.

`java.nio.charset`

Defines classes for encoding sequences of characters into bytes and decoding sequences of bytes into characters, according to the encoding rules of a named charset.

`java.nio.charset.spi`

The service provider interface for charset implementations.

Package `java.nio`

Java 1.4

This package defines buffer classes that are fundamental to the `java.nio` API. See `Buffer` for an overview of buffers and `ByteBuffer` (the most important of the buffer classes) for full documentation of byte buffers. The other type-specific buffer classes are close analogs to `ByteBuffer` and are documented in terms of that class. See the `java.nio.channels` package for classes that perform I/O operations on buffers.

Package *java.nio*

Classes:

```
public abstract class Buffer;  
    public abstract class ByteBuffer extends Buffer implements Comparable;  
        public abstract class MappedByteBuffer extends ByteBuffer;  
    public abstract class CharBuffer extends Buffer implements CharSequence, Comparable;  
    public abstract class DoubleBuffer extends Buffer implements Comparable;  
    public abstract class FloatBuffer extends Buffer implements Comparable;  
    public abstract class IntBuffer extends Buffer implements Comparable;  
    public abstract class LongBuffer extends Buffer implements Comparable;  
    public abstract class ShortBuffer extends Buffer implements Comparable;  
public final class ByteOrder;
```

Exceptions:

```
public class BufferOverflowException extends RuntimeException;  
public class BufferUnderflowException extends RuntimeException;  
public class InvalidMarkException extends IllegalStateException;  
public class ReadOnlyBufferException extends UnsupportedOperationException;
```

Buffer

Java 1.4

java.nio

This class is the abstract superclass of all buffer classes in the *java.nio* API. A buffer is a linear (finite) sequence of primitive values. The *java.nio* package defines a **Buffer** subclass for each primitive type in Java except for *boolean*. **Buffer** itself defines the common, type-independent features of all buffers. **Buffer** and its subclasses are intended for use by a single thread at a time and contain no synchronization code to make them thread-safe.

The purpose of a buffer is to store data, and buffer classes must define methods for reading data from a buffer and writing data into a buffer. Because each **Buffer** subclass stores data of a different primitive type, however, the *get()* and *put()* methods that read and write data must be defined by each of the individual subclasses. See **ByteBuffer** (the most important subclass) for documentation of these methods; all the other subclasses define similar methods that differ only in the datatype of the values being read or written.

Each buffer has four numbers associated with it:

Capacity

A buffer's capacity is its maximum size; it can hold this many values. The capacity is specified when a buffer is created and cannot be changed. It can be queried with the *capacity()* method.

Limit

A buffer's limit is its current size, or the index of the first element that does not contain valid data. Data cannot be read from or written into a buffer beyond the limit. When data is written into a buffer, the limit is usually the same as the capacity. When data is read from a buffer, the limit may be less than the capacity and indicate the amount of valid data contained in the buffer. Two *limit()* methods exist: one to query a buffer's limit and one to set it.

Position

A buffer's position is the index of the element in the buffer at which data is read or written. It is used and updated by the relative `get()` and `put()` methods defined by `ByteBuffer` and the other `Buffer` subclasses. Two `position()` methods exist to query and set the current position of the buffer. A buffer's position is always greater than or equal to 0 and less than or equal to the buffer's limit. The `remaining()` method returns the number of elements between the position and the limit, and `hasRemaining()` returns `true` if this number is greater than 0.

Mark

A buffer's mark is a temporarily saved position. Call `mark()` to set the mark to the current position. Call `reset()` to restore the buffer's position to the marked position.

`Buffer` defines several methods that perform important operations on a buffer:

clear()

This method does not actually clear the contents of the buffer but sets the position to 0, sets the limit to the capacity, and discards any saved mark. This prepares the buffer to have new data written into it.

flip()

This method sets the limit to the position, sets the position to 0, and discards any saved mark. After data has been written into a buffer, this method “flips” the purpose of the buffer and prepares it for reading.

rewind()

This method sets the position to 0 and discards any saved mark. It does not alter the limit and can be used to restart a read operation at the beginning of the buffer.

`Buffer` objects may be read-only, in which case, any attempt to store data in the buffer results in a `ReadOnlyBufferException`. The `isReadOnly()` method returns `true` if a buffer is read-only.

```
public abstract class Buffer {
    // No Constructor
    // Public Instance Methods
    public final int capacity();
    public final Buffer clear();
    public final Buffer flip();
    public final boolean hasRemaining();
    public abstract boolean isReadOnly();
    public final int limit();
    public final Buffer limit(int newLimit);
    public final Buffer mark();
    public final int position();
    public final Buffer position(int newPosition);
    public final int remaining();
    public final Buffer reset();
    public final Buffer rewind();
}
```

Subclasses: `ByteBuffer`, `CharBuffer`, `DoubleBuffer`, `FloatBuffer`, `IntBuffer`, `LongBuffer`, `ShortBuffer`

Returned By: `Buffer.{clear(), flip(), limit(), mark(), position(), reset(), rewind()}`

BufferOverflowException

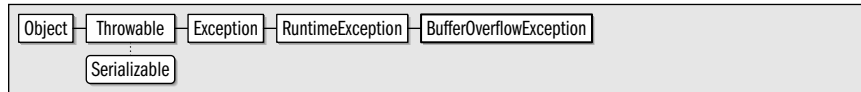
BufferOverflowException

Java 1.4

java.nio

serializable unchecked

This exception signals that a relative `put()` operation on a buffer could not be completed because the number of elements that need to be written exceeds the number of remaining elements between the buffer's position and its limit.



```
public class BufferOverflowException extends RuntimeException {  
    // Public Constructors  
    public BufferOverflowException();  
}
```

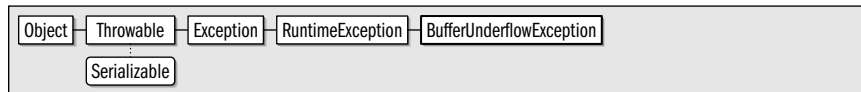
BufferUnderflowException

Java 1.4

java.nio

serializable unchecked

This exception signals that a relative `get()` operation on a buffer could not be completed because the number of elements that need to be read exceeds the number of remaining elements between the buffer's position and its limit.



```
public class BufferUnderflowException extends RuntimeException {  
    // Public Constructors  
    public BufferUnderflowException();  
}
```

ByteBuffer

Java 1.4

java.nio

comparable

`ByteBuffer` holds a sequence of bytes for use in an I/O operation. `ByteBuffer` is an abstract class, so you cannot instantiate one by calling a constructor. Instead, you must use `allocate()`, `allocateDirect()`, or `wrap()`.

`allocate()` returns a `ByteBuffer` with the specified capacity. The position of this new buffer is 0, and its limit is set to its capacity. `allocateDirect()` is like `allocate()` except that it attempts to allocate a buffer that the underlying operating system can use directly. Such direct buffers may be substantially more efficient than normal buffers for low-level I/O operations but may also have significantly larger allocation costs.

If you have already allocated an array of bytes, you can use the `wrap()` method to create a `ByteBuffer` that uses the byte array to store it. In the one-argument version of `wrap()`, you specify only the array; the buffer capacity and limit are set to the array length, and the position is set to 0. In the other form of `wrap()`, you specify the array as well as an offset and length that specify a portion of that array. The capacity of the resulting `ByteBuffer` is again set to the total array length, but its position is set to the specified offset, and its limit is set to the offset plus length.

Once you have obtained a `ByteBuffer`, you can use the various `get()` and `put()` methods to read data from it or write data into it. Several versions of these methods exist to read and write single bytes or arrays of bytes. The single-byte methods come in two forms. Relative `get()` and `put()` methods query or set the byte at the current position and then increment the position. The absolute forms of the methods take an additional argument that specifies the buffer element that will be read or written and do not affect the buffer

position. Two other relative forms of the `get()` method exist to read a sequence of bytes (starting at and incrementing the buffer's position) into a specified byte array or a specified subarray. These methods throw a `BufferUnderflowException` if there are not enough bytes left in the buffer. Two relative forms of the `put()` method copy bytes from a specified array or subarray into the buffer (starting at and incrementing the buffer's position). They throw a `BufferOverflowException` if there is not enough room left in the buffer to hold the bytes. One final form of the `put()` method transfers all the remaining bytes from one `ByteBuffer` into this buffer, incrementing the positions of both buffers.

In addition to the `get()` and `put()` methods, `ByteBuffer` also defines another operation that affects the buffer's content. `compact()` discards any bytes before the buffer position and copies all bytes between the position and limit to the beginning of the buffer. The position is then set to the new limit, and the limit is set to the capacity. This method compacts a buffer by discarding elements that have already been read, and then prepares the buffer for appending new elements to those that remain.

All `Buffer` subclasses, such as `CharBuffer`, `IntBuffer`, and `FloatBuffer`, have analogous methods that are just like the `get()` and `put()` methods except that they operate on different data types. `ByteBuffer` is unique among `Buffer` subclasses in that it has additional methods for reading and writing values of other primitive types from and into the byte buffer. These methods have names such as `getInt()` and `putChar()`, and there are methods for all primitive types except `byte` and `boolean`. Each method reads or writes a single primitive value. Like the `get()` and `put()` methods, they come in relative and absolute variations. The relative methods start with the byte at the buffer's position and increment the position by the appropriate number of bytes (two bytes for a `char`, four bytes for an `int`, eight bytes for a `double`, etc.). The absolute methods take a buffer index (a byte index that is not multiplied by the size of the primitive value) as an argument and do not modify the buffer position. The encoding of multibyte primitive values into a byte buffer can be done most-significant byte to least-significant byte (big-endian byte order) or the reverse (little-endian byte order). The byte order used by these primitive-type `get()` and `put()` methods is specified by a `ByteOrder` object. The byte order for a `ByteBuffer` can be queried and set with the two forms of the `order()` method. The default byte order for all newly created `ByteBuffer` objects is `ByteOrder.BIG_ENDIAN`.

Other methods unique to `ByteBuffer()` are a set of methods that allow a buffer of bytes to be viewed as a buffer of other primitive types. `asCharBuffer()`, `asIntBuffer()` and related methods return "view buffers" that allow the bytes between the position and the limit of the underlying `ByteBuffer` to be viewed as a sequence of characters, integers, or other primitive values. The returned buffers have position, limit, and mark values that are independent of those of the underlying buffer. The initial position of the returned buffer is 0, and the limit and capacity are the number of bytes between the position and limit of the original buffer divided by the size in bytes of the relevant primitive type (two for `char` and `short`, four for `int` and `float`, and eight for `long` and `double`). Note that the returned view buffer is a view of the bytes between the position and limit of the byte buffer. Subsequent changes to the position and limit of the byte buffer do not change the size of the view buffer, but changes to the bytes themselves do change the values that are viewed through the view buffer. View buffers use the byte ordering that was current in the byte buffer when they were created; subsequent changes to the byte order of the byte buffer do not affect the view buffer. If the underlying byte buffer is direct, then the returned buffer is also direct; this is important because `ByteBuffer` is the only buffer class with an `allocateDirect()` method.

`ByteBuffer` defines some additional methods, which, like the `get()` and `put()` methods, have analogs in all `Buffer` subclasses. `duplicate()` returns a new buffer that shares the content with the original buffer. The two buffers have independent position, limit, and mark values, although the duplicate buffer starts off with the same values as the original buffer. The duplicate buffer is direct if the original is direct and is read-only if the

ByteBuffer

original is read-only. The buffers share content, and content changes made to either buffer are visible through the other. `asReadOnlyBuffer()` is like `duplicate()` except that the returned buffer is read-only, and all of its `put()` and related methods throw a `ReadOnlyBufferException`. `slice()` is also somewhat like `duplicate()` except that the returned buffer represents only the content between the current position and limit. The returned buffer has a position of 0, a limit and capacity equal to the number of remaining elements in this buffer, and an undefined mark. `isDirect()` is a simple method that returns `true` if a buffer is a direct buffer and `false` otherwise. If this buffer has a backing array and is not a read-only buffer (e.g., if it was created with the `allocate()` or `wrap()` methods), then `hasArray()` returns `true`, `array()` returns the backing array, and `arrayOffset()` returns the offset within that array of the first element of the buffer. If `hasArray()` returns `false`, then `array()` and `arrayOffset()` may throw an `UnsupportedOperationException` or a `ReadOnlyBufferException`.

Finally, `ByteBuffer` and other `Buffer` subclasses override several standard object methods. The `equals()` method compares the elements between the position and limit of two buffers and returns `true` only if there are the same number and have the same value. Note that elements before the position of the buffer are not considered. The `hashCode()` method is implemented to match the `equals()` method: the hashcode is based upon only the elements between the position and limit of the buffer. This means that the hashcode changes if either the contents or position of the buffer changes. This means that instances of `ByteBuffer` and other `Buffer` subclasses are not usually useful as keys for hashtables or `java.util.Map` objects. `toString()` returns a string summary of the buffer, but the precise contents of the string are unspecified. `ByteBuffer` and each of the other `Buffer` subclasses also implement the `Comparable` interface and define a `compareTo()` method that performs an element-by-element comparison operation on the buffer elements between the position and the limit of the buffer.



```
public abstract class ByteBuffer extends Buffer implements Comparable {
// No Constructor
// Public Class Methods
    public static ByteBuffer allocate(int capacity);
    public static ByteBuffer allocateDirect(int capacity);
    public static ByteBuffer wrap(byte[] array);
    public static ByteBuffer wrap(byte[] array, int offset, int length);
// Property Accessor Methods (by property name)
    public abstract char getChar();
    public abstract char getChar(int index);
    public abstract boolean isDirect();
    public abstract double getDouble();
    public abstract double getDouble(int index);
    public abstract float getFloat();
    public abstract float getFloat(int index);
    public abstract int getInt();
    public abstract int getInt(int index);
    public abstract long getLong();
    public abstract long getLong(int index);
    public abstract short getShort();
    public abstract short getShort(int index);
// Public Instance Methods
    public final byte[] array();
    public final int arrayOffset();
    public abstract CharBuffer asCharBuffer();
```

```

public abstract DoubleBuffer asDoubleBuffer();
public abstract FloatBuffer asFloatBuffer();
public abstract IntBuffer asIntBuffer();
public abstract LongBuffer asLongBuffer();
public abstract ByteBuffer asReadOnlyBuffer();
public abstract ShortBuffer asShortBuffer();
public abstract ByteBuffer compact();
public abstract ByteBuffer duplicate();
public abstract byte get();
public abstract byte get(int index);
public ByteBuffer get(byte[] dst);
public ByteBuffer get(byte[] dst, int offset, int length);
public final boolean hasArray();
public final ByteOrder order();
public final ByteBuffer order(ByteOrder bo);
public ByteBuffer put(ByteBuffer src);
public abstract ByteBuffer put(byte b);
public final ByteBuffer put(byte[] src);
public abstract ByteBuffer put(int index, byte b);
public ByteBuffer put(byte[] src, int offset, int length);
public abstract ByteBuffer putChar(char value);
public abstract ByteBuffer putChar(int index, char value);
public abstract ByteBuffer putDouble(double value);
public abstract ByteBuffer putDouble(int index, double value);
public abstract ByteBuffer putFloat(float value);
public abstract ByteBuffer putFloat(int index, float value);
public abstract ByteBuffer putInt(int value);
public abstract ByteBuffer putInt(int index, int value);
public abstract ByteBuffer putLong(long value);
public abstract ByteBuffer putLong(int index, long value);
public abstract ByteBuffer putShort(short value);
public abstract ByteBuffer putShort(int index, short value);
public abstract ByteBuffer slice();
// Methods Implementing Comparable
public int compareTo(Object ob);
// Public Methods Overriding Object
public boolean equals(Object ob);
public int hashCode();
public String toString();
}

```

Subclasses: MappedByteBuffer

Passed To: Too many methods to list.

Returned By: Too many methods to list.

ByteOrder

Java 1.4

java.nio

This class is a type-safe enumeration of byte orders used by the `ByteBuffer` class. The two constant fields define the two legal byte order values. `BIG_ENDIAN` byte order means most-significant byte first. `LITTLE_ENDIAN` means least-significant byte first. The static `nativeOrder()` method returns whichever of these two constants represents the native byte order of the underlying operating system and hardware. Finally, the `toString()` method returns the string “BIG_ENDIAN” or “LITTLE_ENDIAN”.

ByteOrder

```
public final class ByteOrder {  
    // No Constructor  
    // Public Constants  
    public static final ByteOrder BIG_ENDIAN;  
    public static final ByteOrder LITTLE_ENDIAN;  
    // Public Class Methods  
    public static ByteOrder nativeOrder();  
    // Public Methods Overriding Object  
    public String toString();  
}
```

Passed To: ByteBuffer.order(), javax.imageio.stream.ImageInputStream.setByteOrder(),
javax.imageio.stream.ImageInputStreamImpl.setByteOrder()

Returned By: ByteBuffer.order(), ByteOrder.nativeOrder(), CharBuffer.order(), DoubleBuffer.order(),
FloatBuffer.order(), IntBuffer.order(), LongBuffer.order(), ShortBuffer.order(),
javax.imageio.stream.ImageInputStream.getByteOrder(),
javax.imageio.stream.ImageInputStreamImpl.getByteOrder()

Type Of: ByteOrder.{BIG_ENDIAN, LITTLE_ENDIAN},
javax.imageio.stream.ImageInputStreamImpl.byteOrder

CharBuffer

Java 1.4

java.nio

comparable

CharBuffer holds a sequence of Unicode character values for use in an I/O operation. Most of this class's methods are directly analogous to methods defined by ByteBuffer except that they use char and char[] argument and return values instead of byte and byte[] values. See ByteBuffer for details.

In addition to the ByteBuffer analogs, this class also implements the java.lang.CharSequence interface so it can be used with java.util.regex regular expression operations or anywhere else a CharSequence is expected.

Note that CharBuffer is an abstract class and does not define a constructor. There are three ways to obtain a CharBuffer:

- By calling the static allocate() method. Note that there is no allocateDirect() method as there is for ByteBuffer.
- By calling one of the static wrap() methods to create a CharBuffer that uses the specified char array or CharSequence for its content. Note that wrapping a CharSequence results in a read-only CharBuffer.
- By calling the asCharBuffer() method of a ByteBuffer to obtain a CharBuffer view of the underlying bytes. If the underlying ByteBuffer is direct, then the CharBuffer view will also be direct.

Note that this class holds a sequence of 16-bit Unicode characters and does not represent text in any other encoding. Classes in the java.nio.charset package can be used to encode a CharBuffer of Unicode characters into a ByteBuffer or decode the bytes in a ByteBuffer into a CharBuffer of Unicode text.



```
public abstract class CharBuffer extends Buffer implements CharSequence, Comparable {  
    // No Constructor
```



```
// Public Class Methods
public static CharBuffer allocate(int capacity);
public static CharBuffer wrap(CharSequence csq);
public static CharBuffer wrap(char[] array);
public static CharBuffer wrap(char[] array, int offset, int length);
public static CharBuffer wrap(CharSequence csq, int start, int end);

// Public Instance Methods
public final char[] array();
public final int arrayOffset();
public abstract CharBuffer asReadOnlyBuffer();
public abstract CharBuffer compact();
public abstract CharBuffer duplicate();
public abstract char get();
public abstract char get(int index);
public CharBuffer get(char[] dst);
public CharBuffer get(char[] dst, int offset, int length);
public final boolean hasArray();
public abstract boolean isDirect();
public abstract ByteOrder order();
public final CharBuffer put(String src);
public abstract CharBuffer put(char c);
public CharBuffer put(CharBuffer src);
public final CharBuffer put(char[] src);
public abstract CharBuffer put(int index, char c);
public CharBuffer put(char[] src, int offset, int length);
public CharBuffer put(String src, int start, int end);
public abstract CharBuffer slice();

// Methods Implementing CharSequence
public final char charAt(int index);
public final int length();
public abstract CharSequence subSequence(int start, int end);
public String toString();

// Methods Implementing Comparable
public int compareTo(Object ob);

// Public Methods Overriding Object
public boolean equals(Object ob);
public int hashCode();
}
```

Passed To: CharBuffer.put(), java.nio.charset.Charset.encode(),
 java.nio.charset.CharsetDecoder.{decode(), decodeLoop(), flush(), implFlush()},
 java.nio.charset.CharsetEncoder.{encode(), encodeLoop()}

Returned By: Too many methods to list.

DoubleBuffer

Java 1.4

java.nio

comparable

DoubleBuffer holds a sequence of double values for use in an I/O operation. Most of this class's methods are directly analogous to methods defined by ByteBuffer except that they use double and double[] argument and return values instead of byte and byte[] values. See ByteBuffer for details.

DoubleBuffer

DoubleBuffer is abstract and has no constructor. Create one by calling the static `allocate()` or `wrap()` methods, which are also analogs of `ByteBuffer` methods, or create a view `DoubleBuffer` by calling the `asDoubleBuffer()` method of an underlying `ByteBuffer`.



```
public abstract class DoubleBuffer extends Buffer implements Comparable {
// No Constructor
// Public Class Methods
    public static DoubleBuffer allocate(int capacity);
    public static DoubleBuffer wrap(double[] array);
    public static DoubleBuffer wrap(double[] array, int offset, int length);
// Public Instance Methods
    public final double[] array();
    public final int arrayOffset();
    public abstract DoubleBuffer asReadOnlyBuffer();
    public abstract DoubleBuffer compact();
    public abstract DoubleBuffer duplicate();
    public abstract double get();
    public abstract double get(int index);
    public DoubleBuffer get(double[] dst);
    public DoubleBuffer get(double[] dst, int offset, int length);
    public final boolean hasArray();
    public abstract boolean isDirect();
    public abstract ByteOrder order();
    public DoubleBuffer put(DoubleBuffer src);
    public abstract DoubleBuffer put(double d);
    public final DoubleBuffer put(double[] src);
    public abstract DoubleBuffer put(int index, double d);
    public DoubleBuffer put(double[] src, int offset, int length);
    public abstract DoubleBuffer slice();
// Methods Implementing Comparable
    public int compareTo(Object ob);
// Public Methods Overriding Object
    public boolean equals(Object ob);
    public int hashCode();
    public String toString();
}
```

Passed To: DoubleBuffer.put()

Returned By: ByteBuffer.asDoubleBuffer(), DoubleBuffer.{allocate(), asReadOnlyBuffer(), compact(), duplicate(), get(), put(), slice(), wrap()}

FloatBuffer

Java 1.4

java.nio

comparable

FloatBuffer holds a sequence of `float` values for use in an I/O operation. Most of the methods of this class are directly analogous to methods defined by `ByteBuffer` except that they use `float` and `float[]` argument and return values instead of `byte` and `byte[]` values. See `ByteBuffer` for details.

FloatBuffer is abstract and has no constructor. Create one by calling the static `allocate()` or `wrap()` methods, which are also analogs of `ByteBuffer` methods, or create a view `FloatBuffer` by calling the `asFloatBuffer()` method of an underlying `ByteBuffer`.



```

public abstract class FloatBuffer extends Buffer implements Comparable {
    // No Constructor
    // Public Class Methods
    public static FloatBuffer allocate(int capacity);
    public static FloatBuffer wrap(float[] array);
    public static FloatBuffer wrap(float[] array, int offset, int length);
    // Public Instance Methods
    public final float[] array();
    public final int arrayOffset();
    public abstract FloatBuffer asReadOnlyBuffer();
    public abstract FloatBuffer compact();
    public abstract FloatBuffer duplicate();
    public abstract float get();
    public abstract float get(int index);
    public FloatBuffer get(float[] dst);
    public FloatBuffer get(float[] dst, int offset, int length);
    public final boolean hasArray();
    public abstract boolean isDirect();
    public abstract ByteOrder order();
    public FloatBuffer put(FloatBuffer src);
    public abstract FloatBuffer put(float f);
    public final FloatBuffer put(float[] src);
    public abstract FloatBuffer put(int index, float f);
    public FloatBuffer put(float[] src, int offset, int length);
    public abstract FloatBuffer slice();
    // Methods Implementing Comparable
    public int compareTo(Object ob);
    // Public Methods Overriding Object
    public boolean equals(Object ob);
    public int hashCode();
    public String toString();
}

```

Passed To: `FloatBuffer.put()`

Returned By: `ByteBuffer.asFloatBuffer()`, `FloatBuffer.allocate()`, `asReadOnlyBuffer()`, `compact()`, `duplicate()`, `get()`, `put()`, `slice()`, `wrap()`

IntBuffer

Java 1.4

java.nio

comparable

`IntBuffer` holds a sequence of `int` values for use in an I/O operation. Most of the methods of this class are directly analogous to methods defined by `ByteBuffer` except that they use `int` and `int[]` argument and return values instead of `byte` and `byte[]` values. See `ByteBuffer` for details.

IntBuffer

IntBuffer is abstract and has no constructor. Create one by calling the static `allocate()` or `wrap()` methods, which are also analogs of `ByteBuffer` methods, or create a view `IntBuffer` by calling the `asIntBuffer()` method of an underlying `ByteBuffer`.



```
public abstract class IntBuffer extends Buffer implements Comparable {
// No Constructor
// Public Class Methods
    public static IntBuffer allocate(int capacity);
    public static IntBuffer wrap(int[ ] array);
    public static IntBuffer wrap(int[ ] array, int offset, int length);
// Public Instance Methods
    public final int[ ] array();
    public final int arrayOffset();
    public abstract IntBuffer asReadOnlyBuffer();
    public abstract IntBuffer compact();
    public abstract IntBuffer duplicate();
    public abstract int get();
    public abstract int get(int index);
    public IntBuffer get(int[ ] dst);
    public IntBuffer get(int[ ] dst, int offset, int length);
    public final boolean hasArray();
    public abstract boolean isDirect();
    public abstract ByteOrder order();
    public IntBuffer put(IntBuffer src);
    public abstract IntBuffer put(int i);
    public final IntBuffer put(int[ ] src);
    public abstract IntBuffer put(int index, int i);
    public IntBuffer put(int[ ] src, int offset, int length);
    public abstract IntBuffer slice();
// Methods Implementing Comparable
    public int compareTo(Object ob);
// Public Methods Overriding Object
    public boolean equals(Object ob);
    public int hashCode();
    public String toString();
}
```

Passed To: `IntBuffer.put()`

Returned By: `ByteBuffer.asIntBuffer()`, `IntBuffer.allocate()`, `asReadOnlyBuffer()`, `compact()`, `duplicate()`, `get()`, `put()`, `slice()`, `wrap()`

InvalidMarkException

Java 1.4

java.nio

serializable unchecked

This exception signals that a buffer's position cannot be `reset()` because there is no mark defined.



```
public class InvalidMarkException extends IllegalStateException {
// Public Constructors
```

```

    public InvalidMarkException();
}

```

LongBuffer

Java 1.4

java.nio

comparable

LongBuffer holds a sequence of long values for use in an I/O operation. Most of the methods of this class are directly analogous to methods defined by ByteBuffer except that they use long and long[] argument and return values instead of byte and byte[] values. See ByteBuffer for details.

LongBuffer is abstract and has no constructor. Create one by calling the static allocate() or wrap() methods, which are also analogs of ByteBuffer methods, or create a “view” LongBuffer by calling the asLongBuffer() method of an underlying ByteBuffer.



```

public abstract class LongBuffer extends Buffer implements Comparable {

```

```

    // No Constructor

```

```

    // Public Class Methods

```

```

        public static LongBuffer allocate(int capacity);

```

```

        public static LongBuffer wrap(long[] array);

```

```

        public static LongBuffer wrap(long[] array, int offset, int length);

```

```

    // Public Instance Methods

```

```

        public final long[] array();

```

```

        public final int arrayOffset();

```

```

        public abstract LongBuffer asReadOnlyBuffer();

```

```

        public abstract LongBuffer compact();

```

```

        public abstract LongBuffer duplicate();

```

```

        public abstract long get();

```

```

        public abstract long get(int index);

```

```

        public LongBuffer get(long[] dst);

```

```

        public LongBuffer get(long[] dst, int offset, int length);

```

```

        public final boolean hasArray();

```

```

        public abstract boolean isDirect();

```

```

        public abstract ByteOrder order();

```

```

        public LongBuffer put(LongBuffer src);

```

```

        public abstract LongBuffer put(long l);

```

```

        public final LongBuffer put(long[] src);

```

```

        public abstract LongBuffer put(int index, long l);

```

```

        public LongBuffer put(long[] src, int offset, int length);

```

```

        public abstract LongBuffer slice();

```

```

    // Methods Implementing Comparable

```

```

        public int compareTo(Object ob);

```

```

    // Public Methods Overriding Object

```

```

        public boolean equals(Object ob);

```

```

        public int hashCode();

```

```

        public String toString();
    }

```

Passed To: LongBuffer.put()

Returned By: ByteBuffer.asLongBuffer(), LongBuffer.{allocate(), asReadOnlyBuffer(), compact(), duplicate(), get(), put(), slice(), wrap() }

MappedByteBuffer

Java 1.4

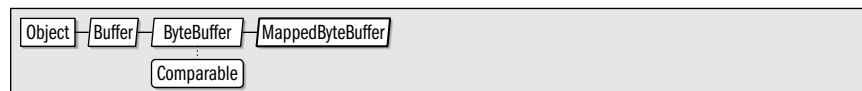
java.nio

comparable

This class is a `ByteBuffer` that represents a memory-mapped portion of a file. Create a `MappedByteBuffer` by calling the `map()` method of a `java.nio.channels.FileChannel`. All `MappedByteBuffer` buffers are direct buffers.

`isLoaded()` returns a hint as to whether the contents of the buffer are currently in primary memory (as opposed to resident on disk). If it returns `true`, then operations on the buffer will probably execute quickly. The `load()` method requests, but does not require, that the operating system load the buffer contents into primary memory. It is not guaranteed to succeed. For buffers mapped in read/write mode, the `force()` method outputs any changes that have been made to the buffer contents to the underlying file. If the file is on a local device, then it is guaranteed to be updated before `force()` returns. No such guarantees can be made for mapped network files.

Note that the underlying file of a `MappedByteBuffer` may be shared, which means that the contents of such a buffer can change asynchronously if the contents of the file are modified by another thread or another process (such as asynchronous changes to the underlying file may or may not be visible through the buffer; this is platform-dependent, and should not be relied on). Furthermore, if another thread or process truncates the file, some or all of the elements of the buffer may no longer map to any content of the file. An attempt to read or write such an inaccessible element of the buffer will cause an implementation-defined exception, either immediately or at some later time.



```

public abstract class MappedByteBuffer extends ByteBuffer {
    // No Constructor
    // Public Instance Methods
    public final MappedByteBuffer force();
    public final boolean isLoaded();
    public final MappedByteBuffer load();
}
  
```

Returned By: `MappedByteBuffer.{force(), load()}`, `java.nio.channels.FileChannel.map()`

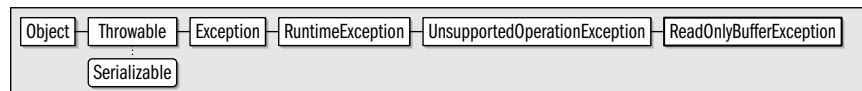
ReadOnlyBufferException

Java 1.4

java.nio

serializable unchecked

This exception signals that a buffer is read-only and that its `put()` or `compact()` methods are not allowed to modify the buffer contents.



```

public class ReadOnlyBufferException extends UnsupportedOperationException {
    // Public Constructors
    public ReadOnlyBufferException();
}
  
```

ShortBuffer

Java 1.4

java.nio

comparable

`ShortBuffer` holds a sequence of `short` values for use in an I/O operation. Most of the methods of this class are directly analogous to methods defined by `ByteBuffer` except that

they use `short` and `short[]` argument and return values instead of `byte` and `byte[]` values. See `ByteBuffer` for details.

`ShortBuffer` is abstract and has no constructor. Create one by calling the static `allocate()` or `wrap()` methods, which are also analogs of `ByteBuffer` methods. Or, create a “view” `ShortBuffer` by calling the `asShortBuffer()` method of an underlying `ByteBuffer`.



```

public abstract class ShortBuffer extends Buffer implements Comparable {
    // No Constructor
    // Public Class Methods
    public static ShortBuffer allocate(int capacity);
    public static ShortBuffer wrap(short[] array);
    public static ShortBuffer wrap(short[] array, int offset, int length);
    // Public Instance Methods
    public final short[] array();
    public final int arrayOffset();
    public abstract ShortBuffer asReadOnlyBuffer();
    public abstract ShortBuffer compact();
    public abstract ShortBuffer duplicate();
    public abstract short get();
    public abstract short get(int index);
    public ShortBuffer get(short[] dst);
    public ShortBuffer get(short[] dst, int offset, int length);
    public final boolean hasArray();
    public abstract boolean isDirect();
    public abstract ByteOrder order();
    public ShortBuffer put(ShortBuffer src);
    public abstract ShortBuffer put(short s);
    public final ShortBuffer put(short[] src);
    public abstract ShortBuffer put(int index, short s);
    public ShortBuffer put(short[] src, int offset, int length);
    public abstract ShortBuffer slice();
    // Methods Implementing Comparable
    public int compareTo(Object ob);
    // Public Methods Overriding Object
    public boolean equals(Object ob);
    public int hashCode();
    public String toString();
}
  
```

Passed To: `ShortBuffer.put()`

Returned By: `ByteBuffer.asShortBuffer()`, `ShortBuffer.allocate()`, `ShortBuffer.asReadOnlyBuffer()`, `compact()`, `duplicate()`, `get()`, `put()`, `slice()`, `wrap()`

Package java.nio.channels

Java 1.4

This package is at the heart of the NIO API. A *channel* is a communication channel for transferring bytes from or to a `java.nio.ByteBuffer`. Channels serve a similar purpose in the `InputStream` and `OutputStream` classes of the `java.io` package but are completely unrelated to those classes and provide important features not available with the `java.io` API. The `Channels` class defines methods that bridge the `java.io` and `java.nio.channels` APIs by returning channels based on streams and streams based on channels.

Package java.nio.channels

The `Channel` interface simply defines methods for testing whether a channel is open and for closing a channel. The other interfaces in the package extend `Channel` and define `read()` and `write()` methods for reading bytes from the channel into one or more byte buffers and for writing bytes from one or more byte buffers to the channel.

The `FileChannel` class defines a channel-based API for reading and writing from files and provides other important file functionality, such as file locking and memory mapping, that is not available through the `java.io` package. `SocketChannel`, `ServerSocketChannel`, and `DatagramChannel` are channels for communication over a network. `Pipe` defines two inner classes that use the channel abstraction for communication between threads.

The network and pipe channels are all subclasses of the `SelectableChannel` class and may be put into nonblocking mode, in which calls to `read()` and `write()` return immediately, even if the channel is not ready for reading or writing. Nonblocking I/O and networking are not possible using the stream abstraction of the `java.io` and `java.net` packages and are perhaps the most important new features of the `java.nio` API. The `Selector` class is crucial in the efficient use of nonblocking channels; it allows a program interested in I/O operations to register on several different channels at once. A call to the `select()` method of a `Selector` will block until one of those channels becomes ready for I/O, and will then wake up. This technique is important for writing scalable, high-performance network servers. See `Selector` and `SelectionKey` for details.

Finally, this package allows for fine-grained error handling by defining a large number of exception classes, several of which may be thrown by only a single method within the `java.nio` API.

Interfaces:

```
public interface ByteChannel extends ReadableByteChannel, WritableByteChannel;  
public interface Channel;  
public interface GatheringByteChannel extends WritableByteChannel;  
public interface InterruptibleChannel extends Channel;  
public interface ReadableByteChannel extends Channel;  
public interface ScatteringByteChannel extends ReadableByteChannel;  
public interface WritableByteChannel extends Channel;
```

Classes:

```
public final class Channels;  
public abstract class DatagramChannel extends java.nio.channels.spi.AbstractSelectableChannel  
    implements ByteChannel, GatheringByteChannel, ScatteringByteChannel;  
public abstract class FileChannel extends java.nio.channels.spi.AbstractInterruptibleChannel  
    implements ByteChannel, GatheringByteChannel, ScatteringByteChannel;  
public static class FileChannel.MapMode;  
public abstract class FileLock;  
public abstract class Pipe;  
public abstract static class Pipe.SinkChannel extends java.nio.channels.spi.AbstractSelectableChannel  
    implements GatheringByteChannel, WritableByteChannel;  
public abstract static class Pipe.SourceChannel  
    extends java.nio.channels.spi.AbstractSelectableChannel  
    implements ReadableByteChannel, ScatteringByteChannel;  
public abstract class SelectableChannel extends java.nio.channels.spi.AbstractInterruptibleChannel  
    implements Channel;  
public abstract class SelectionKey;  
public abstract class Selector;  
public abstract class ServerSocketChannel extends java.nio.channels.spi.AbstractSelectableChannel;
```


public abstract class SocketChannel extends **java.nio.channels.spi.AbstractSelectableChannel**
implements **ByteChannel**, **GatheringByteChannel**, **ScatteringByteChannel**;

Exceptions:

```
public class AlreadyConnectedException extends IllegalStateException;
public class CancelledKeyException extends IllegalStateException;
public class ClosedChannelException extends java.io.IOException;
    public class AsynchronousCloseException extends ClosedChannelException;
        public class ClosedByInterruptException extends AsynchronousCloseException;
public class ClosedSelectorException extends IllegalStateException;
public class ConnectionPendingException extends IllegalStateException;
public class FileLockInterruptedException extends java.io.IOException;
public class IllegalBlockingModeException extends IllegalStateException;
public class IllegalSelectorException extends IllegalArgumentException;
public class NoConnectionPendingException extends IllegalStateException;
public class NonReadableChannelException extends IllegalStateException;
public class NonWritableChannelException extends IllegalStateException;
public class NotYetBoundException extends IllegalStateException;
public class NotYetConnectedException extends IllegalStateException;
public class OverlappingFileLockException extends IllegalStateException;
public class UnresolvedAddressException extends IllegalArgumentException;
public class UnsupportedAddressTypeException extends IllegalArgumentException;
```

AlreadyConnectedException

Java 1.4

java.nio.channels

serializable unchecked

This exception is thrown by a call to `connect()` on a `SocketChannel` that is already connected.



```
public class AlreadyConnectedException extends IllegalStateException {
    // Public Constructors
    public AlreadyConnectedException();
}
```

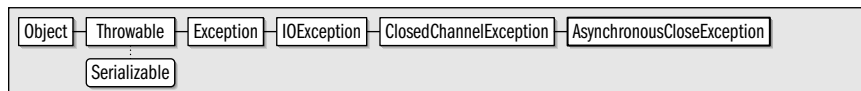
AsynchronousCloseException

Java 1.4

java.nio.channels

serializable checked

This exception signals the termination of a blocked I/O operation because another thread closed the channel asynchronously. See also `ClosedByInterruptException`.



```
public class AsynchronousCloseException extends ClosedChannelException {
    // Public Constructors
    public AsynchronousCloseException();
}
```

Subclasses: `ClosedByInterruptException`

AsynchronousCloseException

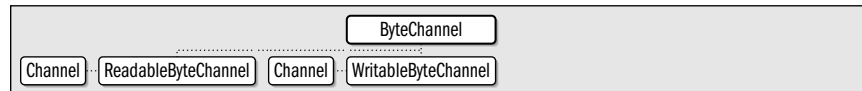
Thrown By: java.nio.channels.spi.AbstractInterruptibleChannel.end()

ByteChannel

Java 1.4

java.nio.channels

This interface extends `ReadableByteChannel` and `WritableByteChannel` but adds no methods or constants of its own. It exists simply as a convenient way to unify the two interfaces.



```
public interface ByteChannel extends ReadableByteChannel, WritableByteChannel {
}
```

Implementations: DatagramChannel, FileChannel, SocketChannel

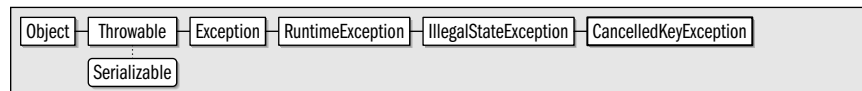
CancelledKeyException

Java 1.4

java.nio.channels

serializable unchecked

This exception signals an attempt to use a `SelectionKey` with a `cancel()` method that has previously been called.



```
public class CancelledKeyException extends IllegalStateException {
    // Public Constructors
    public CancelledKeyException();
}
```

Channel

Java 1.4

java.nio.channels

This interface defines a communication channel for input and output. The `Channel` interface is a high-level generic interface extended by more specific interfaces, such as `ReadableByteChannel` and `WritableByteChannel`. `Channel` defines only two methods: `isOpen()` determines whether a channel is open, and `close()` closes a channel. Channels are open when they are first created. Once closed, a channel remains closed forever, and no further I/O operations may go through it.

Many channel implementations are interruptible and asynchronously closeable and implement the `InterruptibleChannel` interface to advertise this fact. See `InterruptibleChannel` for details.

```
public interface Channel {
    // Public Instance Methods
    public abstract void close() throws java.io.IOException;
    public abstract boolean isOpen();
}
```

Implementations: `InterruptibleChannel`, `ReadableByteChannel`, `SelectableChannel`, `WritableByteChannel`, `java.nio.channels.spi.AbstractInterruptibleChannel`

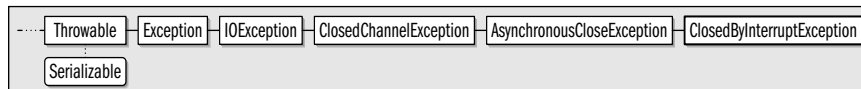
Channels**Java 1.4****java.nio.channels**

This class defines static methods that provide a bridge between the byte stream and character stream classes of the `java.io` package and the channel classes of `java.nio.channels`. `Channels` is never intended to be instantiated; it serves solely as a placeholder for static methods. These methods create byte channels based on `java.io` byte streams and `java.io` byte streams based on byte channels. Note that the channel objects returned by the `newChannel()` methods may not implement `InterruptibleChannel`, and so may not be asynchronously closeable and interruptible like other channel classes in this package. `Channels` also defines methods that create character streams (`java.io.Reader` and `java.io.Writer`) based on the combination of a byte channel and a character encoding. The encoding may be specified by charset name, or with a `CharsetDecoder` or `CharsetEncoder`. See `java.nio.charset`.

```
public final class Channels {
    // No Constructor
    // Public Class Methods
    public static ReadableByteChannel newChannel(java.io.InputStream in);
    public static WritableByteChannel newChannel(java.io.OutputStream out);
    public static java.io.InputStream newInputStream(ReadableByteChannel ch);
    public static java.io.OutputStream newOutputStream(WritableByteChannel ch);
    public static java.io.Reader newReader(ReadableByteChannel ch, String csName);
    public static java.io.Reader newReader(ReadableByteChannel ch, java.nio.charset.CharsetDecoder dec,
                                           int minBufferCap);
    public static java.io.Writer newWriter(WritableByteChannel ch, String csName);
    public static java.io.Writer newWriter(WritableByteChannel ch, java.nio.charset.CharsetEncoder enc,
                                           int minBufferCap);
}
```

ClosedByInterruptException**Java 1.4****java.nio.channels***serializable checked*

An exception of this type is thrown by a thread blocked in an I/O operation on a channel when another thread calls its `interrupt()` method. This exception is a subclass of `AsynchronousCloseException`, and the channel will be closed as a side effect of the thread interruption.



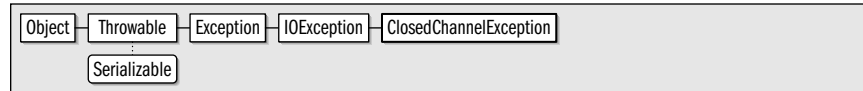
```
public class ClosedByInterruptException extends AsynchronousCloseException {
    // Public Constructors
    public ClosedByInterruptException();
}
```

ClosedChannelException**Java 1.4****java.nio.channels***serializable checked*

This exception signals an attempt to perform I/O on a channel that was closed with the `close()` method or closed for a particular type of I/O operation. (For example, a `SocketChannel` may have its read and write halves shut down independently.) Channels may be closed asynchronously, and threads blocking to complete an I/O operation will

ClosedChannelException

throw a subclass of this exception type. See `AsynchronousCloseException` and `ClosedByInterruptException`.



```
public class ClosedChannelException extends java.io.IOException {  
    // Public Constructors  
    public ClosedChannelException();  
}
```

Subclasses: `AsynchronousCloseException`

Thrown By: `SelectableChannel.register()`, `java.nio.channels.spi.AbstractSelectableChannel.register()`

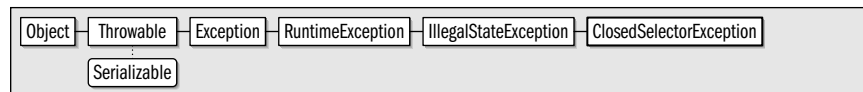
ClosedSelectorException

Java 1.4

`java.nio.channels`

serializable unchecked

This exception signals an attempt to use a `Selector` object with `close()` method that has been called.



```
public class ClosedSelectorException extends IllegalStateException {  
    // Public Constructors  
    public ClosedSelectorException();  
}
```

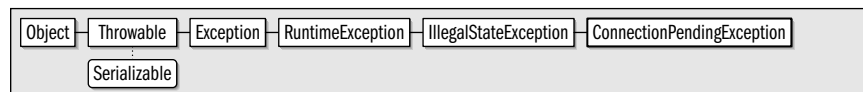
ConnectionPendingException

Java 1.4

`java.nio.channels`

serializable unchecked

This exception signals a call to the `connect()` method of a `SocketChannel` when there is already a connection pending for that channel. See `SocketChannel.isConnectionPending()`.



```
public class ConnectionPendingException extends IllegalStateException {  
    // Public Constructors  
    public ConnectionPendingException();  
}
```

DatagramChannel

Java 1.4

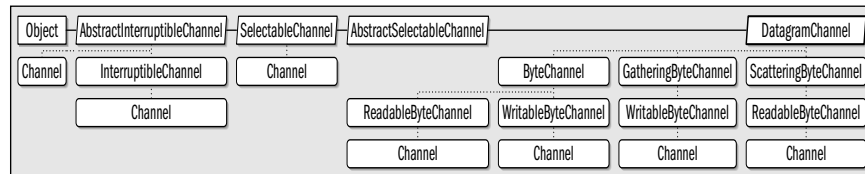
`java.nio.channels`

This class implements a communication channel based on network datagrams. Obtain a `DatagramChannel` by calling the static `open()` method. Call `socket()` to obtain the `java.net.DatagramSocket` object on which the channel is based if you need to set any socket options to control low-level networking details.

The `send()` method sends the remaining bytes of the specified `ByteBuffer` to the host and port specified in the `java.net.SocketAddress` in the form of a datagram. `receive()` does the opposite: it receives a datagram, stores its content into the specified buffer (discarding any bytes that do not fit), and returns a `SocketAddress` that specifies the sender of the datagram (or returns null if the channel was in nonblocking mode and no datagram was waiting).

Typically, the `send()` and `receive()` methods must perform security checks on each invocation to see if the application has permissions to communicate with the remote host. If your application will be using a `DatagramChannel` to exchange datagrams with a single remote host and port, use the `connect()` method to connect to a specified `SocketAddress`. The `connect()` method performs the required security checks once and allows future communication with the specified address without the overhead. Once a `DatagramChannel` is connected, you can use the standard `read()` and `write()` methods defined by the `ReadableByteChannel`, `WritableByteChannel`, `GatheringByteChannel`, and `ScatteringByteChannel` interfaces. Like the `receive()` method, the `read()` methods silently discard any received bytes that do not fit in the specified `ByteBuffer`. The `read()` and `write()` methods throw a `NotYetConnectedException` if `connect()` has not been called.

`DatagramChannel` is a `SelectableChannel`; its `validOps()` method specifies that read and write operations may be selected. `DatagramChannel` objects are thread-safe. Read and write operations may proceed concurrently, but the class ensures that only one thread may read and one thread write at a time.



```
public abstract class DatagramChannel extends java.nio.channels.spi.AbstractSelectableChannel
    implements ByteChannel, GatheringByteChannel, ScatteringByteChannel {

    // Protected Constructors
    protected DatagramChannel(java.nio.channels.spi.SelectorProvider provider);

    // Public Class Methods
    public static DatagramChannel open() throws java.io.IOException;

    // Public Instance Methods
    public abstract DatagramChannel connect(java.net.SocketAddress remote) throws java.io.IOException;
    public abstract DatagramChannel disconnect() throws java.io.IOException;
    public abstract boolean isConnected();
    public abstract java.net.SocketAddress receive(java.nio.ByteBuffer dst) throws java.io.IOException;
    public abstract int send(java.nio.ByteBuffer src, java.net.SocketAddress target) throws java.io.IOException;
    public abstract java.net.DatagramSocket socket();

    // Methods Implementing GatheringByteChannel
    public final long write(java.nio.ByteBuffer[] srcs) throws java.io.IOException;
    public abstract long write(java.nio.ByteBuffer[] srcs, int offset, int length) throws java.io.IOException;

    // Methods Implementing ReadableByteChannel
    public abstract int read(java.nio.ByteBuffer dst) throws java.io.IOException;

    // Methods Implementing ScatteringByteChannel
    public final long read(java.nio.ByteBuffer[] dsts) throws java.io.IOException;
    public abstract long read(java.nio.ByteBuffer[] dsts, int offset, int length) throws java.io.IOException;

    // Methods Implementing WritableByteChannel
    public abstract int write(java.nio.ByteBuffer src) throws java.io.IOException;

    // Public Methods Overriding SelectableChannel
    public final int validOps(); constant
}
```

Returned By: `java.net.DatagramSocket.getChannel()`, `DatagramChannel.{connect(), disconnect(), open()}`, `java.nio.channels.spi.SelectorProvider.openDatagramChannel()`

FileChannel**Java 1.4****java.nio.channels**

This class implements a communication channel for efficiently reading and writing files. It implements the standard `read()` and `write()` methods of the `ReadableByteChannel`, `WritableByteChannel`, `GatheringByteChannel` and `ScatteringByteChannel` methods. In addition, `FileChannel` provides methods for random access to the file, efficient transfer of bytes between the file and another channel, file locking, memory mapping, querying and setting the file size, and forcing buffered updates to be written to disk. (These important features are described in further detail later.) Note that since file operations do not typically block for extended periods the way network operations can, `FileChannel` does not subclass `SelectableChannel` (it is the only channel class that does not) and cannot be used with `Selector` objects.

`FileChannel` has no public constructor and no static factory methods. To obtain a `FileChannel`, first create a `FileInputStream`, `FileOutputStream`, or `RandomAccessFile` object (see the `java.io` package) and then call the `getChannel()` method of that object. If you use a `FileInputStream`, the resulting channel will allow reading but not writing, and if you use a `FileOutputStream`, the channel will allow writing but not reading. If you obtain a `FileChannel` from a `RandomAccessFile`, then the channel will allow reading, or both reading and writing, depending on the *mode* argument to the `RandomAccessFile` constructor.

A `FileChannel` has a position, or file pointer, that specifies the current point in the file. You can set or query the file position with two methods, both of which share the name `position()`. The position of a `FileChannel` and of the stream or `RandomAccessFile` from which it is derived are always the same; changing the position of the channel changes the position of the stream, and vice versa. The initial position of a `FileChannel` is the position of the stream or `RandomAccessFile` when the `getChannel()` method was called. If you create a `FileChannel` from a `FileOutputStream` that was opened in append mode, then any output to the channel always occurs at the end of the file and sets the file position to the end of the file.

Once you have a `FileChannel` object, you can use the standard `read()` and `write()` methods defined by the various channel interfaces. In addition to updating the buffer position as they read and write bytes, these methods also update the file position to or from which those bytes are written or read. These standard `read()` methods return the number of bytes actually read and return `-1` if there are no bytes left in the file to read. The `write()` methods enlarge the file if they write past the current end-of-file.

`FileChannel` also defines position-independent `read()` and `write()` methods that take a file position as an explicit argument; they read or write starting at that position of the file, and although they update the position of the `ByteBuffer`, they do not update the file position of the `FileChannel`. If the specified position is past the end-of-file, the `read()` method does not read any bytes and returns `-1`, and the `write()` method enlarges the file, leaving any bytes between the old end-of-file and the specified position undefined.

It is common to read bytes from a `FileChannel` and then immediately write them out to some other channel (such as a `SocketChannel`: think of a web server, for example), or to read bytes from a channel and immediately write them to a `FileChannel` (consider an FTP client). `FileChannel` provides two methods, `transferTo()` and `transferFrom()`, that do this efficiently, without the need for a temporary `ByteBuffer`. `transferTo()` reads up to the specified number of bytes starting at the specified location from this `FileChannel` and writes them to the specified channel. It does not alter the file position of the `FileChannel` and returns the number of bytes actually transferred. `transferFrom()` does the reverse: it reads up to the specified number of available bytes from the specified channel, writes them to this

`FileChannel` at the specified location without altering the file position of this channel, and returns the actual number of bytes transferred. For both methods, if the destination or source channel is a `FileChannel`, then the file position of that channel is updated.

The `size()` method returns the size (in bytes) of the underlying file. `truncate()` reduces the file size to the specified value, discarding any file content that exceeds that size. If the specified size is greater than or equal to the current file size, the file is unchanged. If the file position is greater than the new size of the file, the position is changed to the new size.

Use the `force()` method to force any buffered modifications to the file that will be written to the underlying storage device. If the file resides on a local device (as opposed to a network filesystem, for example), then `force()` guarantees that any changes to the file made since the channel was opened or since a previous call to `force()` were written to the device. The argument to this method is a hint as to whether file metadata (such as last modification time) will be forced out with file content. If this argument is `true`, the system will force out content and metadata. If `false`, the system may omit updates to metadata. Note that `force()` is required only to output changes made directly through the `FileChannel`. File updates made through a `MappedByteBuffer` returned by the `map()` method (described later) should be forced out with the `force()` method of `MappedByteBuffer`.

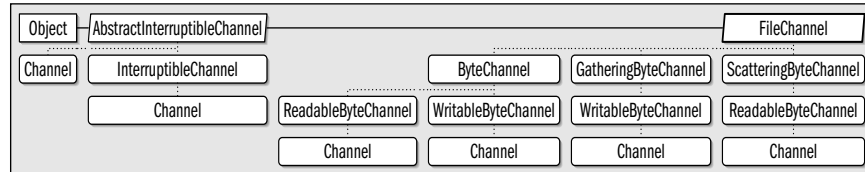
`FileChannel` defines two blocking `lock()` methods and two nonblocking `tryLock()` methods for locking a file or a region of a file against concurrent access by another program. (These methods are not suitable for preventing concurrent access to a file by two threads within the same Java virtual machine.) The no-argument versions of these methods attempt to acquire an exclusive lock on the entire file. The three-argument versions of these methods attempt to lock a specified region of the file and may acquire shared locks in addition to exclusive locks. (A shared lock prevents any other process from acquiring an exclusive lock but does not prevent other shared locks. Typically, you acquire a shared lock when reading a file that should not be concurrently updated and an exclusive lock before writing file content to ensure that no one else is trying to read it at the same time.) The `tryLock()` methods return a `FileLock` object, or `null` if there was already a conflicting lock on the file. The `lock()` methods block if there is already a conflicting lock and never return `null`. (See `FileLock` for more information about locks.) The `FileChannel` file-locking mechanism uses whatever locking capability is provided by the underlying platform. Some operating systems enforce file locking: if one process holds a lock, other processes are prevented by the operating system from accessing the file. Other operating systems merely prevent other processes from acquiring a conflicting lock; in this case, successful file locking requires the cooperation of all processes. Some operating systems do not support shared locks; on these systems, an exclusive lock is returned even when a shared lock is requested.

The `map()` method returns a `MappedByteBuffer` that represents the specified region of the file. File contents can be read directly from the buffer, and bytes placed in the buffer will be written to the file (if the mapping is done in read/write mode). The mapping represented by a `MappedByteBuffer` remains valid until the buffer is garbage collected; the buffer continues to function even if the `FileChannel` from which it was created is closed. File mappings can be done in three different modes that specify whether bytes can be written into the buffer and what happens when this is done. See `FileChannel.MapMode` for a description of these three modes.

The `map()` method relies on the memory-mapping facilities provided by the underlying operating system. This means that a number of details may vary from implementation to implementation. In particular, it is not specified whether changes to the underlying

FileChannel

file made after the call to `map()` are visible through the `MappedByteBuffer`. Typically, a mapped file is more efficient than an unmapped file, but only when the file is large.



public abstract class **FileChannel** extends java.nio.channels.spi.AbstractInterruptibleChannel implements ByteChannel, GatheringByteChannel, ScatteringByteChannel {

// Protected Constructors

protected **FileChannel**();

// Inner Classes

public static class **MapMode**;

// Public Instance Methods

public abstract void **force**(boolean *metaData*) throws java.io.IOException;

public final FileLock **lock**() throws java.io.IOException;

public abstract FileLock **lock**(long *position*, long *size*, boolean *shared*) throws java.io.IOException;

public abstract java.nio.MappedByteBuffer **map**(FileChannel.MapMode *mode*, long *position*, long *size*) throws java.io.IOException;

public abstract long **position**() throws java.io.IOException;

public abstract FileChannel **position**(long *newPosition*) throws java.io.IOException;

public abstract int **read**(java.nio.ByteBuffer *dst*, long *position*) throws java.io.IOException;

public abstract long **size**() throws java.io.IOException;

public abstract long **transferFrom**(ReadableByteChannel *src*, long *position*, long *count*) throws java.io.IOException;

public abstract long **transferTo**(long *position*, long *count*, WritableByteChannel *target*) throws java.io.IOException;

public abstract FileChannel **truncate**(long *size*) throws java.io.IOException;

public final FileLock **tryLock**() throws java.io.IOException;

public abstract FileLock **tryLock**(long *position*, long *size*, boolean *shared*) throws java.io.IOException;

public abstract int **write**(java.nio.ByteBuffer *src*, long *position*) throws java.io.IOException;

// Methods Implementing GatheringByteChannel

public final long **write**(java.nio.ByteBuffer[] *srcs*) throws java.io.IOException;

public abstract long **write**(java.nio.ByteBuffer[] *srcs*, int *offset*, int *length*) throws java.io.IOException;

// Methods Implementing ReadableByteChannel

public abstract int **read**(java.nio.ByteBuffer *dst*) throws java.io.IOException;

// Methods Implementing ScatteringByteChannel

public final long **read**(java.nio.ByteBuffer[] *dsts*) throws java.io.IOException;

public abstract long **read**(java.nio.ByteBuffer[] *dsts*, int *offset*, int *length*) throws java.io.IOException;

// Methods Implementing WritableByteChannel

public abstract int **write**(java.nio.ByteBuffer *src*) throws java.io.IOException;

Passed To: FileLock.FileLock()

Returned By: java.io.FileInputStream.getChannel(), java.io.FileOutputStream.getChannel(), java.io.RandomAccessFile.getChannel(), FileChannel.{position(), truncate()}, FileLock.channel()

FileChannel.MapMode

Java 1.4

java.nio.channels

This class defines three constants that define the legal values of the *mode* argument to the `map()` method of the `FileChannel` class:

READ_ONLY

The memory mapping is read-only. The contents of the `MappedByteBuffer` returned by the `map()` method may be read but not modified.

READ_WRITE

The memory mapping is bidirectional: the contents of the returned buffer can be modified, and any modifications will eventually be written to the underlying file. The `FileChannel` must have been created from a `java.io.RandomAccessFile` opened in read/write mode.

PRIVATE

The returned buffer may be modified, but any such changes are private to the buffer and are never written to the underlying file. This mapping mode is also known as “copy-on-write”.

```
public static class FileChannel.MapMode {
    // No Constructor
    // Public Constants
    public static final FileChannel.MapMode PRIVATE;
    public static final FileChannel.MapMode READ_ONLY;
    public static final FileChannel.MapMode READ_WRITE;
    // Public Methods Overriding Object
    public String toString();
}
```

Passed To: `FileChannel.map()`

Type Of: `FileChannel.MapMode.{PRIVATE, READ_ONLY, READ_WRITE}`

FileLock**Java 1.4****java.nio.channels**

A `FileLock` object is returned by the `lock()` and `tryLock()` methods of `FileChannel` and represents a lock on a file or a region of a file. (See `FileChannel` for more information on file locking with those methods.) When a lock is no longer required, it should be released with the `release()` method. A lock will also be released if the channel is closed, or when the virtual machine terminates. `isValid()` returns `true` if the lock has not yet been released and returns `false` if it has been released.

The `channel()`, `position()`, `size()`, and `isShared()` methods return basic information about the lock: the `FileChannel` that was locked, the region of the file that was locked, and whether the lock is shared or exclusive. If the entire file is locked, then the `size()` method returns a value (`Long.MAX_VALUE`) that is much greater than the actual file size. If the underlying operating system does not support shared locks, then `isShared()` may return `false` even if a shared lock was requested. `overlaps()` is a convenience method that returns `true` if the position and size of this lock overlap the specified position and size.

```
public abstract class FileLock {
    // Protected Constructors
    protected FileLock(FileChannel channel, long position, long size, boolean shared);
    // Public Instance Methods
    public final FileChannel channel();
    public final boolean isShared();
    public abstract boolean isValid();
    public final boolean overlaps(long position, long size);
    public final long position();
    public abstract void release() throws java.io.IOException;
    public final long size();
    // Public Methods Overriding Object
    public final String toString();
}
```

FileLock

Returned By: FileChannel.{lock(), tryLock()}

FileLockInterruptedException

Java 1.4

java.nio.channels

serializable checked

This exception signals that the interrupt() method of a blocked thread that was waiting to acquire a file lock was called. See FileChannel.lock().



```
public class FileLockInterruptedException extends java.io.IOException {  
    // Public Constructors  
    public FileLockInterruptedException();  
}
```

GatheringByteChannel

Java 1.4

java.nio.channels

This interface extends WritableByteChannel and adds two additional write() methods that can gather bytes from one or more buffers and write them out to the channel. These methods are passed an array of ByteBuffer objects and, optionally, an offset and length that define the relevant subarray to be used. The write() method attempts to write all the remaining bytes from all the specified buffers (in the order in which they appear in the buffer array) to the channel. The return value of the method is the number of bytes actually written. See WritableByteChannel for a discussion of exceptions and thread safety that apply to these write() methods.



```
public interface GatheringByteChannel extends WritableByteChannel {  
    // Public Instance Methods  
    public abstract long write(java.nio.ByteBuffer[] srcs) throws java.io.IOException;  
    public abstract long write(java.nio.ByteBuffer[] srcs, int offset, int length) throws java.io.IOException;  
}
```

Implementations: DatagramChannel, FileChannel, Pipe.SinkChannel, SocketChannel

IllegalBlockingModeException

Java 1.4

java.nio.channels

serializable unchecked

This exception signals an attempt to use a channel in the wrong blocking mode. An exception of this type is thrown by SelectableChannel.register() if the channel is not in nonblocking mode.



```
public class IllegalBlockingModeException extends IllegalStateException {  
    // Public Constructors  
    public IllegalBlockingModeException();  
}
```

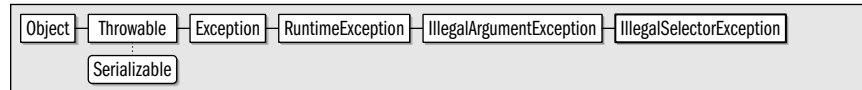
IllegalSelectorException

Java 1.4

java.nio.channels

serializable unchecked

This exception signals an attempt to register a `SelectableChannel` with a `Selector` when the channel and the selector were not created by the same `java.nio.channels.spi.SelectorProvider`.



```

public class IllegalSelectorException extends IllegalArgumentException {
    // Public Constructors
    public IllegalSelectorException();
}
  
```

InterruptibleChannel

Java 1.4

java.nio.channels

Channels that implement this marker interface have two important properties that are relevant to multithreaded programs: they are asynchronously closeable and interruptible. When the `close()` method of an `InterruptibleChannel` is called, any other thread that is blocked while waiting for an I/O operation to complete on that channel will stop blocking and receive an `AsynchronousCloseException`. Furthermore, if a thread is blocked while waiting for an I/O operation to complete on an `InterruptibleChannel`, then another thread may call the `interrupt()` method of the blocked thread. This sets the interrupt status of the blocked thread and causes the thread to wake up and receive a `ClosedByInterruptException` (a subclass of `AsynchronousCloseException`). As the name of this interrupt implies, the channel that the thread was blocked on is closed as a side effect of the thread interruption. There is no way to interrupt a blocked thread without closing the channel upon which it is blocked. This ability to interrupt a blocked thread is particularly noteworthy because it has never worked reliably with the older `java.io` API.

All the concrete channel implementations that are part of this package implement `InterruptibleChannel`. Note, however, that methods such as `Channels.newChannel()` may return channel objects that are not interruptible. You can use the `instanceof` to determine whether an unknown channel object implements this interface.



```

public interface InterruptibleChannel extends Channel {
    // Public Instance Methods
    public abstract void close() throws java.io.IOException;
}
  
```

Implementations: `java.nio.channels.spi.AbstractInterruptibleChannel`

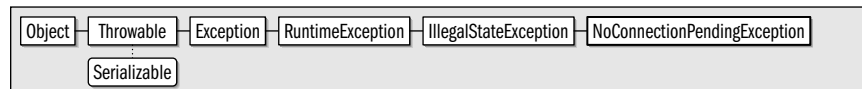
NoConnectionPendingException

Java 1.4

java.nio.channels

serializable unchecked

This exception signals that `SocketChannel.finishConnect()` was called without a previous call to `SocketChannel.connect()`.



NoConnectionPendingException

```
public class NoConnectionPendingException extends IllegalStateException {  
    // Public Constructors  
    public NoConnectionPendingException();  
}
```

NonReadableChannelException

Java 1.4

java.nio.channels

serializable unchecked

This exception signals a call to the `read()` method of a readable channel that is not open for reading, such as a `FileChannel` created from a `FileOutputStream`.



```
public class NonReadableChannelException extends IllegalStateException {  
    // Public Constructors  
    public NonReadableChannelException();  
}
```

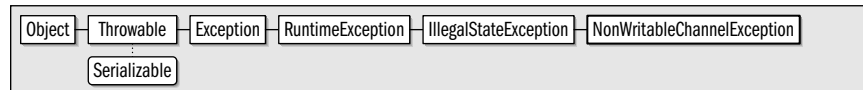
NonWritableChannelException

Java 1.4

java.nio.channels

serializable unchecked

This exception signals a call to a `write()` method of a writable channel that is not open for writing, such as a `FileChannel` created from a `FileInputStream`.



```
public class NonWritableChannelException extends IllegalStateException {  
    // Public Constructors  
    public NonWritableChannelException();  
}
```

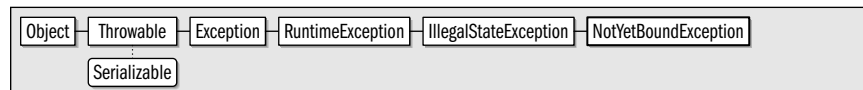
NotYetBoundException

Java 1.4

java.nio.channels

serializable unchecked

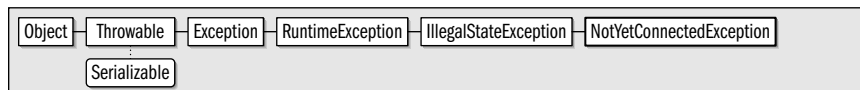
This exception signals a call to `ServerSocketChannel.accept()` before the underlying server socket has been bound to a local port. Call `socket().bind()` to bind the `java.net.ServerSocket` that underlies the `ServerSocketChannel`.



```
public class NotYetBoundException extends IllegalStateException {  
    // Public Constructors  
    public NotYetBoundException();  
}
```

NotYetConnectedException**Java 1.4****java.nio.channels***serializable unchecked*

This exception signals an attempt to `read()` or `write()` on a `SocketChannel` that is not yet connected to a remote host. See `SocketChannel.connect()`.

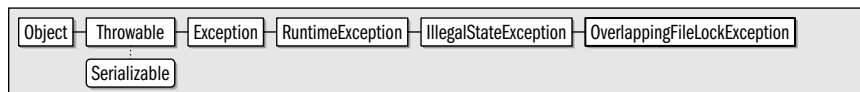


```

public class NotYetConnectedException extends IOException {
    // Public Constructors
    public NotYetConnectedException();
}
  
```

OverlappingFileLockException**Java 1.4****java.nio.channels***serializable unchecked*

This exception is thrown by the `lock()` and `tryLock()` methods of `FileChannel` if the requested lock region overlaps a file lock already held by a thread in this JVM, or if there is already a thread in this JVM waiting to lock an overlapping region of the same file. The `FileChannel` file-locking mechanism is designed to lock files against concurrent access by two separate processes. Two threads within the same JVM should not attempt to acquire a lock on overlapping regions of the same file—any attempt to do so causes an exception of this type to be thrown.



```

public class OverlappingFileLockException extends IOException {
    // Public Constructors
    public OverlappingFileLockException();
}
  
```

Pipe**Java 1.4****java.nio.channels**

A pipe is an abstraction that allows the one-way transfer of bytes from one thread to another. A pipe has a “read end” and a “write end,” which are represented by objects that implement the `ReadableByteChannel` and `WritableByteChannel` interfaces. Create a new pipe with the static `Pipe.open()` method. Call the `sink()` method to obtain the `Pipe.SinkChannel` object that represents the write end of the pipe, and call the `source()` method to obtain the `Pipe.SourceChannel` object that represents the read end of the pipe.

Programmers familiar with only Unix-style pipes may find the names and return values of the `sink()` and `source()` methods confusing. A Unix pipe is an interprocess communication mechanism tied to two specific processes: a source of bytes and a destination, or sink, for those bytes. With this conceptual model of a pipe, you would expect the source to obtain the channel it writes to with the `source()` method and the sink to obtain the channel it reads from with the `sink()` method.

This `Pipe` class is not a Unix-style pipe, however. While it can be used for communication between two threads, the ends of the pipe are not tied to those threads, and a single source thread and a single sink thread are not necessary. Therefore, in the `Pipe` API, it is the pipe itself that serves as the source and sink of bytes: bytes are read from the source end of the pipe and written to the sink end.

Pipe

```
public abstract class Pipe {  
    // Protected Constructors  
    protected Pipe();  
    // Inner Classes  
    public abstract static class SinkChannel extends java.nio.channels.spi.AbstractSelectableChannel implements  
        GatheringByteChannel, WritableByteChannel;  
    public abstract static class SourceChannel extends java.nio.channels.spi.AbstractSelectableChannel implements  
        ReadableByteChannel, ScatteringByteChannel;  
    // Public Class Methods  
    public static Pipe open() throws java.io.IOException;  
    // Public Instance Methods  
    public abstract Pipe.SinkChannel sink();  
    public abstract Pipe.SourceChannel source();  
}
```

Returned By: `Pipe.open()`, `java.nio.channels.spi.SelectorProvider.openPipe()`

Pipe.SinkChannel

Java 1.4

`java.nio.channels`

This public inner class represents the write end of a pipe. Bytes written to a `Pipe.SinkChannel` become available on the corresponding `Pipe.SourceChannel` of the pipe. Obtain a `Pipe.SinkChannel` by creating a `Pipe` object with `Pipe.open()` and calling the `sink()` method of that object. See also the containing `Pipe` class.

`Pipe.SinkChannel` implements `WritableByteChannel` and `GatheringByteChannel` and defines the `write()` methods of those interfaces. This class subclasses `SelectableChannel` so it can be used with a `Selector`. It overrides the abstract `validOps()` method of `SelectableChannel` to return `SelectionKey.OP_WRITE` but defines no new methods of its own.

```
public abstract static class Pipe.SinkChannel extends java.nio.channels.spi.AbstractSelectableChannel  
    implements GatheringByteChannel, WritableByteChannel {  
    // Protected Constructors  
    protected SinkChannel(java.nio.channels.spi.SelectorProvider provider);  
    // Public Methods Overriding SelectableChannel  
    public final int validOps(); constant  
}
```

Returned By: `Pipe.sink()`

Pipe.SourceChannel

Java 1.4

`java.nio.channels`

This public inner class represents the read end of a pipe. Bytes written to the corresponding write end of the pipe (see `Pipe.SinkChannel`) become available for reading through this channel. Obtain a `Pipe.SourceChannel` by creating a `Pipe` object with `Pipe.open()` and then calling the `source()` method of that object. See also the containing `Pipe` class.

`Pipe.SourceChannel` implements `ReadableByteChannel` and `ScatteringByteChannel` and defines the `read()` methods of those interfaces. This class subclasses `SelectableChannel` so it can be used with a `Selector`. It overrides the abstract `validOps()` method of `SelectableChannel` to return `SelectionKey.OP_READ` but defines no new methods of its own.

```
public abstract static class Pipe.SourceChannel extends java.nio.channels.spi.AbstractSelectableChannel  
    implements ReadableByteChannel, ScatteringByteChannel {  
    // Protected Constructors
```

```
protected SourceChannel(java.nio.channels.spi.SelectorProvider provider);
// Public Methods Overriding SelectableChannel
public final int validOps(); constant
}
```

Returned By: Pipe.source()

ReadableByteChannel

Java 1.4

java.nio.channels

This subinterface of **Channel** defines a single-key **read()** method that reads bytes from the channel and stores them in the specified **ByteBuffer**, updating the buffer position as it does so. **read()** attempts to read as many bytes as will fit in the specified buffer (see **Buffer.remaining()**) but may read fewer than this. For example, if the channel is a non-blocking channel, **read()** will return immediately, even if there are no bytes available to be read. **read()** returns the number of bytes actually read (which may be zero in the nonblocking case) or **-1** if there are no more bytes to be read in the channel (for example, if the end of a file has been reached, or the other end of a socket has been closed).

read() is declared to throw an **IOException**. More specifically, it may throw a **ClosedChannelException** if the channel is closed. If the channel is closed asynchronously, or if a blocked thread is interrupted, the **read()** method may terminate with an **AsynchronousCloseException** or a **ClosedByInterruptException**. **read()** may also throw an unchecked **NonReadableChannelException** if it is called on a channel that was not opened or configured to allow reading.

ReadableByteChannel implementations are required to be thread-safe: only one thread may perform a read operation on a channel at a time. If a read operation is in progress, then any call to **read()** will block until the operation is completed. Some channel implementations may allow read and write operations to proceed concurrently, but none will allow two read operations to proceed at the same time.

Channel ReadableByteChannel

```
public interface ReadableByteChannel extends Channel {
// Public Instance Methods
public abstract int read(java.nio.ByteBuffer dst) throws java.io.IOException;
}
```

Implementations: ByteChannel, Pipe.SourceChannel, ScatteringByteChannel

Passed To: Channels.{newInputStream(), newReader()}, FileChannel.transferFrom()

Returned By: Channels.newChannel()

ScatteringByteChannel

Java 1.4

java.nio.channels

This interface extends **ReadableByteChannel** and adds two additional **read()** methods that read bytes for a channel and scatter them to an array (or subarray) of buffers. These methods are passed an array of **ByteBuffer** objects and, optionally, an offset and length that define the region of the array to be used. The **read()** method attempts to read enough bytes from the channel to fill each of the specified buffers in the order in which they appear in the buffer array (the scattering process is actually much more orderly and linear than the name implies). The return value of the method is the number of bytes actually read, which may be different than the sum of the remaining bytes

ScatteringByteChannel

in the buffers. See `ReadableByteChannel` for a discussion of exceptions and thread safety that applies to these `read()` methods as well.

```
Channel ······ ReadableByteChannel ······ ScatteringByteChannel

public interface ScatteringByteChannel extends ReadableByteChannel {
// Public Instance Methods
    public abstract long read(java.nio.ByteBuffer[ ] dsts) throws java.io.IOException;
    public abstract long read(java.nio.ByteBuffer[ ] dsts, int offset, int length) throws java.io.IOException;
}
```

Implementations: `DatagramChannel`, `FileChannel`, `Pipe.SourceChannel`, `SocketChannel`

SelectableChannel

Java 1.4

`java.nio.channels`

This abstract class defines the API for channels that can be used with a `Selector` object to allow a thread to block while waiting for activity on any of a group of channels. All channel classes in the `java.nio.channels` package except for `FileChannel` are subclasses of `SelectableChannel`.

A selectable channel may be registered only with a `Selector` if it is nonblocking, so this class defines the `configureBlocking()` method—pass `false` to this method to put a channel into nonblocking mode, or pass `true` to make calls to its `read()` and/or `write()` methods block. Use `isBlocking()` to determine the current blocking mode of a selectable channel.

Register a `SelectableChannel` with a `Selector` by calling the `register()` method of the channel (not of the selector). There are two versions of this method; both take a `Selector` object and a bitmask that specifies the set of channel operations that will be selected on that channel. (See `SelectionKey` for the constants that can be OR-ed together to form this bitmask). Both methods return a `SelectionKey` object that represents the registration of the channel with the selector. One version of the `register()` method also takes an arbitrary object argument that serves as an attachment to the `SelectionKey` and allows you to associate arbitrary data with it. The `validOps()` method returns a bitmask that specifies the set of operations that a particular channel object allows to be selected. The bitmask passed to `register()` may contain only bits that are set in this `validOps()` value.

Note that `SelectableChannel` does not define a `deregister()` method. Instead, to remove a channel from the set of channels monitored by a `Selector`, you must call the `cancel()` method of the `SelectionKey` returned by `register()`.

Call `isRegistered()` to determine whether a `SelectableChannel` is registered with any `Selector`. (Note that a single channel may be registered with more than one `Selector`.) If you did not keep track of the `SelectionKey` returned by a call to `register()`, you can query it with the `keyFor()` method.

See `Selector` and `SelectionKey` for further details on multiplexing selectable channels.

```
Object ······ AbstractInterruptibleChannel ······ SelectableChannel
Channel ······ Channel ······ InterruptibleChannel ······ Channel

public abstract class SelectableChannel extends java.nio.channels.spi.AbstractInterruptibleChannel
    implements Channel {
// Protected Constructors
    protected SelectableChannel();
// Public Instance Methods
    public abstract Object blockingLock();
    public abstract SelectableChannel configureBlocking(boolean block) throws java.io.IOException;
```



```

public abstract boolean isBlocking();
public abstract boolean isRegistered();
public abstract SelectionKey keyFor(Selector sel);
public abstract java.nio.channels.spi.SelectorProvider provider();
public final SelectionKey register(Selector sel, int ops) throws ClosedChannelException;
public abstract SelectionKey register(Selector sel, int ops, Object att) throws ClosedChannelException;
public abstract int validOps();
}

```

Subclasses: java.nio.channels.spi.AbstractSelectableChannel

Returned By: SelectableChannel.configureBlocking(), SelectionKey.channel(),
java.nio.channels.spi.AbstractSelectableChannel.configureBlocking()

SelectionKey

Java 1.4

java.nio.channels

A **SelectionKey** represents the registration of a **SelectableChannel** with a **Selector** and identifies a selected channel and the operations that are ready to be performed on that channel. After a call to the **select()** method of a selector, the **selectedKeys()** method of the selector returns a **Set** of **SelectionKey** objects to identify the channel or channels that are ready for reading, for writing, or for another operation.

Create a **SelectionKey** by passing a **Selector** object to the **register()** method of a **SelectableChannel**. The **channel()** and **selector()** methods of the returned **SelectionKey** return the **SelectableChannel** and **Selector** objects associated with that key.

When you no longer want the channel to be registered with the selector, call the **cancel()** method of the **SelectionKey**. **isValid()** determines whether a **SelectionKey** is still valid—it returns **true** unless the **cancel()** method has been called, the channel has been closed, or the selector has been closed.

The main purpose of a **SelectionKey** is to hold the “interest set” of channel operations that the selector should monitor for the channel and the “ready set” of operations that the selector has determined are ready to proceed on the channel. Both sets are represented as integer bitmasks (not as **java.util.Set** objects) formed by OR-ing together any of the **OP_** constants defined by this class. Those constants are:

OP_READ

In the interest set, this bit specifies an interest in read operations. In the ready set, this bit specifies that the channel has bytes available for reading, has reached the end-of-stream, has been remotely closed, or that an error has occurred.

OP_WRITE

In the interest set, this bit specifies an interest in write operations. In the ready set, this bit specifies that the channel is ready to have bytes written, has been closed, or that an error has occurred.

OP_CONNECT

In the interest set, this bit specifies an interest in socket connection operations. In the ready set, it indicates that a socket channel is ready to connect or that an error has occurred.

OP_ACCEPT

In the interest set, this bit specifies an interest in server socket accept operations. In the ready set, it indicates that a server socket channel is ready to accept a connection or that an error has occurred.

SelectionKey

The no-argument version of the `interestOps()` method allows you to query the interest set. The initial value of the interest set is the bitmask that was passed to the `register()` method of the channel. It can be changed, however, by passing a new bitmask to the one-argument version of `interestOps()`. (Note that the same method name is used to both query and set the interest set.) The current state of the ready set can be queried with `readyOps()`. You can also use the convenience methods `isReadable()`, `isWritable()`, `isConnectable()`, and `isAcceptable()` to test whether individual operation bits are set in the ready set bitmask. There is no way to explicitly set the state of the ready set—each call to the `select()` method updates the ready set for you. Note, however, that you must remove a `SelectionKey` object from the `Set` returned by `Selector.selectedKeys()` for the bits of the ready set that will be cleared at the start of the next selection operation. If you never remove the `SelectionKey` from the set of selected keys, the `Selector` assumes that none of the I/O readiness conditions represented by the ready set have been handled yet, and leaves their bits set.

Use `attach()` to associate an arbitrary object with a `SelectionKey` and call `attachment()` to query that object. This ability to associate data with a selection key is often useful when using a `Selector` with multiple channels: it can provide the context necessary to process a `SelectionKey` that has been selected.

```
public abstract class SelectionKey {
    // Protected Constructors
    protected SelectionKey();

    // Public Constants
    public static final int OP_ACCEPT;           =16
    public static final int OP_CONNECT;         =8
    public static final int OP_READ;            =1
    public static final int OP_WRITE;           =4

    // Property Accessor Methods (by property name)
    public final boolean isAcceptable();
    public final boolean isConnectable();
    public final boolean isReadable();
    public abstract boolean isValid();
    public final boolean isWritable();

    // Public Instance Methods
    public final Object attach(Object ob);
    public final Object attachment();
    public abstract void cancel();
    public abstract SelectableChannel channel();
    public abstract int interestOps();
    public abstract SelectionKey interestOps(int ops);
    public abstract int readyOps();
    public abstract Selector selector();
}
```

Subclasses: `java.nio.channels.spi.AbstractSelectionKey`

Returned By: `SelectableChannel.{keyFor(), register()}`, `SelectionKey.interestOps()`,
`java.nio.channels.spi.AbstractSelectableChannel.{keyFor(), register()}`,
`java.nio.channels.spi.AbstractSelector.register()`

Selector

Java 1.4

`java.nio.channels`

A `Selector` is an object that monitors multiple nonblocking `SelectableChannel` objects and selects the channel or channels (after blocking, if necessary) ready for I/O. Create a new `Selector` with the static `open()` method. Next, register the channels that it will monitor; a channel is registered by passing the `Selector` to the `register()` method of the channel

(`register()` is defined by the abstract `SelectableChannel` class). In addition to the `Selector`, you must pass a bitmask that specifies which I/O operations (reading, writing, connecting, or accepting) that the `Selector` will monitor for that channel. Each call to this `register()` method returns a `SelectionKey` object. (The `SelectionKey` class also defines the constants used to form the bitmask of I/O operations.) Note that before a `SelectableChannel` can be registered, it must be in nonblocking mode, which can be accomplished with the `configureBlocking()` method of `SelectableChannel`.

Once the channels are registered with the `Selector`, call `select()` to block until one or more of the channels is ready for I/O. One version of `select()` takes a timeout value and returns if the specified number of milliseconds elapses without any channels becoming ready for I/O. These methods also return if any of the channels are closed, if an error occurs on any channel, if the `wakeup()` method of the `Selector` is called, or if the `interrupt()` method of the blocked thread is called. There is also a `selectNow()` method similar to `select()`: it does not block but simply polls each of the channels and determines which are ready for I/O. The return value of `selectNow()` and of both `select()` methods is the number of channels ready for I/O. It is possible for this return value to be 0.

The `select()` and `selectNow()` methods return the number of channels ready for I/O; they do not return the channels themselves. To obtain this information, you must call the `selectedKeys()` method, which returns a `java.util.Set` containing `SelectionKey` objects. After calling `select()` and `selectedKeys()`, applications typically obtain a `java.util.Iterator` for the `Set` and use it to loop through the `SelectionKey` objects that represent the channels ready for I/O. Use the `channel()` method of the `SelectionKey` to determine which channel is ready and call `readyOps()`, `isReadable()`, `isWritable()`, or related methods of the `SelectionKey` to determine the kind of I/O operation that is ready on the channel. `SelectionKey` objects remain in the `selectedKeys()` set until explicitly removed, so after performing the I/O operation for a given `SelectionKey`, remove that key from the `Set` returned by `selectedKeys()` (use the `remove()` method of the `Set` of its `Iterator`).

In addition to the `selectedKeys()` method, `Selector` defines a `keys()` method, which also returns a `Set` of `SelectionKey` objects. This set represents the complete set of channels that are monitored by the `Selector` and may not be modified, except by closing or deregistering the channel by calling the `cancel()` method of the associated `SelectionKey`. Canceled keys are removed from the `keys()` set on the next call to `select()` or `selectNow()`.

Call `wakeup()` to have another thread blocked in a call to `select()` wake up and return immediately. If `wakeup()` is called, but no thread is currently blocked in a `select()` call, then the next call to `select()` or `selectNow()` will return immediately.

When a `Selector` object is no longer needed, close it by calling `close()`. If any thread is blocked in a `select()` call, it will return immediately as if `wakeup()` had been called. After calling `close()`, you should not call any other methods of a `Selector`. `isOpen()` returns true if a `Selector` is still open and false if it has been closed.

The `Selector` class is thread-safe. Note, however, that the `Set` object returned by `selectedKeys()` is not; it should be used by only one thread at a time.

```
public abstract class Selector {
    // Protected Constructors
    protected Selector();
    // Public Class Methods
    public static Selector open() throws java.io.IOException;
    // Public Instance Methods
    public abstract void close() throws java.io.IOException;
    public abstract boolean isOpen();
    public abstract java.util.Set keys();
    public abstract java.nio.channels.spi.SelectorProvider provider();
}
```

Selector

```
public abstract int select() throws java.io.IOException;
public abstract int select(long timeout) throws java.io.IOException;
public abstract java.util.Set selectedKeys();
public abstract int selectNow() throws java.io.IOException;
public abstract Selector wakeup();
}
```

Subclasses: java.nio.channels.spi.AbstractSelector

Passed To: SelectableChannel.{keyFor(), register()},
java.nio.channels.spi.AbstractSelectableChannel.{keyFor(), register()}

Returned By: SelectionKey.selector(), Selector.{open(), wakeup()}

ServerSocketChannel

Java 1.4

java.nio.channels

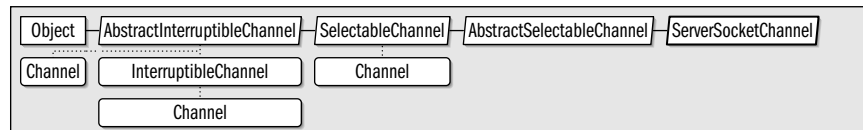
This class is the java.nio version of java.net.ServerSocket. It is a selectable channel that can be used by servers to accept connections from clients. Unlike other channel classes in this package, this class cannot be used for reading or writing bytes; it does not implement any of the ByteChannel interfaces and exists only to accept and establish connections with clients, not to communicate with those clients. ServerSocketChannel differs from java.net.ServerSocket in two important ways: it can be put into nonblocking mode and used with a Selector, and its accept() method returns a SocketChannel rather than a Socket, so communication with the client whose connection was just accepted can be done with the java.nio APIs.

Create a new ServerSocketChannel with the static open() method. Next, call socket() to obtain the associated ServerSocket object and use its bind() method to bind the server socket to a specific port on the local host. You can also call any other ServerSocket methods to configure other socket options at this point.

To accept a new connection through this ServerSocketChannel, simply call accept(). If the channel is in blocking mode, this method will block until a client connects and will then return a SocketChannel connected to the client. In nonblocking mode (see the inherited configureBlocking() method), accept() returns a SocketChannel only if a client is currently waiting to connect; otherwise, it immediately returns null. To be notified when a client is waiting to connect, use the inherited register() method to register a nonblocking ServerSocketChannel with a Selector and specify an interest in accept operations with the SelectionKey.OP_ACCEPT constant. See Selector and SelectionKey for further details.

Note that the SocketChannel object returned by the accept() method is always in non-blocking mode, regardless of the blocking mode of the ServerSocketChannel.

ServerSocketChannel is thread-safe; only one thread may call the accept() method at a time. When a ServerSocketChannel is no longer required, close it with the inherited close() method.



```
public abstract class ServerSocketChannel extends java.nio.channels.spi.AbstractSelectableChannel {
    // Protected Constructors
    protected ServerSocketChannel(java.nio.channels.spi.SelectorProvider provider);
}
```

```
// Public Class Methods
public static ServerSocketChannel open() throws java.io.IOException;
// Public Instance Methods
public abstract SocketChannel accept() throws java.io.IOException;
public abstract java.net.ServerSocket socket();
// Public Methods Overriding SelectableChannel
public final int validOps();
}
```

Returned By: `java.net.ServerSocket.getChannel()`, `ServerSocketChannel.open()`,
`java.nio.channels.spi.SelectorProvider.openServerSocketChannel()`

SocketChannel

Java 1.4

`java.nio.channels`

This class is a channel for communicating over a `java.net.Socket`. It implements `ReadableByteChannel` and `WritableByteChannel`, as well as `GatheringByteChannel` and `ScatteringByteChannel`. It is a subclass of `SelectableChannel` and can be used with a `Selector`.

Create a new `SocketChannel` with one of the static `open()` methods. The no-argument version of `open()` creates a new `SocketChannel` but does not connect it to a remote host. The other version of `open()` opens a new channel and connects it to the specified `java.net.SocketAddress`. If you create an unconnected socket, you can explicitly connect it with the `connect()` method. The main reason to open the channel and connect to the remote host in separate steps is if you want to do a nonblocking connect. To do this, first put the channel into nonblocking mode with the inherited `configureBlocking()` method. Next, call `connect()`; it will return immediately, without waiting for the connection to be established. Then register the channel with a `Selector`, specifying that you are interested in `SelectionKey.OP_CONNECT` operations. When you are notified that your channel is ready to connect (see `Selector` and `SelectionKey` for details), simply call the nonblocking `finishConnect()` method to complete the connection. `isConnected()` returns `true` once a connection is established, and `false` otherwise. `isConnectionPending()` returns `true` if `connect()` has been called in blocking mode and has not yet returned, or if `connect()` has been called in nonblocking mode, but `finishConnect()` has not been called yet.

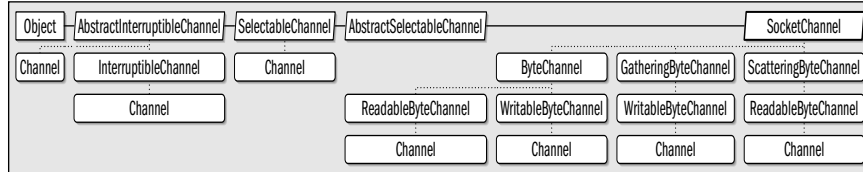
Once you have opened and connected a `SocketChannel`, you can read and write bytes to it with the various `read()` and `write()` methods. `SocketChannel` is thread-safe: read and write operations may proceed concurrently, but `SocketChannel` will not allow more than one read operation and more than one write operation to proceed at the same time. If you put a `SocketChannel` into nonblocking mode, you can register it with a `Selector` using the `SelectionKey` constants `OP_READ` and `OP_WRITE`, which will have the `Selector` tell you when the channel is ready for reading or writing.

The `socket()` method returns the `java.net.Socket` associated with the `SocketChannel`. You can use this `Socket` object to configure socket options, bind the socket to a specific local address, close the socket, or shut down its input or output sides. (See `java.net.Socket`.) Note that although all `SocketChannel` objects have associated `Socket` objects, the reverse is not true: you cannot obtain a `SocketChannel` from a `Socket` unless the `Socket` was created with the `SocketChannel` by a call to `SocketChannel.open()`.

When you are done with a `SocketChannel`, close it with the `close()` method. You can also independently shut down the read and write portions of the channel with `socket().shutdownInput()` and `socket().shutdownOutput()`. When the input is shut down, any future reads (and any blocked read operation) will return `-1` to indicate that the end-of-stream has been reached. When the output is shut down, any future writes throw a `ClosedChannelEx-`

SocketChannel

ception, and any write operation that was blocked at the time of shutdown throws a `AsynchronousCloseException`.



```

public abstract class SocketChannel extends java.nio.channels.spi.AbstractSelectableChannel
    implements ByteChannel, GatheringByteChannel, ScatteringByteChannel {
    // Protected Constructors
    protected SocketChannel(java.nio.channels.spi.SelectorProvider provider);
    // Public Class Methods
    public static SocketChannel open() throws java.io.IOException;
    public static SocketChannel open(java.net.SocketAddress remote) throws java.io.IOException;
    // Public Instance Methods
    public abstract boolean connect(java.net.SocketAddress remote) throws java.io.IOException;
    public abstract boolean finishConnect() throws java.io.IOException;
    public abstract boolean isConnected();
    public abstract boolean isConnectionPending();
    public abstract java.net.Socket socket();
    // Methods Implementing GatheringByteChannel
    public final long write(java.nio.ByteBuffer[] srcs) throws java.io.IOException;
    public abstract long write(java.nio.ByteBuffer[] srcs, int offset, int length) throws java.io.IOException;
    // Methods Implementing ReadableByteChannel
    public abstract int read(java.nio.ByteBuffer dst) throws java.io.IOException;
    // Methods Implementing ScatteringByteChannel
    public final long read(java.nio.ByteBuffer[] dsts) throws java.io.IOException;
    public abstract long read(java.nio.ByteBuffer[] dsts, int offset, int length) throws java.io.IOException;
    // Methods Implementing WritableByteChannel
    public abstract int write(java.nio.ByteBuffer src) throws java.io.IOException;
    // Public Methods Overriding SelectableChannel
    public final int validOps();
}
  
```

Returned By: `java.net.Socket.getChannel()`, `ServerSocketChannel.accept()`, `SocketChannel.open()`, `java.nio.channels.spi.SelectorProvider.openSocketChannel()`

UnresolvedAddressException

Java 1.4

java.nio.channels

serializable unchecked

This exception signals the use of a `java.net.SocketAddress` that could not be resolved—for example, a `java.net.InetSocketAddress` that contains an unknown hostname.



```

public class UnresolvedAddressException extends IllegalArgumentException {
    // Public Constructors
    public UnresolvedAddressException();
}
  
```

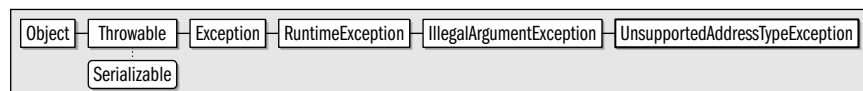
UnsupportedAddressTypeException

Java 1.4

java.nio.channels

serializable unchecked

This exception signals the use of a `java.net.SocketAddress` subclass unknown to or not supported by the implementation. It is safe to assume that addresses of the type `java.net.InetSocketAddress` are universally supported.



```

public class UnsupportedAddressTypeException extends IllegalArgumentException {
    // Public Constructors
    public UnsupportedAddressTypeException();
}
  
```

WritableByteChannel

Java 1.4

java.nio.channels

This subinterface of `Channel` defines a single-key `write()` method which writes bytes from a specified `ByteBuffer` to the channel, updating the buffer position as it goes. If possible, it writes all remaining bytes in the buffer (see `Buffer.remaining()`). This is not always possible (with nonblocking channels, for example), so the `write()` method returns the number of bytes it was actually able to write to the channel.

`write()` is declared to throw an `IOException`. More specifically, it may throw a `ClosedChannelException` if the channel is closed. If the channel is closed asynchronously, or if a blocked thread is interrupted, the `write()` method may terminate with an `AsynchronousCloseException` or a `ClosedByInterruptException`. `write()` may also throw an unchecked `NonWritableChannelException` if it is called on a channel that was not opened or configured to allow writing (such as a `FileChannel`).

`WritableByteChannel` implementations are required to be thread-safe: only one thread may perform a write operation on a channel at a time. If a write operation is in progress, then any call to `write()` will block until the operation is completed. Some channel implementations may allow read and write operations to proceed concurrently; some may not.



```

public interface WritableByteChannel extends Channel {
    // Public Instance Methods
    public abstract int write(java.nio.ByteBuffer src) throws java.io.IOException;
}
  
```

Implementations: `ByteChannel`, `GatheringByteChannel`, `Pipe.SinkChannel`

Passed To: `Channels.{newOutputStream(), newWriter()}()`, `FileChannel.transferTo()`

Returned By: `Channels.newChannel()`

Package java.nio.channels.spi

Java 1.4

This package defines four classes used by implementors of channels and selector classes of `java.nio.channels`. It also defines the `SelectorProvider` class, which allows a custom implementation of channels and selectors to be specified instead of the default implementation. Application programmers should never need to use this package, except in

Package *java.nio.channels.spi*

rare circumstances to explicitly install a `SelectorProvider` implementation with the `SelectorProvider.provider()` method.

Classes:

public abstract class **AbstractInterruptibleChannel** implements `java.nio.channels.Channel`,
`java.nio.channels.InterruptibleChannel`;
public abstract class AbstractSelectableChannel extends `java.nio.channels.SelectableChannel`;
public abstract class AbstractSelectionKey extends `java.nio.channels.SelectionKey`;
public abstract class AbstractSelector extends `java.nio.channels.Selector`;
public abstract class SelectorProvider;

AbstractInterruptibleChannel

Java 1.4

`java.nio.channels.spi`

This class exists as a convenience for implementors of new channel classes. Application programmers should never need to use or subclass it.



```
public abstract class AbstractInterruptibleChannel implements java.nio.channels.Channel,  
    java.nio.channels.InterruptibleChannel {  
    // Protected Constructors  
    protected AbstractInterruptibleChannel();  
    // Methods Implementing Channel  
    public final void close() throws java.io.IOException;  
    public final boolean isOpen();  
    // Protected Instance Methods  
    protected final void begin();  
    protected final void end(boolean completed) throws java.nio.channels.AsynchronousCloseException;  
    protected abstract void implCloseChannel() throws java.io.IOException;  
}
```

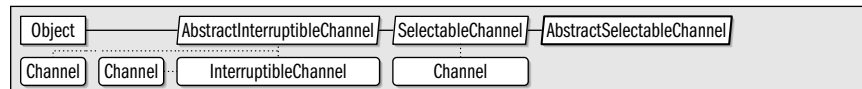
Subclasses: `java.nio.channels.FileChannel`, `java.nio.channels.SelectableChannel`

AbstractSelectableChannel

Java 1.4

`java.nio.channels.spi`

This class exists as a convenience for implementors of new selectable channel classes. It defines common methods of `SelectableChannel` in terms of protected methods with names that begin with `impl`. Application programmers should never need to use or subclass this class.



```
public abstract class AbstractSelectableChannel extends java.nio.channels.SelectableChannel {  
    // Protected Constructors  
    protected AbstractSelectableChannel(SelectorProvider provider);  
    // Public Methods Overriding SelectableChannel  
    public final Object blockingLock();  
    public final java.nio.channels.SelectableChannel configureBlocking(boolean block) throws java.io.IOException;  
    public final boolean isBlocking();  
    public final boolean isRegistered();  
    public final java.nio.channels.SelectionKey keyFor(java.nio.channels.Selector sel);  
}
```



```

    public final SelectorProvider provider();
    public final java.nio.channels.SelectionKey register(java.nio.channels.Selector sel, int ops, Object att)
        throws java.nio.channels.ClosedChannelException;
    // Protected Methods Overriding AbstractInterruptibleChannel
    protected final void implCloseChannel() throws java.io.IOException;
    // Protected Instance Methods
    protected abstract void implCloseSelectableChannel() throws java.io.IOException;
    protected abstract void implConfigureBlocking(boolean block) throws java.io.IOException;
}

```

Subclasses: java.nio.channels.DatagramChannel, java.nio.channels.Pipe.SinkChannel,
java.nio.channels.Pipe.SourceChannel, java.nio.channels.ServerSocketChannel,
java.nio.channels.SocketChannel

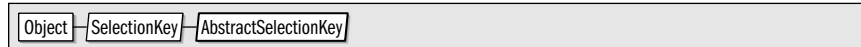
Passed To: AbstractSelector.register()

AbstractSelectionKey

Java 1.4

java.nio.channels.spi

This class exists as a convenience for implementors of new SelectionKey classes. Application programmers should never need to use or subclass this class.



```

public abstract class AbstractSelectionKey extends java.nio.channels.SelectionKey {
    // Protected Constructors
    protected AbstractSelectionKey();
    // Public Methods Overriding SelectionKey
    public final void cancel();
    public final boolean isValid();
}

```

Passed To: AbstractSelector.deregister()

AbstractSelector

Java 1.4

java.nio.channels.spi

This class exists as a convenience for implementors of new Selector classes. Application programmers should never need to use or subclass this class.



```

public abstract class AbstractSelector extends java.nio.channels.Selector {
    // Protected Constructors
    protected AbstractSelector(SelectorProvider provider);
    // Public Methods Overriding Selector
    public final void close() throws java.io.IOException;
    public final boolean isOpen();
    public final SelectorProvider provider();
    // Protected Instance Methods
    protected final void begin();
    protected final java.util.Set canceledKeys();
    protected final void deregister(AbstractSelectionKey key);
    protected final void end();
    protected abstract void implCloseSelector() throws java.io.IOException;
    protected abstract java.nio.channels.SelectionKey register(AbstractSelectableChannel ch, int ops, Object att);
}

```

AbstractSelector

Returned By: SelectorProvider.openSelector()

SelectorProvider

Java 1.4

java.nio.channels.spi

This class is the central service-provider class for the channels and selectors of the java.nio.channels API. A concrete subclass of **SelectorProvider** implements factory methods that return open socket channels, server socket channels, datagram channels, pipes (with their two internal channels), and **Selector** objects. There is one default **SelectorProvider** object per JVM; this object can be obtained with the static **SelectorProvider.provider()** method.

You can specify a custom **SelectorProvider** implementation by setting its class name as the value of the system property **java.nio.channels.spi.SelectorProvider**. Or you can put the class name in a file called *META-INF/services/java.nio.channels.spi.SelectorProvider* in your application's JAR file. The **provider()** method first looks for the system property then looks for the JAR file entry. If it finds neither, it instantiates the implementation's default **SelectorProvider**.

Applications are not required to use the default **SelectorProvider** exclusively. It is legal to instantiate other **SelectorProvider** objects and explicitly invoke their **open()** methods to create channels in this way.

```
public abstract class SelectorProvider {  
    // Protected Constructors  
    protected SelectorProvider();  
    // Public Class Methods  
    public static SelectorProvider provider();  
    // Public Instance Methods  
    public abstract java.nio.channels.DatagramChannel openDatagramChannel() throws java.io.IOException;  
    public abstract java.nio.channels.Pipe openPipe() throws java.io.IOException;  
    public abstract AbstractSelector openSelector() throws java.io.IOException;  
    public abstract java.nio.channels.ServerSocketChannel openServerSocketChannel() throws java.io.IOException;  
    public abstract java.nio.channels.SocketChannel openSocketChannel() throws java.io.IOException;  
}
```

Passed To: java.nio.channels.DatagramChannel.DatagramChannel(),
java.nio.channels.Pipe.SinkChannel.SinkChannel(),
java.nio.channels.Pipe.SourceChannel.SourceChannel(),
java.nio.channels.ServerSocketChannel.ServerSocketChannel(),
java.nio.channels.SocketChannel.SocketChannel(),
AbstractSelectableChannel.AbstractSelectableChannel(), AbstractSelector.AbstractSelector()

Returned By: java.nio.channels.SelectableChannel.provider(), java.nio.channels.Selector.provider(),
AbstractSelectableChannel.provider(), AbstractSelector.provider(), SelectorProvider.provider()

Package java.nio.charset

Java 1.4

This package contains classes that represent character sets or encodings and defines methods that encode characters into bytes and decode bytes into characters. The key class is **Charset**. You can obtain a **Charset** object for a named character encoding with the static **forName()** method. **Charset** defines **encode()** and **decode()** convenience methods, but for full control over the encoding and decoding process, you can also obtain a **CharsetEncoder** or **CharsetDecoder** object from the **Charset**.

The Java platform has had a character encoding and decoding facility since Java 1.1 and defines a number of classes and methods that perform character encoding or decoding. Some of these classes and methods are specified to use the default charset

for the locale; others take the name of a charset as a method or constructor argument. (For example, see the `String()`, `java.io.InputStreamReader()`, and `java.io.OutputStreamWriter()` constructors.) In Java 1.4, the `java.nio.charset` package defines a public API for the character encoding and decoding facility and allows applications to work with it explicitly. Most applications will not have to do this, however, and can simply continue to rely on the default charset or can continue to supply charset names where needed. Even applications that use the `java.nio.channels` package can avoid explicit character encoding and decoding by passing the name of a desired charset to the `newReader()` and `newWriter()` methods of `java.nio.channels.Channels`.

Classes:

```
public abstract class Charset implements Comparable;
public abstract class CharsetDecoder;
public abstract class CharsetEncoder;
public class CodeResult;
public class CodingErrorAction;
```

Exceptions:

```
public class CharacterCodingException extends java.io.IOException;
    public class MalformedInputException extends CharacterCodingException;
    public class UnmappableCharacterException extends CharacterCodingException;
public class IllegalCharsetNameException extends IllegalArgumentException;
public class UnsupportedCharsetException extends IllegalArgumentException;
```

Errors:

```
public class CoderMalfunctionError extends Error;
```

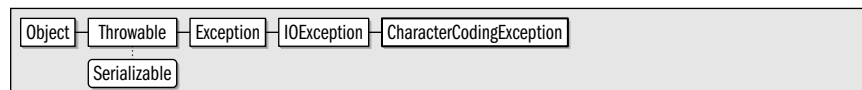
CharacterCodingException

Java 1.4

`java.nio.charset`

serializable checked

This class signals a problem encoding or decoding characters or bytes. It is a generic superclass for more specific exception types. Note that the one-argument versions of `CharsetEncoder.encode()` and `CharsetDecoder.decode()` may throw an exception of this type, but that the three-argument versions of the same method report encoding problems through their `CodeResult` return value. Also note that the `encode()` and `decode()` convenience methods of `Charset` do not throw this exception because they specify that malformed input and unmappable characters or bytes should be replaced. See `CodingErrorAction`.



```
public class CharacterCodingException extends java.io.IOException {
// Public Constructors
    public CharacterCodingException();
}
```

Subclasses: `MalformedInputException`, `UnmappableCharacterException`

Thrown By: `CharsetDecoder.decode()`, `CharsetEncoder.encode()`, `CodeResult.throwException()`

Charset**Java 1.4****java.nio.charset****comparable**

A **Charset** represents a character set or encoding. Each **Charset** has a canonical name, returned by `name()`, and a set of aliases, returned by `aliases()`. You can look up a **Charset** by name or alias with the static method `Charset.forName()` method, which throws an `UnsupportedCharsetException` if the named charset is not installed on the system. Check whether a charset specified by name or alias is supported with the static `isSupported()`. Obtain the complete set of installed charsets with `availableCharsets()`, which returns a sorted map from canonical names to **Charset** objects. Note that charset names are not case-sensitive, and you can use any capitalization for charset names you pass to `isSupported()` and `forName()`. Also note that there are a number of classes and methods in the Java platform that specify charsets by name rather than by **Charset** object. (For example, see `java.io.InputStreamReader`, `java.io.OutputStreamWriter`, `String.getBytes()`, and `java.nio.channels.Channels.newWriter()`.) When working with classes and methods such as these, there is no need to use a **Charset** object.

All implementations of Java are required to support at least the following six charsets:

<i>Canonical name</i>	<i>Description</i>
US-ASCII	7-bit ASCII.
ISO-8859-1	The 8-bit superset of ASCII that includes the characters used in most Western European languages. Also known as ISO-LATIN-1.
UTF-8	An 8-bit encoding of Unicode characters that is compatible with US-ASCII.
UTF-16BE	A 16-bit encoding of Unicode characters, using big-endian byte order.
UTF-16LE	A 16-bit encoding of Unicode characters, using little-endian byte order.
UTF-16	A 16-bit encoding of Unicode characters with byte order specified by a "byte order mark" character. Assumes big-endian when decoding if there is no byte order mark. Encodes using big-endian byte order and outputs an appropriate byte order mark.

Once you have obtained a **Charset** with `forName()` or `availableCharsets()`, you can use the `encode()` method to encode a `String` or `CharBuffer` of text into a `ByteBuffer` or use the `decode()` method to convert the bytes in a `ByteBuffer` into characters in a `CharBuffer`. These convenience methods create a new `CharsetEncoder` or `CharsetDecoder`, specify that malformed input or unmappable characters or bytes should be replaced with the default replacement string or bytes, and then invoke the `encode()` or `decode()` method of the encoder or decoder. For full control over the encoding and decoding process, you may want to obtain your own `CharsetEncoder` or `CharsetDecoder` object with `newEncoder()` or `newDecoder()`. See `CharsetDecoder` for details.

Instead of using a **Charset**, `CharsetEncoder`, or `CharsetDecoder` directly, you may also pass an encoder or decoder to the static methods of `java.nio.channels.Channels` to obtain a `java.io.Reader` or `java.io.Writer` that you can use to read or write characters from or to a byte-oriented `Channel`.

Note that not all **Charset** objects support encoding ("auto-detect" charsets can determine the source charset when decoding but have no way to encode). Use `canEncode()` to determine whether a given **Charset** can encode.

Charset also defines, implements, or overrides various other methods. `displayName()` returns a localized name for the charset or returns the canonical name if there is no localization. `toString()` returns an implementation-dependent textual representation of the

charset. The `equals()` method compares two charsets by comparing their canonical names. `Charset` implements `Comparable`, and its `compareTo()` method orders charsets by their canonical name. `contains()` returns `true` if a specified charset is “contained in” this charset, that is, if every character that can be represented in the specified charset can also be represented in this charset. Note that these representations need not be the same, however. `isRegistered()` returns `true` if the charset is registered with the IANA charset registry (see <http://www.iana.org/assignments/character-sets>).

Object	Charset	Comparable
--------	---------	------------

```

public abstract class Charset implements Comparable {
    // Protected Constructors
    protected Charset(String canonicalName, String[] aliases);
    // Public Class Methods
    public static java.util.SortedMap availableCharsets();
    public static Charset forName(String charsetName);
    public static boolean isSupported(String charsetName);
    // Public Instance Methods
    public final java.util.Set aliases();
    public boolean canEncode(); constant
    public abstract boolean contains(Charset cs);
    public final java.nio.CharBuffer decode(java.nio.ByteBuffer bb);
    public String displayName();
    public String displayName(java.util.Locale locale);
    public final java.nio.ByteBuffer encode(java.nio.CharBuffer cb);
    public final java.nio.ByteBuffer encode(String str);
    public final boolean isRegistered();
    public final String name();
    public abstract CharsetDecoder newDecoder();
    public abstract CharsetEncoder newEncoder();
    // Methods Implementing Comparable
    public final int compareTo(Object ob);
    // Public Methods Overriding Object
    public final boolean equals(Object ob);
    public final int hashCode();
    public final String toString();
}

```

Passed To: `java.io.InputStreamReader.InputStreamReader()`,
`java.io.OutputStreamWriter.OutputStreamWriter()`, `Charset.contains()`,
`CharsetDecoder.CharsetDecoder()`, `CharsetEncoder.CharsetEncoder()`

Returned By: `Charset.forName()`, `CharsetDecoder.{charset(), detectedCharset()}`,
`CharsetEncoder.charset()`, `java.nio.charset.spi.CharsetProvider.charsetForName()`

CharsetDecoder

Java 1.4

java.nio.charset

A `CharsetDecoder` is a “decoding engine” that converts a sequence of bytes into a sequence of characters based on the encoding of a charset. Obtain a `CharsetDecoder` from the `Charset` that represents the charset to be decoded. If you have a complete sequence of bytes to be decoded in a `ByteBuffer`, you can pass that buffer to the one-argument version of `decode()`. This convenience method decodes the bytes and stores the resulting characters into a newly allocated `CharBuffer`, resetting and flushing the decoder as necessary. It throws an exception if there are problems with the bytes that will be decoded.

CharsetDecoder

Typically, however, the three-argument version of `decode()` is used in a three-step decoding process:

1. Call the `reset()` method, unless this is the first time the `CharsetDecoder` has been used.
2. Call the three-argument version of `decode()` one or more times. The third argument should be `true` on, and only on, the last invocation of the method. The first argument to `decode()` is a `ByteBuffer` that contains bytes to be decoded. The second argument is a `CharBuffer` into which the resulting characters are stored. The return value of the method is a `CoderResult` object that specifies the state of the ongoing decoding operation. (The possible `CoderResult` return values are detailed later.) In a typical case, however, `decode()` returns after it has decoded all of the bytes in the input buffer. In this case, you would then fill the input buffer with more bytes to be decoded and read characters from the output buffer, calling its `compact()` method to make room for more. If an unexpected problem arises in the `CharsetDecoder` implementation, `decode()` throws a `CoderMalfunctionError`.
3. Pass the output `CharBuffer` to the `flush()` method to allow any remaining characters to be output.

The `decode()` method returns a `CoderResult` that indicates the state of the decoding operation. If the return value is `CoderResult.UNDERFLOW`, this means that `decode()` returned because all bytes from the input buffer have been read, and more input is required. If the return value is `CoderResult.OVERFLOW`, this means that `decode()` returned because the output `CharBuffer` is full, and no more characters can be decoded into it. Otherwise, the return value is a `CoderResult` with an `isError()` method that returns `true`. There are two basic types of decoding errors. If `isMalformed()` returns `true`, then the input included bytes that are not legal for the charset. These bytes start at the position of the input buffer and continue for `length()` bytes. Otherwise, if `isUnmappable()` returns `true`, then the input bytes include a character for which there is no representation in Unicode. The relevant bytes start at the position of the input buffer and continue for `length()` bytes.

By default, a `CharsetDecoder` reports all malformed input and unmappable character errors by returning a `CoderResult` object, as described previously. This behavior can be altered, however, by passing a `CodingErrorAction` to `onMalformedInput()` and `onUnmappableCharacter()`. (Query the current action for these types of errors with `malformedInputAction()` and `unmappableCharacterAction()`.) `CodingErrorAction` defines three constants that represent the three possible actions. The default action is `REPORT`. The `IGNORE` action tells the `CharsetDecoder` to ignore (i.e., skip) malformed input and unmappable characters. The `REPLACE` action tells the `CharsetDecoder` to replace malformed input and unmappable characters with the replacement string. This replacement string can be set with `replaceWith()` and queried with `replacement()`.

`averageCharsPerByte()` and `maxCharsPerByte()` return the average and maximum number of characters produced by this decoder per decoded byte. These values can help you choose the size of the `CharBuffer` to allocate for decoding.

`CharsetDecoder` is not a thread-safe class; only one thread should use an instance at a time.

`CharsetDecoder` is an abstract class. Implementors defining new charsets will need to subclass `CharsetDecoder` and define the abstract `decodeLoop()` method, which is invoked by `decode()`.

```
public abstract class CharsetDecoder {  
    // Protected Constructors  
    protected CharsetDecoder(Charset cs, float averageCharsPerByte, float maxCharsPerByte);
```

```
// Public Instance Methods
public final float averageCharsPerByte();
public final Charset charset();
public final java.nio.CharBuffer decode(java.nio.ByteBuffer in) throws CharacterCodingException;
public final CoderResult decode(java.nio.ByteBuffer in, java.nio.CharBuffer out, boolean endOfInput);
public Charset detectedCharset();
public final CoderResult flush(java.nio.CharBuffer out);
public boolean isAutoDetecting(); constant
public boolean isCharsetDetected();
public CodingErrorAction malformedInputAction();
public final float maxCharsPerByte();
public final CharsetDecoder onMalformedInput(CodingErrorAction newAction);
public final CharsetDecoder onUnmappableCharacter(CodingErrorAction newAction);
public final String replacement();
public final CharsetDecoder replaceWith(String newReplacement);
public final CharsetDecoder reset();
public CodingErrorAction unmappableCharacterAction();
// Protected Instance Methods
protected abstract CoderResult decodeLoop(java.nio.ByteBuffer in, java.nio.CharBuffer out);
protected CoderResult implFlush(java.nio.CharBuffer out);
protected void implOnMalformedInput(CodingErrorAction newAction); empty
protected void implOnUnmappableCharacter(CodingErrorAction newAction); empty
protected void implReplaceWith(String newReplacement); empty
protected void implReset(); empty
}
```

Passed To: java.io.InputStreamReader.InputStreamReader(), java.nio.channels.Channels.newReader()

Returned By: Charset.newDecoder(), CharsetDecoder.{onMalformedInput(), onUnmappableCharacter(), replaceWith(), reset()}

CharsetEncoder

Java 1.4

java.nio.charset

A `CharsetEncoder` is an “encoding engine” that converts a sequence of characters into a sequence of bytes using character encoding. Obtain a `CharsetEncoder` with the `newEncoder()` method of the `Charset` that represents the desired encoding.

A `CharsetEncoder` works like a `CharsetDecoder` in reverse. Use the `encode()` method to encode characters read from a `CharBuffer` into bytes stored in a `ByteBuffer`. See `CharsetDecoder`, which documents this process in detail.

```
public abstract class CharsetEncoder {
// Protected Constructors
protected CharsetEncoder(Charset cs, float averageBytesPerChar, float maxBytesPerChar);
protected CharsetEncoder(Charset cs, float averageBytesPerChar, float maxBytesPerChar, byte[] replacement);
// Public Instance Methods
public final float averageBytesPerChar();
public boolean canEncode(CharSequence cs);
public boolean canEncode(char c);
public final Charset charset();
public final java.nio.ByteBuffer encode(java.nio.CharBuffer in) throws CharacterCodingException;
public final CoderResult encode(java.nio.CharBuffer in, java.nio.ByteBuffer out, boolean endOfInput);
public final CoderResult flush(java.nio.ByteBuffer out);
public boolean isLegalReplacement(byte[] repl);
public CodingErrorAction malformedInputAction();
public final float maxBytesPerChar();
}
```

CharsetEncoder

```
public final CharsetEncoder onMalformedInput(CodingErrorAction newAction);
public final CharsetEncoder onUnmappableCharacter(CodingErrorAction newAction);
public final byte[] replacement();
public final CharsetEncoder replaceWith(byte[] newReplacement);
public final CharsetEncoder reset();
public CodingErrorAction unmappableCharacterAction();
// Protected Instance Methods
protected abstract CoderResult encodeLoop(java.nio.CharBuffer in, java.nio.ByteBuffer out);
protected CoderResult implFlush(java.nio.ByteBuffer out);
protected void implOnMalformedInput(CodingErrorAction newAction); empty
protected void implOnUnmappableCharacter(CodingErrorAction newAction); empty
protected void implReplaceWith(byte[] newReplacement); empty
protected void implReset(); empty
}
```

Passed To: java.io.OutputStreamWriter.OutputStreamWriter(), java.nio.channels.Channels.newWriter()

Returned By: Charset.newEncoder(), CharsetEncoder.{onMalformedInput(), onUnmappableCharacter(), replaceWith(), reset()}

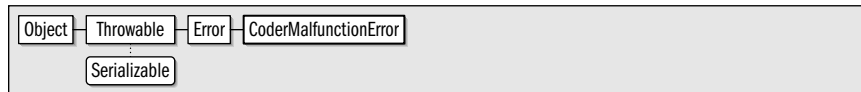
CoderMalfunctionError

Java 1.4

java.nio.charset

serializable error

This signals a malfunction—typically, an unknown and unrecoverable error—in a CharsetEncoder or CharsetDecoder. An error of this type is thrown by the encode() and decode() methods when the protected encodeLoop() or decodeLoop() methods upon which they are implemented throw an exception of an unexpected type.



```
public class CoderMalfunctionError extends Error {
// Public Constructors
public CoderMalfunctionError(Exception cause);
}
```

CoderResult

Java 1.4

java.nio.charset

A CoderResult object specifies the results of a call to CharsetDecoder.decode() or CharsetEncoder.encode(). There are four possible reasons why a call to the decode() or encode() would return:

- If all the bytes have been decoded or all the characters have been encoded, and the input buffer is empty, then the return value is the constant object CoderResult.UNDERFLOW, indicating that coding stopped because there was no more data to code. Calling the isUnderflow() method on the returned object returns true, and calling isError() returns false. This is a normal return value.
- If there is more data to be coded but no more room in the output buffer to store the coded data, then the return value is the constant object CoderResult.OVERFLOW. Calling isOverflow() on the returned object returns true, and calling isError() returns false. This is a normal return value.
- If the input data was malformed, containing characters or bytes that are not legal for the charset, and the CharsetEncoder or CharsetDecoder has not specified that malformed input should be ignored or replaced, then the returned value is a

`CoderResult` object with `isError()` and `isMalformed()` methods that both return `true`. The position of the input buffer is at the first malformed character or byte, and the `length()` method of the returned object specifies the number of characters or bytes that are malformed.

- If the input was well-formed but contained characters or bytes that were unmappable—that cannot be encoded or decoded in the specified charset—and if the `CharsetEncoder` or `CharsetDecoder` has not specified that unmappable characters should be ignored or replaced, then the returned value is a `CoderResult` object with `isError()` and `isUnmappable()` methods that both return `true`. The input buffer is positioned at the first unmappable character or byte, and the `length()` method of the `CoderResult` specifies the number of unmappable characters or bytes.

```
public class CoderResult {
// No Constructor
// Public Constants
    public static final CoderResult OVERFLOW;
    public static final CoderResult UNDERFLOW;
// Public Class Methods
    public static CoderResult malformedForLength(int length);
    public static CoderResult unmappableForLength(int length);
// Property Accessor Methods (by property name)
    public boolean isError();
    public boolean isMalformed();
    public boolean isOverflow();
    public boolean isUnderflow();
    public boolean isUnmappable();
// Public Instance Methods
    public int length();
    public void throwException() throws CharacterCodingException;
// Public Methods Overriding Object
    public String toString();
}
```

Returned By: `CharsetDecoder`.{`decode()`, `decodeLoop()`, `flush()`, `implFlush()`},
`CharsetEncoder`.{`encode()`, `encodeLoop()`, `flush()`, `implFlush()`}, `CoderResult`.{`malformedForLength()`,
`unmappableForLength()`}

Type Of: `CoderResult`.{`OVERFLOW`, `UNDERFLOW`}

CodingErrorAction

Java 1.4

java.nio.charset

This class is a typesafe enumeration that defines three constants that serve as the legal argument values to the `onMalformedInput()` and `onUnmappableCharacter()` methods of `CharsetDecoder` and `CharsetEncoder`. These constants specify how malformed input and unmappable error conditions should be handled. The values are:

CodingErrorAction.REPORT

Specifies that the error should be reported. This is done by returning a `CoderResult` object from the three-argument version of `decode()` or `encode()` or by throwing a `MalformedInputException` or `UnmappableCharacterException` from the one-argument version of `decode()` or `encode()`. This is the default action for both error types for `CharsetDecoder` and `CharsetEncoder`.

CodingErrorAction

CodingErrorAction.IGNORE

Specifies that the malformed input or unmappable input character should simply be skipped, with no output.

CodingErrorAction.REPLACE

Specifies that the malformed input or unmappable character should be skipped and the replacement string or replacement bytes should be appended to the output.

See `CharsetDecoder` for more information.

```
public class CodingErrorAction {  
    // No Constructor  
    // Public Constants  
    public static final CodingErrorAction IGNORE;  
    public static final CodingErrorAction REPLACE;  
    public static final CodingErrorAction REPORT;  
    // Public Methods Overriding Object  
    public String toString();  
}
```

Passed To: `CharsetDecoder`.{`implOnMalformedInput()`, `implOnUnmappableCharacter()`, `onMalformedInput()`, `onUnmappableCharacter()`}, `CharsetEncoder`.{`implOnMalformedInput()`, `implOnUnmappableCharacter()`, `onMalformedInput()`, `onUnmappableCharacter()`}

Returned By: `CharsetDecoder`.{`malformedInputAction()`, `unmappableCharacterAction()`}, `CharsetEncoder`.{`malformedInputAction()`, `unmappableCharacterAction()`}

Type Of: `CodingErrorAction`.{`IGNORE`, `REPLACE`, `REPORT`}

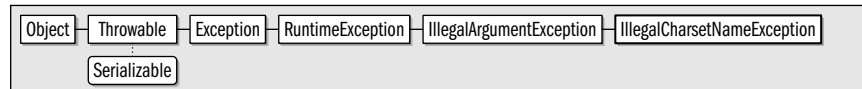
IllegalCharsetNameException

Java 1.4

`java.nio.charset`

serializable unchecked

This signals that a charset name is illegal (for example, one passed to `Charset.forName()` or `Charset.isSupported()`). Charset names may contain only the characters A–Z (upper- and lowercase), the digits 0–9, hyphens, underscores, colons, and periods. They must begin with a letter or a digit, not with a punctuation character.



```
public class IllegalCharsetNameException extends IllegalArgumentException {  
    // Public Constructors  
    public IllegalCharsetNameException(String charsetName);  
    // Public Instance Methods  
    public String getCharsetName();  
}
```

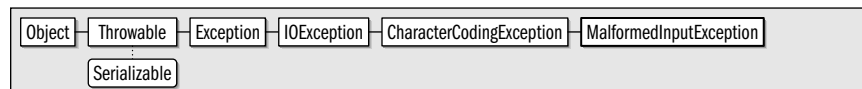
MalformedInputException

Java 1.4

`java.nio.charset`

serializable checked

This signals that input to the `CharsetDecoder.decode()` or `CharsetEncoder.encode()` method was malformed.



```

public class MalformedInputException extends CharacterCodingException {
// Public Constructors
    public MalformedInputException(int inputLength);
// Public Instance Methods
    public int getInputLength();
// Public Methods Overriding Throwable
    public String getMessage();
}

```

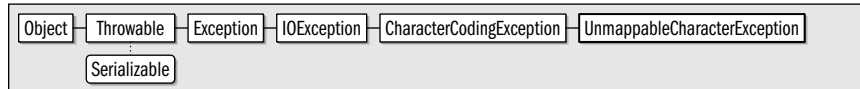
UnmappableCharacterException

Java 1.4

java.nio.charset

serializable checked

This signals that input to the `CharsetDecoder.decode()` or `CharsetEncoder.encode()` method contained a character or byte sequence that is not mappable in the specified charset.



```

public class UnmappableCharacterException extends CharacterCodingException {
// Public Constructors
    public UnmappableCharacterException(int inputLength);
// Public Instance Methods
    public int getInputLength();
// Public Methods Overriding Throwable
    public String getMessage();
}

```

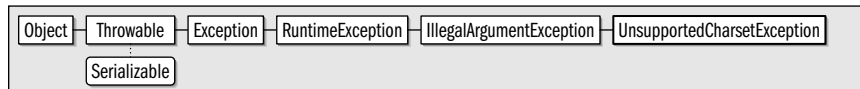
UnsupportedCharsetException

Java 1.4

java.nio.charset

serializable unchecked

This signals that the requested charset is not supported on the current platform. This exception is thrown by `Charset.forName()` when no `Charset` object can be obtained for the named charset. See also `Charset.isSupported()`.



```

public class UnsupportedCharsetException extends IllegalArgumentException {
// Public Constructors
    public UnsupportedCharsetException(String charsetName);
// Public Instance Methods
    public String getCharsetName();
}

```

Package java.nio.charset.spi

Java 1.4

This package defines a “provider” class for system developers who are defining new `Charset` implementations and want to make them available to the system. Application programmers never need to use this package or the class it defines.

Package java.nio.charset.spi

Classes:

public abstract class **CharsetProvider**;

CharsetProvider

Java 1.4

java.nio.charset.spi

System programmers developing new **Charset** implementations should implement this class to make these charsets available to the system. **charsetForName()** should return a **Charset** instance for the given name. **charsets()** should return a **java.util.Iterator** that allows the caller to iterate through the set of **Charset** objects defined by the provider.

A **CharsetProvider** and its associated **Charset** implementations should be packaged in a JAR file and made available to the system in the *jre/lib/ext/* extensions directory (or in another extensions location). The JAR file should contain a file named *META-INF/services/java.nio.charset.spi.CharsetProvider*, which contains the class name of the **CharsetProvider** implementation.

```
public abstract class CharsetProvider {  
    // Protected Constructors  
    protected CharsetProvider();  
    // Public Instance Methods  
    public abstract java.nio.charset.Charset charsetForName(String charsetName);  
    public abstract java.util.Iterator charsets();  
}
```