



User Guide

- [Introduction](#)
 - [Related projects](#)
 - [JAXB](#)
- [Features](#)
- [Working Example](#)
 - [Usage example](#)
 - [Defining mappings](#)
 - [Generating mappings from an XML Schema](#)
- [Mapping XML to JavaScript Objects](#)
 - [Basic concepts](#)
 - [Modules](#)
 - [Element declarations](#)
 - [Types](#)
 - [Simple types](#)
 - [Built-in simple types](#)
 - [Deriving simple types by list](#)
 - [Deriving simple types by union](#)
 - [Deriving simple types by restriction](#)
 - [Defining custom simple types](#)
 - [Complex types](#)
 - [Defining complex types by extension](#)
 - [Defining complex types by restriction](#)
 - [Properties](#)
 - [Basic property characteristics](#)
 - [Property name](#)
 - [Property cardinality](#)
 - [Mixed properties](#)
 - [Wrapper elements](#)
 - [Defining properties](#)
 - [Value property](#)
 - [Defining complex type with simple content](#)
 - [Attribute property](#)
 - [Any attribute property](#)
 - [Element property](#)
 - [Element property example - single element](#)
 - [Elements property](#)
 - [Element map property](#)
 - [Element reference property](#)
 - [Scoped elements](#)
 - [Substitution groups](#)
 - [Element references property](#)
 - [Any element property](#)
 - [Any element property example - lax processing](#)
- [Using Jsonix in your JavaScript program](#)
 - [Including Jsonix scripts](#)
 - [Using Jsonix](#)
 - [Creating Jsonix](#)
 - [Marshalling](#)
 - [Unmarshalling](#)
- [Generating mappings from XML Schema](#)
 - [Command-line tool](#)
 - [XJC plugin](#)
 - [Maven usage](#)
 - [Ant usage](#)

Introduction

🟢 Take a look at [this question](#) on Stackoverflow which inceptioned the development of Jsonix.

Jsonix (JSON interfaces for XML) is a JavaScript library which allows you to convert between XML and JSON structures.

With Jsonix you can parse XML into JavaScript objects (this process is called *unmarshalling*) or serialize JavaScript objects in XML form (this is called *marshalling*).

These conversions are based on simple XML/JSON mappings which can be written manually or generated from an XML Schema.

- ✔ Strictly speaking, Jsonix works with JavaScript objects, which are not limited to JSON. But for the sake of simplicity we'll use *JSON* to denote these "plain old simple JavaScript objects".

In short, with Jsonix you can:

- parse XML into JSON;
- serialize JSON into XML;
- define mappings between XML and JSON declaratively;
- generate these mappings from XML Schemas.

Related projects

JAXB

Jsonix is inspired by and based on [JAXB](#) which is a great tool to convert between XML and Java objects. Jsonix is literally a JAXB analog for JavaScript.

Jsonix mappings are heavily influenced by [JAXB annotations](#).

Jsonix schema compiler is based on XJC, schema compiler from the [JAXB Reference Implementation](#).

Features

- Runs in almost **any** modern **browser**
- Runs in [node.js](#)
- Implements **marshalling** (serializing a JavaScript object into XML)
 - Supports **string** data and **DOM** nodes as result
- Implements **unmarshalling** (parsing a JavaScript object from XML)
 - Supports **string** data, **DOM** nodes, **URLs** or **files** (with node.js) as source
- Driven by **declarative** XML/object **mappings** which control how JavaScript object is converted into XML or vice versa
- **Mappings** can be **automatically generated** based on an **XML Schema**
- **Strongly-structured**
 - XML/object mappings describe structures of JavaScript objects
- **Strongly-typed**
 - Conversion between string content on XML side and values on the JavaScript side is controlled by declared property types.
- Provides **extensible type system**
 - Supports most XML Schema simple types
 - Supports enumerations, list and union simple types
 - Allows adding own simple types
 - Supports complex types consisting of several properties
 - Supports deriving complex types by extensions
- Provides **advanced property system**
 - Value, attribute, element, element reference properties for string processing of XML content
 - Any attribute, any element properties for "lax" processing for XML content

Working Example

This chapter demonstrates the usage of Jsonix in a classic "purchase order" example.

Assume you need to develop a JavaScript program which should process an XML in the following XML Schema:

- [Purchase order schema](#) from the [XML Schema Primer](#).

Here's an example of XML for this schema:

```
<purchaseOrder orderDate="1999-10-20">
  <shipTo country="US">
    <name>Alice Smith</name>
    <street>123 Maple Street</street>
```

```

<city>Mill Valley</city>
<state>CA</state>
<zip>90952</zip>
</shipTo>
<billTo country="US">
  <name>Robert Smith</name>
  <street>8 Oak Avenue</street>
  <city>Old Town</city>
  <state>PA</state>
  <zip>95819</zip>
</billTo>
<comment>Hurry, my lawn is going wild!</comment>
<items>
  <item partNum="872-AA">
    <productName>Lawnmower</productName>
    <quantity>1</quantity>
    <USPrice>148.95</USPrice>
    <comment>Confirm this is electric</comment>
  </item>
  <item partNum="926-AA">
    <productName>Baby Monitor</productName>
    <quantity>1</quantity>
    <USPrice>39.98</USPrice>
    <shipDate>1999-05-21</shipDate>
  </item>
</items>
</purchaseOrder>

```

Usage example

Here's how you would parse this XML document with Jsonix:

```

// The PO variable provides Jsonix mappings for the purchase order test case
// Its definition will be shown in the next section

var PO = {
  // ... Declaration of Jsonix mappings for the purchase order schema ...
};

// First we construct a Jsonix context - a factory for unmarshaller (parser)
// and marshaller (serializer)
var context = new Jsonix.Context([PO]);

// Then we create a unmarshaller
var unmarshaller = context.createUnmarshaller();

// Unmarshal an object from the XML retrieved from the URL
unmarshaller.unmarshalURL('po.xml',
// This callback function will be provided with the result
// of the unmarshalling
function (unmarshalled) {
  console.log(unmarshalled.value.shipTo.name); // Alice Smith
  console.log(unmarshalled.value.items.item[1].productName); // Baby Monitor
});

```

The callback function will receive the result of the unmarshalling in a form of a JavaScript object. Here's how it would look like in JavaScript:

```

{
  name: {
    localPart: "purchaseOrder"
  },
  value: {
    orderDate: new Date(1999, 10, 20),
    shipTo: {
      country: "US",
      name: "Alice Smith",
      street: "123 Maple Street",
      city: "Mill Valley",
      state: "CA",
      zip: 90952
    },
    billTo: {
      name: "Robert Smith",
      street: "8 Oak Avenue",
      city: "Old Town",
      state: "PA",
      country: "US",
      zip: 95819
    }
  }
}

```

```

        zip: 90952
    },
    comment: 'Hurry, my lawn is going wild!',
    items: {
        item: [{
            partNum: "872-AA",
            productName: "Lawnmower",
            quantity: 1,
            usPrice: 148.95,
            comment: "Confirm this is electric"
        }, {
            partNum: '926-AA',
            productName: 'Baby Monitor',
            quantity: 1,
            usPrice: 39.98,
            shipDate: new Date(1999, 4, 21)
        }]
    }
}
}
}
}

```

Here's how marshalling of a JavaScript object into XML would look like:

```

// Create a marshaller
var marshaller = context.createMarshaller();
// Marshal a JavaScript Object as XML (DOM Document)
var doc = marshaller.marshallDocument({
    name: {
        localPart: "purchaseOrder"
    },
    value: {
        orderDate: {
            year: 1999,
            month: 10,
            day: 20
        },
        shipTo: {
            country: "US",
            name: "Alice Smith",
            street: "123 Maple Street",
            city: "Mill Valley",
            state: "CA",
            zip: 90952
        },
        billTo: {
            name: "Robert Smith",
            street: "8 Oak Avenue",
            city: "Old Town",
            state: "PA",
            country: "US",
            zip: 95819
        }, // ...
    }
});

```

Try it online in a [fiddle](#).

Defining mappings

Now let us take a look at the XML/object mappings, the part we skipped previously:

```

var PO = {
    name: 'PO',
    typeInfos: [{
        type: 'classInfo',
        localName: 'PurchaseOrderType',
        propertyInfos: [{
            type: 'element',
            name: 'shipTo',
            elementName: 'shipTo',
            typeInfo: 'PO.USAddress'
        }, {
            type: 'element',
            name: 'billTo',
            elementName: 'billTo',
            typeInfo: 'PO.USAddress'
        }, {
            type: 'element',
            name: 'comment',
            elementName: 'comment',
            typeInfo: 'String'
        }
    ]
}

```

```

    }, {
      type: 'element',
      name: 'items',
      elementName: 'items',
      typeInfo: 'PO.Items'
    }, {
      name: 'orderDate',
      typeInfo: 'Calendar',
      attributeName: 'orderDate',
      type: 'attribute'
    }
  ]
}, {
  type: 'classInfo',
  localName: 'Items',
  propertyInfos: [{
    type: 'element',
    name: 'item',
    collection: true,
    elementName: 'item',
    typeInfo: 'PO.Item'
  }]
}, {
  type: 'classInfo',
  localName: 'USAddress',
  propertyInfos: [{
    type: 'element',
    name: 'name',
    elementName: 'name',
    typeInfo: 'String'
  }, {
    type: 'element',
    name: 'street',
    elementName: 'street',
    typeInfo: 'String'
  }, {
    type: 'element',
    name: 'city',
    elementName: 'city',
    typeInfo: 'String'
  }, {
    type: 'element',
    name: 'state',
    elementName: 'state',
    typeInfo: 'String'
  }, {
    type: 'element',
    name: 'zip',
    elementName: 'zip',
    typeInfo: 'Decimal'
  }, {
    name: 'country',
    typeInfo: 'String',
    attributeName: 'country',
    type: 'attribute'
  }]
}, {
  type: 'classInfo',
  localName: 'Item',
  propertyInfos: [{
    type: 'element',
    name: 'productName',
    elementName: 'productName',
    typeInfo: 'String'
  }, {
    type: 'element',
    name: 'quantity',
    elementName: 'quantity',
    typeInfo: 'Int'
  }, {
    type: 'element',
    name: 'usPrice',
    elementName: 'USPrice',
    typeInfo: 'Decimal'
  }, {
    type: 'element',
    name: 'comment',
    elementName: 'comment',
    typeInfo: 'String'
  }, {
    type: 'element',
    name: 'shipDate',
    elementName: 'shipDate',
    typeInfo: 'Calendar'
  }, {
    name: 'partNum'
  }

```

```

        name: 'partNum',
        typeInfo: 'String',
        attributeName: 'partNum',
        type: 'attribute'
    })
  ]],
  elementInfos: [{
    elementName: 'purchaseOrder',
    typeInfo: 'PO.PurchaseOrderType'
  }, {
    elementName: 'comment',
    typeInfo: 'String'
  }]
};
// If we're in node.js environment, export the mappings
if (typeof require === 'function') {
  module.exports.PO = PO;
}

```

Basically, Jsonix mappings is a JavaScript object which describes how XML constructs (simple and complex types, elements, attributes) should be represented in object form. From the other hand, Jsonix mappings define the target object structure: objects, properties, their types and cardinalities. XML/object mappings are required to guarantee strongly-structured and strongly-typed mapping.

Generating mappings from an XML Schema

As we've seen above, Jsonix needs XML/object mappings to operate. These mappings can be created manually, they are just simple JavaScript programs which use Jsonix API.

There is, however, another possibility for creating Jsonix mappings: you can generate them automatically from an XML Schema. Jsonix provides a *schema compiler* which take an XML Schema as input and generates Jsonix mappings for it. So instead of writing `PO.js` per hand you can generate it from an XML Schema (`po.xsd`) using the Jsonix schema compiler:

```

java
-jar jsonix-full-<VERSION>.jar // Run executable Java archive
-d mappings // Target directory
-p PO // Package/Module name
purchaseorder.xsd // Schema

```

You can [download](#) the complete example.

Mapping XML to JavaScript Objects

Jsonix needs XML/object mappings to operate. These mappings can be created manually or generated from an XML Schema. Either way, they are just simple JavaScript objects which define how XML should be mapped to properties of objects and vice versa.

This sections explains concepts of these mappings and describes how to create them.

Basic concepts

Jsonix mappings are defined in a *module object* which provides information about declared types and XML elements which they are mapped to. Below is a very simple module `One` which declares a complex type `One.ValueType` (containing a single property `value`) and maps this type to the global XML element `value`:

```

var One = {
  name: 'One',
  typeInfos: [{
    type: 'classInfo',

    localName: 'ValueType',
    propertyInfos: [{
      name: 'data',
      type: 'value',
      typeInfo: 'String'
    }]
  }],
  elementInfos: [{
    elementName: 'value',
    typeInfo: 'One.ValueType'
  }]
};

```

Provided this module object, we can create a Jsonix context and use it for marshalling or unmarshalling:

```

var context = new Jsonix.Context([One]);
var unmarshaller = context.createUnmarshaller();

```

```
var data = unmarshaller.unmarshalString('<value>Some text.</value>');
console.log(data);
```

See the [Fiddle](#) for this example.

Now we can enumerate basic components of Jsonix mappings:

- [Modules](#)
- [Types](#) and [Properties](#)
- [Element declarations](#)

These components will be described in the following sections.

Modules

Jsonix module is essentially just a simple JavaScript object which declares a set of XML/object mappings.

```
// Module declaration syntax
var MyModule = {
  // Name of the module, required
  name: 'MyModule',
  // Array of types declared by the module, optional
  typeInfos: [ /*...*/ ],
  // Array of element mappings declared by the module, optional
  elementInfos: [ /*...*/ ],
  // Default namespace URI for elements, optional
  defaultElementNamespaceURI: 'http://www.mymodule.org/elements',
  // Default namespace URI for attributes, optional
  defaultAttributeNamespaceURI: 'http://www.mymodule.org/attributes'
};
```

[Fiddle](#).

Name

Each module must have a string `name` property which names a module. The name is useful for locally-named declarations. For instance, in the code below the full name of the `PurchaseOrderType` type info will be `PO.PurchaseOrderType`.

```
var PO = {
  name: 'PO',
  typeInfos: [
    // Full name of the class info is PO.PurchaseOrderType
    {
      type: 'classInfo',
      localName: 'PurchaseOrderType',

      // ...
    }, // ...
  ], // ...
};
```

[Fiddle](#).

- ✔ For backwards-compatibility, `name` property the module is technically not required (you'll get no error if you pass a module without a name). However it is highly recommended to declare this property. We may implement in strict check for name in future versions.

Type infos

Each module may declare zero or more [types](#) using the `typeInfos` property.

Types are roughly equivalent to the global simple and complex types of the XML Schema.

```
var PO = {
  name: 'PO',
  typeInfos: [{
    type: 'classInfo',
    localName: 'PurchaseOrderType',
    // ...
  }, {
    type: 'classInfo',
    localName: 'Items',
    // ...
  }, {
    type: 'classInfo',
```

```

        localName: 'USAddress',
        // ...
    }, {
        type: 'classInfo',
        localName: 'Item',
        // ...
    }],
    elementInfos: [ /* ... */ ]
};

```

[Fiddle.](#)

If type info is declared with a local name, it will get a "full" name based on the pattern `<ModuleName>.<LocalName>`, ex.

`PO.PurchaseOrderType`.

See the [types](#) section for more information.

Element infos

Each module may declare zero or more [element declarations](#).

Element declarations are roughly equivalent to global elements of the XML Schema.

```

var PO = {
    name: 'PO',
    typeInfos: [ /* ... */ ],
    elementInfos: [{
        elementName: 'purchaseOrder',
        typeInfo: 'PO.PurchaseOrderType'
    }, {
        elementName: 'comment',
        typeInfo: 'String'
    }]
};

```

[Fiddle.](#)

The mapping above basically says that `<purchaseOrder .../>` element should be processed using the `PO.PurchaseOrderType` type and `<comment.../>` using the (built-in) string type.

See the [element declarations](#) section for more information.

Default element and attribute namespaces

Element and attribute names can be declared using simple strings, for instance `elementName: 'comment'`. If you use namespaces (I hope you do), you can use the `defaultElementNamespaceURI` or `defaultAttributeNamespaceURI` to declare the namespace for such names. Consider the following example of the mapping.

```

var Qualified = {
    name: 'Qualified ',
    defaultElementNamespaceURI: 'urn:qualified',
    elementInfos: [{
        elementName: 'comment',
        typeInfo: 'String'
    }]
};

```

This will suit XML like:

```
<q:comment xmlns:q="urn:qualified">Some text.</q:comment>
```

[Fiddle.](#)



An alternative would have to declare the `elementName` like this:

```

var Qualified = {
    name: 'Qualified ',
    elementInfos: [{
        elementName: {
            namespaceURI: 'urn:qualified',
            localPart: 'comment'
        },
        typeInfo: 'String'
    }]
};

```



```
    }]  
};
```

[Fiddle.](#)

Which is a little bit more cumbersome.

Element declarations

Every valid XML document has a single root element which is called the *document element*. When unmarshalling an XML document, Jsonix runtime needs to know onto which type does the root element of this document map. For instance that the element `value` maps onto the type `One.ValueType` in the module `One`.

This mapping is defined by the `elementInfos` property of the module object. The `elementInfos` property is an array of element declarations. Each element declaration is an object with the following structure:

```
var MyModule = {  
  // ...  
  elementInfos: [  
    // Element declaration syntax  
    {  
      // Qualified name of the element  
      name: {  
        namespaceURI: 'urn:myNamespaceURI',  
        localPart: 'element'  
      },  
      // Target type of the element  
      typeInfo: 'MyModule.MyType',  
      // Element scope (optional)  
      scope: 'MyModule.AnotherType',  
      // Substitution group (optional)  
      substitutionGroup: {  
        namespaceURI: 'urn:myNamespaceURI',  
        localPart: 'substitutableElement'  
      }  
    }  
  ]  
};
```

[Fiddle.](#)

The `name` property provides the name of the element to be mapped. This can be a qualified name defined by an object with properties `namespaceURI`, `localPart` (and maybe `prefix`) or a string. If name is given as a string, it will be resolved to the qualified name using the `defaultNamespaceURI` of the module.

See [defining element and attribute names](#) for details about name resolution. (TODO)

The `typeInfo` property defines the type which is associated with the given element. It can be a string (name of the type) or an object (full mapping of the type).

See [referencing types](#) for more information about type resolution. (TODO)

- ✔ Element declaration has two more options, [scope](#) and [substitutionHead](#) which will be explained later on.

For example, consider the following element declaration:

```
{  
  name: {  
    namespaceURI: 'urn:myNamespaceURI',  
    localPart: 'element'  
  },  
  typeInfo: 'MyModule.MyType',  
}
```

This declaration maps the following element:

```
<my:element xmlns:my="urn:myNamespaceURI" ...>  
  ...  
</my:element>
```

Onto the type `MyModule.MyType`. So when Jsonix unmarshals such an element it will produce a result like:

```
{  
  name: {
```

```

name: {
  namespaceURI: 'urn:myNamespaceURI',
  localPart: 'myLocalPart'
},
value: {
  // Contents according to the MyModule.MyType
},
TYPE_NAME : 'MyModule.MyType'
}

```

You mostly need to declare only your global elements in `MyModule.elementInfos`. All other elements, attributes etc. mappings are done via properties.

✓ However, you may also need to use `MyModule.elementInfos` to declare [scoped elements](#).

Types

A concept of *type* is a central concept in Jsonix mappings. Element declarations map XML elements onto types; most of the properties have a target type and so on.

Jsonix distinguished two categories of types: [simple](#) and [complex](#) types. The difference between them is that complex types contain [properties](#) whereas simple types don't.

Either way, types can convert between XML structures (elements, attributes, character data) and JavaScript structures (objects, arrays, strings, numbers etc.).

Each type *may* have a name which can be used to reference this type in mappings.

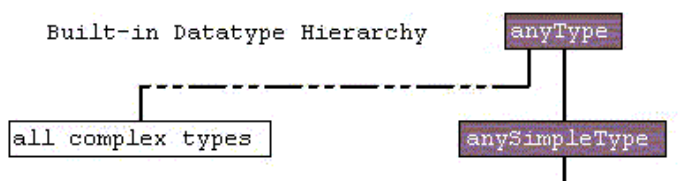
Simple types

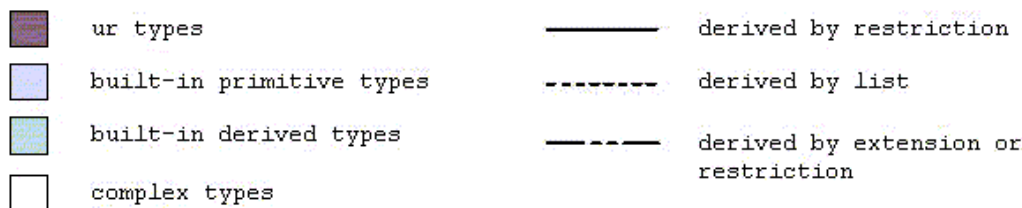
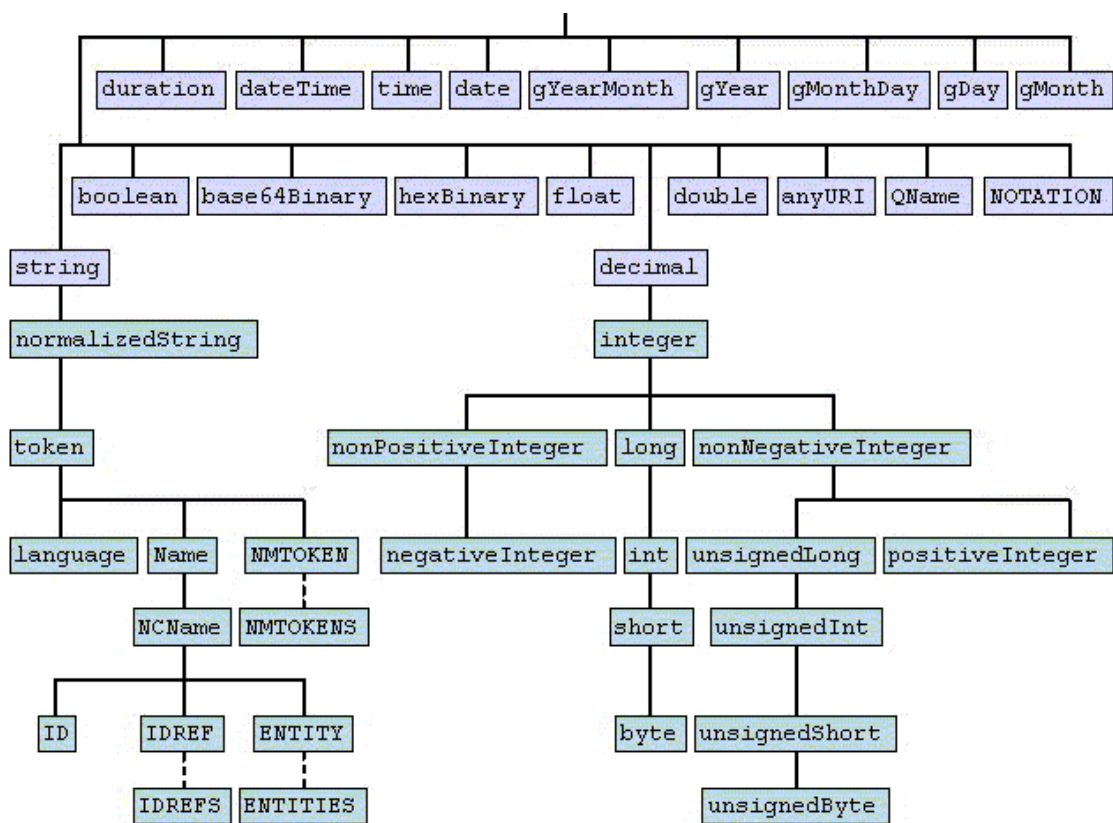
Simple types convert between character data on XML side and primitive or basic types on the JavaScript side. For instance, Jsonix boolean type converts between `"true"` or `"false"` text on XML side and `true` or `false` boolean values on JavaScript.

Jsonix provides supports most simple types defined in the XML Schema [out of the box](#). You can also define your own simple types using [derivation by list](#), [by union](#), defining [enumerations](#) or writing a [custom simple type](#).

Built-in simple types

Jsonix supports most simple types defined in the XML Schema. These types are called *built-in* simple types and are based on the following hierarchy of types:





To support this hierarchy, Jsonix declares an individual JavaScript class for each of these types. Each of the classes also has a pre-instantiated instance (ex. `Jsonix.Schema.XSD.String.INSTANCE`) which can be reference by name (ex. `String`). Below is the type mapping table:

XML Schema Type	Jsonix JavaScript Class	Jsonix Type Name
anySimpleType	Jsonix.Schema.XSD.AnySimpleType	AnySimpleType
string	Jsonix.Schema.XSD.String	String
normalizedString	Jsonix.Schema.XSD.NormalizedString	NormalizedString
token	Jsonix.Schema.XSD.Token	Token
language	Jsonix.Schema.XSD.Language	Language
Name	Jsonix.Schema.XSD.Name	Name
NCName	Jsonix.Schema.XSD.NCName	NCName
boolean	Jsonix.Schema.XSD.Boolean	Boolean
base64Binary	Jsonix.Schema.XSD.Base64Binary	Base64Binary
hexBinary	Jsonix.Schema.XSD.HexBinary	HexBinary
float	Jsonix.Schema.XSD.Float	Float
decimal	Jsonix.Schema.XSD.Decimal	Decimal
integer	Jsonix.Schema.XSD.Integer	Integer
nonPositiveInteger	Jsonix.Schema.XSD.NonPositiveInteger	NonPositiveInteger
negativeInteger	Jsonix.Schema.XSD.NegativeInteger	NegativeInteger
long	Jsonix.Schema.XSD.Long	Long
int	Jsonix.Schema.XSD.Int	Int

short	Jsonix.Schema.XSD.Short	Short
byte	Jsonix.Schema.XSD.Byte	Byte
nonNegativeInteger	Jsonix.Schema.XSD.NonNegativeInteger	NonNegativeInteger
unsignedLong	Jsonix.Schema.XSD.UnsignedLong	UnsignedLong
unsignedInt	Jsonix.Schema.XSD.UnsignedInt	UnsignedInt
unsignedShort	Jsonix.Schema.XSD.UnsignedShort	UnsignedShort
unsignedByte	Jsonix.Schema.XSD.UnsignedByte	UnsignedByte
positiveInteger	Jsonix.Schema.XSD.PositiveInteger	PositiveInteger
double	Jsonix.Schema.XSD.Double	Double
anyURI	Jsonix.Schema.XSD.AnyURI	AnyURI
QName	Jsonix.Schema.XSD.QName	QName
duration	Jsonix.Schema.XSD.Duration	Duration
dateTime	Jsonix.Schema.XSD.DateTime	DateTime
time	Jsonix.Schema.XSD.Time	Time
date	Jsonix.Schema.XSD.Date	Date
gYearMonth	Jsonix.Schema.XSD.GYearMonth	GYearMonth
gYear	Jsonix.Schema.XSD.GYear	GYear
gMonthDay	Jsonix.Schema.XSD.GMonthDay	GMonthDay
gDay	Jsonix.Schema.XSD.GDay	GDay
gMonth	Jsonix.Schema.XSD.GMonth	GMonth

Consider the following declaration of the global element:

```
var MyModule = {
  elementInfos: [{
    elementName: 'comment',
    typeInfo: 'String'
  }]
};
```

This maps the following element:

```
<comment>Some text</comment>
```

Onto the following data:

```
{
  name: {
    localPart: 'comment'
  }
  value: 'Some text'
}
```

[Fiddle](#).

✔ Although all classes are defined, not all the types are already implemented. See [JSNX-1](#) and [JSNX-2](#) issues.

Deriving simple types by list

In addition to atomic simple types Jsonix supports list simple types. Such list types map an *array* of values to string (delimiter-separated items of the array).

Derived type is defined as follows:

```
// List type declaration syntax
{
  // Indicates this is a "list" type info
  type: 'list',
```

```

// Type of the list elements, required
typeInfo: 'String',
// Name of the type, optional. Defaults to typeInfo.name + '*', ex. 'String*'
name: 'Strings',
// Separator characters, optional. Defaults to ' '
separator: ' '
}

```

Example:

```

var MyModule = {
  elementInfos: [{
    elementName: 'comment',
    typeInfo: {
      type: 'list',
      typeInfo: 'String'
    }
  }]
};

```

XML element:

```
<comment>Some text</comment>
```

Data:

```

{
  name: {
    localPart: 'comment'
  }
  value: ['Some', text]
}

```

[Fiddle.](#)



Here's XML Schema analog:

```

<xsd:simpleType>
  <xsd:list itemType="xsd:string"/>
</xsd:simpleType>

```

Unlike XML Schema, Jsonix allows deriving list types from non-atomic types (ex. other list types). Below is an example of a list of lists of doubles:

```

{
  type: 'list',
  typeInfo: {
    type: 'list',
    typeInfo: 'Double'
  },
  separator: ','
}

```

You can use this type to convert between the string `0 0, 0 1, 1 1, 1 0, 0 0` and JavaScript array structure `[[0, 0], [0, 1], [1, 1], [1, 0], [0, 0]]`.

[Fiddle.](#)

Deriving simple types by union

This feature is planned but not supported, see [JSNX-9](#).

Deriving simple types by restriction

Definition of complex types by restriction is not supported at the moment.

Defining custom simple types

Type system in Jsonix is extensible, so if your requirements are not covered by the built-in simple types, you can write custom simple types to match your needs.

Jsonix requires an *instance* of a simple types to provide the following properties and functions:

- `name` Logical name of the type to be used in mappings

- `name` - Logical name of the type to be used in mappings.
- `typeName` - Qualified name of the type (as if you'd define it in an XML Schema). Optional.
- `CLASS_NAME` - String property which provides the name of the class. Required.
- `unmarshal` - Function which accepts `Jsonix.Context` and `Jsonix.XML.Input` and returns unmarshalled value.
- `marshal` - Function which accepts `Jsonix.Context`, `value` and `Jsonix.XML.Output` and marshalls the given value into the output.

In most cases you can just inherit from `Jsonix.Schema.XSD.AnySimpleType` and implement `print` and `parse` methods. See the following custom yes/no boolean type for example:

```
var MyModule = {
  name: 'MyModule',
  typeInfos: [new(Jsonix.Class(Jsonix.Schema.XSD.AnySimpleType, {
    name: 'MyModule.YesNo',
    typeName: new Jsonix.XML.QName('urn:my', 'YesNo'),
    print: function (value) {

      Jsonix.Util.Ensure.ensureBoolean(value);
      return value ? 'yes' : 'no';
    },
    parse: function (text) {
      Jsonix.Util.Ensure.ensureString(text);
      if (text.toLowerCase() === 'yes') {
        return true;
      } else if (text.toLowerCase() === 'no') {
        return false;
      } else {
        throw "Either [yes] or [no] expected as boolean value.";
      }
    },
  },
  CLASS_NAME: 'MyModule.YesNo'
  )))(),
  elementInfos: [{
    elementName: 'data',
    typeInfo: 'MyModule.YesNo'
  }]
};
```

[Fiddle](#).

Complex types

Complex type has a name and contains a number of properties.

```
// Complex type declaration syntax
{
  // Local name of the type, required
  localName: 'ExtendedType',
  // Array of properties of this complex type, required
  propertyInfos: [ /* ... */ ],
  // Base type info, optional
  baseTypeInfo: 'MyModule.BaseType'
}
```

The name is defined using the `localName` property. The full name of the type will be composed of the name of the module and the local name of the complex type, with `.` as delimiter (`MyModule.DataType` in the example above). Name of the type can be used to reference this type in mappings (note the declaration of the `data` element above).

Properties are provided using the `propertyInfos` property. In the example above, we define two properties, `key` (mapped onto the `key` attribute) and `value` (mapped onto the textual contents of the element).

Complex types can be defined using the `typeInfos` property of the module:

```
var MyModule = {
  name: 'MyModule',
  typeInfos: [{
    type: 'classInfo',
    localName: 'DataType',
    propertyInfos: [{
      type: 'value',
      name: 'value',
      typeInfo: 'Integer'
    }, {
      type: 'attribute',
      name: 'key',
      attributeName: 'key',
      typeInfo: 'String'
    }]
  }]
};
```

```

    },
    elementInfos: [{
        elementName: 'data',
        typeInfo: 'MyModule.DataType'
    }]
};

```

[Fiddle.](#)

The mapping above converts between the following XML:

```
<data key="one">1</data>
```

And the following JavaScript object:

```

{
  name: {
    localPart: 'data'
  },
  value: {
    key: 'one',
    value: 1
  }
}

```

[Fiddle.](#)

See [Properties](#) for more information on defining properties.

Properties declared in a complex type define both the structure of the JavaScript object as well as structure of the XML it will be mapped onto.

Defining complex types by extension

Complex types can be defined as extensions for other content types. This is achieved by setting the `baseTypeInfo` property of the extending type to point to the base type. Consider the following example:

```

var MyModule = {
  name: 'MyModule',
  typeInfos: [{
    type: 'classInfo',
    localName: 'BaseType',
    propertyInfos: [{
      type: 'element',
      name: 'alpha',
      elementName: 'Alpha',
      typeInfo: 'String'
    }, {
      type: 'element',
      name: 'beta',
      elementName: 'Beta',
      collection: true,
      typeInfo: 'Integer'
    }]
  }, {
    type: 'classInfo',
    localName: 'ExtendedType',
    baseTypeInfo: 'MyModule.BaseType',
    propertyInfos: [{
      type: 'element',
      name: 'gamma',
      elementName: 'Gamma',
      typeInfo: 'AnyURI'
    }, {
      type: 'element',
      name: 'delta',
      elementName: 'Delta',
      collection: true,
      typeInfo: 'Date'
    }]
  }],
  elementInfos: [{
    elementName: 'Base',
    typeInfo: 'MyModule.BaseType'
  }, {
    elementName: 'Extended',
    typeInfo: 'MyModule.ExtendedType'
  }]
};

```

In this example the base type has the properties `alpha` and `beta` whereas the extended type has four properties `alpha`, `beta`, `gamma` and `delta`. This corresponds to the following XML Schema:

```
<xs:complexType name="BaseType">
  <xs:sequence>
    <xs:element name="Alpha" type="xs:string" minOccurs="0"/>
    <xs:element name="Beta" type="xs:integer" minOccurs="0" maxOccurs="unbounded"/>
  </xs:sequence>
</xs:complexType>

<xs:complexType name="ExtendedType">
  <xs:complexContent>
    <xs:extension base="BaseType">
      <xs:sequence>
        <xs:element name="Gamma" type="xs:anyURI" minOccurs="0"/>
        <xs:element name="Delta" type="xs:date" minOccurs="0" maxOccurs="unbounded"/>
      </xs:sequence>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
```

Here's a couple of examples of XML and equivalent JavaScript objects.

```
<Base>
  <Alpha>one</Alpha>
  <Beta>2</Beta>
  <Beta>3</Beta>
</Base>
```

Turns into:

```
{
  name : { localPart: 'Base' },
  value : {
    alpha : 'one',
    beta : [ 2, 3 ]
  },
  TYPE_NAME: 'MyModule.BaseType'
}
```

Next,

```
<Extended>
  <Alpha>one</Alpha>
  <Beta>2</Beta>
  <Beta>3</Beta>
  <Gamma>urn:four</Gamma>
  <Delta>2005-06-07</Delta>
  <Delta>2008-09-10</Delta>
</Extended>
```

turns into:

```
{
  name : { localPart : 'extended' },
  value : {
    alpha : 'one',
    beta : [ 2, 3 ],
    gamma : 'urn:four',
    delta : [ new Date(2005, 5, 7), new Date(2008, 8, 9) ]
  }
}
```

[Fiddle.](#)

Defining complex types by restriction

Definition of complex types by restriction is not supported.

Properties

Properties define contents of a complex type. From one hand, they configure the structure of a JavaScript object which is mapped by this complex type. From the other hand, they describe, how this object will be presented in an XML form.

Jsonix allows you to map character content, attributes and elements using following property types:

- Character content
 - [Value property](#)
- Attributes
 - [Attribute property](#)
 - [Any attribute property](#)
- Elements
 - [Element property](#)
 - [Elements property](#)
 - [Element map property](#)
 - [Element reference property](#)
 - [Element references property](#)
 - [Any element property](#)

Basic property characteristics

Property types enumerated above have different functionality. However, there are some basic characteristics shared by most properties.

Property name

Every property must have a name (string). Primary function of this name is to define the name of the matching JavaScript object property.

Consider the following example:

```
var MyModule = {
  name: 'MyModule',
  typeInfos: [{
    type: 'classInfo',
    localName: 'MyType',
    propertyInfos: [{
      type: 'element',
      name: 'data',
      elementName: 'content'
    }]
  }],
  elementInfos: [{ /* ... */ }]
};
```

The property named `data` is mapped to the element `content`. So if we'll unmarshal the following element:

```
<root>
  <content>one</content>
</root>
```

We'll get the `data` property in the JavaScript object:

```
{
  name: {
    localPart: 'root'
  },
  value: {
    data: 'one',
    TYPE_NAME: 'MyModule.MyType'
  }
}
```

[Fiddle](#).

- ✔ Name of the property is also used by [attribute](#), [element](#) and [element reference](#) properties to default the target XML attribute or element names if they are omitted.

Property cardinality

Element properties also have the cardinality characteristic; they can be collection or single properties.

- ✔ [Value](#), [attribute](#) and [any attribute](#) properties are always single.

Collection properties handle repeatable elements.

Consider the following collection property declaration:

```
{
  type: 'element',
```

```
name: 'data',
elementName: 'content',
collection : true
}
```

This will unmarshal the following XML:

```
<root>
  <content>one</content>
  <content>two</content>
  <content>three</content>
</root>
```

Into the following JavaScript object:

```
{
  data : ['one', 'two', 'three']
}
```

✔ Note that this is different from [deriving types by list](#):

```
<root>
  <content>one two three</content>
</root>
```

Mixed properties

Some of the properties (namely [Element reference/references](#) and [any element](#) properties) can be declared as *mixed*. Mixed properties can handle elements together with character content. Consider the following example:

```
var MyModule = {
  name: 'MyModule',
  typeInfos: [{
    type: 'classInfo',
    localName: 'MyType',
    propertyInfos: [{
      type: 'elementRef',
      name: 'data',
      elementName: 'content',
      collection : true,
      mixed: true
    }]
  }],
  elementInfos: [{
    elementName: 'root',
    typeInfo: 'MyModule.MyType'
  }]
};
```

Here's an example of XML:

```
<root>
  <content>one</content>two<content>three</content>
</root>
```

And the equivalent JavaScript object:

```
{
  name: {
    localPart: 'root'
  },
  value: {
    data: [{
      name: {
        localPart: 'content'
      },
      value: 'one'
    },
    'two', {
      name: {
        localPart: 'content'
      },
      value: 'three'
    }
  ]
}
```

```
}
}
```

[Fiddle.](#)

Wrapper elements

A common XML Schema design pattern is the usage of wrapper elements to enclose repeated elements, for instance:

```
<root>
  <contents>
    <content>one</content>
    <content>two</content>
    <content>three</content>
  </contents>
</root>
```

The `contents` element on its own has no meaning, it only encloses the `content` subelements. You can model such XML by using the `wrapperElementName` option in element properties:

```
var MyModule = {
  name: 'MyModule',
  typeInfos: [{
    type: 'classInfo',
    localName: 'MyType',
    propertyInfos: [{
      type: 'element',
      name: 'data',
      wrapperElementName: 'contents',
      elementName: 'content',
      collection: true
    }]
  }],
  elementInfos: [{
    elementName: 'root',
    typeInfo: 'MyModule.MyType'
  }]
};
```

The XML sample above will be marshalled into the following JavaScript object:

```
{
  name: {
    localPart: 'root'
  },
  value: {
    data: [ 'one', 'two', 'three' ]
  }
}
```

[Fiddle.](#)

Defining properties

Value property

Property declaration syntax:

```
{
  type: 'value',

  // Name of the JavaScript object property
  name: 'data',

  // Type of the property
  typeInfo: 'Double'
}
```

Value property maps to the textual content of the XML element:

```
var MyModule = {
  name: 'MyModule',
  typeInfos: [{
    type: 'classInfo',
    localName: 'MyValueType',
    propertyInfos: [{
      type: 'value',
```

```

    --
    typeInfo: 'Double',
    name: 'data'
  }]
}],
elementInfos: [{
  elementName: 'root',
  typeInfo: 'MyModule.MyValueType'
}]
}];

```

XML:

```
<root>1.234</root>
```

JavaScript Object:

```

{
  name: {
    localPart: 'root'
  },
  value: {
    data: 1.234,
    TYPE_NAME: 'MyModule.MyValueType'
  }
}

```

[Fiddle.](#)

Usage constraints:

- Complex type can define at most one value property.
- Value property can be used with [attribute](#) or [any attribute](#) properties. It can not be used with:
 - [Element property](#)
 - [Elements property](#)
 - [Element reference property](#)
 - [Element references property](#)
 - [Element references property](#)
 - [Mixed properties](#) without wrapper elements

Defining complex type with simple content

The [value property](#) can be used to define complex type with simple content. Consider the following XML Schema fragment:

```

<xs:element name="root">
  <xs:complexType>
    <xs:simpleContent>
      <xs:extension base="xs:double"/>
    </xs:simpleContent>
  </xs:complexType>
</xs:element>

```

The anonymous complex type within the `root` element is a complex type with simple content. This is how it can be mapped with Jsonix:

```

{
  type: 'classInfo',
  localName: 'RootType',
  propertyInfos: [{
    type: 'value',
    typeInfo: 'Double',
    name: 'value'
  }]
}

```

[Fiddle.](#)

Attribute property

Property declaration syntax:

```

{
  type: 'attribute',

  // Name of the JavaScript object property
  name: 'data',

  // Attribute name, string or QName, defaults to the name of the property

```

```

    attributeName : 'myAttribute',
    // Or as QName
    // attributeName : { localPart: 'myAttribute', namespaceURI : 'urn:mynamespace' }

    // Type of the property
    typeInfo: 'Double'
  }

```

Mapping example:

```

var MyModule = {
  name: 'MyModule',
  typeInfos: [{
    type: 'classInfo',
    localName: 'InputType',
    propertyInfos: [{
      type: 'attribute',
      typeInfo: 'Boolean',
      name: 'checked'
    }]
  }],
  elementInfos: [{
    elementName: 'input',
    typeInfo: 'MyModule.InputType'
  }]
};

```

XML:

```

<input checked="false"/>

```

JavaScript object:

```

{
  name: {
    localPart: 'input'
  },
  value: {
    checked: false,
    TYPE_NAME: 'MyModule.InputType'
  }
}

```

[Fiddle.](#)

Usage constraints:

- Complex type can define at most one attribute property for the given attribute name.

Any attribute property

Property declaration syntax:

```

{
  type: 'anyAttribute',

  // Name of the JavaScript object property
  name: 'attributes'
}

```

"Any attribute" property maps to the attributes of the XML element.

Value of the property is a map of the form:

```

{
  attribute0: value0,
  attribute1: value1,
  attribute2: value2,
  ...
}

```

Where attributeX is the string representation of the qualified attribute name, valueX is the string value of the attribute.

Mapping example:

```

var MyModule = {
  name: 'MyModule',

```

```

typeInfos: [{
  type: 'classInfo',
  localName: 'AnyAttributeType',
  propertyInfos: [{
    type: 'anyAttribute',
    name: 'attributes'
  }]
}],
elementInfos: [{
  elementName: 'anyAttribute',
  typeInfo: 'MyModule.AnyAttributeType'
}]
};

```

XML:

```

<anyAttribute
  a="a"
  b:b="b" xmlns:b="urn:b"
  c:c="c" xmlns:c="urn:c"/>

```

JavaScript Object:

```

{
  name: {
    localPart: 'anyAttribute'
  },
  value: {
    attributes: {
      a: 'a',
      '{urn:b}b': 'b',
      '{urn:c}c': 'c'
    }
  }
}

```

[Fiddle](#).

Usage constraints:

- Complex type can define at most one "any attribute" property.

Element property

Property declaration syntax:

```

{
  type: 'element',

  // Name of the property, required
  name: 'element',

  // Whether the property is collection or not, defaults to false
  collection: false,

  // Name of the element, optional, defaults to the name of the property
  elementName: 'myElement',
  // Or, as QName
  // elementName : { localPart: 'myElement', namespaceURI: 'urn:mynamespace' }

  // Name of the wrapper element, defaults to null
  wrapperElementName: 'myElements',
  // Or, as QName
  // wrapperElementName : { localPart: 'myElements', namespaceURI: 'urn:mynamespace' }

  // Type of the property (can be simple or complex), required
  typeInfo: 'String'
}

```

Element property maps a JavaScript object property onto the XML element.

See [Wrapper elements](#) for an explanation of the `wrapperElementName` option.

Usage constraints:

- Element property can not be used with:
 - [Value properties](#)
 - [Mixed properties](#) without wrapper elements

Element property example - single element

Mapping example:

```
var MyModule = {
  name: 'MyModule',
  typeInfos: [{
    type: 'classInfo',
    localName: 'ElementType',
    propertyInfos: [{
      type: 'element',
      name: 'element',
      typeInfo: 'String'
    }]
  }],
  elementInfos: [{
    elementName: 'elements',
    typeInfo: 'MyModule.ElementType'
  }]
};
```

XML:

```
<elements>
  <element>fire</element>
</elements>
```

JavaScript object:

```
{
  name: {
    localPart: 'elements'
  },
  value: {
    element: 'fire'
  }
}
```

[Fiddle.](#)

Elements property

Property declaration syntax:

```
{
  type: 'elements',

  // Name of the property, required
  name: 'elements',

  // Whether the property is collection or not, defaults to false
  collection: true,

  // Name of the wrapper element, defaults to null
  wrapperElementName: 'myElements',

  // Element mappings, required
  elementTypeInfos: [{
    // Name of the element, required (can be string or a QName)
    elementName: 'string',

    // Type of the property , required
    typeInfo: 'String'
  }, {
    elementName: 'integer',
    typeInfo: 'Integer'
  }]
}
```

Elements property maps several XML elements onto one JavaScript object property.

Elements property is provided with `elementTypeInfos`, an array of element/type mappings. These mappings are objects carrying `elementName`, string qualified name of the element and `typeInfo`, type of the element.

When unmarshalling an element, this property uses the name of the element to find the corresponding type and then uses this type for actual unmarshalling.

When marshalling a value this property searches for a matching type for this value and then uses the corresponding element name to

When marshalling a value, the property is searched for a matching type for the value and then uses the corresponding element name to create the outgoing XML element.

Mapping example:

```
var MyModule = {
  name: 'MyModule',
  typeInfos: [{
    type: 'classInfo',
    localName: 'ElementsType',
    propertyInfos: [{
      type: 'elements',
      name: 'elements',
      wrapperElementName: 'elements',
      collection: true,
      elementTypeInfos: [{
        elementName: 'string',
        typeInfo: 'String'
      }, {
        elementName: 'integer',
        typeInfo: 'Integer'
      }]
    }]
  }],
  elementInfos: [{
    elementName: 'root',
    typeInfo: 'MyModule.ElementsType'
  }]
};
```

XML:

```
<root>
  <elements>
    <string>one</string>
    <integer>2</integer>
    <string>three</string>
  </elements>
</root>
```

JavaScript object:

```
{
  name: {
    localPart: 'root'
  },
  value: {
    elements: ['one', 2, 'three']
  }
}
```

[Fiddle.](#)

As you see, we're getting elements of different types (strings, integers) from differently-named elements (`string`, `integer`) in one array property `elements`. The `elementTypeInfos` definition of our `elements` property allows Jsonix to understand that `string` elements must be unmarshalled as strings, `integer` elements - as integers. During marshalling, Jsonix tries to find a matching type (that is, a type for which this given value would be an instance of) and then use the corresponding element name for marshalling.

The "instance of" operator is implemented differently for simple and complex types.

For simple types the "instance of" operator checks that value has an appropriate JavaScript type (like, `string` for strings, `number` for numeric types, `boolean` for booleans and so on). Simple type also checks that value is actually allowed (ex. integers are round numbers, bytes are in range from -128 to +127 and so on). This can surely be ambiguous, so caution is advised when mixing compatible simple types in one `elements` property.

Values of complex types are objects so there is no reliable way to determine if a given object is considered as "instance of" a certain complex type. To overcome this difficulty, objects may carry a special-purpose `TYPE_NAME` property, for instance:

```
{
  value : 'two',
  TYPE_NAME : 'MyModule.ValueType'
}
```

Complex type thinks that value is an "instance of" itself if the value is an object and it has a string `TYPE_NAME` property matching the name of the complex type.

Element map property

Element map property

Property declaration syntax:

```
{
  type: 'elementMap',

  // Name of the property, required
  name: 'element',

  // Whether the property is collection or not, defaults to false
  collection: true,

  // Name of the element, optional, defaults to the name of the property (string or a QName)
  elementName: 'myElement',

  // Name of the wrapper element, defaults to null (string or a QName)
  wrapperElementName: 'myElements',

  // Declaration of the key property
  key: {
    type: 'attribute',
    name: 'key',
    typeInfo: 'String'
  },

  // Declaration of the value property
  value: {
    type: 'value',
    name: 'value',
    typeInfo: 'String'
  }
}
```

Element map property allows mapping one or more elements to an object/hashmap-valued property. Element map property is configured with two further properties which describe, what should be taken as a key of the hashmap and what as value.

Since version 1.1 element map properties can be collections. In this case, values in the hashmap will be arrays. This allows modelling multimaps.

Mapping example:

```
var MyModule = {
  name: 'MyModule',
  typeInfos: [{
    type: 'classInfo',
    localName: 'ElementMapType',
    propertyInfos: [{
      type: 'elementMap',
      name: 'element',
      key: {
        type: 'attribute',
        name: 'key',
        typeInfo: 'String'
      },
      value: {
        type: 'value',
        name: 'value',
        typeInfo: 'String'
      }
    }
  ]
}, {
  type: 'elementMap',
  name: 'elements',
  wrapperElementName: 'elements',
  elementName: 'element',
  key: {
    type: 'attribute',
    name: 'key',
    typeInfo: 'String'
  },
  value: {
    type: 'value',
    name: 'value',
    typeInfo: 'String'
  }
}, {
  type: 'elementMap',
  name: 'elementCollection',
  collection: true,
  key: {
    type: 'attribute',
    name: 'key',
    typeInfo: 'String'
  }
}
```

```

    }, {
      value: {
        type: 'value',
        name: 'value',
        typeInfo: 'String'
      }
    }, {
      type: 'elementMap',
      name: 'elementsCollection',
      wrapperElementName: 'elementsCollection',
      elementName: 'element',
      collection: true,
      key: {
        type: 'attribute',
        name: 'key',
        typeInfo: 'String'
      },
      value: {
        type: 'value',
        name: 'value',
        typeInfo: 'String'
      }
    }
  ]],
  elementInfos: [{
    elementName: 'elementMap',
    typeInfo: 'MyModule.ElementMapType'
  }]
};

```

XML:

```

<elementMap>
  <element key="one">earth</element>
  <element key="two">wind</element>
  <elements>
    <element key="three">fire</element>
    <element key="four">wood</element>
  </elements>
  <elementCollection key="one">1</elementCollection>
  <elementCollection key="one">I</elementCollection>
  <elementCollection key="two">2</elementCollection>
  <elementCollection key="two">II</elementCollection>
  <elementsCollection>
    <element key="three">3</element>
    <element key="three">III</element>
    <element key="four">4</element>
    <element key="four">IV</element>
  </elementsCollection>
</elementMap>

```

JavaScript object:

```

{
  name: {
    localPart: 'elementMap'
  },
  value: {
    element: {
      'one': 'earth',
      'two': 'wind'
    },
    elements: {
      'three': 'fire',
      'four': 'wood'
    },
    elementCollection: {
      one: ['1', 'I'],
      two: ['2', 'II']
    },
    elementsCollection: {
      three: ['3', 'III'],
      four: ['4', 'IV']
    }
  }
}

```

[Fiddle.](#)

Element reference property

Property declaration syntax:

```
{
  type: 'elementRef',

  // Name of the property, required
  name: 'elementRef',

  // Whether the property is collection or not, defaults to false
  collection: false,

  // Name of the element, optional, defaults to the name of the property, string or QName
  elementName: 'myElementRef',

  // Name of the wrapper element, defaults to null, string or QName
  wrapperElementName: 'myElementRefs',

  // Type of the property (can be simple or complex), required
  typeInfo: 'String'
}
```

Element reference property maps a JavaScript object property onto XML element. This is similar to the [element property](#), however what's different is content representation in the JavaScript object. Consider the following properties:

```
var MyModule = {
  name: 'MyModule',
  typeInfos: [{
    type: 'classInfo',
    localName: 'ElementRefType',
    propertyInfos: [{
      type: 'element',
      name: 'a',
      typeInfo: 'String'
    }, {
      type: 'elementRef',
      name: 'b',
      typeInfo: 'String'
    }
  ]
}],
  elementInfos: [{
    elementName: 'data',
    typeInfo: 'MyModule.ElementRefType'
  }, {
    elementName: 'c',
    substitutionHead: 'b',
    typeInfo: 'String'
  }
]
};
```

XML elements for both properties is the same:

```
<data>
  <a>1</a>
  <b>2</b>
</data>
```

However, content representation in the JavaScript object is different:

```
{
  name: { localPart: 'data' },
  value: {
    a: '1',
    b: {
      name: { localPart: 'b' },
      value: '2'
    }
  }
}
```

In the example above the element reference property `b` is represented in the JavaScript object by the following construct:

```
{
  name: { localPart: 'b' },
  value: '2'
}
```

The advantage of this representation is that you can choose the name of the XML element dynamically:

```
{
  name: {
    localPart: 'data'
  },
  value: {
    a: '1',
    b: {
      name: {
        localPart: 'c'
      },
      value: '2'
    }
  }
}
```

Despite property name is still `b`, it will be marshalled as the element `c`:

```
<data>
  <a>1</a>
  <c>2</c>
</data>
```

Note that to do this trick we had to declare an element mapping for the element `c`:

```
elementInfos: [{
  elementName: 'data',
  typeInfo: 'MyModule.ElementRefType'
}, {
  elementName: 'c',
  typeInfo: 'String'
}]
```

This lets Jsonix know that the element `c` can substitute the element `b` and it should be processed as `String`.

[Fiddle.](#)

Scoped elements

There is one problem with element declaration above:

```
{
  elementName: 'c',
  typeInfo: 'String'
}
```

This makes `c` a global element. So you can now unmarshal the following XML:

```
<c>3</c>
```

This may or may not be the desired effect. To overcome this difficulty, you can define a *scope* for this element declaration.

Scope is essentially a complex type which will limit the applicability of the given element declaration. In the example above, we can limit the scope of the element `c` to the enclosing type `MyModule.ElementRefType`:

```
{
  elementName: 'c',
  scope: 'MyModule.ElementRefType',
  typeInfo: 'String'
}
```

[Fiddle.](#)

Substitution groups

You might have noticed that although marshalling

```
{
  name: { localPart: 'c' },
  value: '2'
}
```

worked as expected, unmarshalling

```
<data>
  <a>1</a>
```

```
<c>2</c>
</data>
```

did not. The reason is that Jsonix sees the element `c` and can unmarshal it via the global element declaration, *but* it does not know which property it should be mapped to. Our complex type `MyModule.ElementRefType` only declares properties `a` and `b`.

To fix this, we can provide the `substitutionHead` property in the element declaration. The `substitutionHead` name the element (either via string or QName) which can be *substituted* by the given element. For instance, if we can define the element declaration as follows:

```
{
  elementName: 'c',
  substitutionHead: 'b',
  typeInfo: 'String'
}
```

This will let Jsonix know that the element `c` substitutes the element `b`. And since our complex type `MyModule.ElementRefType` has an element reference property for the element `b`, Jsonix will know that the unmarshalled `c` should be assigned to the property `b`.

[Fiddle](#).

Element references property

Property declaration syntax:

```
{
  type: 'elementRefs',

  // Name of the property, required
  name: 'elementRefs',

  // Whether the property is collection or not, defaults to false
  collection: true,

  // Name of the wrapper element, defaults to null, string or QName
  wrapperElementName: 'myElementRefs',

  // Element mappings, required
  elementTypeInfos: [{
    // Name of the element, required, string or QName
    elementName: 'string',

    // Type of the property, required
    typeInfo: 'String'
  }, {
    elementName: 'integer',
    typeInfo: 'Integer'
  }]
}
```

Element references property maps several XML elements onto one JavaScript object property. This is similar to [elements property](#), but for [references](#).

Example:

```
var MyModule = {
  name: 'MyModule',
  typeInfos: [{
    type: 'classInfo',
    localName: 'ElementRefsType',
    propertyInfos: [{
      type: 'elementRefs',
      name: 'data',
      wrapperElementName: 'numbers',
      collection: true,
      elementTypeInfos: [{
        elementName: 'long',
        typeInfo: 'Long'
      }, {
        elementName: 'integer',
        typeInfo: 'Integer'
      }]
    }]
  }],
  elementInfos: [{
    elementName: 'root',
    typeInfo: 'MyModule.ElementRefsType'
  }]
}
```

```
};
```

XML:

```
<root>
  <numbers>
    <long>1</long>
    <integer>2</integer>
  </numbers>
</root>
```

JavaScript object:

```
{
  name: {
    localPart: 'root'
  },
  value: {
    data: [{
      name: {
        localPart: 'long'
      },
      value: 1
    }, {
      name: {
        localPart: 'integer'
      },
      value: 2
    }]
  }
}
```

[Fiddle](#).

Any element property

Property declaration syntax:

```
{
  type: 'anyElement',

  // Name of the property
  name: "any",

  // Whether the property is collection or not, defaults to false
  collection: false,

  // Whether the property allows DOM nodes, default to true
  allowDom: true,

  // Whether the property allows typed objects, default to true
  allowTypedObject: true,





  // If the property is a mixed property, default to true
  mixed: true
}
```

Any element property can handle arbitrary XML elements. Depending on the processing types and whether these elements are known within the current Jsonix context, you'll get objects, DOM nodes or strings on the JavaScript side.

Any element property handles unmarshalling as follows:

- When unmarshalling character data:
 - If this property is mixed, character data is unmarshalled as string.
 - Otherwise an error is reported.
- When unmarshalling an element:
 - If this property allows typed objects, check if this element is know to the context via [element mapping](#).
 - If it is known, unmarshal as typed object.
 - Otherwise if this property allows DOM, simply return current element as a DOM element.
 - Otherwise report an error.

Below is the correspondence between `xs:any` processing types and `allowTypedObject/allowDom` processing settings.

Processing type	allowTypedObject	allowDom
lax		
		

strict		
skip		

Usage constraints:

- Complex type may declare at most one "any element" property.

Any element property example - lax processing

Consider the following mapping example:

```
var MyModule = {
  name: 'MyModule',
  typeInfos: [{
    type: 'classInfo',
    localName: 'AnyElementType',
    propertyInfos: [{
      type: 'anyElement',
      name: 'any',
      collection: true
    }]
  }, {
    type: 'classInfo',
    localName: 'ValueType',
    propertyInfos: [{
      type: 'value',
      typeInfo: 'Double',
      name: 'data'
    }]
  }],
  elementInfos: [{
    elementName: 'root',
    typeInfo: 'MyModule.AnyElementType'
  }, {
    elementName: 'string',
    typeInfo: 'String'
  }, {
    elementName: 'value',
    typeInfo: 'MyModule.ValueType'
  }]
};
```

The `allowsDom`, `allowsTypedObject` and `mixed` options are defaulted to `true`, so this property will produce:

- typed objects for elements known in this context;
- DOM nodes for elements which are not known to this context;
- strings for character data.

XML:

```
<root>
  <string>one</string>
  <value>2</value>
  three
  <node>4</node>
</root>
```

Since we have declared global elements `string` and `value` in our module, these elements will be unmarshalled as typed objects. The element `node` is not known in this context - it will be returned as DOM. Character data `three` will be unmarshalled as string.

```
{
  name: {
    localPart: 'root'
  },
  value: {
    any: [{
      name: {
        localPart: 'string'
      },
      value: 'one'
    }, {
      name: {
        localPart: 'value'
      },
      value: {
        data: 2,
        TYPE_NAME: 'MyModule.ValueType'
      }
    }],
  },
}
```

```

    },
    'three',
    // <node>4</node> as a DOM element
    Jsonix.DOM.parse('<node>4</node>').documentElement]
  }
}

```

Using Jsonix in your JavaScript program

- [Download](#) Jsonix or [install](#) it with npm in node.js
- Add/import/include/require Jsonix scripts into your program/page.
- [Write](#) or [generate](#) Jsonix mappings.
- Create Jsonix context from these mappings.
 - To marshal (serialize JavaScript objects as XML):
 - Create marshaller.
 - Use `marshalString`, `marshalDocument` etc. methods of marshaller.
 - To unmarshal (parse JavaScript objects from XML):
 - Create unmarshaller.
 - Use `unmarshalString`, `unmarshalDocument`, `unmarshalURL` etc. methods of unmarshaller.

Including Jsonix scripts in a web page

In production you'll normally want to use the minified version of Jsonix:

```

<html>
  <head>
    <script type="text/javascript" src="../js/Jsonix/Jsonix-min.js"></script>
    <!-- ... -->
  </head>
  <!-- ... -->
</html>

```

Available versions:

- `Jsonix-min.js` - aggregated, minified version.
- `Jsonix-all.js` - aggregated, not minified version.
- `lib/Jsonix.js` - not aggregated, not minified development version.

Installing Jsonix in node.js

Install:

```
npm install jsonix
```

Or add to dependencies of your package:

package.js

```

{
  "name": "mypackage",
  // ...
  "dependencies": {
    "jsonix": "~<VERSION>",
    // ...
  }
}

```

Using Jsonix

Creating Jsonix Context

In order to marshal or unmarshal you'll first need to create Jsonix context:

```

var context = new Jsonix.Context(
  // Array of mapping modules
  [ MyMappings1, MyMappings2 ],
  // Optional properties
  {
    // Default namespace/prefix mappings (optional)
  }
);

```



```

        // default namespace/prefix mappings (optional)
        namespacePrefixes : {
            'http://acme.com/foo' : 'foo',
            'http://acme.com/bar' : 'bar'
        }
    };

```

Jsonix context is a factory which produces marshallers or unmarshallers. Jsonix context is thread-safe and reusable.

Once Jsonix context is created you can use it to produce marshallers or unmarshallers:

```

var marshaller = context.createMarshaller();
var unmarshaller = context.createUnmarshaller();

```

Unlike the context itself, marshaller and unmarshallers *neither* thread-safe *nor* reusable.

Marshalling

Once you have a marshaller, you can marshal your object as XML:

```

// Marshal as string
var objectAsXMLString = marshaller.marshalString(myObject);
// Marshal as document
var objectAsXMLDocument = marshaller.marshalDocument(myObject);

```

Unmarshalling

Unmarshaller can parse your object from XML:

```

// Unmarshal from string
var objectFromXMLString = unmarshaller.unmarshalString(myString);
// Unmarshal from document
var objectFromXMLDocument = unmarshaller.unmarshalDocument(myDocument);
// Unmarshal from URL via AJAX
unmarshaller.unmarshalURL(myURL,
    function (data) {
        var objectFromURL = data;
    });

```

Unmarshalling a file with node.js

✔ Since 2.0

If you're running Jsonix in a [node.js](#) environment, you can also unmarshal from a file:

```

// Unmarshal from file via node.js file system API
unmarshaller.unmarshalFile(fileName,
    function (data) {
        var objectFromFile = data;
    },
    options);

```

At the moment, the file will be loaded as a string, then parsed into DOM document and finally unmarshalled from the parsed document.

The optional argument `options` is passed directly to the `fs.readFile(...)` call. See [node.js FileSystem API](#).

Generating mappings from XML Schema

⚠ You need a Java environment to generate mappings from XML Schemata.

Command-line tool

```

java
-jar jsonix-full-<VERSION>.jar // Run executable Java archive
-d src/main/webapp/js // Target directory
src/main/resources/purchaseorder.xsd // Schema
-b src/main/resources/bindings.xjb // Bindings

```

XJC plugin

If you're already using XJC to compile your schemas, you'll just need to use the `jsonix` plugin for XJC. The plugin can be downloaded [here](#). It is activated using the following command-line option:

```
-Xjsonix
```

Maven usage

- Set `extension=true`
- Add `-Xjsonix` to `args/arg`

```
<plugin>
  <groupId>org.jvnet.jaxb2.maven2</groupId>
  <artifactId>maven-jaxb2-plugin</artifactId>
  <configuration>
    <extension>true</extension>
    <args>
      <arg>-Xjsonix</arg>
    </args>
    <plugins>
      <plugin>
        <groupId>org.hisrc.jsonix</groupId>
        <artifactId>jsonix-schema-compiler</artifactId>
        <version><VERSION></version>
      </plugin>
    </plugins>
  </configuration>
</plugin>
```

Ant usage

```
<xjc destdir="${basedir}/target/generated-sources/xjc" extension="true">
  <binding dir="${basedir}/src/main/resources">
    <include name="**/*.xjb"/>
  </binding>
  <schema dir="${basedir}/src/main/resources">
    <include name="**/*.xsd"/>
  </schema>
  <!-- Plugins -->
  <classpath>
    <fileset dir="${basedir}/lib">
      <include name="jsonix-<VERSION>.jar"/>
    </fileset>
  </classpath>
  <arg line="-Xjsonix"/>
</xjc>
```

Bindings files

Now you may wonder, what the bindings file does. Bindings customize schema compilation. For instance, you can instruct Jsonix schema compiler to generate the `PO` module (by default written to the `PO.js` file).

Here's how a typical bindings file looks like.

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<jaxb:bindings
  version="2.1"
  xmlns:jaxb="http://java.sun.com/xml/ns/jaxb"
  xmlns:jsonix="http://jsonix.hisrc.org/customizations"
  xmlns:xs="http://www.w3.org/2001/XMLSchema"

  jaxb:extensionBindingPrefixes="jsonix">

  <jaxb:bindings schemaLocation="purchaseorder.xsd" node="/xs:schema">
    <jaxb:schemaBindings>
      <jaxb:package name="org.hisrc.jsonix.demos.po"/>
    </jaxb:schemaBindings>

    <jsonix:packageMapping
      packageName="org.hisrc.jsonix.demos.po"
      spaceName="PO"/>
  </jaxb:bindings>
```

```
</jaxb:bindings>
```

```
</jaxb:bindings>
```

This bindings file basically says two things:

- The `purchaseorder.xsd` schema will get the package `org.hisrc.jsonix.demos.po`
- The package `org.hisrc.jsonix.demos.po` will have the associated space name (module) `PO`

This might look a bit cumbersome, but this is due to certain limitations in the underlying technologies.

Using command-line tool in node.js

Add schema generation as a `preinstall` script.

package.json

```
{
  ...,
  "dependencies": {
    "jsonix": "~<VERSION>"
  },
  "scripts": {
    "preinstall" : "java -jar jsonix-full.jar -d mappings purchaseorder.xsd -b bindings.xjb"
  }
}
```

Powered by a free **Atlassian Confluence Open Source Project License** granted to disy Informationssysteme GmbH. Evaluate Confluence today.

Powered by [Atlassian Confluence](#) 2.10.2, the [Enterprise Wiki](#). [Bug/feature request](#) – [Atlassian news](#) – [Contact administrators](#)