Java Mapping for Classes

On this page:

Basic Java Mapping for Classes

A Slice class is mapped to a Java class with the same name. The generated class contains a public data member for each Slice data member (just as for structures and exceptions), and a member function for each operation. Consider the following class definition:

Slice

```
class TimeOfDay {
    short hour;         // 0 - 23
    short minute;       // 0 - 59
    short second;       // 0 - 59
    string format();    // Return time as hh:mm:ss
};
```

The Slice compiler generates the following code for this definition:

Java

```
public interface _TimeOfDayOperations {
    String format(Ice.Current current);
}

public interface _TimeOfDayOperationsNC {
    String format();
}

public abstract class TimeOfDay extends Ice.ObjectImpl
                                implements _TimeOfDayOperations, _TimeOfDayOperationsNC {
    public short hour;
    public short minute;
    public short second;

    public TimeOfDay();
    public TimeOfDay(short hour, short minute, short second);
    // ...
}
```

There are a number of things to note about the generated code:

1. The compiler generates "operations interfaces" called `_TimeOfDayOperations` and `_TimeOfDayOperationsNC`. These interfaces contain a method for each Slice operation of the class.
2. The generated class `TimeOfDay` inherits (indirectly) from `Ice.Object`. This means that all classes implicitly inherit from `Ice.Object`, which is the ultimate ancestor of all classes. Note that `Ice.Object` is *not* the same as `Ice.ObjectPrx`. In other words, you *cannot* pass a class where a proxy is expected and vice versa.
   If a class has only data members, but no operations, the compiler generates a non-abstract class.
3. The generated class contains a public member for each Slice data member.
4. The generated class inherits member functions for each Slice operation from the operations interfaces.
5. The generated class contains two constructors.

There is quite a bit to discuss here, so we will look at each item in turn.

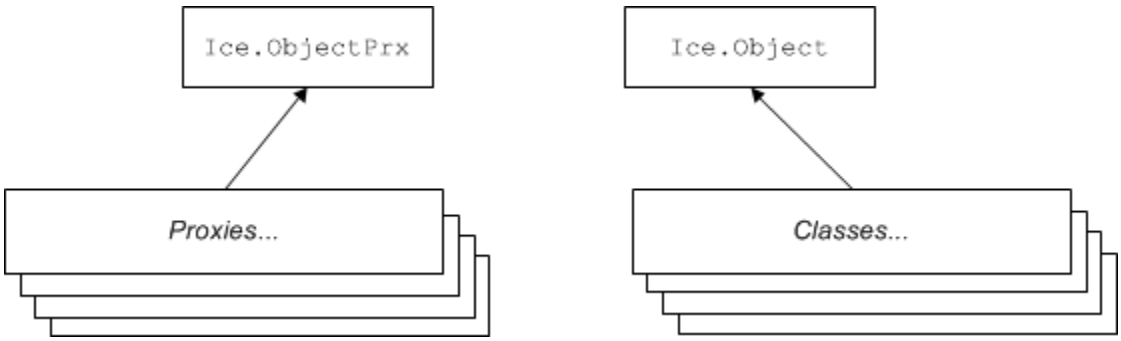
Operations Interfaces in Java

The methods in the `_<interface-name>Operations` interface have an additional trailing parameter of type `Ice.Current`, whereas the methods in the `_<interface-name>OperationsNC` interface lack this additional trailing parameter. The methods without the `Current` parameter simply forward to the methods with a `Current` parameter, supplying a default `Current`. For now, you can ignore this parameter and pretend it does not exist.

If a class has only data members, but no operations, the compiler omits generating the `<_interface-name>Operations` and `_<interface-name>OperationsNC` interfaces.

Inheritance from `Ice.Object` in Java

Like interfaces, classes implicitly inherit from a common base class, `Ice.Object`. However, as shown in the illustration below, classes inherit from `Ice.Object` instead of `Ice.ObjectPrx` (which is at the base of the inheritance hierarchy for proxies). As a result, you cannot pass a class where a proxy is expected (and vice versa) because the base types for classes and proxies are not compatible.



Inheritance from `Ice.ObjectPrx` and `Ice.Object`.

`Ice.Object` contains a number of member functions:

Java

```
package Ice;

public interface Object
{
    boolean ice_isA(String s);
    boolean ice_isA(String s, Current current);

    void ice_ping();
    void ice_ping(Current current);

    String[] ice_ids();
    String[] ice_ids(Current current);

    String ice_id();
    String ice_id(Current current);

    void ice_preMarshal();
    void ice_postUnmarshal();

    DispatchStatus ice_dispatch(Request request, DispatchInterceptorAsyncCallback cb);
}
```

The member functions of `Ice.Object` behave as follows:

- `ice_isA`
  This function returns `true` if the object supports the given type ID, and `false` otherwise.
- `ice_ping`
  As for interfaces, `ice_ping` provides a basic reachability test for the class.

- `ice_ids`
  This function returns a string sequence representing all of the type IDs supported by this object, including `::Ice::Object`.
- `ice_id`
  This function returns the actual run-time type ID for a class. If you call `ice_id` through a reference to a base instance, the returned type id is the actual (possibly more derived) type ID of the instance.
- `ice_preMarshal`
  The Ice run time invokes this function prior to marshaling the object's state, providing the opportunity for a subclass to validate its declared data members.
- `ice_postUnmarshal`
  The Ice run time invokes this function after unmarshaling an object's state. A subclass typically overrides this function when it needs to perform additional initialization using the values of its declared data members.
- `ice_dispatch`
  This function dispatches an incoming request to a servant. It is used in the implementation of dispatch interceptors.

Note that the generated class does not override `hashCode` and `equals`. This means that classes are compared using shallow reference equality, not value equality (as is used for structures).

All Slice classes derive from `Ice.Object` via the `Ice.ObjectImpl` abstract base class. `ObjectImpl` implements the `java.io.Serializable` interface to support Java's serialization facility. `ObjectImpl` also supplies an implementation of `clone` that returns a shallow memberwise copy.

Class Data Members in Java

By default, data members of classes are mapped exactly as for structures and exceptions: for each data member in the Slice definition, the generated class contains a corresponding public data member. A JavaBean-style API is used for optional data members, and you can customize the mapping to force required members to use this same API.

If you wish to restrict access to a data member, you can modify its visibility using the `protected` metadata directive. The presence of this directive causes the Slice compiler to generate the data member with protected visibility. As a result, the member can be accessed only by the class itself or by one of its subclasses. For example, the `TimeOfDay` class shown below has the `protected` metadata directive applied to each of its data members:

Slice

```
class TimeOfDay {
    ["protected"] short hour;   // 0 - 23
    ["protected"] short minute; // 0 - 59
    ["protected"] short second; // 0 - 59
    string format();    // Return time as hh:mm:ss
};
```

The Slice compiler produces the following generated code for this definition:

Java

```
public abstract class TimeOfDay extends Ice.ObjectImpl
                                implements _TimeOfDayOperations,
                                           _TimeOfDayOperationsNC {
    protected short hour;
    protected short minute;
    protected short second;

    public TimeOfDay();
    public TimeOfDay(short hour, short minute, short second);
    // ...
}
```

For a class in which all of the data members are protected, the metadata directive can be applied to the class itself rather than to each member individually. For example, we can rewrite the `TimeOfDay` class as follows:

Slice

```
["protected"] class TimeOfDay {
    short hour;         // 0 - 23
    short minute;       // 0 - 59
    short second;       // 0 - 59
    string format();    // Return time as hh:mm:ss
};
```

Note that you can optionally customize the mapping for data members to use getters and setters instead.

Class Operations in Java

Operations of classes are mapped to abstract member functions in the generated class. This means that, if a class contains operations (such as the `format` operation of our `TimeOfDay` class), you must provide an implementation of the operation in a class that is derived from the generated class. For example:

Java

```
public class TimeOfDayI extends TimeOfDay {
    public String format(Ice.Current current) {
        DecimalFormat df = (DecimalFormat)DecimalFormat.getInstance();
        df.setMinimumIntegerDigits(2);
        return new String(df.format(hour) + ":" + df.format(minute) + ":" +
                          df.format(second));
    }
}
```

Class Factories in Java

Having created a class such as `TimeOfDayI`, we have an implementation and we can instantiate the `TimeOfDayI` class, but we cannot receive it as the return value or as an out-parameter from an operation invocation. To see why, consider the following simple interface:

Slice

```
interface Time {
    TimeOfDay get();
};
```

When a client invokes the `get` operation, the Ice run time must instantiate and return an instance of the `TimeOfDay` class. However, `TimeOfDay` is an abstract class that cannot be instantiated. Unless we tell it, the Ice run time cannot magically know that we have created a `TimeOfDayI` class that implements the abstract `format` operation of the `TimeOfDay` abstract class. In other words, we must provide the Ice run time with a factory that knows that the `TimeOfDay` abstract class has a `TimeOfDayI` concrete implementation. The `Ice::Communicator` interface provides us with the necessary operations:

Slice

```
module Ice {
    local interface ObjectFactory {
        Object create(string type);
        void destroy();
    };

    local interface Communicator {
        void addObjectFactory(ObjectFactory factory, string id);
        ObjectFactory findObjectFactory(string id);
        // ...
    };
};
```

To supply the Ice run time with a factory for our `TimeOfDayI` class, we must implement the `ObjectFactory` interface:

Java

```
class ObjectFactory implements Ice.ObjectFactory {
    public Ice.Object create(String type) {
        if (type.equals(M.TimeOfDay.ice_staticId())) {
            return new TimeOfDayI();
        }
        assert(false);
        return null;
    }

    public void destroy() {
        // Nothing to do
    }
}
```

The object factory's `create` method is called by the Ice run time when it needs to instantiate a `TimeOfDay` class. The factory's `destroy` method is called by the Ice run time when its communicator is destroyed.

The `create` method is passed the type ID of the class to instantiate. For our `TimeOfDay` class, the type ID is `"::M::TimeOfDay"`. Our implementation of `create` checks the type ID: if it matches, the method instantiates and returns a `TimeOfDayI` object. For other type IDs, the method asserts because it does not know how to instantiate other types of objects.

Note that we used the `ice_staticId` method to obtain the type ID rather than embedding a literal string. Using a literal type ID string in your code is discouraged because it can lead to errors that are only detected at run time. For example, if a Slice class or one of its enclosing modules is renamed and the literal string is not changed accordingly, a receiver will fail to unmarshal the object and the Ice run time will raise `NoObjectFactoryException`. By using `ice_staticId` instead, we avoid any risk of a misspelled or obsolete type ID, and we can discover at compile time if a Slice class or module has been renamed.

Given a factory implementation, such as our `ObjectFactory`, we must inform the Ice run time of the existence of the factory:

Java

```
Ice.Communicator ic = ...;
ic.addObjectFactory(new ObjectFactory(), M.TimeOfDay.ice_staticId());
```

Now, whenever the Ice run time needs to instantiate a class with the type ID `"::M::TimeOfDay"`, it calls the `create` method of the registered `ObjectFactory` instance.

The `destroy` operation of the object factory is invoked by the Ice run time when the communicator is destroyed. This gives you a chance to clean up any resources that may be used by your factory. Do not call `destroy` on the factory while it is registered with the communicator — if you do, the Ice run time has no idea that this has happened and, depending on what your `destroy` implementation is doing, may cause undefined behavior when the Ice run time tries to next use the factory.

The run time guarantees that `destroy` will be the last call made on the factory, that is, `create` will not be called concurrently with `destroy`, and `create` will not be called once `destroy` has been called. However, calls to `create` can be made concurrently.

Note that you cannot register a factory for the same type ID twice: if you call `addObjectFactory` with a type ID for which a factory is registered, the Ice run time throws an `AlreadyRegisteredException`.

Finally, keep in mind that if a class has only data members, but no operations, you need not create and register an object factory to transmit instances of such a class. Only if a class has operations do you have to define and register an object factory.

Class Constructors in Java

Classes have a default constructor that default-constructs each data member. This means members of primitive type are initialized to the equivalent of zero, and members of reference type are initialized to null. Note that applications must always explicitly initialize members of structure and enumerated types because the Ice run time does not accept null as a legal value for these types.

If you wish to ensure that data members of primitive and enumerated types are initialized to specific values, you can declare default values in your Slice definition. The default constructor initializes each of these data members to its declared value.

The generated class also contains a second constructor that accepts one argument for each member of the class. This allows you to create and initialize a class in a single statement, for example:

Java

```
TimeOfDayI tod = new TimeOfDayI(14, 45, 00); // 14:45pm
```

For derived classes, the constructor requires an argument for every member of the class, including inherited members. For example, consider the the definition from Class Inheritance once more:

Slice

```
class TimeOfDay {
    short hour;         // 0 - 23
    short minute;       // 0 - 59
    short second;       // 0 - 59
};

class DateTime extends TimeOfDay {
    short day;          // 1 - 31
    short month;        // 1 - 12
    short year;         // 1753 onwards
};
```

The constructors for the generated classes are as follows:

Java

```
public class TimeOfDay extends Ice.ObjectImpl {
    public TimeOfDay() {}

    public TimeOfDay(short hour, short minute, short second)
    {
        this.hour = hour;
        this.minute = minute;
        this.second = second;
    }

    // ...
}

public class DateTime extends TimeOfDay
{
    public DateTime()
    {
        super();
    }

    public DateTime(short hour, short minute, short second,
                    short day, short month, short year)
    {
        super(hour, minute, second);
        this.day = day;
        this.month = month;
        this.year = year;
    }

    // ...
}
```

If you want to instantiate and initialize a `DateTime` instance, you must either use the default constructor or provide values for all of the data members of the instance, including data members of any base classes.

See Also

- Classes
- Class Inheritance
- Java Mapping for Optional Data Members
- Type IDs
- Serializable Objects in Java
- JavaBean Mapping
- The Current Object
- Dispatch Interceptors