JavaScript Mapping for Classes

Previous Next

On this page:

Basic JavaScript Mapping for Classes

A Slice class is mapped to a JavaScript class with the same name. For each Slice data member, the JavaScript instance contains a corresponding property (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
}
```

The Slice compiler generates the following code for this definition:

JavaScript

```
class TimeOfDay extends Ice.Value
{
    constructor(hour = 0, minute = 0, second = 0)
    {
        super();
        this.hour = hour;
        this.minute = minute;
        this.second = second;
    }
}
```

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

1. The generated `TimeOfDay` class inherits from `Ice.Value`. Note that `Ice.Value` 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 generated class provides a constructor that accepts a value for each data member.
3. The generated class defines a property for each Slice data member.

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

Back to Top ^

Inheritance from `Ice.Value` in JavaScript

As for Slice interfaces, the generated type for a Slice class implicitly inherits from a common base type. However, the type inherits from `Ice.Value` 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.

`Ice.Value` defines a number of member functions:

JavaScript

```
class Ice.Value
{
    static ice_staticId() {}
    ice_id() {}
    ice_preMarshal() {}
    ice_postUnmarshal() {}
    ice_getSlicedData() {}
}
```

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

- `ice_staticId`
  This function returns the static type ID of a class.
- `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 subtype to validate its declared data members.
- `ice_postUnmarshal`
  The Ice run time invokes this function after unmarshaling an object's state. A subtype typically overrides this function when it needs to perform additional initialization using the values of its declared data members.
- `ice_getSlicedData`
  This functions returns the `SlicedData` object if the value has been sliced during un-marshaling or `null` otherwise.

Back to Top ^

Class Constructors in JavaScript

The type generated for a Slice class provides a constructor that initializes each data member to a default value appropriate for its type:

Data Member Type	Default Value
`string`	Empty string
`enum`	First enumerator in enumeration
`struct`	Default-constructed value
Numeric	Zero
`bool`	False
`sequence`	Null
`dictionary`	Null
`class/interface`	Null

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 constructor initializes each of these data members to its declared value instead.

The constructor accepts one argument for each member of the class. This allows you to create and initialize an instance in a single statement, for example:

JavaScript

```
let 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 generated for these classes are similar to the following:

JavaScript

```
class TimeOfDay extends Ice.Value
{
    constructor(hour = 0, minute = 0, second = 0)
    {
        this.hour = hour;
        this.minute = minute;
        this.second = second;
    }
}

class DateTime extends TimeOfDay
{
    constructor(hour, minute, second, day = 0, month = 0, year = 0)
    {
        super(hour, minute, second);
        this.day = day;
        this.month = month;
        this.year = year;
    }
}
```

Pass `undefined` as the value of any optional data member that you wish to remain unset.

Back to Top ^

Class Data Members in JavaScript

By default, data members of classes are mapped exactly as for structures and exceptions: for each data member in the Slice definition, the generated type defines a corresponding property.

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

JavaScript

```
let v = ...
if(v.optionalMember === undefined)
{
    console.log("optionalMember is unset")
}
else
{
    console.log("optionalMember =", v.optionalMember)
}
```

Back to Top ^

Class Operations in JavaScript

> **Deprecated Feature**
>
> Operations on classes are deprecated as of Ice 3.7. Skip this section unless you need to communicate with old applications that rely on this feature.

With the JavaScript mapping, operations in classes are not mapped at all into the corresponding JavaScript class. The generated JavaScript class is the same whether the Slice class has operations or not.

The Slice to JavaScript compiler also generates a separate `<class-name>Disp` class, which can be used to implement an Ice object with these operations. For example:

Slice

```
class FormattedTimeOfDay
{
    short hour;         // 0 - 23
    short minute;       // 0 - 59
    short second;       // 0 - 59
    string tz;
    string format();
}
```

JavaScript

```
class FormattedTimeOfDay extends Ice.Value
{
     // ... operation format() not mapped at all here
}

// Disp class for servant implementation
class FormattedTimeOfDayDisp extends Ice.Object
{
     // ...
}
```

Back to Top ^

Value Factories in JavaScript

> While value factories are necessary when using classes with operations (a now deprecated feature), value factories may be used for any kind of class and are *not* deprecated.

Value factories allow you to create classes derived from the JavaScript class generated by the Slice compiler, and tell the Ice run time to create instances of these classes when unmarshaling. For example, with the following simple interface:

Slice

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

The Ice run time will by default create and return a plain `TimeOfDay` instance.

If you wish, you can create your own custom derived class, and tell Ice to create and return these instances instead. For example:

JavaScript

```
var communicator = ...;
communicator.getValueFactoryManager().add(
  type => {
    if(type === TimeOfDay.ice_staticId())
      return new TimeOfDayI();
    return null;
  }, TimeOfDay.ice_staticId());
```

Now, whenever the Ice run time needs to instantiate an object with the type ID `"::M::TimeOfDay"`, it calls the registered factory, which returns a `TimeOfDayI` instance to the Ice run time.

Back to Top ^

See Also

- Classes
- Class Inheritance
- Type IDs
- Value Factories

Previous Next