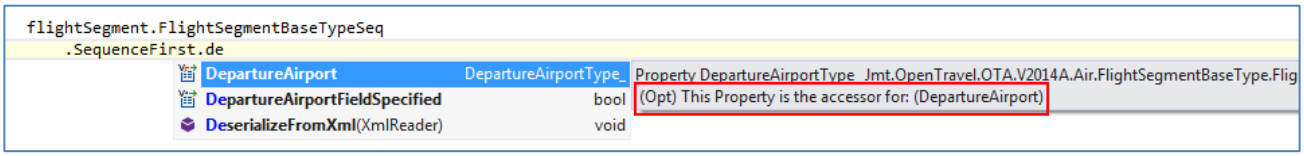
Programming with JMT Types
Coding Guidelines

As with all complex type landscapes, it is helpful to get comfortable and productive with coding by having some guidance in how to use the various types on offer. This TechNote gathers together some topics that will help any developer getting into synch with JMT types. If not otherwise noted, schemas from the OpenTravel Alliance v2014A specification are used in this document[1].

Optionality

Use the property's Intellisense comment as an indication of a field's optionality "(Opt)" which means that it will have been set to null in the enclosing type's instantiation:



Use the class' constructor in conjunction with object initializer to set up value of an optional field.

```
flightSegment.FlightSegmentBaseTypeSeq.SequenceFirst.DepartureAirport = new FlightSegmentBaseType.FlightSegmentBaseSeqType_.D
{
    AirportLocationGroup =
    {
        LocationCode = new StringLength1to8Type
        {
            Text = "ZRH"
        }
    }
};
```

Mandatory

This is basically the reverse of Optional. If in the Intellisense the documentation does not show the prefix "Opt", then the item is mandatory. In this case whatever the type, there is no need to instantiate it before use.

Sequence

Nested structure of sequences is reflected in the JMT types. Consider the OTA_AirAvailRQ.xsd as a schema. The top-level type, a message, is **OTA_AirAvailRQ** and this contains a complex type having a sequence with a range of elements, one of which is the **POS** element which we want to populate in this example:

[1] http://www.opentravel.org

```
<xs:element name="OTA_AirAvailRQ">
  <xs:annotation>...</xs:annotation>
  <xs:complexType>
    <xs:sequence>
      <xs:element name="POS" type="POS_Type">
        <xs:annotation>...</xs:annotation>
      </xs:element>
```

If you continued exploring you will find that the **POS_Type** contains yet another sequence:

```
<xs:complexType name="POS_Type">
  <xs:annotation>...</xs:annotation>
  <xs:sequence>
    <xs:element name="Source" type="SourceType" maxOccurs="10">
      <xs:annotation>...</xs:annotation>
    </xs:element>
  </xs:sequence>
</xs:complexType>
```

And in turn the **SourceType** type also exposes a sequence which is of interest:

```
<xs:complexType name="SourceType">
  <xs:annotation>...</xs:annotation>
  <xs:sequence>
    <xs:element name="RequestorID" minOccurs="0">...</xs:element>
    <xs:element name="Position" minOccurs="0">...</xs:element>
    <xs:element name="BookingChannel" minOccurs="0">...</xs:element>
    <xs:element ref="TPA_Extensions" minOccurs="0"/>
  </xs:sequence>
  <!-- attributes removed for clarity... -->
</xs:complexType>
```

This nested structure of the schemas is reflected in the structure of the JMT types. Firstly, one can access the sequence of the top-level types by the **OTA_AirAvailRQSeq** property. Please note that the **Seq** suffix in the property name indicates that that the property represents a sequence. To access the contents of the sequence the Items property may be used in conjunction with one of the LINQ to Objects extension methods or the **SequenceFirst** property:

```
var message = new OTA_AirAvailRQ();
var sequence = message.OTA_AirAvailRQSeq.Items.First();
var sequence2 = message.OTA_AirAvailRQSeq.SequenceFirst;
```

Using this approach we can access the inner sequence within the **POS_Type** type:

```
var message = new OTA_AirAvailRQ();
var posSequence = message.OTA_AirAvailRQSeq.SequenceFirst
    .POS.POS_TypeSeq.SequenceFirst;
```

Continuing in this manner we can access the Source element by using the **SourceList** property. Please note that the Source element is defined with the *maxOccurs="10"* quantifier which in JMT type landscape is represented as a list:

```
var message = new OTA_AirAvailRQ();
var source1 = message.OTA_AirAvailRQSeq.SequenceFirst
    .POS.POS_TypeSeq.SequenceFirst
    .SourceList.Items.First();
```

It leads us to the following guidelines:

- The **`Seq`** suffix applies to properties that correspond to sequences
- Use the **`List`** suffix to discover collections that represent elements that may occur more than once in the XML (`maxOccurs > 1`)
- Use the **`SequenceFirst`** property to easily access the first entity in a sequence container

The hierarchy of sequences defined by the schema is preserved by the types. For example the type **`HotelSearchCriterionType`** from the `OTA_HotelCommonTypes.xsd` contains a complex content with an extension as well as a sequence:

```
<xs:complexType name="HotelSearchCriterionType">
  <xs:annotation>...</xs:annotation>
  <xs:complexContent>
    <xs:extension base="ItemSearchCriterionType">
      <xs:sequence>...</xs:sequence>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
```

You can use both sequences which are exposed as properties:

```
criterion.seq
    HotelSearchCriterionTypeSeq   GuardedList<HotelSearchCriterionSeqType_>
    ItemSearchCriterionTypeSeq    GuardedList<ItemSearchCriterionSeqType_>
```

It should be noted that sequences can have `min-/max-Occurs` parameters. These are handled in JMT types.

Guarded List

We will now take a look at the **`TravelInfoSummaryType`** of the **`TravelInfoSummary`** element:

```
<xs:complexType name="TravelerInfoSummaryType">
  <xs:annotation>...</xs:annotation>
  <xs:sequence>
    <xs:element name="SeatsRequested" type="xs:nonNegativeI" minOccurs="0" maxOccurs="99">...</xs:element>
    <xs:element name="AirTravelerAvail" type="TravelerInformationType" minOccurs="0" maxOccurs="99">
      <xs:annotation>
        <xs:documentation xml:lang="en">Specifies passenger numbers and types.</xs:documentation>
      </xs:annotation>
    </xs:element>
    <xs:element name="PriceRequestInf" minOccurs="0">...</xs:element>
    <xs:element ref="TPA_Extensions" minOccurs="0"/>
  </xs:sequence>
</xs:complexType>
```

First thing to notice is that number of the **`AirTravelerAvail`** elements is restricted from 0 to 99. In the JMT types world the elements are held in a collection of type **`GuardedList<T>`** where T is the type representing the type of the element. In this particular case the **`minOccurs`** of the element is 0 which means it is *optional* and hence, by default, the collection is null. It has to be initiated in order to add elements to it:

```
var message = new OTA_AirAvailRQ();
var travelInfoSummary = message.OTA_AirAvailRQSeq.SequenceFirst
    .TravelerInfoSummary.TravelerInfoSummaryTypeSeq.SequenceFirst;

travelInfoSummary.AirTravelerAvailList = new GuardedList<TravelerInformationType>(0, 99);
```

Please note that the parameters of the constructor correspond to the `minOccurs` and `maxOccurs` attributes of the element. This information is provided to you with in the Intellisense documentation of the type and doesn't have to be obtained from the schema. Hovering over the type name in Visual Studio will cause the relevant information to be shown as a suffix to the entity documentation ("`[minOccurs, maxOccurs]`"). Once the **`GuardedList`** is instantiated, you can use it just as you would any other .NET collection you are familiar with:

```
travelInfoSummary.AirTravelerAvailList.Add(new TravelerInformationType());
```

It leads us to the following guidelines:

- Use the comments of the property of type **`GuardedList<T>`** to know what parameters of the constructor you should use
- For unbounded restrictions (without the `maxOccurs` quantifier) use the `Int.MaxValue` as the second parameter of the constructor
- If the list is not optional (the `minOccurs` is more than 0 or is not specified) then the collection doesn't have to be initialized and can be used right away

W3C primitives

In the world of W3C Schema there are a group of fundamental types that are available to the Schema designer, the so-called Primitives. These are defined precisely by the W3C organization and have a very clearly defined set of values that can be assigned to each type. So, for example, the primitive type, displayed in a Schema like `xs:nonNegativeInteger` has the definition that its valid value MUST be in the set (0, 1, 2, 3…). In the case of `xs:date` the valid values possible are specified as being in the form "YYYY-MM-DD".
All JMT libraries come with a full set of strong-type Primitives. These appear, in general, throughout a Schema set, but may be used quite independent in a general application development context.

Local Type Extensions

As an example of local type extensions, a basic idiom of W3C Schema, consider the **`OriginDestinationInformationType`** type:

```
<xs:complexType name="OriginDestinationInformationType">
  <xs:annotation>...</xs:annotation>
  <xs:complexContent>
    <xs:extension base="TravelDateTimeType">
      <xs:sequence>
        <xs:element name="OriginLocation">
          <xs:annotation>...</xs:annotation>
          <xs:complexType>
            <xs:complexContent>
              <xs:extension base="LocationType">
                <xs:attribute name="MultiAirportCit" type="xs:boolean" use="optional">...</xs:attribute>
                <xs:attribute name="AlternateLocati" type="xs:boolean" use="optional">...</xs:attribute>
              </xs:extension>
            </xs:complexContent>
          </xs:complexType>
        </xs:element>
        <xs:element name="DestinationLoca">...</xs:element>
        <xs:element name="ConnectionLocat" type="ConnectionType" minOccurs="0">...</xs:element>
      </xs:sequence>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
```

The **`OriginLocation`** is defined as a local type with complex content which extends the **`LocationType`** type. In the world of JMT types accessing the properties of the type being extended is simple as using the **`SimpleTypeExtension`** property. To set the **`LocationCode`** attribute from the **`LocationGroup`** attribute group defined in the **`LocationType`** type only one line of code is required:

```
originDestinationInformation.OriginLocation.SimpleTypeExtension.LocationGroup.LocationCode = new StringLength1to16Type
{
    Text = "LHR"
};
```

Use the **`SimpleTypeExtension`** property to access contents of an extension

Schema SimpleTypes

Use the "Text" property in an object initializer or property setter to access the value of a SimpleType:

```
// Object initializer
message.DirectAndStopsGroup.DirectFlightsOnly = new W3cBoolean
{
    Text = "true"
};

// Property setter
message.DirectAndStopsGroup.DirectFlightsOnly = new W3cBoolean();
message.DirectAndStopsGroup.DirectFlightsOnly.Text = "true";
```

Constructor calls or calls to property setter may be simplified using an extension method:

```
public static T Set<T, V>(this T @this, Expression<Func<T, V>> getter, object value)
    where T : class
    where V : class, ISimpleType, new()
{
    // Gets a PropertyInfo object from the Expression.
    PropertyInfo property = GetPropertyInfo(@this, getter);

    var newValue = new V {Text = value.ToString()};

    property.SetValue(@this, newValue);

    return @this;
}
```

Which simplifies the setting of a SimpleType value to:

```
message.DirectAndStopsGroup.Set(p => p.DirectFlightsOnly, true);
```

The instantiation of objects described above allows you to "chain" method calls in a "fluent" way. Such an approach simplifies the setting of a range of attributes in (for example) AttributeGroups:

```
var message = new OTA_AirAvailRQ();
message.OTA_PayloadStdAttributes
    .Set(x => x.AltLangID, "en-US")
    .Set(x => x.EchoToken, "12345");
```

Choice entities

Choice is another Schema idiom which is very prevalent in OpenTravel, as well as other Schema sets. An example of a choice in a schema is that which occurs in `OTA_AirAvailRS` (`OTA_AirAvailRS.xsd`):

```
<xs:element name="OTA_AirAvailRS">
  <xs:annotation>...</xs:annotation>
  <xs:complexType>
    <xs:choice>
      <xs:sequence>...</xs:sequence>
      <xs:element name="Errors" type="ErrorsType">...</xs:element>
    </xs:choice>
    <xs:attributeGroup ref="OTA_PayloadStdAttributes"/>
  </xs:complexType>
```

The choice defined by the schema is between a sequence and an "Errors" element. In the JMT world the choice is represented by the **`OTA_AirAvailRSChoice`** (please note the Choice suffix) property.

Use the **`Choice`** suffix to discover properties that represent choices:

```
var message = new OTA_AirAvailRS();
var choice = message.OTA_AirAvailRSChoice;
```

Use the **`Keys`** property to discover possible keys of a choice:

```
var message = new OTA_AirAvailRS();
var keys = message.OTA_AirAvailRSChoice.ChoiceKeys;
```

keys {string[2]}
[0] "GuardedList<OTA_AirAvailRSChoiceSeqType_>"
[1] "Errors"

The keys represent the concrete types of the individual choice elements. Note that in this example the first possible element to choose is a sequence which is represented by a `GuardedList<T>`.

Use the **`Choose()`** method with an appropriate key to select a particular element.
Note that the value returned by the method (the object representing the chosen element) is of type `object`. The reason is that various elements may be a part of the choice and they not necessarily share a common base class. To work with the object it has to be cast to the correct type (consult the keys of the choice to see the choice element types):

```
var message = new OTA_AirAvailRS();
var myChoice = message.OTA_AirAvailRSChoice.Choose("GuardedList<OTA_AirAvailRSChoiceSeqType_>").Value
    as GuardedList<OTA_AirAvailRS.OTA_AirAvailRSChoiceType_.OTA_AirAvailRSChoiceSeqType_>;
```

Use the **`ChosenElement`** property and cast it's value to the appropriate type to obtain the value:

```
var somebodysChoice = message.OTA_AirAvailRSChoice.ChosenElement.Value
    as GuardedList<OTA_AirAvailRS.OTA_AirAvailRSChoiceType_.OTA_AirAvailRSChoiceSeqType_>;
```

Unit Testing

Unit tests play a fundamental role in any large-scale project. The guidelines here will allow you to write them in a more robust and maintainable way.

General approach

The triple A (Arrange, Act, Assert – AAA) approach for writing unit test[2] is the most common and widely used approach. It's also the recommended way to write unit tests for messages as defined your Schema.
A generic template for a unit tests for a message (`OTA_AirAvailRQ`) would look like as follows:

```
[TestMethod]
public void Deserialize_Test()
{
    // Arrange
    var message = new OTA_AirAvailRQ();

    // Act
    message.Deserialize("Test.xml");

    // Assert
    Assert.IsNotNull(message.OTA_AirAvailRQSeq);
}
```

In this deserialization test, an empty object is instantiated in the "Arrange" section which is then populated with data from a test file, the "Act" section, and finally the object is checked in the "Assert" section. In this very simple example we see that a check is made that the object is "NotNull", whereas in practice the tests would check the content, the data, contained in the object.
Unit testing of message serialization is more complicated as it requires comparing an expectation (an XML exemplar) with the actual value (the result of the serialization). A general template for such a test would look like this:

```
[TestMethod]
public void Serialize_Test()
{
    // Arrange
    var message = CreateMessage();

    // Act
    message.Serialize("TestOutput.xml");

    // Assert
    var xmlComparer = new XmlComparer("Test.xml", "TestOutput.xml");
    Assert.IsTrue(xmlComparer.FilesEqual);
}
```

Please note that an *XmlComparer* class is used in this example. The role of this class would be to compare two XML files and decide about their equality. To implement such a class, a two-step implementation process is required:

- Finding a common comparable format for the two files
- Comparing the files and deciding about their semantic equality

The first step may be implemented by transforming the files to their canonical form (using, for example, the *XmlDsigExcC14NTTransform*[3] class from the .NET Framework). The second part may be done using a 3rd party library like

[2] http://xp123.com/articles/3a-arrange-act-assert/
[3] http://msdn.microsoft.com/en-us/library/system.security.cryptography.xml.xmldsigexcc14ntransform%28v=vs.110%29.aspx

Microsoft "XML Diff And Patch Tool"[4]. It is also necessary that the unit tests are run in total isolation from the physical file system.

Improved approach

Due to the nature of the unit tests which involves checking the contents of the serialized or deserialized XML messages they may become less readable using the basic approach. Eventually, a typical test would end with a set of assertions about particular parts of a message:

```
[TestMethod]
public void Deserialize_Test()
{
    // Arrange
    var driver = new TestDriver("OTA_AirAvailRQ");

    // Act
    var result = driver.TestDeserialization<OTA_AirAvailRQ>();

    // Assert
    var sequence = result.OTA_AirAvailRQSeq.SequenceFirst;
    var pos = sequence.POS.POS_TypeSeq.SequenceFirst;
    var source = pos.SourceList.SequenceFirst;

    Assert.AreEqual(source.AgentSine, "BSIA1234PM");
    Assert.AreEqual(source.PseudoCityCode, "2U8");
    Assert.AreEqual(source.ISOCountry, "US");
    Assert.AreEqual(source.ISOCurrency, "USD");
}
```

As noted above, although this approach is fine for a starting point, it does become less readable and maintainable as the object against which the tests are written becomes complex. An improved approach would be to abandon individual assertions in favor of **verifying our expectations** towards the result of the deserialization. Instead of asserting individual parts of the deserialized message one would create a set of **expectations** and **verify** them. An example unit test using this approach would look as follows

```
[TestMethod]
public void Deserialize_Test()
{
    var driver = new TestDriver("OTA_AirAvailRQ");
    var result = driver.TestDeserialization<OTA_AirAvailRQ>();

    var sequence = result.OTA_AirAvailRQSeq.SequenceFirst;
    var pos = sequence.POS.POS_TypeSeq.SequenceFirst;
    var source = pos.SourceList.SequenceFirst;

    // Expectations
    var sourceExpectation = new Expectations<SourceType>()
        .Expect(p => p.AgentSine, "BSIA1234PM")
        .Expect(p => p.PseudoCityCode, "2U8")
        .Expect(p => p.ISOCountry, "US")
        .Expect(p => p.ISOCurrency, "USD");

    source.Verify(sourceExpectation);
}
```

[4] http:/msdn.microsoft.com/en-US/library/aa302294.aspx

Our expectations towards the object of type ***SourceType*** are specified using the *Expectations<T>* class. It allows the developer to express the expectations using lambda expressions as parameters of the *Expect* method. Individual calls may be chained due to the fluent interface exposed by the Expectations class. Once the set of expectations is ready, they can be verified against the real object using the *Verify* method.
The *Verify* method would actually execute all the expectations defined in the specification:

```
public static void Verify<T>(this T @this, Expectations<T> exp)
    where T : class
{
    exp.Execute(@this);
}
```

The implementation of the *Execute* method would materialize the lambda expressions and exercise assertions against actual values of the object under test:

```
public void Execute(T subject)
{
    foreach (var expectation in _expectations)
    {
        var expected = expectation.Item2 is bool
            ? expectation.Item2.ToString().ToLower()
            : expectation.Item2.ToString();

        if (subject == null)
        {
            Assert.Fail(FormatErrorMessage(expectation.Item1.ToString(), expected));
        }

        var actual = expectation.Item1.Compile()(subject);

        var actualSt = actual as ISimpleType;
        if (actualSt != null)
        {
            Assert.AreEqual(expected, actualSt.Text, FormatErrorMessage(expectation.Item1.ToString(), expected, actualSt.Text));
        }
        else
        {
            Assert.AreEqual(expected, actual, FormatErrorMessage(expectation.Item1.ToString(), expected, actual));
        }
    }
}
```

Jet Messaging Technologies AG
Rotwandstrasse 35, 8004 Zurich,
Switzerland
Phone +41 79 303 05 63

Email info@jet-messaging.com
www.jet-messaging.com