XML BASCIS FOR NIBRS

XML Basics Training

Introduction

Why does NIBRS use XML?

The transition of law enforcement agencies to Incident-Based (NIBRS) reporting across the country in recent years has created a need for UCR program managers and others to be able to have a basic understanding of XML.

The historical transmission method for NIBRS data to the state and federal level has been the flat file format.

Given that this file type has existed since the onset of NIBRS, service providers, law enforcement agencies and UCR Program Managers are accustomed to reading and troubleshooting flat file issues.

However, it can be cumbersome to translate these files and flat files are typically associated with legacy systems.

Flat files are brittle. Validating flat files is very limited, so mistakes can be difficult to find. Flat files are limited in ability. There's no way to represent structure. There's no easy way to handle dynamic data, where you might have one instance of a piece of data, but you might have multiple instances.

A flat file typically is submitted to the FBI monthly and many vendors do not provide the ability for law enforcement agencies to resubmit single incidents for error correction to the state UCR Program. This creates a significant lag time in data submissions. More real time submissions make it easier for law enforcement agencies to make error corrections timely.

To address these limitations, the FBI and state programs have ramped up to XML submissions as the preferred format.

XML is designed for computers to parse and understand, while still being meaningful to a person looking at it. However, this can be a struggle for those who are not technical and trained in XML concepts.

The goal of this training is to provide that background in XML, so make it easier for developers to more easily and effectively support NIBRS using XML.

Overview of NIBRS IEPD Structure

Here's an example of a sample NIBRS exchange. We'll use it throughout the training to highlight different aspects of XML. You don't need to understand it all right away. But XML is readable enough that even newcomers will be able to make some sense of it.

Important sections include:

- The Submission itself
- Message Metadata, including
  - date and timestamps
  - IDs
  - versioning information
  - the submitting organization
- The Report, including information about
  - the report itself with category codes, the report date, and the reporting organization
  - an Incident with IDs, datestamps, and clearance information
  - one or more Offenses with UCR and other codes, and force information
  - one or more Locations with category codes
  - one or more Persons with age, race, residency, and sex
  - one or more Victims with category codes and injury

- one or more Subjects
- relationships between these objects

What is XML?

XML is a means of making information self-describing, by wrapping different pieces of data in tags that describe the date. It shares roots with HTML and looks very similar, but it pickier about how you format things. That's to ensure that important data is organized and described correctly so that entities on either end of an exchange understand what each piece of data means.

Also different from HTML is that XML as much about formatting documents for people to view as it is about describing information so that systems can produce and consume the data in an intelligent fashion.

XML Schema is a technology to define what XML documents need to look like.

The combination of XML and XML Schema provides a variety of features:

- Hierarchical organization of information
- Sophisticated data typing, supporting a variety of strings, numbers, dates, booleans, and code tables
- Object-oriented types that can be used as a base for more specialized types
- Cardinality constrants
- Referencing across the hierarchical organization

What follows is, first, and introduction to XML itself, followed by a detailed explanation of XML Schema. By the end of this training, you'll be able to understand XML when you see it, recognize XML Schema, and be able to see how different structures in XML Schema define how XML documents, including NIBRS exchanges, must look.

Element Tag Names

XML elements are defined by tags. Pieces of information are surrounded by tags, an opening tag and a closing tag.

In its simplest form, a starting tag begins with an opening angle bracket, followed by the name of the element, followed by a closing angle bracket.

Closing tags are similar with one difference. A closing tag begins with an opening angle bracket, followed by a forward slash, followed by the name of the element, followed by a closing angle bracket.

Here's a very simple example:

```
<FirstName>Fred</FirstName>
```

The name of this element is "FirstName". The information it holds is "Fred".

There are rules for tag names:

- Can contain letters, numbers, and other certain characters
  - Non-English letters like “é ò á” are legal, but may not be supported by implementations
  - Avoid dashes, periods, and colons
  - Colons are used with namespaces, a topic for the end
- Must not start with number or punctuation
- Must not start with `xml`, `XML`, or `Xml`
- Cannot contain spaces

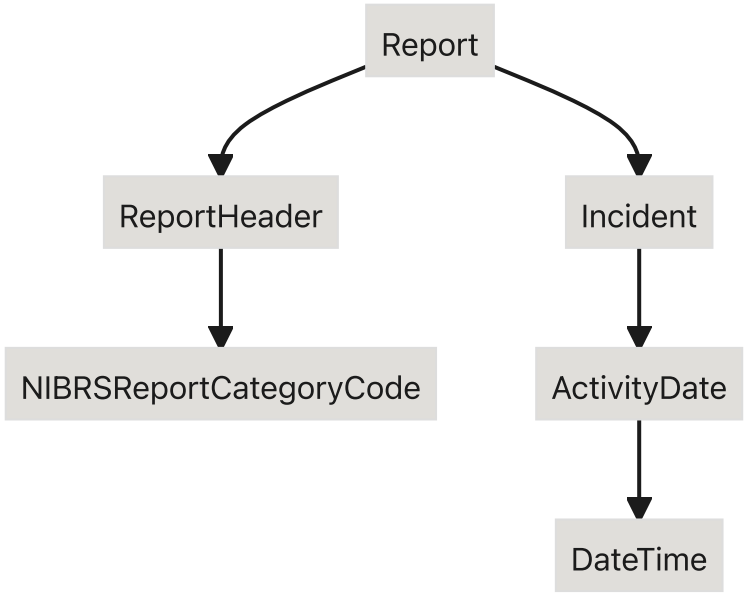
Tag names are case sensitive. `<FirstName>` is not the same as `<firstname>`.

Tag names should be descriptive, long enough to make sense, but no longer than needed.

Element Structure and Hierarchy

Elements can also hold other elements, forming a hierarchy. That's a very basic feature of XML. Let's look at part of the example:

```
<nibrs:Report>
        <nibrs:ReportHeader>
                <!-- Submission Type -->
                <nibrs:NIBRSReportCategoryCode>GROUP A INCIDENT
REPORT</nibrs:NIBRSReportCategoryCode>
        </nibrs:ReportHeader>
        <nc:Incident>
                <nc:ActivityDate>
                        <nc:DateTime>2022-05-01T10:00:00</nc:DateTime>
                </nc:ActivityDate>
        </nc:Incident>
</nibrs:Report>
```



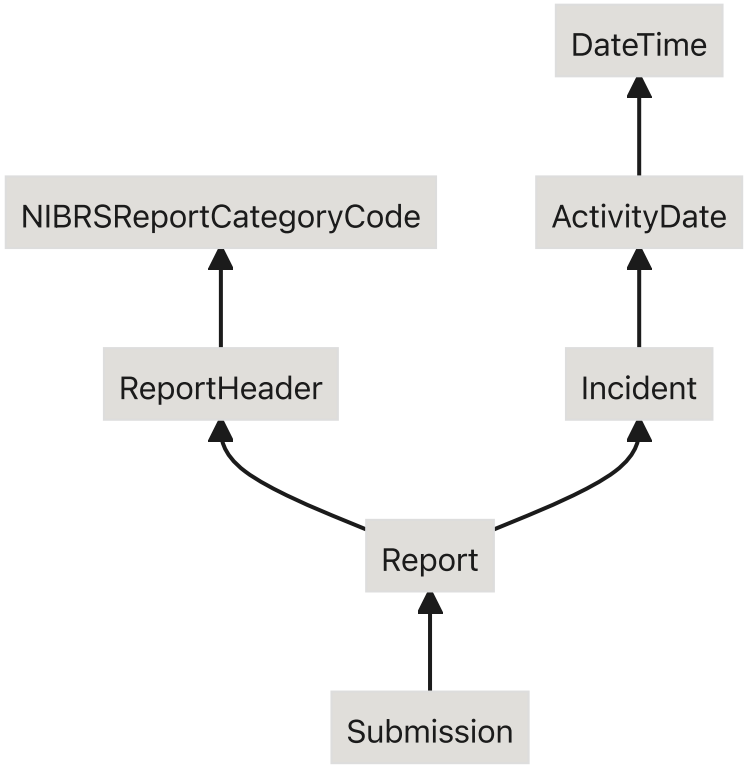
You can see that some elements hold other elements, forming parent/child relationships. The `Report` element is the *parent* of the `ReportHeader` and `Incident` elements. The `ReportHeader` and `Incident` elements are *children* of the `Report` element

The `ReportHeader` element is the parent of the `NIBRSReportCategoryCode` element, and the `NIBRSReportCategoryCode` element is the *child* of the `ReportHeader` element.

Similarly relationships exist between `Incident`, `ActivityDate`, and `DateTime`.

Root Element

When you look at the broader example, you see that `Report` is a *child* of `Submission`. `Submission` has no parent. It only has children. It's called the "root" of the document. (If you make a tree graph of the structure of the document, the root element is, indeed, the root of the tree.)



Every XML document has *one and only one root* element. This is different from some other data serializations. JSON, for example, has no concept of a root key.

Attributes

In addition to have data in between opening and closing tags, XML can include data inside the opening tag via an attribute.

Let's look near the end of our example:

```
<j:Victim s:id="Victim2">
        <nc:RoleOfPerson s:ref="PersonVictim2"/>
```

```
        <j:VictimSequenceNumberText>2</j:VictimSequenceNumberText>
        <j:VictimCategoryCode>I</j:VictimCategoryCode>
</j:Victim>
```

`Victim` holds a `RoleOfPerson`, a `VictimSequenceNumberText`, and a `VictimCategoryCode`. It also has an attribute inside the opening tag itself called `id`. Elements can have more than one attribute; this example only includes one. However, an element can only have one instance of a particular attribute. A single `Victim` element could *not* have two `id` attributes. Elements that hold other elements can have attributes, but so can elements that just hold simple data.

Suppose you had a `<City>` tag, and you're dealing with Virginia, where some cities are independent entities, not inside any county. For example, Williamsburg, Virginia, is surrounded by James City County and York County, but is part of neither. You could represent that with something like:

```
<City status="independent">Williamsburg</City>
```

You can also use single quotes for attribute values:

```
<City status='independent'>Williamsburg</City>
```

Quotes in attribute values can be escaped by using HTML entities, but good development environments will take care of that for you.

Singleton Tags

An element can contain nothing between the tags. Maybe just the existance of the element is meaningful. Maybe all the info that the element holds is held in attributes. In these cases, you can use a shortcut form of the element, the

singleton tag.

Continuing from the earlier snippet, note that the `RoleOfPerson` element only has a `ref` attribute. The XML for it could look like this:

```
<nc:RoleOfPerson s:ref="PersonVictim2"></nc:RoleOfPerson>
```

Instead, you can use the singleton form, which essentially takes the slash from the closing tag and inserts it at the end of the opening tag, like this:

```
<nc:RoleOfPerson s:ref="PersonVictim2"/>
```

Both examples are identical in meaning.

Elements vs Attributes

Elements and attributes look like they do the same thing, and until you learn more about XML Schema, they pretty much do. But there are significant difference, which we'll list now and explain in more detail later:

Elements	Attributes
Can hold simple data in the form of strings, numbers, booleans, dates, etc.	Can hold simple data in the form of strings, numbers, booleans, dates, etc.
Can hold other elements	Can hold simple data...
Can have attributes	Can hold simple data...
Can have arbitrary cardinality constrains in XML Schema, including ranges	Can be required, optional, or forbidden in XML Schema

XML Declaration

XML documents start with a declaration. This declaration tells us two things:

1. Which version of XML is being used (which is always "1.0")
2. Which encoding the text uses (usually either "US-ASCII" or "UTF-8")
    - US-ASCII is the unadorned 26 letters of the english alphabet, numbers, and a few puncuation marks
    - UTF-8 is Unicode and supports accented characters
        - And tons of different alphabets
        - Is backwards compatible with US-ASCII

Here's the start of our example:

```
<?xml version="1.0" encoding="UTF-8"?>
<nibrs:Submission xmlns:nibrs="http://fbi.gov/cjis/nibrs/2019"
        xmlns:cjis="http://fbi.gov/cjis/2.0"
        xmlns:cjiscodes="http://fbi.gov/cjis/cjis-codes/2.0"
        xmlns:i="http://release.niem.gov/niem/appinfo/3.0/"
        xmlns:ucr="http://release.niem.gov/niem/codes/fbi_ucr/3.2/"
        xmlns:j="http://release.niem.gov/niem/domains/jxdm/5.2/"
        xmlns:term="http://release.niem.gov/niem/localTerminology/3.0/"
        xmlns:nc="http://release.niem.gov/niem/niem-core/3.0/"
        xmlns:niem-xsd="http://release.niem.gov/niem/proxy/xsd/3.0/"
        xmlns:s="http://release.niem.gov/niem/structures/3.0/"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xmlns:xsd="http://www.w3.org/2001/XMLSchema"
        xmlns:nibrscodes="http://fbi.gov/cjis/nibrs/nibrs-codes/2019"
```

```
        xsi:schemaLocation="http://fbi.gov/cjis/nibrs/2019 ../base-
xsd/nibrs/2019/nibrs.xsd">
        <cjis:MessageMetadata>
                <cjis:MessageDateTime>2017-03-23T03:00:00</cjis:MessageDateTime>
                <cjis:MessageIdentification>
                        <nc:IdentificationID>123456</nc:IdentificationID>
                </cjis:MessageIdentification>
                <!-- NIBRS IEPD Version -->

<cjis:MessageImplementationVersion>2019</cjis:MessageImplementationVersion>
```

The XML declaration is that first line. Here you can see that the version is 1.0 and the encoding is UTF-8. This declaration tells a parser that this is, indeed, XML, rather than something that might just look a lot like XML, like an HTML document.

XML Comments

XML allows for comments in the document. Again, let's look at the start of our example:

```
<nibrs:Report>
        <nibrs:ReportHeader>
                <!-- Submission Type -->
                <nibrs:NIBRSReportCategoryCode>GROUP A INCIDENT
REPORT</nibrs:NIBRSReportCategoryCode>
                <!-- Submission Action Type -->
                <nibrs:ReportActionCategoryCode>I</nibrs:ReportActionCategoryCode>
                <!-- Year/Month Of Report -->
```

```
                <nibrs:ReportDate>
                        <nc:YearMonthDate>2022-05</nc:YearMonthDate>
                </nibrs:ReportDate>
```

The comments are the parts that read `<!-- Submission Type -->`, `<!-- Submission Action Type -->`, and `<!-- Year/Month Of Report -->`. It's an odd format, an opening angle bracket, followed by an exclamation mark and a pair of dashes. After the text of the comment, it's closed off by two more dashes and a closing angle bracket.

Comments can go nearly anywhere other than *inside* an XML tag.

Well Formed XML

If you follow all the rules above, you'll get what's called "Well Formed XML." Being well-formed is the bottom line for XML. If your XML isn't well-formed, it's not really XML. Being well-formed means that a parser can read through the XML and parse it into pieces. Being well-formed does not mean that the XML makes any sense, though. Here's an example:

```
<?xml version="1.0" encoding="UTF-8"?>
<kladafas>
        <afkkjkljf>klslkajfakljd</afkkjkljf>
        <jklffjkasd>
                <poiufda>
                        <werafdasf/>
                </poiufda>
        </jklffjkasd>
</kladafas>
```

That's perfectly well-formed XML, but it means nothing. To start adding meaning to XML, we need to be able to define, for a particular purpose, what the tags should be called, how they should fit together, and what sorts of data they should hold.

And that's the topic for the rest of this training, definitional technologies, ways of defining how an XML document must look, specifically XML Schema.

XML Schema

XML Schema is a means of defining what an XML instance document needs to look like. It's a popular one, so this is where our focus will be.

XML Schema comes in two versions, 1.0 and 1.1. XML Schema 1.1 is much more powerful than 1.0, but does not enjoy broad tool support. We will be looking at XML Schema 1.0 only.

To start, let's take a look at the some XML Schema, for example:

> nibrs.xsd
>
> ```
> <?xml version="1.0" encoding="UTF-8"?>
> <!--
>     Document: $Id: nibrs.xsd $
>     NIEM version  : 3.2
>     NIBRS version : 2019
>     Namespace     : xmlns:nibrs="http://fbi.gov/cjis/nibrs/2019"
>     Description     : NIBRS 2019 Extension Schema
> -->
> <xsd:schema  xmlns:nibrs="http://fbi.gov/cjis/nibrs/2019"
> ```

```
    xmlns:cjis="http://fbi.gov/cjis/2.0"
    xmlns:cjiscodes="http://fbi.gov/cjis/cjis-codes/2.0"
    xmlns:ct="http://release.niem.gov/niem/conformanceTargets/3.0"
    xmlns:i="http://release.niem.gov/niem/appinfo/3.0/"
    xmlns:ucr="http://release.niem.gov/niem/codes/fbi_ucr/3.2/"
    xmlns:j="http://release.niem.gov/niem/domains/jxdm/5.2/"
    xmlns:term="http://release.niem.gov/niem/localTerminology/3.0/"
    xmlns:nc="http://release.niem.gov/niem/niem-core/3.0/"
    xmlns:niem-xsd="http://release.niem.gov/niem/proxy/xsd/3.0/"
    xmlns:s="http://release.niem.gov/niem/structures/3.0/"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:nibrscodes="http://fbi.gov/cjis/nibrs/nibrs-codes/2019"
    targetNamespace="http://fbi.gov/cjis/nibrs/2019" version="2019"
ct:conformanceTargets="http://reference.niem.gov/niem/specification/naming-and-
design-rules/3.0/#ExtensionSchemaDocument">
    <xsd:annotation>
        <xsd:documentation>NIBRS Exchange Schema</xsd:documentation>
    </xsd:annotation>
    <xsd:import namespace="http://fbi.gov/cjis/nibrs/nibrs-codes/2019"
schemaLocation="nibrs-codes.xsd"/>
    <xsd:import namespace="http://fbi.gov/cjis/2.0"
schemaLocation="../../cjis/2.0/cjis.xsd"/>
    <xsd:import namespace="http://fbi.gov/cjis/cjis-codes/2.0"
schemaLocation="../../cjis/2.0/cjis-codes.xsd"/>
    <xsd:import namespace="http://release.niem.gov/niem/codes/fbi_ucr/3.2/"
schemaLocation="../../niem/codes/fbi_ucr/3.2/fbi_ucr.xsd"/>
```

```
    <xsd:import
namespace="http://release.niem.gov/niem/conformanceTargets/3.0/"
schemaLocation="../../niem/conformanceTargets/3.0/conformanceTargets.xsd"/>
    <xsd:import namespace="http://release.niem.gov/niem/domains/jxdm/5.2/"
schemaLocation="../../niem/domains/jxdm/5.2/jxdm.xsd"/>
    <xsd:import namespace="http://release.niem.gov/niem/localTerminology/3.0/"
schemaLocation="../../niem/localTerminology/3.0/localTerminology.xsd"/>
    <xsd:import namespace="http://release.niem.gov/niem/niem-core/3.0/"
schemaLocation="../../niem/niem-core/3.0/niem-core.xsd"/>
    <xsd:import namespace="http://release.niem.gov/niem/proxy/xsd/3.0/"
schemaLocation="../../niem/proxy/xsd/3.0/xs.xsd"/>
    <xsd:import namespace="http://release.niem.gov/niem/structures/3.0/"
schemaLocation="../../niem/structures/3.0/structures.xsd"/>

    <xsd:complexType name="ReportType">
        <xsd:annotation>
            <xsd:documentation>A data type for a CJIS
report</xsd:documentation>
        </xsd:annotation>
        <xsd:complexContent>
            <xsd:extension base="s:ObjectType">
                <xsd:sequence>
                    <xsd:element ref="nibrs:ReportHeader" minOccurs="1"
maxOccurs="1"/>
                    <xsd:element ref="nc:Incident" minOccurs="0"
maxOccurs="1"/>
                    <xsd:element ref="j:Offense" minOccurs="0"
```

```
maxOccurs="unbounded"/>
                    <xsd:element ref="nc:Location" minOccurs="0"
maxOccurs="unbounded"/>
                    <xsd:element ref="nc:Item" minOccurs="0"
maxOccurs="unbounded"/>
                    <xsd:element ref="nc:Substance" minOccurs="0"
maxOccurs="unbounded"/>
                    <xsd:element ref="nc:Person" minOccurs="0"
maxOccurs="unbounded" />
                    <xsd:element ref="j:EnforcementOfficial" minOccurs="0"
maxOccurs="unbounded"/>
                    <xsd:element ref="j:Victim" minOccurs="0"
maxOccurs="unbounded"/>
                    <xsd:element ref="j:Subject" minOccurs="0"
maxOccurs="unbounded"/>
                    <xsd:element ref="j:Arrestee" minOccurs="0"
maxOccurs="unbounded"/>
                    <xsd:element ref="j:Arrest" minOccurs="0"
maxOccurs="unbounded"/>
                    <xsd:element ref="j:ArrestSubjectAssociation" minOccurs="0"
maxOccurs="unbounded"/>
                    <xsd:element ref="j:OffenseLocationAssociation"
minOccurs="0" maxOccurs="unbounded"/>
                    <xsd:element ref="j:OffenseVictimAssociation" minOccurs="0"
maxOccurs="unbounded"/>
                    <xsd:element ref="j:SubjectVictimAssociation" minOccurs="0"
maxOccurs="unbounded"/>
```

```
                </xsd:sequence>
            </xsd:extension>
        </xsd:complexContent>
    </xsd:complexType>
    <xsd:complexType name="ReportHeaderType">
       <xsd:annotation>
           <xsd:documentation>A data type for header information for the
report.</xsd:documentation>
       </xsd:annotation>
       <xsd:complexContent>
           <xsd:extension base="s:ObjectType">
               <xsd:sequence>
                   <xsd:element ref="nibrs:NIBRSReportCategoryCode"
minOccurs="1" maxOccurs="1"/>
                   <xsd:element ref="nibrs:ReportActionCategoryCode"
minOccurs="1" maxOccurs="1"/>
                   <xsd:element ref="nibrs:ReportDate" minOccurs="1"
maxOccurs="1"/>
                   <xsd:element ref="nibrs:ReportingAgency" minOccurs="1"
maxOccurs="1"/>
               </xsd:sequence>
           </xsd:extension>
       </xsd:complexContent>
   </xsd:complexType>

    <xsd:complexType name="SubmissionType">
        <xsd:annotation>
```

```
            <xsd:documentation>The root element for a NIBRS
exchange</xsd:documentation>
        </xsd:annotation>
        <xsd:complexContent>
            <xsd:extension base="s:ObjectType">
                <xsd:sequence>
                    <xsd:element ref="cjis:MessageMetadata" minOccurs="1"
maxOccurs="1"/>
                    <xsd:element ref="nibrs:Report" minOccurs="1"
maxOccurs="1"/>
                </xsd:sequence>
            </xsd:extension>
        </xsd:complexContent>
    </xsd:complexType>

       <xsd:element name="VictimToSubjectRelationshipCode"
type="nibrscodes:VictimToSubjectRelationshipCodeType"
substitutionGroup="j:VictimToSubjectRelationship">
        <xsd:annotation>
            <xsd:documentation>A code that identifies the victim's relationship
to subject who perpetrated a crime against them, depicting who the victim was
to the offender.</xsd:documentation>
        </xsd:annotation>
    </xsd:element>

    <xsd:element name="ChargeUCRCode" type="nibrscodes:OffenseCodeType"
nillable="false" substitutionGroup="j:ChargeUCR">
```

```
        <xsd:annotation>
            <xsd:documentation>An offense within the Uniform Crime Report (UCR)
system.</xsd:documentation>
        </xsd:annotation>
    </xsd:element>

    <xsd:element name="CriminalActivityCategoryCode"
type="nibrscodes:CriminalActivityCategoryCodeType" nillable="false"
substitutionGroup="j:CriminalActivityCategory">
        <xsd:annotation>
            <xsd:documentation>A kind of criminal activity.</xsd:documentation>
        </xsd:annotation>

    </xsd:element>

    <xsd:element name="LocationCategoryCode"
type="nibrscodes:LocationCategoryCodeType" nillable="false"
substitutionGroup="nc:LocationCategory">
        <xsd:annotation>
            <xsd:documentation>A kind or functional description of a location.
</xsd:documentation>
        </xsd:annotation>
    </xsd:element>

    <xsd:element name="NIBRSReportCategoryCode"
type="nibrscodes:NIBRSReportCategoryCodeType" nillable="false">
        <xsd:annotation>
```

```
            <xsd:documentation>A kind of Report for the National Incident Based
Reporting System (NIBRS).</xsd:documentation>
        </xsd:annotation>
    </xsd:element>

    <xsd:element name="OffenseUCRCode" type="nibrscodes:OffenseCodeType"
nillable="false" substitutionGroup="j:OffenseDesignation">
        <xsd:annotation>
            <xsd:documentation>An offense designation as specified by FBI's
Uniform Crime Reporting (UCR) program.</xsd:documentation>
        </xsd:annotation>
    </xsd:element>

    <xsd:element name="Report" type="nibrs:ReportType" nillable="false">
        <xsd:annotation>
            <xsd:documentation>A report being submitted tot he National
Incident Based Report System (NIBRS) program.</xsd:documentation>
        </xsd:annotation>
    </xsd:element>

    <xsd:element name="ReportActionCategoryCode"
type="nibrscodes:ReportActionCategoryCodeType" nillable="false">
        <xsd:annotation>
            <xsd:documentation>A kind of action to be performed by the NIBRS
system, on the submitted report.</xsd:documentation>
        </xsd:annotation>
    </xsd:element>
```

```
    <xsd:element name="ReportDate" type="nc:DateType">
        <xsd:annotation>
            <xsd:documentation>The date the report was created.
</xsd:documentation>
        </xsd:annotation>
    </xsd:element>

    <xsd:element name="ReportingAgency" type="nc:OrganizationType">
        <xsd:annotation>
            <xsd:documentation>The agency that created the report.
</xsd:documentation>
        </xsd:annotation>
    </xsd:element>

    <xsd:element name="ReportHeader" type="nibrs:ReportHeaderType">
        <xsd:annotation>
            <xsd:documentation>A container for header information for the
report.</xsd:documentation>
        </xsd:annotation>
    </xsd:element>

    <xsd:element name="Submission" type="nibrs:SubmissionType">
        <xsd:annotation>
            <xsd:documentation>A NIBRS data submission.</xsd:documentation>
        </xsd:annotation>
```

```
    </xsd:element>
</xsd:schema>
```

Kinds of Content

There are two types of content, simple and complex. Simple content consists of things that we think of as simple data types, strings, numbers, booleans, dates, etc. In our example, `NIBRSReportCategoryCode` holds simple content. The contents is just a string.

Complex content consists of other objects. In our example, `Submission` holds complex content; it holds other objects like `MessageMetadata` and `Report`. Each of those is also contianing other objects. `MessageMetadata` for example holds `MessageDateTime`, `MessageIdentification`, `MessageImplementationVersion`, and `MessageSubmittingOrganization`.

You can also have mixed content, where an object can hold data and more objects. It's heavily used in markup languages. XHTML is a good example of this, where you could have something like:

```
<p>This is <strong>strong</strong> text!</p>
```

The `p` tag holds both strongs and the `strong` object.

Our focus is XML in a information exchange context, where mixed content is very difficult to handle, so we won't be seeing mixed content being used in the examples.

Kinds of Types

To handle these two types of content, there are two kinds of XML Schema types, however, it's not quite as simple as it seems, as there are *three* combinations.

Simple type (with simple content)

Simple types are types that hold data. XML Schema provides a large number of different sorts of data types, but here are some of the basics:

- strings
- integers
- decimals
- dates
- booleans (literally the strings "true" and "false")

XML Schema refines these basic types into many varieties. Intergers are refined into, for example, non-negative integers for representing quantities. Dates are just strings, but come in many varieties, including versions with date and time.

XML Schema validators will check that simple data follows the rules for that particular data type.

Any of the elements in our example that hold text are simple types. Here's just one example:

```
<xs:element name="IdentificationID" type="niem-xs:string">
        <xs:annotation>
                <xs:documentation>An identifier.</xs:documentation>
        </xs:annotation>
</xs:element>
```

Complex type with complex content

If, instead of simple data, an element contains other elements, then you need a complex type for that complex content. Here's a simplified version of `IncidentAugmentation` from our example:

```
<xsd:element name="IncidentAugmentation">
        <xsd:complexType>
                <xsd:complexContent>
                        <xsd:sequence>
                                <xsd:element
ref="cjis:IncidentReportDateIndicator" type="niem-xsd:boolean" minOccurs="0"
maxOccurs="1"/>
                                <xsd:element ref="j:OffenseCargoTheftIndicator"
type="niem-xsd:boolean" minOccurs="0" maxOccurs="1"/>
                        </xsd:sequence>
                </xsd:complexContent>
        </xsd:complexType>
</xsd:element>
```

That was a simple example, where the things element held were all simple content, the `IncidentReportDateIndicator` and `OffenseCargoTheftIndicator`. If we go up the hierarchy one level, we'll see a lot more complexity, and a mixture of complex content and simple content. Here's a simplified version of the definition for `Incident`:

```
<xsd:element name="Incident">
        <xsd:complexType>
                <xsd:sequence>
```

```
                        <xsd:element name="nc:ActivityIdentification"
type="nc:IdentificationType" minOccurs="0" maxOccurs="1"/>
                        <xsd:element name="nc:ActivityDate" type="nc:DateType"
minOccurs="0" maxOccurs="1"/>
                        <xsd:element name="j:IncidentAugmentation"
type="j:IncidentAugmentationType" minOccurs="0" maxOccurs="unbounded"/>
                </xsd:sequence>
        </xsd:complexType>
</xsd:element>
```

Note that `Incident` holds a variety of objects. The `ActivityIdentification`, `ActivityDate` and `IncidentAugmentation` are all complex content themselves, which can be seen in the matching instance document:

```
<nc:Incident>
        <nc:ActivityIdentification>
                <nc:IdentificationID>MULTI-OFFM</nc:IdentificationID>
        </nc:ActivityIdentification>
        <nc:ActivityDate>
                <nc:DateTime>2022-05-01T10:00:00</nc:DateTime>
        </nc:ActivityDate>
        <j:IncidentAugmentation>

<j:IncidentExceptionalClearanceCode>A</j:IncidentExceptionalClearanceCode>
                <j:IncidentExceptionalClearanceDate>
                        <nc:Date>2022-05-05</nc:Date>
                </j:IncidentExceptionalClearanceDate>
```

```
        </j:IncidentAugmentation>
</nc:Incident>
```

Complex type with simple content

The third option is to attach attributes to elements that hold simple content. We saw an example of attributes earlier:

```
<j:Victim s:id="Victim2">
        <nc:RoleOfPerson s:ref="PersonVictim2"/>
        <j:VictimSequenceNumberText>2</j:VictimSequenceNumberText>
        <j:VictimCategoryCode>I</j:VictimCategoryCode>
</j:Victim>
```

While `Victim` holds complex content, we could also put attributes on simple data, like something that is just a string. Suppose we wanted `VictimSequenceNumberText` to have an attribute telling us whether the number represented is Arabic or Roman. Here's the XML Schema for this:

```
<xs:element name="VictimSequenceNumberText">
        <xs:simpleContent>
                <xs:extension base="xs:string">
                        <xs:attribute name="numberformat" type="xs:string"/>
                </xs:extension>
        </xs:simpleContent>
</xs:element>
```

This definition takes an existing type, a string, and extends it by adding an attribute. We'll see more about extensions soon.

Now we could have this instead:

```
<j:Victim s:id="Victim2">
        <nc:RoleOfPerson s:ref="PersonVictim2"/>
        <j:VictimSequenceNumberText
numberformat="Roman">II</j:VictimSequenceNumberText>
        <j:VictimCategoryCode>I</j:VictimCategoryCode>
</j:Victim>
```

Complex types with simple content can cause issues when moving from XML to other serializations, such as JSON or RDF. Other serializations treat data as either objects or simple data types. Complex types with simple content are an oddball mix from the perspective of other serializations.

Complex Type Compositors

Let's revisit the schema example:

> nibrs.xsd
>
> ```
> <?xml version="1.0" encoding="UTF-8"?>
> <!--
>     Document: $Id: nibrs.xsd $
>     NIEM version  : 3.2
>     NIBRS version : 2019
>     Namespace     : xmlns:nibrs="http://fbi.gov/cjis/nibrs/2019"
> ```

```
    Description     : NIBRS 2019 Extension Schema
-->
<xsd:schema  xmlns:nibrs="http://fbi.gov/cjis/nibrs/2019"
    xmlns:cjis="http://fbi.gov/cjis/2.0"
    xmlns:cjiscodes="http://fbi.gov/cjis/cjis-codes/2.0"
    xmlns:ct="http://release.niem.gov/niem/conformanceTargets/3.0"
    xmlns:i="http://release.niem.gov/niem/appinfo/3.0/"
    xmlns:ucr="http://release.niem.gov/niem/codes/fbi_ucr/3.2/"
    xmlns:j="http://release.niem.gov/niem/domains/jxdm/5.2/"
    xmlns:term="http://release.niem.gov/niem/localTerminology/3.0/"
    xmlns:nc="http://release.niem.gov/niem/niem-core/3.0/"
    xmlns:niem-xsd="http://release.niem.gov/niem/proxy/xsd/3.0/"
    xmlns:s="http://release.niem.gov/niem/structures/3.0/"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:nibrscodes="http://fbi.gov/cjis/nibrs/nibrs-codes/2019"
    targetNamespace="http://fbi.gov/cjis/nibrs/2019" version="2019"
ct:conformanceTargets="http://reference.niem.gov/niem/specification/naming-and-
design-rules/3.0/#ExtensionSchemaDocument">
    <xsd:annotation>
        <xsd:documentation>NIBRS Exchange Schema</xsd:documentation>
    </xsd:annotation>
    <xsd:import namespace="http://fbi.gov/cjis/nibrs/nibrs-codes/2019"
schemaLocation="nibrs-codes.xsd"/>
    <xsd:import namespace="http://fbi.gov/cjis/2.0"
schemaLocation="../../cjis/2.0/cjis.xsd"/>
    <xsd:import namespace="http://fbi.gov/cjis/cjis-codes/2.0"
```

```
schemaLocation="../../cjis/2.0/cjis-codes.xsd"/>
    <xsd:import namespace="http://release.niem.gov/niem/codes/fbi_ucr/3.2/"
schemaLocation="../../niem/codes/fbi_ucr/3.2/fbi_ucr.xsd"/>
    <xsd:import
namespace="http://release.niem.gov/niem/conformanceTargets/3.0/"
schemaLocation="../../niem/conformanceTargets/3.0/conformanceTargets.xsd"/>
    <xsd:import namespace="http://release.niem.gov/niem/domains/jxdm/5.2/"
schemaLocation="../../niem/domains/jxdm/5.2/jxdm.xsd"/>
    <xsd:import namespace="http://release.niem.gov/niem/localTerminology/3.0/"
schemaLocation="../../niem/localTerminology/3.0/localTerminology.xsd"/>
    <xsd:import namespace="http://release.niem.gov/niem/niem-core/3.0/"
schemaLocation="../../niem/niem-core/3.0/niem-core.xsd"/>
    <xsd:import namespace="http://release.niem.gov/niem/proxy/xsd/3.0/"
schemaLocation="../../niem/proxy/xsd/3.0/xs.xsd"/>
    <xsd:import namespace="http://release.niem.gov/niem/structures/3.0/"
schemaLocation="../../niem/structures/3.0/structures.xsd"/>

    <xsd:complexType name="ReportType">
        <xsd:annotation>
            <xsd:documentation>A data type for a CJIS
report</xsd:documentation>
        </xsd:annotation>
        <xsd:complexContent>
            <xsd:extension base="s:ObjectType">
                <xsd:sequence>
                    <xsd:element ref="nibrs:ReportHeader" minOccurs="1"
maxOccurs="1"/>
```

```
                    <xsd:element ref="nc:Incident" minOccurs="0"
maxOccurs="1"/>
                    <xsd:element ref="j:Offense" minOccurs="0"
maxOccurs="unbounded"/>
                    <xsd:element ref="nc:Location" minOccurs="0"
maxOccurs="unbounded"/>
                    <xsd:element ref="nc:Item" minOccurs="0"
maxOccurs="unbounded"/>
                    <xsd:element ref="nc:Substance" minOccurs="0"
maxOccurs="unbounded"/>
                    <xsd:element ref="nc:Person" minOccurs="0"
maxOccurs="unbounded" />
                    <xsd:element ref="j:EnforcementOfficial" minOccurs="0"
maxOccurs="unbounded"/>
                    <xsd:element ref="j:Victim" minOccurs="0"
maxOccurs="unbounded"/>
                    <xsd:element ref="j:Subject" minOccurs="0"
maxOccurs="unbounded"/>
                    <xsd:element ref="j:Arrestee" minOccurs="0"
maxOccurs="unbounded"/>
                    <xsd:element ref="j:Arrest" minOccurs="0"
maxOccurs="unbounded"/>
                    <xsd:element ref="j:ArrestSubjectAssociation" minOccurs="0"
maxOccurs="unbounded"/>
                    <xsd:element ref="j:OffenseLocationAssociation"
minOccurs="0" maxOccurs="unbounded"/>
                    <xsd:element ref="j:OffenseVictimAssociation" minOccurs="0"
```

```
maxOccurs="unbounded"/>
                    <xsd:element ref="j:SubjectVictimAssociation" minOccurs="0"
maxOccurs="unbounded"/>
                </xsd:sequence>
            </xsd:extension>
        </xsd:complexContent>
    </xsd:complexType>
    <xsd:complexType name="ReportHeaderType">
       <xsd:annotation>
           <xsd:documentation>A data type for header information for the
report.</xsd:documentation>
       </xsd:annotation>
       <xsd:complexContent>
           <xsd:extension base="s:ObjectType">
               <xsd:sequence>
                   <xsd:element ref="nibrs:NIBRSReportCategoryCode"
minOccurs="1" maxOccurs="1"/>
                   <xsd:element ref="nibrs:ReportActionCategoryCode"
minOccurs="1" maxOccurs="1"/>
                   <xsd:element ref="nibrs:ReportDate" minOccurs="1"
maxOccurs="1"/>
                   <xsd:element ref="nibrs:ReportingAgency" minOccurs="1"
maxOccurs="1"/>
               </xsd:sequence>
           </xsd:extension>
       </xsd:complexContent>
   </xsd:complexType>
```

```
    <xsd:complexType name="SubmissionType">
        <xsd:annotation>
            <xsd:documentation>The root element for a NIBRS
exchange</xsd:documentation>
        </xsd:annotation>
        <xsd:complexContent>
            <xsd:extension base="s:ObjectType">
                <xsd:sequence>
                    <xsd:element ref="cjis:MessageMetadata" minOccurs="1"
maxOccurs="1"/>
                    <xsd:element ref="nibrs:Report" minOccurs="1"
maxOccurs="1"/>
                </xsd:sequence>
            </xsd:extension>
        </xsd:complexContent>
    </xsd:complexType>

       <xsd:element name="VictimToSubjectRelationshipCode"
type="nibrscodes:VictimToSubjectRelationshipCodeType"
substitutionGroup="j:VictimToSubjectRelationship">
        <xsd:annotation>
            <xsd:documentation>A code that identifies the victim's relationship
to subject who perpetrated a crime against them, depicting who the victim was
to the offender.</xsd:documentation>
        </xsd:annotation>
    </xsd:element>
```

```
    <xsd:element name="ChargeUCRCode" type="nibrscodes:OffenseCodeType"
nillable="false" substitutionGroup="j:ChargeUCR">
        <xsd:annotation>
            <xsd:documentation>An offense within the Uniform Crime Report (UCR)
system.</xsd:documentation>
        </xsd:annotation>
    </xsd:element>

    <xsd:element name="CriminalActivityCategoryCode"
type="nibrscodes:CriminalActivityCategoryCodeType" nillable="false"
substitutionGroup="j:CriminalActivityCategory">
        <xsd:annotation>
            <xsd:documentation>A kind of criminal activity.</xsd:documentation>
        </xsd:annotation>

    </xsd:element>

    <xsd:element name="LocationCategoryCode"
type="nibrscodes:LocationCategoryCodeType" nillable="false"
substitutionGroup="nc:LocationCategory">
        <xsd:annotation>
            <xsd:documentation>A kind or functional description of a location.
</xsd:documentation>
        </xsd:annotation>
    </xsd:element>
```

```
    <xsd:element name="NIBRSReportCategoryCode"
type="nibrscodes:NIBRSReportCategoryCodeType" nillable="false">
        <xsd:annotation>
            <xsd:documentation>A kind of Report for the National Incident Based
Reporting System (NIBRS).</xsd:documentation>
        </xsd:annotation>
    </xsd:element>

    <xsd:element name="OffenseUCRCode" type="nibrscodes:OffenseCodeType"
nillable="false" substitutionGroup="j:OffenseDesignation">
        <xsd:annotation>
            <xsd:documentation>An offense designation as specified by FBI's
Uniform Crime Reporting (UCR) program.</xsd:documentation>
        </xsd:annotation>
    </xsd:element>

    <xsd:element name="Report" type="nibrs:ReportType" nillable="false">
        <xsd:annotation>
            <xsd:documentation>A report being submitted tot he National
Incident Based Report System (NIBRS) program.</xsd:documentation>
        </xsd:annotation>
    </xsd:element>

    <xsd:element name="ReportActionCategoryCode"
type="nibrscodes:ReportActionCategoryCodeType" nillable="false">
        <xsd:annotation>
            <xsd:documentation>A kind of action to be performed by the NIBRS
```

```
system, on the submitted report.</xsd:documentation>
        </xsd:annotation>
    </xsd:element>

    <xsd:element name="ReportDate" type="nc:DateType">
        <xsd:annotation>
            <xsd:documentation>The date the report was created.
</xsd:documentation>
        </xsd:annotation>
    </xsd:element>

    <xsd:element name="ReportingAgency" type="nc:OrganizationType">
        <xsd:annotation>
            <xsd:documentation>The agency that created the report.
</xsd:documentation>
        </xsd:annotation>
    </xsd:element>

    <xsd:element name="ReportHeader" type="nibrs:ReportHeaderType">
        <xsd:annotation>
            <xsd:documentation>A container for header information for the
report.</xsd:documentation>
        </xsd:annotation>
    </xsd:element>

    <xsd:element name="Submission" type="nibrs:SubmissionType">
        <xsd:annotation>
```

```
            <xsd:documentation>A NIBRS data submission.</xsd:documentation>
        </xsd:annotation>
    </xsd:element>
</xsd:schema>
```

Note that each `complexType` contains a `sequence` object. The `sequence` object is called a compositor. It meaningfully collects together elements that this object needs to hold.

While `sequence` is a popular way of combining a set of XML elements into a type and the one used in our examples, it's not the only one. There are four "compositors" in total.

Sequence

The beauty of `sequence` is that it's easy to understand and parse. It lists the elements that need to be in things of a type, in order. If something required is missing, or if the elements are out of order, validation fails. This is great for information exchange, where you want to define the exchange as tightly as you can.

Choice

The `choice` compositor lists a number of elements, one of which needs to be in the instance document. This can be used, for example, to give a choice of different data formats, maybe a string versus an enumeration.

All

The `all` compositor requires that all listed elements be included, but they can be included in any order. This can be resource intensive to parse. Using `all` does not help nail down data like `sequence` does.

Any

The `any` compositor is a wildcard. Anything can show up! This is not great for data integrity, but it is good for allowing arbitrary content to be included, which can be useful. One example would be including an arbitrary document inside the larger instance document.

Cardinality

Elements

The compositors above give you different means of specifying what elements need to be inside other elements of a certain type, but what if you want some elements to be optional, others to be required, and others still to be able to appear more than once? That's where cardinality comes in.

XML Schema doesn't mark elements as optional or required. Instead it defines a minimum number of occurance and a maximum number of occurances. The attributes `minOccurs` and `maxOccurs` are used for this. The `minOccurs` value can be any non-negative integer, so zero on up. The `maxOccurs` value can also be any non-negative integer. It can also be the string "unbounded" which means you can have as many as you want.

Setting `minOccurs` to zero for an element makes it optional. Setting it to one (or higher) makes the element required.

Setting `maxOccurs` to zero is legal and it forbids the element from appearing. There really isn't a good reason to do this, but it's legal. Typically `maxOccurs` will be the number one or "unbounded", although there are use cases where you might want a maximum that's more than one but not quite unbounded.

And example might be address lines. Maybe the systems exchanging data can only handle up to three address lines. Then you would want `maxOccurs` set to three, like so:

```
<xs:complexType name="LocationType">
        <xs:sequence>
                <xs:element name="Street" type="xs:string" minOccurs="0"
maxOccurs="3"/>
                <xs:element name="City" type="xs:string" minOccurs="0"/>
                <xs:element name="State" type="xs:string" minOccurs="0"/>
                <xs:element name="ZIP" type="xs:string" minOccurs="0"/>
        </xs:sequence>
        <xs:attribute name="id" type="xs:ID"/>
</xs:complexType>
```

If `minOccurs` is left out, it defaults to one. If `maxOccurs` is left out, it also defaults to one. You can leave them both out, meaning the element needs to occur once and only once. You can also just leave one of them out, depending on your needs.

Attributes

An attribute can only appear in an element once, so it doesn't get the same fancy cardinality of an element. It can be "required", "optional", or "prohibited". That's one of the major difference between elements and attributes, and you'll see it in the table above when we first looked at them.

If no cardinality is defined, the default is "optional". Note that this is differnt than elements. Elements are required by default; attributes are optional by default.

Anonymous Types

There are a couple of different way to define elements in XML Schema. Let's look at a portion of the example schema done two different ways.

The simplest way is to specify the types of elements inside the element definition itself. Each elements tells us directly what kinds of things it can hold. The typing has no name; it's just anonymously defined inside the element. Here the definition of `Incident` done this way

```
<xsd:element name="Incident">
        <xsd:complexType>
                <xsd:complexContent>
                        <xsd:sequence>
                                <xsd:element name="nc:ActivityIdentification">
                                        <xsd:complexType>
                                                <xsd:complexContent>
                                                        <xsd:sequence>
                                                                <xsd:element
name="nc:IdentificationID" type="xsd:string"/>
                                                        </xsd:sequence>
                                                </xsd:complexContent>
                                        </xsd:complexType>
                                </xsd:element>
                                <xsd:element name="nc:ActivityDate">
                                        <xsd:complexType>
                                                <xsd:complexContent>
                                                        <xsd:sequence>
                                                                <xsd:element
name="nc:DateTime" type="xsd:dateTime"/>
                                                        </xsd:sequence>
                                                </xsd:complexContent>
                                        </xsd:complexType>
```

```
                                </xsd:element>
                        </xsd:sequence>
                </xsd:complexContent>
        </xsd:complexType>
</xsd:element>
```

This kind of XML Schema is very straighforward to follow, but it comes at a cost. With an element's type defined in the element, there's no name to call it by. So a type can't be reused elsewhere. If you have five elements all with the same type, you'll be repeating that type definition five times.

Named Types

The other method is to give names to types, allowing you to reuse those types, or even build upon them. Here's the same XML Schema, only using named types.

```
<xsd:complexType name="IncidentType">
        <xsd:complexContent>
                <xsd:sequence>
                        <xsd:element ref="nc:ActivityIdentification"/>
                        <xsd:element ref="nc:ActivityDate"/>
                </xsd:sequence>
        </xsd:complexContent>
</xsd:complexType>

<xsd:complexType name="DateType">
        <xsd:complexContent>
                <xsd:sequence>
```

```
                        <xsd:element ref="DateTime"/>
                </xsd:sequence>
        </xsd:complexContent>
</xsd:complexType>

<xsd:complexType name="IdentificationType">
        <xsd:complexContent>
                <xsd:sequence>
                        <xsd:element ref="IdentificationID"/>
                </xsd:sequence>
        </xsd:complexContent>
</xsd:complexType>

<xsd:element name="ActivityDate" type="nc:DateType">
<xsd:element name="ActivityIdentification" type="nc:IdentificationType">
<xsd:element name="DateTime" type="xsd:dateTime"/>
<xsd:element name="IdentificationID" type="xsd:string"/>
<xsd:element name="Incident" type="nc:IncidentType">
```

This allows us to reuse types. There may be several things in an exchange that have an ID attached to them. We can reuse `nc:IdentificationType` for those instead of repeating the definition over and over. And if we change what we need in an `Identification`, we can change it just once in the type rather than everywhere it's used.

Extensions

Name types provide more than just a means of grouping common definitions. XML Schema lets you take an existing type and build new types from it, adding additional pieces of information about the new object.

We've been simplifying `Incident` all along, but it's actually more complex that we've shown. `IncidentType` extends from a more generic `ActivityType`. Other sorts of things that a special forms of activities could also derive from `ActivityType`.

```
<xs:complexType name="ActivityType">
        <xs:complexContent>
                <xs:extension base="structures:ObjectType">
                        <xs:sequence>
                                <xs:element ref="nc:ActivityIdentification"
minOccurs="0" maxOccurs="1"/>
                                <xs:element ref="nc:ActivityDateRepresentation"
minOccurs="0" maxOccurs="1"/>
                        </xs:sequence>
                </xs:extension>
        </xs:complexContent>
</xs:complexType>

<xs:complexType name="IncidentType">
        <xs:complexContent>
                <xs:extension base="nc:ActivityType">
                        <xs:sequence>
                                <xs:element ref="j:IncidentAugmentation"
minOccurs="0" maxOccurs="unbounded"/>
                        </xs:sequence>
                </xs:extension>
        </xs:complexContent>
</xs:complexType>
```

```
<xs:element name="Incident" type="nc:IncidentType"/>
```

The `Incident` object inherits `ActivityIdentification` and `ActivityDateRepresentation` from `ActivityType` as well as getting `IncidentAugmentationPoint` from `IncidentType`, resulting in an object like:

```
<nc:Incident>
        <nc:ActivityIdentification>
                <nc:IdentificationID>MULTI-OFFM</nc:IdentificationID>
        </nc:ActivityIdentification>
        <nc:ActivityDate>
                <nc:DateTime>2022-05-01T10:00:00</nc:DateTime>
        </nc:ActivityDate>
        <j:IncidentAugmentation>

<j:IncidentExceptionalClearanceCode>A</j:IncidentExceptionalClearanceCode>
                <j:IncidentExceptionalClearanceDate>
                        <nc:Date>2022-05-05</nc:Date>
                </j:IncidentExceptionalClearanceDate>
        </j:IncidentAugmentation>
</nc:Incident>
```

Another example is `Substance`, which is a special kind of `Item`.

```
<xs:complexType name="ItemType">
        <xs:complexContent>
```

```
                <xs:extension base="structures:ObjectType">
                        <xs:sequence>
                                <xs:element ref="nc:ItemStatus" minOccurs="0"
maxOccurs="1"/>
                                <xs:element ref="nc:ItemValue" minOccurs="0"
maxOccurs="1"/>
                                <xs:element ref="nc:ItemCategory" minOccurs="0"
maxOccurs="1"/>
                                <xs:element ref="nc:ItemQuantity" minOccurs="0"
maxOccurs="1"/>
                        </xs:sequence>
                </xs:extension>
        </xs:complexContent>
</xs:complexType>

<xs:complexType name="SubstanceType">
        <xs:complexContent>
                <xs:extension base="nc:ItemType">
                        <xs:sequence>
                                <xs:element ref="nc:SubstanceCategory"
minOccurs="0" maxOccurs="3"/>
                                <xs:element ref="nc:SubstanceQuantityMeasure"
minOccurs="0" maxOccurs="3"/>
                        </xs:sequence>
                </xs:extension>
        </xs:complexContent>
</xs:complexType>
```

```
<xs:element name="Substance" type="nc:SubstanceType">
        <xs:annotation>
                <xs:documentation>A matter or substance of which something
consists.</xs:documentation>
        </xs:annotation>
</xs:element>
```

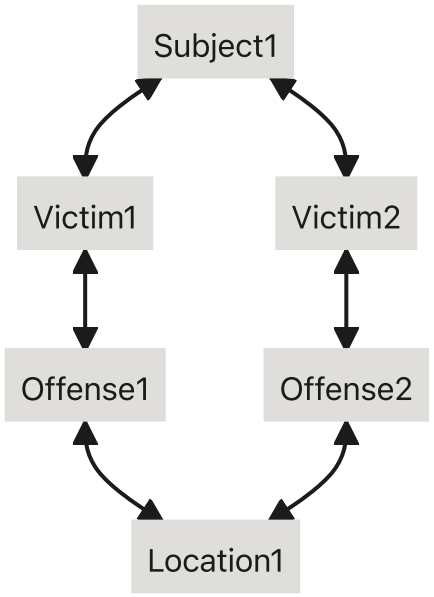
Resulting in a `Substance` object like this:

```
<nc:Substance>
        <nc:ItemStatus>
                <cjis:ItemStatusCode>SEIZED</cjis:ItemStatusCode>
        </nc:ItemStatus>
        <nc:ItemValue>
                <nc:ItemValueDate>
                        <nc:DateTime>2022-05-01T10:00:00</nc:DateTime>
                </nc:ItemValueDate>
        </nc:ItemValue>
        <nc:ItemQuantity>5</nc:ItemQuantity>
        <nc:SubstanceQuantityMeasure>
                <nc:MeasureDecimalValue>1.5</nc:MeasureDecimalValue>
                <j:SubstanceUnitCode>GM</j:SubstanceUnitCode>
        </nc:SubstanceQuantityMeasure>
</nc:Substance>
```

Referencing

Most XML is strictly hierarchical, but XML Schema also allows for making connections between elements regardless of their location in the overall document.

Looking at the example we can see a number of links that join a number of objects together like this:



To do this, XML Schema provides three types:

- ID: Assigns an ID to an element
- IDREF: Refers to an ID set on an element
- IDREFS: Refers to multiple IDs set on elements

These are all attributes that can be used to assign unique IDs to elements and have other elements refer to those IDs. The IDs are not real-world IDs. They just need to be unique within a document. They're just strings. They cannot contain spaces.

We can add these attributes to elements in a friendlier manner with small bits of XML Schema, like this:

```
<xsd:attribute name="id" type="xs:ID"/>
<xsd:attribute name="ref" type="xs:IDREF"/>
```

Then add those attributes wherever needed. To add the `id` and `ref` to `Location` objects, we add it to `LocationType`:

```
<xs:complexType name="LocationType">
        <xs:annotation>
                <xs:documentation>A data type for geospatial location.
</xs:documentation>
        </xs:annotation>
        <xs:complexContent>
                <xs:extension base="structures:ObjectType">
                        <xs:sequence>
                                <xs:element ref="nc:LocationAddress" minOccurs="0"
maxOccurs="1"/>
                                <xs:element ref="nc:LocationCategory"
minOccurs="0" maxOccurs="1"/>
                        </xs:sequence>
                        <xs:attribute ref="structures:id"/>
                        <xs:attribute ref="structures:ref"/>
                </xs:extension>
        </xs:complexContent>
</xs:complexType>
```

Now we can assign an `id` to one `Location` object and have another `Location` object point to it using a `ref` that matches the `id`:

```
<j:OffenseLocationAssociation>
        <j:Offense s:ref="Offense1"/>
        <nc:Location s:ref="Location1"/>
</j:OffenseLocationAssociation>

<j:Offense s:id="Offense1">
        <nibrs:OffenseUCRCode>13A</nibrs:OffenseUCRCode>
        <nibrs:CriminalActivityCategoryCode>N</nibrs:CriminalActivityCategoryCode>
        <j:OffenseFactorBiasMotivationCode>NONE</j:OffenseFactorBiasMotivationCode
>
        <j:OffenseFactor>
                <j:OffenseFactorCode>N</j:OffenseFactorCode>
        </j:OffenseFactor>
        <j:OffenseForce>
                <j:ForceCategoryCode>11</j:ForceCategoryCode>
        </j:OffenseForce>
        <j:OffenseAttemptedIndicator>false</j:OffenseAttemptedIndicator>
</j:Offense>

<nc:Location s:id="Location1">
        <nibrs:LocationCategoryCode>12</nibrs:LocationCategoryCode>
</nc:Location>
```

Note that these object aren't inside each other. They could be located anywhere in the XML document.

If an object you want to use solely as a reference is required to hold other objects, you can mark the object as being "nil" which tells a parser to ignore that this object would normally be required to hold other objects. If

`nc:Location` was required to hold, say, a ZIP code, then you would need to mark it as "nil" when using it solely as a reference, like so:

```
<nc:Location s:ref="Location1" nil="true"/>
```

Why use referencing? Three reasons:

1. It can avoid replication of information.
2. It establishes equivalency.
3. It allows for non-hierarchical linking of objects.

In our example, both `Offense` objects are linked to the same `Location`. This can be better than replicating the information, especially for larger objects. Additionally, this better shows that the two locations are the *same* location without resorting to matching information.

Referencing like this is not something people commonly do with XML, but XML Schema supports it. It's important to know the capability is available.

Namespaces

Namespaces are a way to organize elements into different contexts. Consider a word like "Case." What does that mean? It depends on the context. Here are just a few potential contexts:

- courts
- social work
- medical
- detective fiction
- containers

Namespaces allow us to organize elements into different contexts. XML Schema supports namespaces by allowing you to declare different contexts, defined by separate XML Schema documents, and give each a nickname to prefix onto elements.

Our example uses a variety of contexts:

- NIBRS-specific content
- FBI's CJIS standard
- Related CJIS code tables
- Other FBI code tables
- Content from the National Information Exhange Model
  - Generic content
  - Justice-specific content
- Data types from XML Schema

To ensure we know which elements come from which contexts, we declare them all, giving the a globally unique name, and a nickname for easy reference. Those declarations are attributes in the root element.

```
<xsd:schema
        xmlns:nibrs="http://fbi.gov/cjis/nibrs/2019"
        xmlns:cjis="http://fbi.gov/cjis/2.0"
    xmlns:cjiscodes="http://fbi.gov/cjis/cjis-codes/2.0"
    xmlns:ucr="http://release.niem.gov/niem/codes/fbi_ucr/3.2/"
    xmlns:nc="http://release.niem.gov/niem/niem-core/3.0/"
    xmlns:j="http://release.niem.gov/niem/domains/jxdm/5.2/"
    xmlns:niem-xsd="http://release.niem.gov/niem/proxy/xsd/3.0/">
```

Each of these attributes defines a context. Consider `xmlns:nibrs="http://fbi.gov/cjis/nibrs/2019"`. The `xmlns:` says that it's declaring a namespace. The `"http://fbi.gov/cjis/nibrs/2019"` is the globally unique name for the namespace. The `nibrs` is a nickname for that longer unique name. The long name is the actual name, but the nickname is used as shorthand in the elements.

Up until now, we've glossed over these prefixes, but now we can look through our example and see where each element and type comes from.

Each namespace is defined in a separate schema document. The objects with a `nibrs:` prefix are defined in the nibrs.xsd schema document. The objects with a `nc:` prefix are defined in the niem-core.xsd schema. One schema brings in related schemas with an `import` statement that lists the full name of the namespace being imported and where to fine the file that defines it. Here's how the schema file for the `nibrs:` prefix pulls in the `nc:` schema:

```
<xs:import namespace="http://release.niem.gov/niem/niem-core/3.0/"
schemaLocation="../../niem/niem-core/3.0/niem-core.xsd"/>
```

It's just as important to declare and use namespaces correctly as it is to spell the tag names correctly.

If you don't correctly declare namespaces, using the correct URIs, and don't correctly import them, then systems receiving the XML will not know what to do with the data.

Omitting or using incorrect namespace information is as bad, from a system perspective, as using `<Persom>` as the tag name for Person objects. Computers can't just figure it out.

When in doubt refer to valid NIBRS examples.

Data Restrictions

Enumerations

There are many ways to contrain what a value can be in XML. A very commonly used method is using enumerations to create a code table. Code tables are a great way to ensure that data sent belongs to a set of valid values. They're easy to create in XML Schema. Here's a code table for the FBI's sex codes:

```
<xs:simpleType name="SEXCodeSimpleType">
        <xs:annotation>
                <xs:documentation>A data type for Sex.</xs:documentation>
        </xs:annotation>
        <xs:restriction base="xs:token">
                <xs:enumeration value="F">
                        <xs:annotation>
                                <xs:documentation>Female</xs:documentation>
                        </xs:annotation>
                </xs:enumeration>
                <xs:enumeration value="M">
                        <xs:annotation>
                                <xs:documentation>Male</xs:documentation>
                        </xs:annotation>
                </xs:enumeration>
                <xs:enumeration value="U">
                        <xs:annotation>
                                <xs:documentation>Unknown - For Unidentified
Only</xs:documentation>
                        </xs:annotation>
                </xs:enumeration>
```

```
        </xs:restriction>
</xs:simpleType>

<xs:element name="PersonSexCode" type="ncic:SEXCodeType"
substitutionGroup="nc:PersonSex">
        <xs:annotation>
                <xs:documentation>A gender or sex of a person.</xs:documentation>
        </xs:annotation>
</xs:element>
```

A code table starts with a simple type. It's based on `xs:token`, which is just a string that's had the whitespace removed. That makes it easier for a parser to match it to the allowable values. Instead of adding to a type, as we've seen before, this is a restriction. We're restricting the string to one of a number of valid choices. Each valid choice is detailed in an `xs:enumeration`. The `value` attribute is the code, the valid string. Each enumeration can also have a definition, so that you know what the code stands for. For state codes, you can usually guess, but a lot of real-world code tables are simply numbers that map to longer strings. So it's good to have those definitions in the schema.

XML Schema validators will check that the values in an instance document actually match one of the values in the code table. This example would pass the FBI sex code table test.

```
<nc:Person>
        <j:PersonSexCode>M</j:PersonSexCode>
</nc:Person>
```

This example would fail. There is no `enumeration` with a `value` of "X".

```
<nc:Person>
        <j:PersonSexCode>X</j:PersonSexCode>
</nc:Person>
```

Enumerations are very common, and our example uses several. These other means of restricting values aren't as common and aren't used in our example, but are important to know about regardless.

Patterns

Another way to restrict the values a string can hold is through patterns. Patterns define a regular expression that describes what the string need to look like. Suppose we wanted to ensure that any ZIP code sent was in the full 5+4 format. Here's a type for that:

```
<xs:simpleType name="ZIPCodeType">
        <xs:annotation>
                <xs:documentation>A data type for ZIP codes.</xs:documentation>
        </xs:annotation>
        <xs:restriction base="xs:token">
                <xs:pattern value="[0-9][0-9][0-9][0-9][0-9]-[0-9][0-9][0-9][0-
9]">
                        <xs:annotation>
                                <xs:documentation>Five digits, a dash, and four
more digits.</xs:documentation>
                        </xs:annotation>
                </xs:pattern>
        </xs:restriction>
</xs:simpleType>
```

Ranges

Numeric types can be contrained to a range. Here's an example constraining an age to a reasonable range:

```
<xs:element name="PersonAge">
  <xs:simpleType>
    <xs:restriction base="xs:integer">
      <xs:minInclusive value="0"/>
      <xs:maxInclusive value="120"/>
    </xs:restriction>
  </xs:simpleType>
</xs:element>
```

The full selection of contraints is defined in the schema for XML Schema itself.

String Lengths

Strings can contrained by length:

```
<xs:element name="password">
  <xs:simpleType>
    <xs:restriction base="xs:string">
      <xs:minLength value="8"/>
      <xs:maxLength value="16"/>
    </xs:restriction>
  </xs:simpleType>
</xs:element>
```

The full selection of contraints is defined in the schema for XML Schema itself.

Substitution Groups

Sometimes you might have a concept that can be represented in multiple ways. A simple example is the difference in writing dates between the US and Europe. The US writes dates with the month, followed by day, then year. Europe starts with the day, followed by the month, then year. (Both are wrong, incidentally. XML Schema's built-in date type uses ISO format, YYYY-MM-DD, which is unambiguous and textually sorts correctly.)

Another example is when you have code tables, like the sex codes we used as an example above, but you might also want to let a document use a full textual description of a sex as string. You *could* have two separate elements included inside a type for that, but substitution groups are a cleaner solution.

With substitution groups, you define an element to act as the concept. That's the substitution group head. They you define substitution group members. One of those members can then substitute for the head.

Here's our example, with support for describing sex by either a full string or by the code table we described earlier:

```
<xs:element name="PersonSex" abstract="true">
        <xs:annotation>
                <xs:documentation>A data concept for a gender or sex of a person.
</xs:documentation>
        </xs:annotation>
</xs:element>

<xs:element name="PersonSexCode" type="ncic:SEXCodeType"
substitutionGroup="nc:PersonSex">
        <xs:annotation>
```

```
                <xs:documentation>A gender or sex of a person.</xs:documentation>
        </xs:annotation>
</xs:element>

<xs:element name="PersonSexText" type="nc:TextType"
substitutionGroup="nc:PersonSex">
        <xs:annotation>
                <xs:documentation>A gender or sex of a person.</xs:documentation>
        </xs:annotation>
</xs:element>
```

The `PersonSex` element has an attribute called `abstract` with a value of "true". This means the `PersonSex` element has no type and can't actually appear in the XML document. Instead, it would be substituted by either of the members of its group. The members of its group are identified by the `substitutionGroup` attribute. That attribute's value is the element name that the group member can substitute for.

The resulting XML *could* be:

```
<nc:Person>
        <j:PersonSexCode>M</j:PersonSexCode>
</nc:Person>
```

Or it *could* be:

```
<nc:Person>
        <nc:PersonSexText>Male</nc:PersonSexText>
</nc:Person>
```

Substitution groups are great for XML Schema that is distributed in terms of governance. One body could be maintaining one substitution group element, while another maintains the other. The substitution group head could be maintained by a third body. None need to be concerned with the other.

However, substitution groups are not always well supported by development environments, especially if the group members are in different namespaces. In most cases, you would be better served by using the `choice` compositor described earlier.

This XML Schema functions the same as the substitution group examples above:

```
<xs:complexType name="PersonSexType">
        <xs:choice>
                <xs:element ref="j:PersonSexCode"/>
                <xs:element ref="nc:PersonSexText"/>
        </xs:choice>
</xs:complexType>
```

Annotations

XML Schema also supports ways to include additional information about elements, definitions essentially.

For documentation for humans to read, there's the `documentation` elements, which goes inside an `annotation` element. It provides human-readable documentation about the element. It's just text and is ignored when validating instance documents. Here's an example:

```
<xsd:element name="Report" type="nibrs:ReportType" nillable="false">
        <xsd:annotation>
                <xsd:documentation>A report being submitted tot he National
```

```
Incident Based Report System (NIBRS) program.</xsd:documentation>
        </xsd:annotation>
</xsd:element>
```

The `annotation` element can be applied to pretty much anything, elements, types, enumerations in a code table. They're good practice to use, to better describe what information each element and type is meant to represent.

Conclusion

You should now have a solid overview of XML and XML Schema. Now you know how the combination allow for the definition and creation of XML documents that contain information to be exchanged, be it a NIBRS submission or some other sort of information exchange.

XML and XML Schema are not the end!

But XML and XML Schema aren't the end. There are plenty of business rules in any exchange that can't be reflected in XML Schema. Those rules can be additional constraints for the XML itself, or wider business rules that impact how the exchanges are implemented. For NIBRS, some to pay special attention to are:

- Don't uploading multiple incidents in a single XML file. Crime reporting via XML is transaction-based, and only one incident and arrest can be reported in a single XML file. Each file should be transmitted individually.

- The importance of Action Types. Don't modify an Incident by deleting it and re-adding it (sending ‘D’ and then ‘I’). Instead, replace it (by directly sending ‘R’). This can minimize the load on end systems. For larger states like CA, this is a big issue.

- Instead of dumping data simultaneously, it is preferable to send incidents sequentially from the RMS system to Repository. This supports the sequential approach used when extracting data from the Repository to the FBI.

- As a good practice, send only the latest version of incidents. Don't upload multiple versions of an incident at the same time, e.g. ‘I’, “D’, ‘I’, “D’, ‘I’, ‘D’, ‘I’ at the **same split second**. Instead, upload only the latest version with ‘I’ or ‘R’.

Our Vision is to support public sector *Mission Priorities* conducting process and policy assessments with the goal of transformation to improved information sharing that results in *safer, healthier, and happier communities*.

www.ijis.org

Questions?

Info@ijis.org