PHP Mapping for Classes

On this page:

Basic PHP Mapping for Classes

A Slice class maps to a Ruby class with the same name. For each Slice data member, the generated class contains a member variable, 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 PHP mapping generates the following code for this definition:

PHP

```
abstract class TimeOfDay extends Ice_ObjectImpl
{
    public function __construct($hour=0, $minute=0, $second=0)
    {
        $this->hour = $hour;
        $this->minute = $minute;
        $this->second = $second;
    }

    abstract public function format();

    public static function ice_staticId()
    {
        return '::TimeOfDay';
    }

    public function __toString()
    {
        // ...
    }

    public $hour;
    public $minute;
    public $second;
}
```

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

1. The generated class `TimeOfDay` inherits from `Ice_ObjectImpl`. 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 initializes an instance variable for each Slice data member.
3. The class includes an abstract function declaration corresponding to the Slice operation `format`.

4. The class defines the class method `ice_staticId`.

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

Inheritance from `Object` in PHP

Like interfaces, classes implicitly inherit from a common base class, `Ice_Object`. However, classes inherit from `Ice_Object` instead of `Ice_ObjectPrx`, therefore you cannot pass a class where a proxy is expected (and vice versa) because the base types for classes and proxies are not compatible.

`Ice_Object` contains a number of member functions:

PHP

```
interface Ice_Object
{
    public function ice_isA($id);

    public function ice_ping();

    public function ice_ids();

    public function ice_id();

    public function ice_preMarshal();

    public function ice_postUnmarshal();
}
```

The member functions of `Ice_Object` behave as follows:

- `ice_isA`
  This method 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 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_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.

All Slice classes derive from `Ice_Object` via the `Ice_ObjectImpl` abstract base class, which provides default implementations of the `Ice_Object` methods.

Class Data Members in PHP

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 member variable.

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:

PHP

```
$v = ...;
if($v->optionalMember == Ice_Unset)
    echo "optionalMember is unset\n";
else
    echo "optionalMember = " . $v->optionalMember . "\n";
```

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:

PHP

```
abstract class TimeOfDay extends Ice_ObjectImpl
{
    public function __construct($hour=0, $minute=0, $second=0)
    {
        $this->hour = $hour;
        $this->minute = $minute;
        $this->second = $second;
    }

    abstract public function format();

    public static function ice_staticId()
    {
        return '::TimeOfDay';
    }

    public function __toString()
    {
        // ...
    }

    protected $hour;
    protected $minute;
    protected $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
};
```

Class Constructors in PHP

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 PHP

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:

PHP

```
class TimeOfDayI extends TimeOfDay
{
    public function format()
    {
        return strftime("%X");
    }
}
```

Class Factories in PHP

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:

PHP

```
class ObjectFactory implements Ice_ObjectFactory {
    public function create($type) {
        if ($type == TimeOfDay::ice_staticId()) {
            return new TimeOfDayI;
        }
        assert(false);
        return null;
    }

    public function 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 `"::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:

PHP

```
$communicator = ...;
$communicator->addObjectFactory(new ObjectFactory, TimeOfDay::ice_staticId());
```

Now, whenever the Ice run time needs to instantiate a class with the type ID `"::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.

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.

See Also

- Classes
- Type IDs
- Optional Data Members