

# Intercepting Object Insertion and Extraction in C-Sharp

In some situations it may be necessary to intercept the insertion and extraction of Ice objects. For example, the [Ice extension for PHP](#) is implemented using Ice for C++ but represents Ice objects as native PHP objects. The PHP extension accomplishes this by manually encoding and decoding Ice objects as directed by the [data encoding rules](#). However, the extension obviously cannot pass a native PHP object to the C++ stream function `writeObject`. To bridge this gap between object systems, Ice supplies the classes `ObjectReader` and `ObjectWriter`:

## C#

```
namespace Ice
{
    public abstract class ObjectReader : ObjectImpl
    {
        public abstract void read(InputStream inStream, bool rid);
        // ...
    }

    public abstract class ObjectWriter : ObjectImpl
    {
        public abstract void write(OutputStream outStream);
        // ...
    }
}
```

A foreign Ice object is inserted into a stream using the following technique:

1. A C# "wrapper" class is derived from `ObjectWriter`. This class wraps the foreign object and implements the `write` member function.
2. An instance of the wrapper class is passed to `writeObject`. (This is possible because `ObjectWriter` derives from `Ice.Object`.) Eventually, the `write` member function is invoked on the wrapper instance.
3. The implementation of `write` encodes the object's state as directed by the [data encoding rules for classes](#).

It is the application's responsibility to ensure that there is a one-to-one mapping between foreign Ice objects and wrapper objects. This is necessary in order to ensure the proper encoding of object graphs.

Extracting the state of a foreign Ice object is more complicated than insertion:

1. A C# "wrapper" class is derived from `ObjectReader`. An instance of this class represents a foreign Ice object.
2. An [object factory](#) is installed that returns instances of the wrapper class. Note that a single object factory can be used for all Slice types if it is registered with an empty Slice type ID.
3. A C# callback class implements the `ReadObjectCallback` interface. The implementation of `invoke` expects its argument to be either null or an instance of the wrapper class as returned by the object factory.
4. An instance of the callback class is passed to `readObject`.
5. When the stream is ready to extract the state of an object, it invokes `read` on the wrapper class. The implementation of `read` decodes the object's state as directed by the [data encoding rules for classes](#). The boolean argument to `read` indicates whether the function should invoke `readTypeId` on the stream; it is possible that the type ID of the current slice has already been read, in which case this argument is `false`.
6. The callback object passed to `readObject` is invoked, passing the instance of the wrapper object. All other callback objects representing the same instance in the stream (in case of object graphs) are invoked with the same wrapper object.

## See Also

- [Client-Side Slice-to-PHP Mapping](#)
- [Data Encoding](#)
- [Data Encoding for Classes](#)
- [Class Factories in C#](#)