ADOBE® INDESIGN® CS6

ADOBE INDESIGN MARKUP LANGUAGE (IDML) COOKBOOK

Document Update Status		
CS6	Unchanged	Content not guaranteed to be current.

Contents

Adobe InDesign Markup Language (IDML) Cookbook

Introduction

This document contains information about working with the Adobe® InDesign® CS6 markup language, called *InDesign Markup Language (IDML)*. This document is meant to be used in conjunction with the *IDML Language Specification*, which contains a formal description of IDML and the package file format.

IDML is an XML-based format for representing InDesign content. Essentially, it is a major revision of INX, InDesign's previous XML-based file format. IDML addresses INX shortcomings, in an effort to represent InDesign content in a human-readable format—something that can be reasonably assembled and disassembled by a competent XML programmer.

Where is IDML used in the application?

IDML syntax is used in several application features: IDML export (*.IDML), asset libraries (*.INDL), snippets (*.INDS), InCopy stories (*.ICML), and Adobe InCopy® assignment files (*.ICMA).

IDML files represent complete InDesign documents. Because they are ZIP (or, more accurately, UCF) archives, they commonly are called *packages*. These packages contain a hierarchy of XML files representing various parts of the InDesign document. An equivalent XML representation is used in single-text-file (non-archive-based) InCopy story files, snippets, and assignments. These are not ZIP files but rather single XML files that contain all elements necessary to reconstruct a particular piece of content.

How can IDML be used outside of InDesign?

IDML was designed to facilitate the inspection and construction of InDesign content outside of InDesign. The SDK provides many examples, written in both Java and ActionScript 3.0/Air, that demonstrate how IDML can be used outside of InDesign. Most of the samples in this SDK focus on Java, but both languages have the capabilities needed to work with IDML files. These languages often achieve the same result using different approaches. The samples provided highlight those differences.

What is IDMLTools?

IDMLTools is a collection of sample code, tools, and reusable classes written in Java. It contains standalone tools for schema validation, as well as package inspection, compression, and decompression. The IDMLTools sample code demonstrates several operations, including building IDML files from scratch, extracting data, altering content, and adding to existing documents. Most of these samples use XSLT and are built on a more generic set of Java classes.

Where is the IDMLTools sample code?

The IDMLTools sample code is part of the InDesign Products SDK. It can be found in the SDK in the following location:

<SDK>/devtools/sdktools/idmltools (or *<IDMLTOOLS>* for short)

What setup is required for IDMLTools

There are two set-up steps for IDMLTools:

1. Create an environment variable called IDMLTOOLS_HOME that contains the location of your IDMLTools directory.

2. Add *<IDMLTOOLS>*/bin to your system's PATH environment variable. This allows you to run IDMLTools programs from any directory.

On Windows, environment variables are set in the Systems Properties panel. You can get to System Properties via the Control Panel or by right-clicking "My Computer" and selecting Properties. While in System Properties, select the Advanced tab and click "Environment Variables."

There are several ways to add environment variables on Mac OS. If you are using Bash (the default terminal shell), consider adding them to your shell startup file. Look for a file called ~/.bashrc. (If you do not have such a file, create one.) Then define a variable as follows:

```
export IDMLTOOLS_HOME="/sdk/devtools/sdktools/idmltools"
export PATH="$PATH:$IDMLTOOLS_HOME/bin"
```

Change the contents above to match the absolute path to your idmltools folder. You also will need to start a new shell before these changes take effect.

What is in an IDML file?

Following the time-tested practice of "Hello World" sample programs, it is useful to consider a "Hello World" package. The SDK includes two such packages in the *<IDMLTools>*/samples/helloworld directory.

▶ helloworld-1.idml was created by InDesign.

▶ helloworld-2.idml is a very minimal IDML file that was created with a text editor.

To view the content of these files, you need to unzip them. If you have set up IDMLTools, you can use the package tool to decompress these files, as follows:

Windows:

```
package.bat -d helloworld-1.idml helloworld-1
package.bat -d helloworld-2.idml helloworld-2
```

Mac OS:

```
package.sh -d helloworld-1.idml helloworld-1
package.sh -d helloworld-2.idml helloworld-2
```

By design, InDesign's IDML export (helloworld-1.idml) contains all types of IDML package files, including a file for each story, spread, and master spread. For Helloworld-1, that amounts to the following files, which are explained in *IDML File Format Specification*.

▶ designmap.xml

▶ mimetype

▶ MasterSpreads\MasterSpread_uc4.xml

▶ META-INF\container.xml

▶ META-INF\metadata.xml

▶ Resources\Fonts.xml

▶ Resources\Graphics.xml

▶ Resources\Preferences.xml

▶ Resources\Styles.xml

▶ Spreads\Spread_ubd.xml

▶ Stories\Story_ud5.xml

▶ XML\BackingStory.xml

▶ XML\Tags.xml

Within these files, each property is set. The designmap.xml file is over 300 lines long. Each preference, style, and swatch is represented in the files in the Resources folder. This is roughly equivalent to writing a script that sets every property in a blank document, and then creates content, setting every possible property on the new content. This is by design, because it is desirable for documents to be equivalent when opened on machines with different defaults.

While IDML export in InDesign is verbose, import is designed to be very flexible. It does not require each attribute to be set; it works on default values for those items that are left out of the file. The helloworld-2.idml file contains only part of the XML files normally found in the package, and a smaller set of elements and attributes. This is roughly equivalent to writing a script that depends on defaults. For example, it takes only a few lines of code to create a document with one page, containing one text frame. Such a script need not be concerned with preferences, styles, and swatches; it can safely assume that application default values are set. This is much more like writing a small script that creates InDesign content.

Helloworld-2 is very simple in comparison to Helloworld-1. The designmap.xml file is only a few lines long, consisting of a Document element that includes spread and story files using idPkg:Spread and idPkg:Story elements.

```
<?xml version="1.0" encoding="utf-8"?>
<?aid style="50" type="document" readerVersion="6.0" featureSet="257"?><Document
      xmlns:idPkg="http://ns.adobe.com/AdobeInDesign/idml/1.0/packaging"
      DOMVersion="7.0" Self="d">
   <idPkg:Spread src="Spreads/Spread_spread1.xml"/>
   <idPkg:Story src="Stories/Story_story1.xml"/>
</Document>
```

The only other meaningful content is in the spread and story files. The story file is almost as simple. Within an idPkg:Story element, there is a Story element; this element provides an ID for the story in the Self attribute. This makes it possible for a text frame to reference this story elsewhere in the package. We will see that in the spread file. The Story element contains the text "Hello World." This text is formatted with default text attributes.

```
<?xml version="1.0" encoding="utf-8"?>
<idPkg:Story xmlns:idPkg="http://ns.adobe.com/AdobeInDesign/idml/1.0/packaging"
      DOMVersion="7.0">
   <Story Self="story1">
      <Content>Hello World!</Content>
   </Story>
</idPkg:Story>
```

The spread file also is simple, but it requires a little more explanation. First, examine the code:

```
<?xml version="1.0" encoding="utf-8"?>
<idPkg:Spread xmlns:idPkg="http://ns.adobe.com/AdobeInDesign/idml/1.0/packaging"
       DOMVersion="7.0">
    <Spread Self="spread_1" PageCount="1">
        <TextFrame Self="textframe1" ParentStory="story1" ContentType="TextType"
               ItemTransform="1 0 0 1 -612 -396">
            <Properties>
                <PathGeometry>
                    <GeometryPath PathOpen="false">
                        <PathPointArray>
                            <PathPoint Anchor="36 36" LeftDirection="36 36"
                                RightDirection="36 36"/>
                            <PathPoint Anchor="36 186" LeftDirection="36 186"
                                RightDirection="36 186"/>
                            <PathPoint Anchor="172 186" LeftDirection="172 186"
                                RightDirection="172 186"/>
                            <PathPoint Anchor="172 36" LeftDirection="172 36"
                                RightDirection="172 36"/>
                        </PathPointArray>
                    </GeometryPath>
                </PathGeometry>
            </Properties>
        </TextFrame>
    </Spread>
</idPkg:Spread>
```

Like the story file, the spread file's document element is in the idPkg namespace. The actual Spread element is given a unique ID in the Self attribute. The PageCount attribute controls how many pages appear in this spread. Without this attribute, the default number of pages would be used. The ItemTransform for the spread is significant. As you deal with files containing multiple spreads, you will see that each spread is translated vertically down the pasteboard; for example, the spreads in any IDML document. The pagebuilder sample also generates such a document from a template.

Within the spread, there is one text-frame element. This frame identifies itself with the story whose Self ID is "story1," using the ParentStory attribute. If you look back at the story file, notice that this is the ID of the story defined above. The TextFrame also has its own ItemTransform. If this were set to a special matrix called the identity matrix ("1 0 0 1 0 0"), the origin would be placed at the center of the spread binding. The ItemTransform in this TextFrame translates the origin back to the upper left-hand corner of the page. Because the page is 612 points wide by 792 points tall, the origin is moved back from the binding (which by default is on the right of the page) by 612 points, and up half the height of the page (396 points). The page-builder sample demonstrates how to do this with multipage spreads, with the binding in various locations.

The PathPoint array specifies the location of the TextFrame. *IDML File Format Specification* provides information on these elements.

Where are the IDML defaults, and how do they work?

It would not be desirable for IDML files to truly work like small scripts that depend on application default values, as that would mean documents would be rendered differently on machines with different application defaults. To prevent that from happening, IDML maintains a set of static defaults in the following file:

```
<InDesign>/Presets/Default/Predef.iddx
```

This file always is imported just before any IDML file, guaranteeing consistent defaults. The values are similar to what you would get if you created an empty InDesign document just after installing the application.

By design, the Predef.iddx file is not extensible, and it should not be changed. It is a static file, meaning it is not recreated based on your plug-in configuration. Furthermore, you cannot reference items in Predef.iddx in your IDML by InDesign.

What is in an IDMS file?

IDMS is the new extension for IDML-based snippets. A snippet is a single XML file describing a subset of the content within an InDesign document. IDML-based snippets are single files (not packages) that describe content using IDML. For detailed information about snippets, read the "Snippet Fundamentals" chapter of the *Adobe InDesign Programming Guide*.

The best way to understand what's in a snippet is to generate several page items and export them as InDesign Snippets. In short, a snippet contains all objects and properties necessary to reconstruct a given set of objects in a new document. The application determines these dependencies and adds to the export. You'll notice that a snippet is very verbose, and resembles something closer to a full IDML export. Like the package import, it is possible for you to provide simplified files; however, make sure all dependencies are provided within the snippet.

What is in an ICML file?

ICML is the new extension for IDML-based InCopy snippets. These special snippets contain the properties and objects necessary to reconstruct an InCopy story.

What is in an ICMA file?

ICMA is the new extension for IDML-based InCopy assignment files.

Is IDML a closed file format?

No, IDML is an open file format. As described in the *IDML Language Specification*, it is based on scripting. Any plug-in can extend the scripting model and, in doing so, extend IDML. Roughly speaking, scripting objects become elements, and scripting properties become either attributes or child elements. There are exceptions, and this is discussed in more detail in the *Language Specification*.

Element and attribute names are equivalent to the JavaScript representation. For example, the Frame Label SDK sample adds properties to page items. This amounts to adding the following attributes to all page item elements. These particular values were added to a TextFrame element and correspond to what was set in the UI:

```
FramelabelString="My Frame Label" FramelabelSize="12" FramelabelVisibility="true"
FramelabelPosition="FrameLabelBottom"
```

Is INX still supported?

INX export has been removed from the product. You can, however, still import INX files.

IDML Schema Validation

IDML features were designed to support schema validation. The compact form of RELAX NG was selected for its expressiveness, simplicity, and readability. For background on RELAX NG, read *RELAX NG* by Eric van der Vlist (O'Reilly Media, Inc., 2003).

InDesign's IDML support is built on the scripting model and can be extended by third-party plug-ins. This means that IDML cannot be described with a single, static schema. Instead, schemas must be generated to match a particular plug-in configuration. To support this, InDesign provides a means to generate schemas at runtime.

How do I generate a schema?

Solution: Use the GeneratePackageSchema.jsx and GenerateSchema.jsx scripts in the *<IDMLTools>*/scripts directory.

There are two types of schemas: one used for IDML packages; the other, for the consolidated nonpackage files. The app.generateIDMLSchema (JavaScript) event writes either of these schema types to disk. It has two parameters:

▶ The destination directory for the schema.

▶ A Boolean value that controls whether a package or nonpackage schema is generated.

For example, the following JavaScript generates the package schema:

```
app.generateIDMLSchema(Folder("/idml-schema/package"), true);
```

And the following generates the nonpackage schema:

```
app.generateIDMLSchema(Folder("/idml-schema/single"), false);
```

Both schema types contain a common file, called datatype.rnc. This file is equivalent in both schemas and specifies common data types.

A nonpackage schema comprises two files: the datatype.rnc XML file and the IDMarkupLanguage.rnc schema file. The datatype.rnc file is included by IDMarkupLanguage.rnc.

The package schema comprises a collection of the following schemas. There are corresponding schema files for each XML file within an IDML package:

▶ datatype.rnc

▶ designmap.rnc

▶ MasterSpreads\MasterSpread.rnc

▶ Resources\Fonts.rnc

▶ Resources\Graphic.rnc

▶ Resources\Preferences.rnc

▶ Resources\Styles.rnc

▶ Spreads\Spread.rnc

▶ Stories\Story.rnc

▶ XML\BackingStory.rnc

▶ XML\Mapping.rnc

▶ XML\Tags.rnc

In most cases, there is a one-to-one correspondence between schema and package XML files. For example, designmap.rnc describes what is legal or valid in a designmap.xml file. A package can contain many story, spread, and masterspread XML files of various names. The Story.rnc, Spread.rnc, and MasterSpread.rnc files describe what is valid in those files.

How do I validate an IDML file?

NOTE: The SDK does not provide a sample of validating an IDML file using Actionscript/Air.

Solution: Use the validation tool provided in IDMLTools:

1. Set up IDMLTools, as described in the IDMLTools ReadMe file.

2. Create both versions of the schema, as discussed in <u>"How do I generate a schema?"</u>.

3. Call the appropriate platform validation script in *<IDMLTOOLS>*/bin.

IDMLTools contains a Jing-based validation tool. Jing is a Java-based, open-source, schema-validation package that supports the compact form of RELAX NG. The IDMLTools validation tool is implemented in the com.adobe.idml.Validator class. It can be used from Java or executed from the command line.

Validating a nonpackage IDML file is technically very simple: it amounts to running the validation on a single XML file. That is simple to do with Jing alone. The real benefit of using the Validator class comes when dealing with packages. The validation tool automatically decompresses the files to a temporary directory and calls validation on each package file. It also contains additional validations that cannot be expressed in Relax NG; for example, it checks that all referenced package files exist.

Platform shell scripts (validate.bat for Windows and validate.sh for Mac OS) set the class path and execute the Validator class. These scripts are in the following location:

```
<IDMLTOOLS>/bin
```

Tip: To facilitate validation, add this to your PATH environment variable and run validation from any directory. This is why all scripts are structured to use the IDMLTOOLS_HOME environment variable.

To validate a package or nonpackage file, run the validate script with two arguments. The first argument is the location of the schema; the second, the location of the file to validate. The class determines whether it is dealing with a package or nonpackage file and calls Jing to do the actual validation.

Windows:

```
validate.bat c:\idml-schema\package-schema MyFile.idml
validate.sh c:\idml-schema\non-package-schema MyStory.icml
```

Mac OS:

```
validate.sh /idml-schema/package-schema/ MyFile.idml
validate.sh /idml-schema/non-package-schema/ MyStory.icml
```

For an up-to-date usage message, run the script with the -h option.

Tools

What external tools does Adobe recommend for working with IDML?

You will need some type of ZIP software for working with IDML. Commercial ZIP applications like WinZip should work fine. The InDesign Products SDK provides a free Java-based library and application that can be used to compress and decompress IDML packages.

When working with IDML, it's important to have a good XML editor. We found the Oxygen XML Editor to be a particularly effective tool for dealing with IDML. It contains support for the Compact RELAX NG schema and, after associating IDML with the ZIP file type, it allows you to browse IDML packages directly. This ZIP-file support makes it much easier to experiment with IDML. You can browse and change files in the IDML package directly. The archive is updated when the file is saved.

There is an open-source plug-in for Eclipse, called the "Eclipse Zip Editor." This plug-in provides a ZIP-editing experience similar to what is available commercially in Oxygen.

NOTE: The ZIP archive support was added in Oxygen version 9.3. If you purchase Oxygen, it's a good idea to buy the maintenance pack, as they regularly add new features.

What tools are included in the SDK?

The InDesign Products SDK contains Java-based tools, source code, and samples that will help you work with IDML. This collection of code, known as *IDMLTools*, is in the following location:

`<SDK>/devtools/sdktools/idmltools`

Before using IDMLTools, complete the following setup steps:

▶ Copy IDMLTools to somewhere on your hard drive.

▶ Add *<IDMLTOOLS>*/bin to your PATH environment variable.

▶ Add an environment variable called IDMLTOOLS_HOME that contains the path to your installation of IDMLTools.

These setup steps are described more thoroughly in the IDMLTools Readme file:

`<IDMLTOOLS>/Readme.txt`

Code constructs are described in the JavaDocs. To get the JavaDocs, unzip the following file:

`<IDMLTOOLS>/docs.zip`

Finally, several sample projects are in the following directory:

`<IDMLTOOLS>/samples`

The InDesign Products SDK also contains ActionScript-based samples for working with IDML, in the following location:

`<SDK>/idml flex samples`

What if my IDML file passes schema validation but does not work as expected?

Solution: Use the INXErrorLogging sample to see whether any scripting errors are reported.

Like INX, IDML is designed to quietly ignore unexpected data and scripting errors. This makes it possible to open a file in a previous release or with missing plug-ins; however, this relaxed behavior makes it difficult to debug IDML files. Through a plug-in, you can capture such errors. This is done by providing an implementation of the IINXErrorHandler interface. The INXErrorLogging sample demonstrates how to implement this interface and logs errors to a file. You probably will find it to be an adequate debugging tool.

The INXErrorLogging plug-in is a model-only plug-in that is compatible with InDesign, InCopy, and InDesign Server. It can be controlled via scripting as follows. The following JavaScript enables and specifies the location of the log file using a cross-platform path:

```
app.inxerrlogOn = true;
app.inxerrlogPath = File("/c/temp.txt");
```

To disable:

```
app.inxerrlogOn = false;
```

There also is an accompanying user-interface plug-in, INXErrorLoggingUI. It provides a simple user interface that enables and disables the INXErrorLogging sample via a menu item (Plug-Ins > SDK > INXErrorLoggingUI > Install INX Error Handler[US]). When the logger is enabled via the menu, a dialog is displayed enabling the user to select the log file. Having access to these errors is extremely useful when debugging IDML files. Along with schema validation, this is useful when debugging problems in your externally created IDML files. The INXErrorLogging sample is in the InDesign Products SDK. Copy the built plug-in to the application plug-ins folder.

How can I use InDesign to produce a package from a directory?

Solution: Use scripting with the packageUCF and unpackageUCF events.

To unpackage an IDML file to a directory:

```
var fromIDMLFile = new File( "~/Desktop/blankDoc.idml" );
var toFolder = new Folder( "~/Desktop/blankDocUnpackaged" );
app.unpackageUCF( fromIDMLFile, toFolder );
```

To produce a package from the contents of a directory:

```
var fromFolder = new Folder( "~/Desktop/blankDocUnpackaged" );
var toIDMLFile = new File( "~/Desktop/blankDocUnpackagedRepackaged.idml" );
app.packageUCF( fromFolder, toIDMLFile );
```

How do I produce an IDML package outside of InDesign?

Solution: Use the com.adobe.idml.Package class (found in IDML Tools) or another compression utility.

IDML packages are thoroughly described in the *IDML Language Specification*. These archives are Adobe UCF (Universal Container Format) files. UCF is built on top of the ZIP format. In addition to being a ZIP file, a UCF file follows several rules, described here.

The rules for building the archive are as follows:

1. A file called mimetype must be the first file in the archive, and it must be stored uncompressed. It must contain the following line:

   ```
   application/vnd.adobe.indesign-idml-package
   ```

2. The META-INF directory can contain several files expected by UCF. IDML uses two such files: container.xml describes the file type and points to the root file (desinmap.xml), and metadata.xml contains XMP metadata.

3. The hierarchy of files as described in the *IDML Language Specification* must exist at the top level of the archive. (A common mistake is to compress the files into a subdirectory.) For example:

   ```
   mimetype
   META-INF/container.xml
   META-INF/metadata.xml
   designmap.xml
   MasterSpreads/MasterSpread_A.xml
   Resources/Fonts.xml
   Resources/Graphic.xml
   Resources/Preferences.xml
   Resources/Styles.xml
   Spreads/Spread_spread1.xml
   Spreads/Spread_spread2.xml
   Stories/Story_story0.xml
   Stories/Story_story1.xml
   XML/BackingStory.xml
   XML/Mapping.xml
   XML/Tags.xml
   ```

4. Build the ZIP file with UNIX path component separators (/), even on Windows. With Java, it is possible to build ZIP files with \ separators on Windows; however, InDesign cannot use such files. The fact that these files exist in a package presents an initial implementation hurdle. The first steps in working with IDML involve examining a package and deciding how to deal with the archive files.

Several options are available for building packages:

▶ IDMLTools contains Java code that demonstrates compressing and decompressing packages using Java's built-in ZIP support. See the com.adobe.idml.Package class for an example of how packages are handled in the Java samples. For your convenience, a command-line version of the class can be driven using the package-wrapper scripts. Setup and use of these tools is described in the IDML ReadMe file.

▶ InDesign provides scripting support for creating a package from the files on the file system.

▶ ActionScript 3.0/Air can compress and decompress packages. For an example, see the CompressionUtils.as class in any ActionScript sample.

▶ You can use off-the-shelf, ZIP-compliant software like WinZip. WinZip has been shown to be compatible with InDesign; however, it is more difficult to control the order and compression of individual files in the archive. To work around this, you first create a ZIP file containing only the uncompressed mimetype file, then you append the remaining files.

You may find all this unnecessary if you are simply feeding the files into InDesign. You must follow these rules, however, if you expect the file to correctly expose its XMP metadata and act as a first-class citizen in Adobe Bridge.

Recipes and Pointers

Spreads and pages

Adding a spread to an existing IDML document

1. Add an XML file to the "Spreads" directory. For example, add Spreads/MySpread.xml.

2. Add content to the new spread XML file. For an example of a simple spread file, see the simplified "Hello World" example in ["What is in an IDML file?"](#).

3. Be sure to leave off the Spread element's ItemTransform attribute. This will be computed for you based on the order of spreads.

4. Add an idPkg:Spread element for the new spread to designmap.xml. The actual spread order is determined by the order in which spreads are referenced in the designmap.xml file; for example:

    ```
    <idPkg:Spread src="Spreads/Spread_spread1.xml"/>
    ```

Adding a page to a spread

1. Set the Spread element's PageCount attribute to the new number of pages on the spread.

2. Add a page element in the position of the new page. This can be empty, or you can specify page properties.

Controlling the page binding location

The page-binding location is controlled by the BindingLocation attribute on the Spread element. All pages with an index less than the BindingLocation are left pages; all pages with an index greater, right pages. Both the BindingLocation and page index are zero-based.

The Page Builder sample demonstrates the creation of spreads with various number of pages and binding locations.

Page Items

Adding page items in page coordinates

By default, Page items are in spread coordinates. The origin lies at the center point of the page binding. Often, it is easier to work in page coordinates. This can be achieved in IDML by adjusting the page item's ItemTransform matrix. Conceptually, this is a matter of moving the origin from the spread center point to the top-left corner of the page, as follows:

1. Determine the zero-based page binding location, as described above.

2. Determine the zero-based page index.

3. Determine the page width.

4. Determine the page height.

5. Calculate the *x* (or horizontal) translation as follows: xTranslation = (pageIndex - pageBinding) * pageWidth

6. Calculate the *y* (or vertical) translation as follows: yTranslation = pageHeight / 2

7. Add yTranslation.

For example, the following TextFrame is on a page just to the left of the binding; therefore, it is transformed horizontally -612 points and vertically -396 points. This moves the origin from the center of the binding to the left 612 points and up 396 points.

```
<TextFrame ... ItemTransform="1 0 0 1 -612 -396">
   ...
   <PathPointArray>
   <PathPointType Anchor="36 36" LeftDirection="36 36"
      RightDirection="36 36"/>
...
```

This pattern is used in numerous samples. For an example, see the Page Builder's spread XSL file at pagebuilder/xsl/Spreads/spread.xsl.

Adding lines, paths, and shapes

There are many elements that represent paths and shapes. The GraphicLine element represents a straight line; the Oval element, ovals; and the Rectangle element, rectangles. Irregular paths and shapes are represented by the Polygon element. Each of these types contains an array of path points and can be open or closed.

Adding an image

An image is either a child element of one of the shape elements or a Polygon element. The Type attribute of the shape or polygon is set to "GraphicType," and an Image element is added as the last child element of the shape or polygon. This element contains many attributes and child elements, but most importantly, it contains a child element called Link, which describes how to access the actual image data.

Several samples demonstrate working with images. The Page Builder sample demonstrates adding a JPEG image to a document it is building from scratch. The Add Catalog pages sample demonstrates adding several images to a spread. The Java Replace Images sample, ActionScript/Air Replace Images sample, and Flex Air IDML Content Editor demonstrate extracting and changing the image Link element.

Fitting images to frames

IDML does not automatically fit images to frames. There is a frame-fitting feature that is controlled by the FrameFittingOption element that can appear on a page item. This does not automatically fit on IDML import; instead, it controls the behavior the next time an image is placed in the user interface.

The samples do demonstrate a technique for fitting an image to a frame. They fit images using the following steps:

1. The rectangle bounds are set to match the proportions of the image. The rectangle can be any size, but it should be the same proportions as the image.

2. The Image element's ItemTransform attribute is set to "1 0 0 1 0 0" (the identity matrix).

3.   The image bounds are set to match the frame.

Adding text frames

Text frames, whether irregularly shaped or rectangular, always are represented as TextFrame elements. Their ParentStory attribute identifies which story the text belongs to and the PreviousTextFrame and NextTextFrame provide information on linked text frames.

```
<TextFrame Self="ue1" ParentStory="ue3" PreviousTextFrame="ua1"
    NextTextFrame="uf1" ... />
```

To add a text frame:

1.   Add a story XML file to the Stories directory. For an example of a minimal story, see the "Hello World" sample in <u>"What is in an IDML file?"</u>. This story must have a unique ID specified in its Self attribute.

2.   Add an idPkg:Story element for the new story to designmap.xml; for example:

```
<idPkg:Story src="Stories/MyStory.xml"/>
```

3.   Add a TextFrame element to the desired spread file. This element should reference the Self attribute of the new story in its ParentStory attribute. The most difficult thing here is setting the geometry; see the simplified "Hello World" example.

4.   Use the PreviousTextFrame and NextTextFrame attributes to link to or from existing frames. You must change the relevant attributes in frames you are linking to or from.

The text for a text frame is saved in an associated story XML file.

Applying object styles

Object styles are applied by setting the AppliedObjectStyle attribute on the page-item element to the Self ID of the desired Object Style:

```
<TextFrame ... AppliedObjectStyle="ObjectStyle/Object Style 1" ... />
```

Text

Adding text

Text content and formatting is handled in the Story element of a story XML file or snippet. Text content always appears within a Content element. The Replace Story sample demonstrates how to replace an entire story.

Formatting text

Text is formatted using the ParagraphStyleRange and CharacterStyleRange elements. Names notwithstanding, these elements control both the text styles and overrides applied.

The Add Catalog Pages sample is a good example of formatting text.

Updating styles

Styles are saved in Resources/Styles.xml. You can update the styles in a document by updating the elements in this file. The Copy Styles sample demonstrates how to copy styles from one document to another.

Snippets and styles

Snippets contain enough information to reconstruct part of an InDesign document; however, if you place a snippet into a document that has styles with the same name applied, it will not replace those styles. Instead, it will use the styles in the document.

Adding a carriage return

Carriage returns can be added using the Br element; however, this must occur outside a Content element.

Special characters

Some special characters are not supported in XML; therefore, they are encoded using a processing instruction. For example, the page-number character is represented as <?ACE 18?>. There is no reference for these characters, so you will have to experiment with the application if you need to add such a character.

Turning conditional text on and off

Controlling conditional text in IDML is more complicated than doing so in the user interface. You must set the Conditional Text preference and add or remove HiddenText elements in the story content.

To turn a text condition off:

1.   Locate the condition in the designmap.xml file, and set its Visible attribute to "false."

2.   Wrap all occurrences in a HiddenText element. This amounts to wrapping the content inside each CharacterStyleRange that has an AppliedCondition attribute set to the condition that is being turned off.

To turn a text condition on:

1.   Locate the condition in the designmap.xml file and set its Visible attribute to "true".

2.   Locate and remove any HiddenText elements that are part of a CharacterStyle range that contains an AppliedCondition set to the condition being turned on.

The ConditionalText sample demonstrates turning conditions on and off with XSLT.

Tables

InDesign has its own table model, which is reflected in IDML. The table's row (BodyRowCount), column (BodyColumnCount), header row (HeaderRowCount), and footer row (FooterRowCount) counts are specified as attributes in the Table element. The model supports cells that span multiple rows and columns. A cell's position and size is determined by the Name, RowSpan, and ColumnSpan attributes on

the Cell element. The Name will be something like 1:1. This determines the starting point of the cell. The RowSpan and ColumnSpan attributes determine how many rows and columns the cell spans. Finally, the InDesign model allows you to specify table and cell styles.

The ICMLBuilder sample demonstrates converting a simple XHTML table to IDML (ICML).

XML

Adding tags

Tags are defined in the Tags.xml file. To add a tag to a document:

1. Add an XML/Tags.xml file if it does not already exist. The document element is idPkg:Tags. For an example, see below.

2. Add an XMLTag element to the XML/Tags.xml file. The new tag must have a unique Self attribute and name, and it should have a unique color specified.

The following is an example of a Tags.xml file with one tag defined:

```
<idPkg:Tags xmlns:idPkg="http://ns.adobe.com/AdobeInDesign/idml/1.0/packaging"
        DOMVersion="7.0">
    <XMLTag Self="XMLTag/ItemHeading" Name="ItemHeading">
        <Properties>
            <TagColor type="enumeration">Green</TagColor>
        </Properties>
    </XMLTag>
</idPkg:Tags>
```

Mapping tags and styles

To map tags to styles:

1. Add an XML/Mapping.xml file if it does not already exist. The document element should be idPkg:Mapping. For an example, see below.

2. Define each mapping by adding an XMLImportMap element containing a unique Self attribute.

3. Add a MarkupTag attribute set to the Self attribute of the desired style.

4. Add a MappedStyle attribute set to the Self attribute of the desired style.

To map styles to tags:

1. Define each mapping by adding an XMLExportMap element containing a unique Self attribute.

2. Add a MarkupTag attribute set to the Self attribute of the desired style.

3. Add a MappedStyle attribute set to the Self attribute of the desired style.

The following is an example of a simple Mapping.xml file:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<idPkg:Mapping xmlns:idPkg="http://ns.adobe.com/AdobeInDesign/idml/1.0/packaging"
        DOMVersion="7.0">
    <XMLExportMap Self="did5" MarkupTag="XMLTag/Tag2"
        MappedStyle="CharacterStyle/Character Style 1"
        IncludeMasterPageStories="false" IncludePasteboardStories="false"
        IncludeEmptyStories="false"/>
    <XMLImportMap Self="did2" MarkupTag="XMLTag/Tag1"
        MappedStyle="CharacterStyle/Character Style 1"/>
</idPkg:Mapping>
```

Adjusting a document's structure

A document's structure is stored in the XML/BackingStory.xml file and within the story XML files that are marked up. The BackingStory contains top-level associations between content and structure. Text that is marked up within a story is in the individual story files. Detailing this relationship is beyond the scope of this document. The Import XML Template sample demonstrates performing the equivalent of an XML import completely outside InDesign.

Programming with Java and XSLT

XSLT patterns

Most changes that needs to be done to the XML files in an IDML package can be handled with XSLT. (When writing the samples, we originally used some manual DOM manipulations, but we found that XSLT was much more efficient and maintainable.)

Several patterns come into play when generating and manipulating IDML content with XSLT:

▶ *Template* — An XSLT stylesheet contains a combination of static and dynamic content. The style sheet is used as a template and is expanded into some new IDML content. The Page Builder and ICML Builder samples follow this pattern.

▶ *Transform* — An XSLT stylesheet contains instructions for manipulating an existing IDML file. In this case, the input is an IDML file. The style sheet does not contain significant static content; instead, it contains instructions for altering the IDML file. Most of the time, this amounts to an identity transformation, with one or more transformations. Most of the other samples use this pattern; the Conditional Text sample is a good example to consider.

▶ *Hybrid* —Some solutions require both the Templates and Transform patterns. The Add Catalog Pages sample does both. The transform pattern is used to update the designmap.xml file; this is necessary to add new spreads and stories. The Template pattern is then followed to generate new spread and story files.

▶ *Parameters* — Flexible solutions based on XSLT must be written to handle variable input. XSLT provides a facility in the <xsl:param> element for passing data to the style sheet. Most or all of the samples pass parameters to the IDML. This is especially important because information like IDs can change from export to export. A typical pattern is to use Java to discover information about an IDML file, and pass it to stylesheets using parameters. The PackageInspector and XmlUtils classes contain code for discovering information in the IDML file, and the FileTransformer class contains support for specifying parameters and performing transformations.

▶ *Multiple outputs* — Because an IDML file contains many files, it is beneficial to break it down into many different templates. The <xsl:document> element can be used to create multiple output files. This is done by the Page Builder.

▶ *Multiple inputs* —Sometimes it is necessary to use one IDML file as an input for the transformation of another. XSLT facilitates this with the document() function. The Copy Styles sample uses this to copy a source document's styles into the document that is being transformed.

The role of Java

The samples use Java to accomplish those tasks that are not easily achieved in XSLT. Another programming language could be substituted easily. In the IDMLTools samples, Java is used as follows:

▶ Accept command line and user interface input.

▶ Compress and decompress ZIP files (see the Package class).

▶ Manage temporary files and folders (see the FileUtils class).

▶ Capture details of the package, like where to find particular package files (see PackageXmlLocator).

▶ Match XSL and XML files in a package (see PackageXslLocator).

▶ Start XSLT transformations (see the FileTransformer and PackageTransformer classes).

▶ Call the Jing Java Validation Library (see the Validator class).

▶ Discover data in an existing IDML file (see the PackageInspector and XmlUtils classes. For more information, see the next subsection.)

Using Java to discover data

IDMLTools contains two classes for discovering data in an IDML package. The high-level PackageInspector class contains many methods for extracting information from an IDML package. For example, if you need to know the page height, width, or count, you can simply construct a PackageInspector from a path to an IDML file or directory containing an extracted IDML file, and call the GetPageHeight(), GetPageWidth(), or GetPageCount() methods.

```
PackageInspector inspector = new PackageInspector(tempDir);
double pageHeight = inspector.GetPageHeight();
```

The PackageInspector is really just a convenience wrapper to the more powerful XmlUtils class that combines Java and XPATH to extract information from a package. The following are examples of how you can discover data using XmlUtils.

How to get a layer ID from its name

XmlUtils provides a way for you to easily extract attributes from an IDML file. In this case, we know (from the schema) that this data is saved in the designmap.xml file. To get the data from a known XML file, we call the version of XmlUtils.getAttribute that takes one file path and an XPATH statement. The XPATH statement identifies the location of the data; in this case, it is in the Self attribute of a Layer item that is in the Document element. The path qualifies that we are looking for such a Layer whose Name attribute is "text."

To extract the ID for a layer called text, we do the following:

```
XmlUtils.getAttribute(xmlLoc.getDesignMapFilePath(),
    "/Document/Layer[@Name = 'text']/@Self");
```

The xmlLoc (which is an instance of PackageXmlLocator) is used to identify the path to the designmap.xml file in a string.

How to get a master spread ID from its name

XmlUtils also provides a way to find one attribute from a set of XML files. In this case, there is a collection of master-spread XML files. The following call finds the master-spread Self attribute that has a Name attribute equal to "D-catalog."

```
String masterSpreadID = XmlUtils.getAttribute(xmlLoc.getMasterSpreadXmlFiles(),
 "/idPkg:MasterSpread/MasterSpread[@Name = 'D-catalog']/@Self");
```

A PackageException is thrown if the XPATH statement returns anything other than a single attribute.

How to get multiple attributes

XmlUtils also contain methods that return an ArrayList of attributes. These methods are similar to the above, but instead of returning a single string, they return an ArrayList of strings. There are versions that operate on both a single file and an ArrayList of files.

How to get a HashTable that maps names or IDs to values

XmlUtils contains a method called getAttributePairsForElement() that creates a HashTable that maps a unique attribute like Self to another attribute. An example of this can be seen in the implementation of PackageInspector.GetObjectStyleIDNamePairs().

Programming with ActionScript/Air

With ActionScript/Air, it is possible to build self-contained applications that open, modify, and save IDML files, without depending on other technologies. For a scripting or programming language to work with IDML, it needs to provide the capabilities to interact with the file system, compress and decompress files, and modify XML files. All this can be done with ActionScript/Air.

Compressing and decompressing IDML files

ActionScript does not support compression operations natively. To compress and decompress files, we use the third-party library nochump by David Chang, included in the SDK samples. (See the Zip.swc file in the lib folder of any ActionScript/Air sample projects.) The CompressionUtils class found in all ActionScript/Air samples contains functions for interacting with this library.

Using E4X to read and modify XML files

ActionScript 3.0 does not support XSLT. To modify the individual XML files inside an IDML package, we used E4X, a feature in ActionScript 3.0 that allows XML files to be easily read and modified with few lines of code. There are many resources in print and on the Web that thoroughly explore E4X in ActionScript 3.0. The following examples demonstrate how to perform some useful IDML-related tasks with E4X.

Get the Story ID associated with a script label

The following code demonstrates how to look up the storyID of a text frame object containing a script label with the text "Sample Script Label" in an XML file in the Spreads directory. This script assumes that script labels are unique. If not, the first script label with a matching name is returned. Since there can be multiple-spread XML files in the Spreads folder, this code iterates through each file until a matching script label is found.

```
//The relative directory path from the extracted IDML files
//to the Spreads XML files.
public static const SPREADS_DIR:String = "Spreads";
//Iterate through the spread XML files in the Spreads directory.
for each (var xmlFile:File in
idmlDir.resolvePath(SPREADS_DIR).getDirectoryListing())
{
    //Read the XML file into an XML object.
    var xml:XML = readXmlFromFile(xmlFile);

    //Use E4X to find the Story ID associated with the script label
    //"Sample Script Label."
    storyID = xml..TextFrame.(Properties.Label.KeyValuePair.@Value ==
        "Sample Script Label").@ParentStory;

    //Break the for loop if a matching storyID was found.
    if (storyID != "")
    {
        break;
    }
}
```

Update an image

The following line of code demonstrates using E4X to update the LinkResourceURI attribute of a link element, where the descendant Image element has a Self attribute value of Sample Self ID. Updating this attribute with a new path updates the image link in the InDesign document.

```
xml..Image.(@Self == "Sample Self ID").Link.@LinkResourceURI = "New Image Path";
```

Get a master spread ID when given a spread name

E4X provides a way to find one attribute from a set of XML files. In this case, there is a collection of master-spread XML files. The following call finds the master-spread Self attribute that has a Name attribute equal to Sample Master ID.

```
xml..MasterSpread.(@Name == "Sample Master ID").@Self
```

Read an XML file into an XML object

To process an XML document with E4X, the file needs to read into an XML object. This function demonstrates how to do that.

```
private static function readXmlFromFile(xmlFile:File):XML
{
    var fs:FileStream = new FileStream();
    fs.open(xmlFile, FileMode.READ);
    var xml:XML = XML(fs.readUTFBytes(fs.bytesAvailable));
    fs.close();
    return xml;
}
```

Workflow customizations

Round-trip custom data through IDML import and export

Solution: Use the Script Labels feature, which in scripting goes beyond what you can do in the user interface.

You may want to round-trip your own data through IDML import and export. By design, it is impossible to add arbitrary elements and attributes; unrecognized content is consumed and discarded on IDML import. There are, however, two ways to achieve this in Adobe Creative Suite® 5. First, you can write a plug-in that adds your data to the scripting system. Because this requires significant effort, we recommend the second option, using the Script Label feature.

Script Labels allow you to attach an arbitrary block of text to a scriptable object. They are more powerful than what is exposed in the user interface, where you can add one block of text to a scriptable object. Scripting and IDML allow you to add a set of key-value pairs. Only the value corresponding to the Label key is accessible in the user interface.

```
<Label>
    <KeyValuePair Key="FirstLabel" Value="First label."/>
    <KeyValuePair Key="SecondLabel" Value="Second label."/>
    <KeyValuePair Key="ThirdLabel" Value="Third label."/>
    <KeyValuePair Key="Label" Value="This shows up in the UI."/>
</Label>
```

Script Labels are accessible via scripting. For example, the following JavaScripts show how to get the value for the FirstLabel key:

```
app.documents[0].textFrames[0].extractLabel("FirstLabel");
```

This script shows inserting a key-value pair:

```
app.documents[0].textFrames[0].insertLabel(
    "TestLabel", "Second label. Will this make the round trip?"
    );
```

To see an example of how script labels can be used to mark content, look at the Action Script/Air IDML Content Editor sample.

Controlling story, spread, and master-spread filenames on IDML export

Solution: Use scripting and the idmlComponentName property to specify the filename you want to export. Setting this property controls the part of the name before the .xml file extension.

For example, the following code produces a story file called MyStory.xml:

```
myStory.idmlComponentName = "MyStory";
```

Discussion: IDML Export creates filenames for stories and spreads using its own naming convention, which ultimately is based on the object type and UID (Unique ID) in the InDesign database. These names can change on import and re-export, because it is not guaranteed that the objects will have the same UIDs.

If you are working in C++, you can access this data through the IIDMLComponentName interface on the following boss classes: kTextStoryBoss, kSpreadBoss, and kMasterPageBoss. To change the data, use the kSetIDMLComponentNameCmdBoss.

NOTE: If an imported story file is set to a nonstandard name, IDML Import sets the idmlComponentName property.