Object Identity

On this page:

The `Ice::Identity` Type

Each Ice object has an object identity defined as follows:

Slice

```
module Ice
{
    struct Identity
    {
        string name;
        string category;
    }
}
```

As you can see, an object identity consists of a pair of strings, a `name` and a `category`. The complete object identity is the combination of `name` and `category`, that is, for two identities to be equal, both `name` and `category` must be the same. The `category` member is usually the empty string, unless you are using servant locators, default servants or callbacks with Glacier2.

If `name` is an empty string, `category` must be the empty string as well. (An identity with an empty `name` and a non-empty `category` is illegal.) If a proxy contains an identity in which `name` is empty, Ice interprets that proxy as a null proxy.

Object identities can be represented as strings; the category part appears first and is followed by the name; the two components are separated by a `/` character, for example:

```
Factory/File
```

In this example, `Factory` is the category, and `File` is the name. If the `name` or `category` member themselves contain a `/` character, the stringified representation escapes the `/` character with a `\`, for example:

```
Factories\/Factory/Node\/File
```

In this example, the category is `Factories/Factory` and the name is `Node/File`.

Syntax for Stringified Identities

You rarely need to write identities as strings because, typically, your code will be using the identity helper functions `identityToString` and `stringToIdentity`, or simply deal with proxies instead of identities. However, on occasion, you will need to use stringified identities in configuration files. If the identities happen to contain meta-characters (such as a slash or backslash), or characters outside the printable ASCII range, these characters may need to be escaped in the stringified representation.

Here are rules that the Ice run time applies when parsing a stringified identity:

1. The parser scans the stringified identity for an unescaped slash character (`/`). If such a slash character can be found, the substrings to the left and right of the slash are parsed as the `category` and `name` members of the identity, respectively; if no such slash character can be found, the entire string is parsed as the `name` member of the identity, and the `category` member is the empty string.
2. Each of the `category` (if present) and `name` substrings are parsed like Slice String Literals, except that an escaped slash character (`\/`) is converted into a simple slash (`/`).

Identity Helper Functions

To make conversion of identities to and from strings easier, Ice provides functions to convert an Identity to and from a native string, using the string format described in the preceding paragraph. These helper functions are called `identityToString` (to stringify an identity into a string) and `stringToIdentity` (to parse a stringified identity and create the corresponding identity).

`ToStringMode` Enumeration

When *stringifying* an identity with `identityToString`, you can choose the algorithm, or mode, used in this "to string" implementation. These modes correspond to the enumerators of the `Ice::ToStringMode` enumeration:

Slice

```
module Ice
{
    local enum ToStringMode { Unicode, ASCII, Compat }
}
```

These modes are used only when you create a stringified identity or proxy. The resulting strings are all in the same format.

The selected mode affects only the handling of non-ASCII characters and non-printable ASCII characters, such as the ASCII character with ordinal value 127 (delete).

The table below shows how these characters are encoded depending of the selected `ToStringMode`:

ToStringMode	Non-Printable ASCII Character other than \a, \b, \f, \n, \r, \t and \v	Non-ASCII Character	Notes
`Unicode`	Replaced by its short universal character name. For example \u007F (delete).	Not escaped - kept as is.	The resulting string contains printable ASCII characters plus possibly non-ASCII characters.
`ASCII`	Replaced by its short universal character name, just like `Unicode`.	If the character's code point is in the BMP, it's escaped with its short universal character name (\unnnn); if the character is outside the BMP, it's escaped with its long universal character name (\Unnnnnnnn). For example \u20ac (euro sign) and \U0001F34C (banana symbol).	The resulting string contains only printable ASCII characters.
`Compat`	Replaced by an octal escape sequence with 3 digits, \ooo. For example \177 (delete).	Replaced by a sequence of 3-digit octal escape sequences, that represent the UTF-8 encoding of this character. For example \342\202\254 (euro sign) and \360\237\215\214 (banana symbol)	The resulting string contains only printable ASCII characters.

The default mode is `Unicode`.

The `Compat` mode is provided for backwards-compatibility with Ice 3.6 and earlier. These older versions do not recognize universal character names and reject non-printable ASCII characters in stringified identities.

`Communicator::identityToString`

The local interface Communicator provides an `identityToString` helper:

Slice

```
module Ice
{
    local interface Communicator
    {
        string identityToString(Identity id);
        // ...
    }
}
```

`identityToString` converts an identity to a string. The `Ice.ToStringMode` property of the communicator controls how non-printable ASCII characters and non-ASCII characters are represented in the resulting string.

`identityToString` and `stringToIdentity`

These language-native helper functions are defined as follows:

C++11

```
namespace Ice
{
    std::string identityToString(const Identity&, ToStringMode = ToStringMode::Unicode);
    Identity stringToIdentity(const std::string&);
}
```

C++98

```
namespace Ice
{
    std::string identityToString(const Identity&, ToStringMode = Unicode);
    Identity stringToIdentity(const std::string&);
}
```

C#

```
namespace Ice
{
    public sealed class Util
    {
        public static string identityToString(Identity id, ToStringMode toStringMode = ToStringMode.
Unicode);
        public static Identity stringToIdentity(string s);
    }
}
```

Java

```
package com.zeroc.Ice;

public final class Util
{
    public static String identityToString(Identity id, ToStringMode toStringMode);
    public static String identityToString(Identity id); // calls identityToString with ToStringMode.Unicode
    public static Identity stringToIdentity(String s);
}
```

Java Compat

```
package Ice;

public final class Util
{
    public static String identityToString(Identity id, ToStringMode toStringMode);
    public static String identityToString(Identity id); // calls identityToString with ToStringMode.Unicode
    public static Identity stringToIdentity(String s);
}
```

JavaScript

```
Ice.stringToIdentity = function(s)
Ice.identityToString = function(ident, toStringMode = Ice.ToStringMode.Unicode)
```

MATLAB

```
% in Ice package

function identity = stringToIdentity(s)
function s = identityToString(identity, toStringMode)
```

ObjC

```
@interface ICEUtil : NSObject
+(NSMutableString*) identityToString:(ICEIdentity*)ident toStringMode:(ICEToStringMode)toStringMode;
+(NSMutableString*) identityToString:(ICEIdentity*)ident; // calls identityToString with ICEUnicode
+(NSMutableString*) identityToString:(ICEIdentity*)ident;
@end
```

PHP

```
namespace Ice
{
    function identityToString($ident, $toStringMode=null) // null corresponds to the Unicode mode
    function stringToIdentity($str)
}
```

Python

```
# in Ice module

def identityToString(ident, toStringMode=None) # None corresponds to the Unicode mode
def stringToIdentity(str)
```

Ruby

```
module Ice
    def Ice.identityToString(str, toStringMode=nil) # nil corresponds to the Unicode mode
    def Ice.stringToIdentity(str)
```

Swift

```
// in Ice module

public func stringToIdentity(_ string: String) throws -> Identity {
    // ...
}

public func identityToString(id: Identity, mode: ToStringMode = ToStringMode.Unicode) -> String {
    // ...
}
```

See Also

- Servant Activation and Deactivation
- Servant Locators
- Default Servants
- Glacier2