The OutputStream Interface in C-Sharp

Previous Next

On this page:

Initializing an `OutputStream` in C#

The `OutputStream` class provides a number of overloaded constructors:

C#

```
namespace Ice
{
    public class OutputStream
    {
        public OutputStream();
        public OutputStream(Communicator communicator);
        public OutputStream(Communicator communicator, EncodingVersion version);

        ...
    }
}
```

The constructors optionally accept the following arguments:

- A communicator instance
- An encoding version

We recommend supplying a communicator instance. The stream inspects the communicator's settings to configure several of its own default settings, but you can optionally configure these settings manually using methods that we'll describe later.

If you omit an encoding version, the stream uses the default encoding version of the communicator (if provided) or the most recent encoding version.

If a communicator instance is not available at the time you construct the stream, you can optionally supply it later using one of the overloaded `initialize` methods:

C#

```
namespace Ice
{
    public class OutputStream
    {
        public void initialize(Communicator communicator);
        public void initialize(Communicator communicator, EncodingVersion version);

        ...
    }
}
```

Invoking `initialize` causes the stream to re-initialize its settings based on the configuration of the given communicator.

Use the following method to manually configure the stream:

C#

```
namespace Ice
{
    public class OutputStream
    {
        public void setFormat(FormatType fmt);

        ...
    }
}
```

For instances of Slice classes, the format determines how the slices of an instance are encoded. If the stream is initialized with a communicator, this setting defaults to the value of `Ice.Default.SlicedFormat`, otherwise the setting defaults to the compact format.

Back to Top ^

Inserting into an `OutputStream` in C#

`OutputStream` provides a number of `write` methods that allow you to insert Slice types into the stream.

For example, you can insert a boolean and a sequence of strings into a stream as follows:

C#

```
string[] seq = { "Ice", "rocks!" };
Ice.OutputStream out = new Ice.OutputStream(communicator);
out.writeBool(true);
out.writeStringSeq(seq);
byte[] data = out.finished();
```

Here are the methods for inserting data into a stream:

C#

```
namespace Ice
{
    public class OutputStream
    {
        public void writeBool(bool v);
        public void writeBool(int tag, bool v);
        public void writeBool(int tag, Optional<bool> v);
        public void writeBoolSeq(bool[] v);
        public void writeBoolSeq(int count, IEnumerable<bool> v);
        public void writeBoolSeq(int tag, bool[] v);
        public void writeBoolSeq(int tag, Optional<bool[]> v);
        public void writeBoolSeq<T>(int tag, int count, Ice.Optional<T> v)
            where T : IEnumerable<bool>;
        public void writeBoolSeq(int tag, int count, IEnumerable<bool> v);
        public void rewriteBool(bool v, int dest);

        public void writeByte(byte v);
        public void writeByte(int tag, byte v);
        public void writeByte(int tag, Optional<byte> v);
        public void writeByteSeq(byte[] v);
        public void writeByteSeq(int count, IEnumerable<byte> v);
        public void writeByteSeq(int tag, byte[] v);
        public void writeByteSeq(int tag, Optional<byte[]> v);
        public void writeByteSeq<T>(int tag, int count, Ice.Optional<T> v)
            where T : IEnumerable<byte>;
        public void writeByteSeq(int tag, int count, IEnumerable<byte> v);
        public void rewriteByte(byte v, int dest);

        public void writeShort(short v);
```

```
public void writeShort(int tag, short v);
public void writeShort(int tag, Optional<short> v);
public void writeShortSeq(short[] v);
public void writeShortSeq(int count, IEnumerable<short> v);
public void writeShortSeq(int tag, short[] v);
public void writeShortSeq(int tag, Optional<short[]> v);
public void writeShortSeq<T>(int tag, int count, Ice.Optional<T> v)
    where T : IEnumerable<short>;
public void writeShortSeq(int tag, int count, IEnumerable<short> v);

public void writeInt(int v);
public void writeInt(int tag, int v);
public void writeInt(int tag, Optional<int> v);
public void writeIntSeq(int[] v);
public void writeIntSeq(int count, IEnumerable<int> v);
public void writeIntSeq(int tag, int[] v);
public void writeIntSeq(int tag, Optional<int[]> v);
public void writeIntSeq<T>(int tag, int count, Ice.Optional<T> v)
    where T : IEnumerable<int>;
public void writeIntSeq(int tag, int count, IEnumerable<int> v);
public void rewriteInt(int v, int dest);

public void writeLong(long v);
public void writeLong(int tag, long v);
public void writeLong(int tag, Optional<long> v);
public void writeLongSeq(long[] v);
public void writeLongSeq(int count, IEnumerable<long> v);
public void writeLongSeq(int tag, long[] v);
public void writeLongSeq(int tag, Optional<long[]> v);
public void writeLongSeq<T>(int tag, int count, Ice.Optional<T> v)
    where T : IEnumerable<long>;
public void writeLongSeq(int tag, int count, IEnumerable<long> v);

public void writeFloat(float v);
public void writeFloat(int tag, float v);
public void writeFloat(int tag, Optional<float> v);
public void writeFloatSeq(float[] v);
public void writeFloatSeq(int count, IEnumerable<float> v);
public void writeFloatSeq(int tag, float[] v);
public void writeFloatSeq(int tag, Optional<float[]> v);
public void writeFloatSeq<T>(int tag, int count, Ice.Optional<T> v)
    where T : IEnumerable<float>;
public void writeFloatSeq(int tag, int count, IEnumerable<float> v);

public void writeDouble(double v);
public void writeDouble(int tag, double v);
public void writeDouble(int tag, Optional<double> v);
public void writeDoubleSeq(double[] v);
public void writeDoubleSeq(int count, IEnumerable<double> v);
public void writeDoubleSeq(int tag, double[] v);
public void writeDoubleSeq(int tag, Optional<double[]> v);
public void writeDoubleSeq<T>(int tag, int count, Ice.Optional<T> v)
    where T : IEnumerable<double>;
public void writeDoubleSeq(int tag, int count, IEnumerable<double> v);

public void writeString(string v);
public void writeString(int tag, string v);
public void writeString(int tag, Optional<string> v);
public void writeStringSeq(string[] v);
public void writeStringSeq(int count, IEnumerable<string> v);
public void writeStringSeq(int tag, string[] v);
public void writeStringSeq(int tag, Optional<string[]> v);
public void writeStringSeq<T>(int tag, int count, Ice.Optional<T> v)
    where T : IEnumerable<string>;
public void writeStringSeq(int tag, int count, IEnumerable<string> v);

public void writeSize(int sz);

public void writeProxy(ObjectPrx v);
public void writeProxy(int tag, ObjectPrx v);
public void writeProxy(int tag, Optional<ObjectPrx> v);
```

```
        public void writeValue(Ice.Object v);
        public void writeValue(int tag, Ice.Object v);
        public void writeValue<T>(int tag, Ice.Optional<T> v)
            where T : Ice.Object;

        public void writeEnum(int v, int maxValue);
        public void writeEnum(int tag, int v, int maxValue);

        public void writeBlob(byte[] v);
        public void writeBlob(byte[] v, int off, int len);

        public void writeException(UserException ex);

        public void startValue(SlicedData sd);
        public void endValue();

        public void startException(SlicedData sd);
        public void endException();

        public void startSlice(String typeId, int compactId, bool last);
        public void endSlice();

        public void startEncapsulation(EncodingVersion encoding, FormatType format);
        public void startEncapsulation();
        public void endEncapsulation();
        public void writeEmptyEncapsulation(EncodingVersion encoding);
        public void writeEncapsulation(byte[] v);

        public EncodingVersion getEncoding();

        public void writePendingValues();

        public bool writeOptional(int tag, OptionalFormat format);

        public int pos();
        public void pos(int n);

        public int startSize();
        public void endSize(int pos);

        public byte[] finished();

        public void writeSerializable(object o);

        public void resize(int sz);

        ...
    }
}
```

Member functions are provided for inserting all of the primitive types, as well as sequences of primitive types; these are self-explanatory. The remaining member functions have the following semantics:

- `void write`*`Type`*`(int tag, `*`type`*` v)`
  `void write`*`Type`*`(int tag, Optional<`*`type`*`> v)`
  `void write`*`Type`*`Seq(int tag, `*`type`*`[] v)`
  `void write`*`Type`*`Seq(int tag, Optional<`*`type`*`[]> v)`
  `void write`*`Type`*`Seq(int tag, int count, IEnumerable<`*`type`*`[]> v)`
  Inserts an optional value. Methods that accept `Optional` instances only insert a value if the argument is non-null and contains a value.

- `void rewriteByte(byte v, int dest)`
  `void rewriteBool(bool v, int dest)`
  `void rewriteInt(int v, int dest)`
  Overwrites the byte(s) at an existing location in the buffer at the given destination with a value. These methods do not change the current position of the stream.

- `void writeSize(int sz)`
  The Ice encoding has a compact representation to indicate size. This function converts the given non-negative integer into the proper encoded representation.

- `void writeProxy(Ice.ObjectPrx v)`
  Inserts a proxy.

- `void writeValue(Ice.Object v)`
  Inserts an instance of a Slice class. The Ice encoding for class instances may cause the insertion to be delayed, in which case the stream retains a reference to the given instance and does not insert its state it until `writePendingValues` is invoked on the stream.

- `void writeEnum(int val, int maxValue)`
  Writes the integer value of an enumerator. The `maxValue` argument represents the highest enumerator value in the enumeration. Consider the following definitions:

  Slice

  ```
  enum Color { red, green, blue }
  enum Fruit { Apple, Pear=3, Orange }
  ```

  The maximum value for `Color` is 2, and the maximum value for `Fruit` is 4.

- `void writeBlob(byte[] v)`
  `void writeBlob(byte[] v, int off, int len)`
  Copies the given blob of bytes directly to the stream's internal buffer without modification.

- `void writeException(UserException ex)`
  Inserts a user exception.

- `void startValue(SlicedData sd)`
  `void endValue()`
  When marshaling the slices of a class instance, the application must first call `startValue`, then marshal the slices, and finally call `endValue`. The caller can pass a `SlicedData` object containing the preserved slices of unknown more-derived types, or null if there are no preserved slices.

- `void startException(SlicedData sd)`
  `void endException()`
  When marshaling the slices of an exception, the application must first call `startException`, then marshal the slices, and finally call `endException`. The caller can pass a `SlicedData` object containing the preserved slices of unknown more-derived types, or 0 if there are no preserved slices.

- `void startSlice(String typeId, int compactId, bool last)`
  `void endSlice()`
  Starts and ends a slice of class or exception member data. The call to `startSlice` must include the type ID for the current slice, the corresponding compact ID for the type (if any), and a boolean indicating whether this is the last slice of the class instance or exception. The compact ID is only relevant for class instances; pass a negative value to indicate the encoding should use the string type ID.

- `void startEncapsulation(EncodingVersion encoding, FormatType format)`
  `void startEncapsulation()`
  `void endEncapsulation()`
  Starts and ends an encapsulation, respectively. The first overloading of `startEncapsulation` allows you to specify the encoding version as well as the format to use for any class instances and exceptions marshaled within this encapsulation.

- `void writeEmptyEncapsulation(EncodingVersion encoding)`
  Writes an encapsulation having the given encoding version with no encoded data.

- `void writeEncapsulation(byte[] v)`
  Copies the bytes representing an encapsulation from the given array into the stream.

- `EncodingVersion getEncoding()`
  Returns the encoding version currently being used by the stream.

- `void writePendingValues()`
  Encodes the state of class instances whose insertion was delayed during `writeValue`. This member function must only be called once. For backward compatibility with encoding version 1.0, this function must only be called when non-optional data members or parameters use class types.

- `bool writeOptional(int tag, OptionalFormat fmt)`
  Prepares the stream to write an optional value with the given tag and format. Returns true if the value should be written, or false otherwise. A return value of false indicates that the encoding version in use by the stream does not support optional values. If this method returns true, the data associated with that optional value must be written next. Optional values must be written in order by tag from least to greatest. The `OptionalFormat` enumeration is defined as follows:

  C#

  ```
  namespace Ice
  {
      public enum OptionalFormat
      {
  ```

```
        OptionalFormatF1, OptionalFormatF2, OptionalFormatF4, OptionalFormatF8,
        OptionalFormatSize, OptionalFormatVSize, OptionalFormatFSize,
        OptionalFormatEndMarker
    }
}
```

Refer to the encoding discussion for more information on the meaning of these values.

- `int pos()`
  `void pos(int n)`
  Returns or changes the stream's current position, respectively.

- `int startSize()`
  `void endSize(int n)`
  The encoding for optional values uses a 32-bit integer to hold the size of variable-length types. Calling `startSize` writes a placeholder value for the size and returns the starting position of the size value; after writing the data, call `endSize` to patch the placeholder with the actual size at the given position.

- `byte[] finished()`
  Indicates that marshaling is complete and returns the encoded byte sequence. This member function must only be called once.

- `void writeSerializable(object v)`
  Writes a serializable object to the stream.

- `void resize(int sz)`
  Allocates space for `sz` more bytes in the buffer. The stream implementation internally uses this function prior to copying more data into the buffer.

Back to Top ^

See Also

- Basic Data Encoding
- Data Encoding for Classes
- Data Encoding for Exceptions
- Serializable Objects in C-Sharp

Previous

Next