Setting Properties

The `setProperty` operation on the `Properties` interface sets a property to the specified value:

Slice

```
module Ice {
    local interface Properties {

        void setProperty(string key, string value);

        // ...
    };
};
```

You can clear a property by setting it to the empty string.

For properties that control the Ice run time and its services (that is, properties that start with one of the reserved prefixes), this operation is useful only if you call it *before* you call `initialize`. This is because property values are usually read by the Ice run time only once, when you call `initialize`, so the Ice run time does not pay attention to a property value that is changed after you have initialized a communicator. Of course, this begs the question of how you can set a property value and have it also recognized by a communicator.

To permit you to set properties before initializing a communicator, the Ice run time provides an overloaded helper function called `createProperties` that creates a property set. In C++, the function is in the `Ice` namespace:

C++

```
namespace Ice {

PropertiesPtr createProperties(const StringConverterPtr& = 0);
PropertiesPtr createProperties(StringSeq&,
                               const PropertiesPtr& = 0,
                               const StringConverterPtr& = 0);
PropertiesPtr createProperties(int&, char*[],
                               const PropertiesPtr& = 0,
                               const StringConverterPtr& = 0);

}
```

The `StringConverter` parameter allows you to parse properties whose values contain non-ASCII characters and to correctly convert these characters into the native codeset. The converter that is passed to `createProperties` remains attached to the returned property set for the life time of the property set.

The function is overloaded to accept either an `argc`/`argv` pair or a `StringSeq`, to aid in parsing properties.

In Java, the functions are static methods of the `Util` class inside the `Ice` package:

Java

```
package Ice;

public final class Util
{
    public static Properties
    createProperties();

    public static Properties
    createProperties(StringSeqHolder args);

    public static Properties
    createProperties(StringSeqHolder args, Properties defaults);

    public static Properties
    createProperties(String[] args);

    public static Properties
    createProperties(String[] args, Properties defaults);

    // ...
}
```

In C#, the `Util` class in the `Ice` namespace supplies equivalent methods:

C#

```
namespace Ice {
    public sealed class Util {
        public static Properties createProperties();
        public static Properties createProperties(ref string[] args);
        public static Properties createProperties(ref string[] args, Properties defaults);
    }
}
```

The Python and Ruby methods reside in the `Ice` module:

```
def createProperties(args=[], defaults=None)
```

In PHP, use the `Ice_createProperties` method:

PHP

```
function Ice_createProperties(args=array(), defaults=null)
```

As for `initialize`, `createProperties` strips Ice-related command-line options from the passed argument vector. (For Java, only the versions that accept a `StringSeqHolder` do this.)

The functions behave as follows:

- The parameter-less version of `createProperties` simply creates an empty property set. It does *not* check `ICE_CONFIG` for a configuration file to parse.
- The other overloads of `createProperties` accept an argument vector and a default property set. The returned property set contains all the property settings that are passed as the default, plus any property settings in the argument vector. If the argument vector sets a property that is also set in the passed default property set, the setting in the argument vector overrides the default. The overloads that accept an argument vector also look for the `--Ice.Config` option; if the argument vector specifies a configuration file, the configuration file is parsed. The order of precedence of property settings, from lowest to highest, is:
    - Property settings passed in the default parameter

- Property settings set in the configuration file
- Property settings in the argument vector.
  The overloads that accept an argument vector also look for the setting of the `ICE_CONFIG` environment variable and, if that variable specifies a configuration file, parse that file. (However, an explicit `--Ice.Config` option in the argument vector or the `defaults` parameter overrides any setting of the `ICE_CONFIG` environment variable.)

`createProperties` is useful if you want to ensure that a property is set to a particular value, regardless of any setting of that property in a configuration file or in the argument vector. Here is an example:

C++

```
// Get the initialized property set.
//
Ice::PropertiesPtr props = Ice::createProperties(argc, argv);

// Make sure that network and protocol tracing are off.
//
props->setProperty("Ice.Trace.Network", "0");
props->setProperty("Ice.Trace.Protocol", "0");

// Initialize a communicator with these properties.
//
Ice::InitializationData id;
id.properties = props;
Ice::CommunicatorPtr ic = Ice::initialize(id);

// ...
```

The equivalent Java code looks as follows:

Java

```
Ice.StringSeqHolder argsH = new Ice.StringSeqHolder(args);
Ice.Properties properties = Ice.Util.createProperties(argsH);
properties.setProperty("Ice.Warn.Connections", "0");
properties.setProperty("Ice.Trace.Protocol", "0");
Ice.InitializationData id = new Ice.InitializationData();
id.properties = properties;
communicator = Ice.Util.initialize(id);
```

We first convert the argument array to an initialized `StringSeqHolder`. This is necessary so `createProperties` can strip Ice-specific settings. In that way, we first obtain an initialized property set, then override the settings for the two tracing properties, and then set the properties in the `InitializationData` structure.

The equivalent Python code is shown next:

Python

```
props = Ice.createProperties(sys.argv)
props.setProperty("Ice.Trace.Network", "0")
props.setProperty("Ice.Trace.Protocol", "0")
id = Ice.InitializationData()
id.properties = props
ic = Ice.initialize(id)
```

This is the equivalent code in Ruby:

Ruby

```
props = Ice::createProperties(ARGV)
props.setProperty("Ice.Trace.Network", "0")
props.setProperty("Ice.Trace.Protocol", "0")
id = Ice::InitializationData.new
id.properties = props
ic = Ice::initialize(id)
```

Finally, we present the code in PHP:

PHP

```
$props = Ice_createProperties($args);
$props->setProperty("Ice.Trace.Network", "0");
$props->setProperty("Ice.Trace.Protocol", "0");
$id = new Ice_InitializationData();
$id->properties = $props;
$ic = Ice_initialize($id);
```

See Also

- The Properties Interface
- Using Configuration Files
- Command-Line Parsing and Initialization
- Reading Properties
- Parsing Properties
- Communicator Initialization
- C++ Strings and Character Encoding