The OutputStream Interface in Java

An `OutputStream` is created using the following function:

Java

```
package Ice;

public class Util {
    public static OutputStream
    createOutputStream(Communicator communicator);

    public static OutputStream
    createOutputStream(Communicator communicator, EncodingVersion version);
}
```

You can optionally specify an encoding version for the stream, otherwise the stream uses the communicator's default encoding version.

The `OutputStream` class is shown below.

Java

```
package Ice;

public interface OutputStream {
    Communicator communicator();

    void writeBool(boolean v);
    void writeBoolSeq(boolean[] v);

    void writeByte(byte v);
    void writeByteSeq(byte[] v);

    void writeShort(short v);
    void writeShortSeq(short[] v);

    void writeInt(int v);
    void writeIntSeq(int[] v);

    void writeLong(long v);
    void writeLongSeq(long[] v);

    void writeFloat(float v);
    void writeFloatSeq(float[] v);

    void writeDouble(double v);
    void writeDoubleSeq(double[] v);

    void writeString(String v);
    void writeStringSeq(String[] v);

    void writeSize(int sz);

    void writeProxy(ObjectPrx v);

    void writeObject(Ice.Object v);

    void writeEnum(int v, int maxValue);

    void writeException(UserException ex);

    void startObject(SlicedData sd);
    void endObject();
```

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

    void startSlice(String typeId, boolean last);
    void endSlice();

    void startEncapsulation(EncodingVersion encoding, FormatType format);
    void startEncapsulation();
    void endEncapsulation();

    EncodingVersion getEncoding();

    void writePendingObjects();

    boolean writeOptional(int tag, OptionalFormat format);

    int pos();

    void rewrite(int sz, int pos);

    void startSize();
    void endSize();

    byte[] finished();

    void reset(boolean clearBuffer);

    void writeSerializable(java.io.Serializable o);

    void destroy();
}
```

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 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 writeObject(Ice.Object v)`
  Inserts an Ice object. The Ice encoding for class instances may cause the insertion of this object to be delayed, in which case the stream retains a reference to the given object and does not insert its state it until `writePendingObjects` 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 writeException(UserException ex)`
  Inserts a user exception. The exception argument may be an instance of `UserExceptionWriter`, which allows you to implement your own exception marshaling logic.

- `void startObject(SlicedData sd)`
  `void endObject()`
  When marshaling the slices of an object, the application must first call `startObject`, then marshal the slices, and finally call `endObject`. 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 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, boolean last)`
  `void endSlice()`
  Starts and ends a slice of object or exception member data. The call to `startSlice` must include the type ID for the current slice, and a boolean indicating whether this is the last slice of the object or exception.

- `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 objects and exceptions marshaled within this encapsulation.

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

- `void writePendingObjects()`
  Encodes the state of Ice objects whose insertion was delayed during `writeObject`. 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.

- `boolean 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:

  Java

  ```
  package Ice;
  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 rewrite(int sz, int pos)`
  The `pos` method returns the stream's current position, and `rewrite` allows you to overwrite a 32-bit integer value at the given position in the stream. Calling `rewrite` does not change the stream's current position.

- `void startSize()`
  `void endSize()`
  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; after writing the data, call `endSize` to patch the placeholder with the actual size.

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

- `void reset(boolean clearBuffer)`
  Resets the writing position of the stream to the beginning. The boolean argument `clearBuffer` determines whether the stream releases the internal buffer it allocated to hold the encoded data. If `clearBuffer` is true, the stream releases the buffer in order to make it eligible for garbage collection. If `clearBuffer` is false, the stream retains the buffer to avoid generating unnecessary garbage.

- `void writeSerializable(java.io.Serializable v)`
  Writes a serializable Java object to the stream.

- `void destroy()`
  Applications must call this function in order to reclaim resources.

Here is a simple example that demonstrates how to insert a boolean and a sequence of strings into a stream:

Java

```
final String[] seq = { "Ice", "rocks!" };
Ice.OutputStream out = Ice.Util.createOutputStream(communicator);
try {
    out.writeBool(true);
    out.writeStringSeq(seq);
    byte[] data = out.finished();
} finally {
    out.destroy();
}
```

See Also

- Basic Data Encoding
- Data Encoding for Classes
- Data Encoding for Exceptions
- Serializable Objects in Java