Java Mapping for Interfaces

The mapping of Slice interfaces revolves around the idea that, to invoke a remote operation, you call a member function on a local class instance that is a proxy for the remote object. This makes the mapping easy and intuitive to use because making a remote procedure call is no different from making a local procedure call (apart from error semantics).

On this page:

Java Classes Generated for an Interface

The compiler generates three source files for each Slice interface. In general, for an interface *<interface-name>*, the following source files are created by the compiler:

- *<interface-name>*`.java`
  This source file declares the *<interface-name>* Java interface, which is used in the server-side mapping.
- *<interface-name>*`Prx.java`
  This source file defines the proxy interface *<interface-name>*`Prx`.
- `_`*<interface-name>*`PrxI.java`
  This source file defines an implementation class for the interface's proxy. Applications should not use this type.

Proxy Interfaces

On the client side, a Slice interface maps to a Java interface with methods that correspond to the operations on that interface. Consider the following simple interface:

Slice

```
interface Simple
{
    void op();
}
```

The Slice compiler generates the following definition for use by the client:

Java

```
public interface SimplePrx extends ObjectPrx
{
    void op();
    void op(java.util.Map<String, String> context);
}
```

As you can see, the compiler generates a *proxy interface* `SimplePrx`. In general, the generated name is *<interface-name>*`Prx`.

In the client's address space, an instance of `SimplePrx` is the local ambassador for a remote instance of the `Simple` interface in a server and is known as a proxy instance. All the details about the server-side object, such as its address, what protocol to use, and its object identity are encapsulated in that instance.

Note that `SimplePrx` inherits from `ObjectPrx`. This reflects the fact that all Ice interfaces implicitly inherit from `Ice::Object`.

For each operation in the interface, the proxy interface has a method of the same name. For the preceding example, we find that the operation `op` has been mapped to the method `op`. Also note that `op` is overloaded: the second version of `op` has a parameter `context` of type `java.util.Map<String, String>`. This parameter is for use by the Ice run time to store information about how to deliver a request. You normally do not need to use it. (We examine the `context` parameter in detail in Request Contexts. The parameter is also used by IceStorm.)

Because all the `<interface-name>Prx` types are interfaces, you cannot instantiate an object of such a type. Instead, proxy instances are always instantiated on behalf of the client by the Ice run time, so client code never has any need to instantiate a proxy directly. The proxy references handed out by the Ice run time are always of type `<interface-name>Prx`; the concrete implementation of the interface is part of the Ice run time and does not concern application code.

A value of `null` denotes the null proxy. The null proxy is a dedicated value that indicates that a proxy points "nowhere" (denotes no object).

Interface Inheritance

Inheritance relationships among Slice interfaces are maintained in the generated Java interfaces. For example:

Slice

```
interface A { ... }
interface B { ... }
interface C extends A, B { ... }
```

The generated code for `CPrx` reflects the inheritance hierarchy:

Java

```
public interface CPrx extends APrx, BPrx
{
    ...
}
```

Given a proxy for `C`, a client can invoke any operation defined for interface `C`, as well as any operation inherited from `C`'s base interfaces.

The `ObjectPrx` Interface

All Ice objects have `Object` as the ultimate ancestor type, so all proxies inherit from `ObjectPrx`. `ObjectPrx` provides a number of methods:

Java

```
public interface ObjectPrx
{
    boolean equals(java.lang.Object r);
    Identity ice_getIdentity();
    boolean ice_isA(String id);
    boolean ice_isA(String id, java.util.Map<String, String> context);
    String[] ice_ids();
    String[] ice_ids(java.util.Map<String, String> context);
    String ice_id();
    String ice_id(java.util.Map<String, String> context);
    void ice_ping();
    void ice_ping(java.util.Map<String, String> context);
    // ...
}
```

The methods behave as follows:

- **`equals`**
  This operation compares two proxies for equality. Note that all aspects of proxies are compared by this operation, such as the communication endpoints for the proxy. This means that, in general, if two proxies compare unequal, that does *not* imply that they denote different objects. For example, if two proxies denote the same Ice object via different transport endpoints, `equals` returns `false` even though the proxies denote the same object.

- **`ice_getIdentity`**
  This method returns the identity of the object denoted by the proxy. The identity of an Ice object has the following Slice type:

  Slice

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

  To see whether two proxies denote the same object, first obtain the identity for each object and then compare the identities:

  Java

  ```
  ObjectPrx o1 = ...;
  ObjectPrx o2 = ...;
  Identity i1 = o1.ice_getIdentity();
  Identity i2 = o2.ice_getIdentity();

  if(i1.equals(i2))
  {
      // o1 and o2 denote the same object
  }
  else
  {
      // o1 and o2 denote different objects

  }
  ```

- **`ice_isA`**
  The `ice_isA` method determines whether the object denoted by the proxy supports a specific interface. The argument to `ice_isA` is a type ID. For example, to see whether a proxy of type `ObjectPrx` denotes a `Printer` object, we can write:

  Java

  ```
  ObjectPrx o = ...;
  if(o != null && o.ice_isA("::M:Printer"))
  {
      // o denotes a Printer object
  }
  else
  {
      // o denotes some other type of object

  }
  ```

  Note that we are testing whether the proxy is null before attempting to invoke the `ice_isA` method. This avoids getting a `NullPointerException` if the proxy is null.

- **`ice_ids`**
  The `ice_ids` method returns an array of strings representing all of the type IDs that the object denoted by the proxy supports.

- **ice_id**
  The ice_id method returns the type ID of the object denoted by the proxy. Note that the type returned is the type of the actual object, which may be more derived than the static type of the proxy. For example, if we have a proxy of type BasePrx, with a static type ID of ::M::Base, the return value of ice_id might be ::M::Base, or it might something more derived, such as ::M::Derived.

- **ice_ping**
  The ice_ping method provides a basic reachability test for the object. If the object can physically be contacted (that is, the object exists and its server is running and reachable), the call completes normally; otherwise, it throws an exception that indicates why the object could not be reached, such as ObjectNotExistException or ConnectTimeoutException.

The ice_isA, ice_ids, ice_id, and ice_ping methods are remote operations and therefore support an additional overloading that accepts a request context. Also note that there are other methods in ObjectPrx, not shown here.

Proxy Helper Methods

For each Slice interface, the Slice-to-Java compiler generates static helper methods that support down-casting and type discovery:

Java

```
public interface SimplePrx extends com.zeroc.Ice.ObjectPrx
{
    // ...
    static SimplePrx checkedCast(ObjectPrx b);
    static SimplePrx checkedCast(ObjectPrx b, java.util.Map<String, String> context);
    static SimplePrx checkedCast(ObjectPrx b, String facet);

    static SimplePrx checkedCast(ObjectPrx b, String facet, java.util.Map<String, String> context);
    static SimplePrx uncheckedCast(ObjectPrx b);
    static SimplePrx uncheckedCast(ObjectPrx b, String facet);
    static String ice_staticId();
}
```

For checkedCast, if the passed proxy is for an object of type Simple, or a proxy for an object with a type derived from Simple, the cast returns a non-null reference to a proxy of type SimplePrx; otherwise, if the passed proxy denotes an object of a different type (or if the passed proxy is null), the cast returns a null reference. Overloaded methods allow you to optionally specify a facet and a request context.

Given a proxy of any type, you can use a checkedCast to determine whether the corresponding object supports a given type, for example:

Java

```
ObjectPrx obj = ...;        // Get a proxy from somewhere...

SimplePrx simple = SimplePrx.checkedCast(obj);
if(simple != null)
{
    // Object supports the Simple interface...
}
else
{
    // Object is not of type Simple...

}
```

Note that a checkedCast contacts the server. This is necessary because only the implementation of an object in the server has definite knowledge of the type of an object. As a result, a checkedCast may throw a ConnectTimeoutException or an ObjectNotExistException. (This also explains the need for the helper method: the Ice run time must contact the server, so we cannot use a simple Java down-cast.)

In contrast, an uncheckedCast does not contact the server and unconditionally returns a proxy of the requested type. However, if you do use an uncheckedCast, you must be certain that the proxy really does support the type you are casting to; otherwise, if you get it wrong, you will most likely get a run-time exception when you invoke an operation on the proxy. The most likely error for such a type mismatch is OperationNotExistException. However, other exceptions, such as a marshaling exception are possible as well. And, if the object happens to have an operation with the correct name, but different parameter types, no exception may be reported at all and you simply end up sending the invocation to an object of the wrong type; that object may do rather nonsensical things. To illustrate this, consider the following two interfaces:

Slice

```
interface Process
{
    void launch(int stackSize, int dataSize);
}

// ...

interface Rocket
{
    void launch(float xCoord, float yCoord);
}
```

Suppose you expect to receive a proxy for a `Process` object and use an `uncheckedCast` to down-cast the proxy:

Java

```
ObjectPrx obj = ...;                                // Get proxy...
ProcessPrx process = ProcessPrx.uncheckedCast(obj); // No worries...
process.launch(40, 60);                             // Oops...
```

If the proxy you received actually denotes a `Rocket` object, the error will go undetected by the Ice run time: because `int` and `float` have the same size and because the Ice protocol does not tag data with its type on the wire, the implementation of `Rocket::launch` will simply misinterpret the passed integers as floating-point numbers.

In fairness, this example is somewhat contrived. For such a mistake to go unnoticed at run time, both objects must have an operation with the same name and, in addition, the run-time arguments passed to the operation must have a total marshaled size that matches the number of bytes that are expected by the unmarshaling code on the server side. In practice, this is extremely rare and an incorrect `uncheckedCast` typically results in a run-time exception.

A final warning about down-casts: you must use either a `checkedCast` or an `uncheckedCast` to down-cast a proxy. If you use a Java cast, the behavior is undefined.

Another method generated for every interface is `ice_staticId`, which returns the type ID string corresponding to the interface. As an example, for the Slice interface `Simple` in module `M`, the string returned by `ice_staticId` is `"::M::Simple"`.

Using Proxy Methods

The base proxy class `ObjectPrx` supports a variety of methods for customizing a proxy. Since proxies are immutable, each of these "factory methods" returns a copy of the original proxy that contains the desired modification. For example, you can obtain a proxy configured with a ten second invocation timeout as shown below:

Java

```
ObjectPrx proxy = communicator.stringToProxy(...);
proxy = proxy.ice_invocationTimeout(10000);
```

A factory method returns a new proxy object if the requested modification differs from the current proxy, otherwise it returns the current proxy. With few exceptions, factory methods return a proxy of the same type as the current proxy, therefore it is generally not necessary to repeat a `checkedCast` or `uncheckedCast` after using a factory method. Furthermore, the mapping generates type-specific factory methods so that no casts are necessary:

Java

```
ObjectPrx base = communicator.stringToProxy(...);
HelloPrx hello = HelloPrx.checkedCast(base);
hello = hello.ice_invocationTimeout(10000); // No cast is necessary
hello.sayHello();
```

The only exceptions are the factory methods `ice_facet` and `ice_identity`. Calls to either of these methods may produce a proxy for an object of an unrelated type, therefore they return a base proxy that you must subsequently down-cast to an appropriate type using `checkedCast` or `uncheckedCast`.

Object Identity and Proxy Comparison

Proxies provide an `equals` method that compares proxies:

Java

```
interface ObjectPrx
{
    boolean equals(java.lang.Object r);
}
```

Note that proxy comparison with `equals` uses *all* of the information in a proxy for the comparison. This means that not only the object identity must match for a comparison to succeed, but other details inside the proxy, such as the protocol and endpoint information, must be the same. In other words, comparison with `equals` tests for *proxy* identity, *not* object identity. A common mistake is to write code along the following lines:

Java

```
ObjectPrx p1 = ...;        // Get a proxy...
ObjectPrx p2 = ...;        // Get another proxy...

if(!p1.equals(p2))
{
    // p1 and p2 denote different objects       // WRONG!
}
else
{
    // p1 and p2 denote the same object         // Correct
}
```

Even though `p1` and `p2` differ, they may denote the same Ice object. This can happen because, for example, both `p1` and `p2` embed the same object identity, but each use a different protocol to contact the target object. Similarly, the protocols may be the same, but denote different endpoints (because a single Ice object can be contacted via several different transport endpoints). In other words, if two proxies compare equal with `equals`, we know that the two proxies denote the same object (because they are identical in all respects); however, if two proxies compare unequal with `equals`, we know absolutely nothing: the proxies may or may not denote the same object.

To compare the object identities of two proxies, you can use a helper function in the `Util` class:

Java

```
public final class Util
{
    public static int proxyIdentityCompare(ObjectPrx lhs, ObjectPrx rhs);
    public static int proxyIdentityAndFacetCompare(ObjectPrx lhs, ObjectPrx rhs);
    // ...
}
```

`proxyIdentityCompare` allows you to correctly compare proxies for identity:

Java

```
ObjectPrx p1 = ...;        // Get a proxy...
ObjectPrx p2 = ...;        // Get another proxy...

if(Util.proxyIdentityCompare(p1, p2) != 0)
```

```
{
    // p1 and p2 denote different objects       // Correct
}
else
{
    // p1 and p2 denote the same object         // Correct
}
```

The function returns 0 if the identities are equal, `-1` if `p1` is less than `p2`, and 1 if `p1` is greater than `p2`. (The comparison uses `name` as the major and `category` as the minor sort key.)

The `proxyIdentityAndFacetCompare` function behaves similarly, but compares both the identity and the facet name.

In addition, the Java mapping provides two wrapper classes that allow you to wrap a proxy for use as the key of a hashed collection:

Java

```
public class ProxyIdentityKey
{
    public ProxyIdentityKey(ObjectPrx proxy);
    public int hashCode();
    public boolean equals(java.lang.Object obj);
    public ObjectPrx getProxy();
}

public class ProxyIdentityFacetKey
{
    public ProxyIdentityFacetKey(ObjectPrx proxy);
    public int hashCode();
    public boolean equals(java.lang.Object obj);
    public ObjectPrx getProxy();
}
```

The constructor caches the identity and the hash code of the passed proxy, so calls to `hashCode` and `equals` can be evaluated efficiently. The `getProxy` method returns the proxy that was passed to the constructor.

As for the comparison functions, `ProxyIdentityKey` only uses the proxy's identity, whereas `ProxyIdentityFacetKey` also includes the facet name.

Deserializing Proxies

Proxy objects implement the `java.io.Serializable` interface that enables serialization of proxies to and from a byte stream. You can use the standard class `java.io.ObjectInputStream` to deserialize all Slice types *except* proxies; proxies are a special case because they must be created by a communicator.

To supply a communicator for use in deserializing proxies, an application must use the Ice-provided class `ObjectInputStream`:

Java

```
public class ObjectInputStream extends java.io.ObjectInputStream
{
    public ObjectInputStream(Communicator communicator, java.io.InputStream stream)
        throws java.io.IOException;

    public Communicator getCommunicator();
}
```

The code shown below demonstrates how to use this class:

Java

```
Communicator communicator = ...
byte[] bytes = ... // data to be deserialized
java.io.ByteArrayInputStream byteStream = new java.io.ByteArrayInputStream(bytes);
ObjectInputStream in = new ObjectInputStream(communicator, byteStream);
ObjectPrx proxy = (ObjectPrx)in.readObject();
```

Ice raises `java.io.IOException` if an application attempts to deserialize a proxy without supplying a communicator.

See Also

- Interfaces, Operations, and Exceptions
- Proxies for Ice Objects
- Type IDs
- Java Mapping for Operations
- Request Contexts
- Operations on Object
- Proxy Methods
- Versioning
- IceStorm