Simple Example of Class Encoding

On this page:

Sample Class Definitions

We have separately discussed the primary components of the class encoding: slices, references, and type IDs. To make the preceding discussions more concrete, consider the following class definitions:

Slice

```
interface SomeInterface {
     void op1();
};

class Base {
    int baseInt;
    void op2();
    string baseString;
};

class Derived extends Base implements SomeInterface {
    bool derivedBool;
    string derivedString;
    void op3();
    double derivedDouble;
};
```

Note that `Base` and `Derived` have operations, and that `Derived` also implements the interface `SomeInterface`. Because marshaling of classes is concerned with state, not behavior, the operations `op1`, `op2`, and `op3` are simply ignored during marshaling and the on-the-wire representation is as if the classes had been defined as follows:

Slice

```
class Base {
    int baseInt;
    string baseString;
};

class Derived extends Base {
    bool derivedBool;
    string derivedString;
    double derivedDouble;
};
```

Suppose the sender marshals two instances of `Derived` (for example, as two in-parameters in the same request) with these member values:

First instance:

Member	Type	Value	Marshaled size (in bytes)
`baseInt`	`int`	`99`	4
`baseString`	`string`	`"Hello"`	6

`derivedBool`	`bool`	`true`	1
`derivedString`	`string`	`"World!"`	7
`derivedDouble`	`double`	`3.14`	8

Second instance:

Member	Type	Value	Marshaled size (in bytes)
`baseInt`	`int`	`115`	4
`baseString`	`string`	`"Cave"`	5
`derivedBool`	`bool`	`false`	1
`derivedString`	`string`	`"Canem"`	6
`derivedDouble`	`double`	`6.32`	8

We describe how to marshal these instances using versions 1.0 and 1.1 of the encoding in separate sections below.

Class Encoding version 1.0

The sender arbitrarily assigns a non-zero identity to each instance. Typically, the sender will simply consecutively number the instances starting at `1`. For this example, assume that the two instances have the identities `1` and `2`. The marshaled representation for the two instances (assuming that they are marshaled immediately following each other) is shown below:

Marshaled value	Size in bytes	Type	Byte offset
`1` *(identity)*	4	`int`	0
`0` *(marker for class type ID)*	1	`bool`	4
`"::Derived"` *(class type ID)*	10	`string`	5
`20` *(byte count for slice)*	4	`int`	15
`1` *(`derivedBool`)*	1	`bool`	19
`"World!"` *(`derivedString`)*	7	`string`	20
`3.14` *(`derivedDouble`)*	8	`double`	27
`0` *(marker for class type ID)*	1	`bool`	35
`"::Base"` *(type ID)*	7	`string`	36
`14` *(byte count for slice)*	4	`int`	43
`99` *(`baseInt`)*	4	`int`	47
`"Hello"` *(`baseString`)*	6	`string`	51
`0` *(marker for class type ID)*	1	`bool`	57
`"::Ice::Object"` *(class type ID)*	14	`string`	58
`5` *(byte count for slice)*	4	`int`	72
`0` *(number of dictionary entries)*	1	`size`	76
`2` *(identity)*	4	`int`	77
`1` *(marker for class type ID)*	1	`bool`	81
`1` *(class type ID)*	1	`size`	82
`19` *(byte count for slice)*	4	`int`	83
`0` *(`derivedBool`)*	1	`bool`	87
`"Canem"` *(`derivedString`)*	6	`string`	88
`6.32` *(`derivedDouble`)*	8	`double`	94
`1` *(marker for class type ID)*	1	`bool`	102
`2` *(class type ID)*	1	`size`	103

13 *(byte count for slice)*	4	int	104
115 *(baseInt)*	4	int	108
"Cave" *(baseString)*	5	string	112
1 *(marker for class type ID)*	1	bool	117
3 *(class type ID)*	1	size	118
5 *(byte count for slice)*	4	int	119
0 *(number of dictionary entries)*	1	size	123

Note that, because classes (like exceptions) are sent as a sequence of slices, the receiver of a class can slice off any derived parts of a class it does not understand. Also note that (as shown in the above table) each class instance contains three slices. The third slice is for the type `::Ice::Object`, which is the base type of all classes. The class type ID `::Ice::Object` has the number 3 in this example because it is the third distinct type ID that is marshaled by the sender. (See entries at byte offsets 58 and 118 in the above table.) All class instances have this final slice of type `::Ice::Object`.

Marshaling a separate slice for `::Ice::Object` dates back to Ice versions 1.3 and earlier. In those versions, classes carried a facet map that was marshaled as if it were defined as follows:

Slice

```
module Ice {
    class Object;

    dictionary<string, Object> FacetMap;

    class Object {
        FacetMap facets; // No longer exists
    };
};
```

As of Ice version 1.4, this facet map is always empty, that is, the count of entries for the dictionary that is marshaled in the `::Ice::Object` slice is always zero. If a receiver receives a class instance with a non-empty facet map, it must throw a `MarshalException`.

Note that if a class has no data members, a type ID and slice for that class is still marshaled. The byte count of the slice will be 4 in this case, indicating that the slice contains no data.

Class Encoding version 1.1

A leading size value of `1` marks the beginning of an instance, followed by one or more slices.

Class Encoding in the Sliced Format

The marshaled representation for the two instances (assuming that they are marshaled immediately following each other) in the sliced format is shown below:

Marshaled value	Size in bytes	Type	Byte offset
1 *(instance marker)*	1	size	0
17 *(slice flags: string type ID, size is present)*	1	byte	1
"::Derived" *(type ID - assigned index 1)*	10	string	2
20 *(byte count for slice)*	4	int	12
1 *(derivedBool)*	1	bool	16
"World!" *(derivedString)*	7	string	17
3.14 *(derivedDouble)*	8	double	24
49 *(slice flags: string type ID, size is present, last slice)*	1	byte	32
"::Base" *(type ID - assigned index 2)*	7	string	33
14 *(byte count for slice)*	4	int	40

99 *(baseInt)*	4	int	44
"Hello" *(baseString)*	6	string	48
1 *(instance marker)*	1	size	54
18 *(slice flags: index type ID, size is present)*	1	byte	55
1 *(type ID index for Derived)*	1	size	56
19 *(byte count for slice)*	4	int	57
0 *(derivedBool)*	1	bool	61
"Canem" *(derivedString)*	6	string	62
6.32 *(derivedDouble)*	8	double	68
50 *(slice flags: index type ID, size is present, last slice)*	1	byte	76
2 *(type ID index for Base)*	1	size	77
13 *(byte count for slice)*	4	int	78
115 *(baseInt)*	4	int	82
"Cave" *(baseString)*	5	string	86

The sliced format allows the receiver of a class to slice off any derived parts of a class it does not understand, as in version 1.0 of the encoding. Although the sliced format provides equivalent functionality to that of version 1.0, it is significantly more efficient, requiring only 91 bytes to encode our example compared to the 124 bytes required by version 1.0. We could reduce the encoded size even further, while still retaining the ability to slice off unknown types, by using compact type IDs.

Note that if a class has no data members, a type ID and slice for that class is still marshaled. The byte count of the slice will be 4 in this case, indicating that the slice contains no data.

Class Encoding in the Compact Format

The marshaled representation for the two instances (assuming that they are marshaled immediately following each other) in the compact format is shown below:

Marshaled value	Size in bytes	Type	Byte offset
1 *(instance marker)*	1	size	0
1 *(slice flags: string type ID)*	1	byte	1
"::Derived" *(type ID - assigned index 1)*	10	string	2
1 *(derivedBool)*	1	bool	12
"World!" *(derivedString)*	7	string	13
3.14 *(derivedDouble)*	8	double	20
32 *(slice flags: last slice)*	1	byte	28
99 *(baseInt)*	4	int	29
"Hello" *(baseString)*	6	string	33
1 *(instance marker)*	1	size	39
2 *(slice flags: index type ID)*	1	byte	40
1 *(type ID index for Derived)*	1	size	41
0 *(derivedBool)*	1	bool	42
"Canem" *(derivedString)*	6	string	43
6.32 *(derivedDouble)*	8	double	49
32 *(slice flags: last slice)*	1	byte	57
115 *(baseInt)*	4	int	58
"Cave" *(baseString)*	5	string	62

In an effort to conserve bandwidth, the compact format omits certain details that would allow a receiver to slice off derived parts of a class, such as the slice size and the type IDs for base classes. The result is an encoding that requires only 67 bytes for the two sample instances.

Note that if a class has no data members, a type ID and slice for that class is still marshaled. The byte count of the slice will be 4 in this case, indicating that the slice contains no data.

Class Encoding in the Compact Format with Compact Type IDs

Compact type IDs can be used regardless of the sender's chosen format. For the sake of example, we will use compact type IDs together with the compact format to produce the smallest encoding possible. The Slice definitions below reflect the addition of the compact type IDs:

Slice

```
interface SomeInterface {
     void op1();
};

class Base(10) {
    int baseInt;
    void op2();
    string baseString;
};

class Derived(11) extends Base implements SomeInterface {
    bool derivedBool;
    string derivedString;
    void op3();
    double derivedDouble;
};
```

We assign the compact type ID `10` to `Base` and `11` to `Derived`. Note however that assigning a compact type ID to `Base` does not affect the size of the encoded data in our example because the compact format omits type IDs altogether for base types.

The marshaled representation for the two instances (assuming that they are marshaled immediately following each other) in the compact format is shown below:

Marshaled value	Size in bytes	Type	Byte offset
`1` *(instance marker)*	1	`size`	0
`3` *(slice flags: compact type ID)*	1	`byte`	1
`11` *(compact type ID for `Derived`)*	1	`size`	2
`1` *(`derivedBool`)*	1	`bool`	3
`"World!"` *(`derivedString`)*	7	`string`	4
`3.14` *(`derivedDouble`)*	8	`double`	11
32 *(slice flags: last slice)*	1	`byte`	19
`99` *(`baseInt`)*	4	`int`	20
`"Hello"` *(`baseString`)*	6	`string`	24
`1` *(instance marker)*	1	`size`	30
`3` *(slice flags: compact type ID)*	1	`byte`	31
`11` *(compact type ID for `Derived`)*	1	`size`	32
`0` *(`derivedBool`)*	1	`bool`	33
`"Canem"` *(`derivedString`)*	6	`string`	34
`6.32` *(`derivedDouble`)*	8	`double`	40
32 *(slice flags: last slice)*	1	`byte`	48
`115` *(`baseInt`)*	4	`int`	49
`"Cave"` *(`baseString`)*	5	`string`	53

Substituting a compact type ID for its string equivalent reduces the encoded size for the two instances by another nine bytes to 58, less than half the size of version 1.0.

See Also

- Data Encoding for Classes
- Data Encoding for Exceptions
- Basic Data Encoding
- Type IDs