Ruby Mapping for Classes

On this page:

- Basic Ruby Mapping for Classes
- Inheritance from Ice::Object in Ruby
- Class Data Members in Ruby
- Class Constructors in Ruby
- Class Operations in Ruby
- Receiving Objects in Ruby
- Class Factories in Ruby

Basic Ruby Mapping for Classes

A Slice class maps to a Ruby class with the same name. For each Slice data member, the generated class contains an instance variable and accessors to read and write it, just as for structures and exceptions. 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 Ruby mapping generates the following code for this definition:

Ruby

```
module TimeOfDay_mixin
    include ::Ice::Object_mixin

    # ...

    def inspect
        # ...
    end

    #
    # Operation signatures.
    #
    # def format()

    attr_accessor :hours, :minutes, :seconds
end
class TimeOfDay
    include TimeOfDay_mixin

    def initialize(hour=0, minute=0, second=0)
        @hour = hour
        @minute = minute
        @second = second
    end

    def TimeOfDay.ice_staticId()
        '::M::TimeOfDay'
    end

    # ...
end
```


Copyright © 2017, ZeroC, Inc.

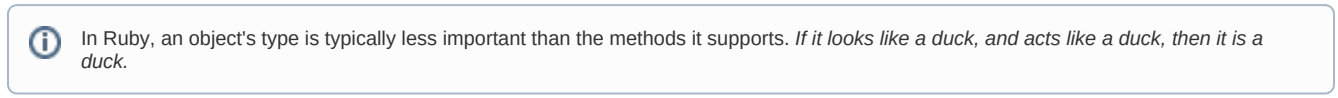
There are a number of things to note about the generated code:

1. The generated class `TimeOfDay` includes the mixin module `TimeOfDay_mixin`, which in turn includes `Ice::Object_mixin`. This reflects the semantics of Slice classes in that all classes implicitly inherit from `Object`, which is the ultimate ancestor of all classes. Note that `Object` is *not* the same as `Ice::ObjectPrx`. In other words, you *cannot* pass a class where a proxy is expected and vice versa.
2. The constructor defines an instance variable for each Slice data member.
3. The class defines the class method `ice_staticId`.
4. A comment summarizes the method signatures for each Slice operation.

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

Inheritance from `Ice::Object` in Ruby

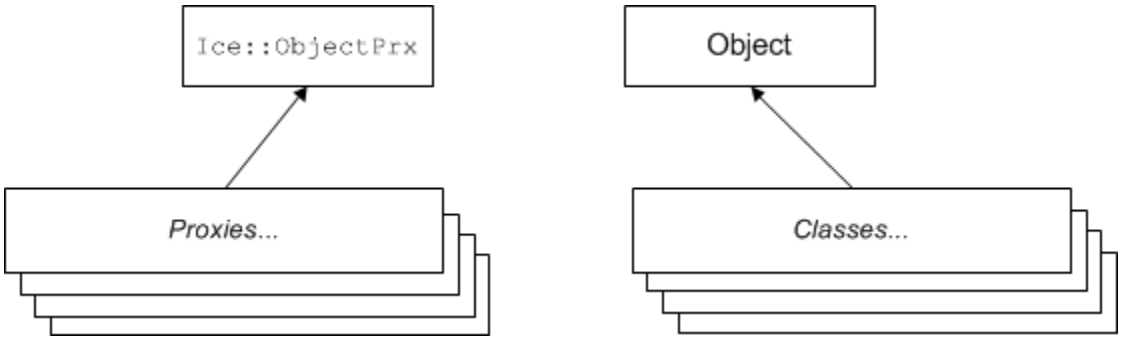
In other language mappings, the inheritance relationship between `Object` and a user-defined Slice class is stated explicitly, in that the generated class derives from a language-specific representation of `Object`. Although its class type allows single inheritance, Ruby's loosely-typed nature places less emphasis on class hierarchies and relies more on *duck typing*.



In Ruby, an object's type is typically less important than the methods it supports. *If it looks like a duck, and acts like a duck, then it is a duck.*

The Slice mapping for a class follows this convention by placing most of the necessary machinery in a mixin module that the generated class includes into its definition. The Ice run time requires an instance of a Slice class to include the mixin module and define values for the declared data members, but does not require that the object be an instance of the generated class.

As shown in the illustration below, classes have no relationship to `Ice::ObjectPrx` (which is at the base of the inheritance hierarchy for proxies), therefore you cannot pass a class where a proxy is expected (and vice versa).



Inheritance from Ice::ObjectPrx and Object.

An instance of a Slice class `C` supports a number of methods:

Ruby

```
def ice_isA(id, current=nil)

def ice_ping(current=nil)

def ice_ids(current=nil)

def ice_id(current=nil)

def C.ice_staticId()

def ice_preMarshal()

def ice_postUnmarshal()
```

The methods behave as follows:

- `ice_isA`
  This method returns `true` if the object supports the given type ID, and `false` otherwise.

Copyright © 2017, ZeroC, Inc.

- `ice_ping`
  As for interfaces, `ice_ping` provides a basic reachability test for the object.
- `ice_ids`
  This method returns a string sequence representing all of the type IDs supported by this object, including `::Ice::Object`.
- `ice_id`
  This method returns the actual run-time type ID of the object. 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_staticId`
  This method returns the static type ID of the class.
- `ice_preMarshal`
  If the object supports this method, the Ice run time invokes it just prior to marshaling the object's state, providing the opportunity for the object to validate its declared data members.
- `ice_postUnmarshal`
  If the object supports this method, the Ice run time invokes it after unmarshaling the object's state. An object typically defines this method when it needs to perform additional initialization using the values of its declared data members.

The mixin module `Ice::Object_mixin` supplies default definitions of `ice_isA` and `ice_ping`. For each Slice class, the generated mixin module defines `ice_ids` and `ice_id`, and the generated class defines the `ice_staticId` method.

Note that neither `Ice::Object` nor the generated class override `hash` and `==`, so the default implementations apply.

Class Data Members in Ruby

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 instance variable and accessor methods.

Optional data members use the same mapping as required data members, but an optional data member can also be set to the marker value `Ice::Unset` to indicate that the member is unset. A well-behaved program must compare an optional data member to `Ice::Unset` before using the member's value:

Ruby

```
v = ...
if v.optionalMember == Ice::Unset
    puts "optionalMember is unset"
else
    puts "optionalMember = " + v.optionalMember
end
```

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:

Copyright © 2017, ZeroC, Inc.

Ruby

```
module TimeOfDay_mixin
    include ::Ice::Object_mixin

    # ...

    #
    # Operation signatures.
    #
    # def format()

    attr_accessor :hours, :minutes, :seconds
    protected :hours, :hours=
    protected :minutes, :minutes=
    protected :seconds, :seconds=
end
class TimeOfDay
    include TimeOfDay_mixin

    def initialize(hour=0, minute=0, second=0)
        @hour = hour
        @minute = minute
        @second = second
    end

    # ...
end
```

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
};
```

Class Constructors in Ruby

Classes have a constructor that assigns to each data member a default value appropriate for its type. You can also declare different default values for data members of primitive and enumerated types.

For derived classes, the constructor has one parameter for each of the base class's data members, plus one parameter for each of the derived class's data members, in base-to-derived order.

Pass the marker value `Ice::Unset` as the value of any optional data members that you wish to be unset.

Class Operations in Ruby

Operations of classes are mapped to methods in the generated class. This means that, if a class contains operations (such as the `format` operation of our `TimeOfDay` class), objects representing instances of `TimeOfDay` must define equivalent methods. For example:

Copyright © 2017, ZeroC, Inc.

Ruby

```
class TimeOfDayI < TimeOfDay
    def format(current=nil)
        sprintf("%02d:%02d:%02d", @hour, @minute, @second)
    end
end
```

In this case our implementation class `TimeOfDayI` derives from the generated class `TimeOfDay`. An alternative is to include the generated mixin module, which makes it possible for the class to derive from a different base class if necessary:

Ruby

```
class TimeOfDayI < SomeOtherClass
    include TimeOfDay_mixin

    def format(current=nil)
        sprintf("%02d:%02d:%02d", @hour, @minute, @second)
    end
end
```

As explained earlier, an implementation of a Slice class must include the mixin module but is not required to derive from the generated class.

Ruby allows an existing class to be reopened in order to augment or replace its functionality. This feature provides another way for us to implement a Slice class: reopen the generated class and define the necessary methods:

Ruby

```
class TimeOfDay
    def format(current=nil)
        sprintf("%02d:%02d:%02d", @hour, @minute, @second)
    end
end
```

As an added benefit, this strategy eliminates the need to define a class factory. The next section describes this subject in more detail.

A Slice class such as `TimeOfDay` that declares or inherits an operation is inherently abstract. Ruby does not support the notion of abstract classes or abstract methods, therefore the mapping merely summarizes the required method signatures in a comment for your convenience.

You may notice that the mapping for an operation adds an optional trailing parameter named `current`. For now, you can ignore this parameter and pretend it does not exist.

Receiving Objects in Ruby

We have discussed the ways you can implement a Slice class, but we also need to examine the semantics of receiving an object as the return value or as an out-parameter from an operation invocation. 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. Unless we tell it otherwise, the Ice run time in Ruby does exactly that: it instantiates the generated class `TimeOfDay`. Although `TimeOfDay` is logically an abstract class because its Slice equivalent defined an operation, Ruby has no notion of abstract classes and therefore it is legal to create an instance of this class. Furthermore, there are situations in which this is exactly the behavior you want:

- when you have reopened the generated class to define its operations, or

Copyright © 2017, ZeroC, Inc.

- when your program uses only the data members of an object and does not invoke any of its operations.

On the other hand, if you have defined a Ruby class that implements the Slice class, you need the Ice run time to return an instance of your class and not an instance of the generated class. The Ice run time cannot magically know about your implementation class, therefore you must inform the Ice run time by installing a class factory.

Class Factories in Ruby

The Ice run time invokes a class factory when it needs to instantiate an object of a particular type. If no factory is found, the Ice run time instantiates the generated class as described above. To install a factory, we use operations provided by the `Ice::Communicator` interface:

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 create an object that supports the `Ice::ObjectFactory` interface:

Ruby

```
class ObjectFactory
    def create(type)
        fail unless type == M::TimeOfDay::ice_staticId()
        TimeOfDayI.new
    end

    def destroy
        # Nothing to do
    end
end
```

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 fails 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:

Ruby

```
ic = ...   # Get Communicator...
ic.addObjectFactory(ObjectFactory.new, M::TimeOfDay::ice_staticId())
```


Copyright © 2017, ZeroC, Inc.

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.

See Also

- Classes
- Type IDs
- Optional Data Members
- The Current Object

Copyright © 2017, ZeroC, Inc.