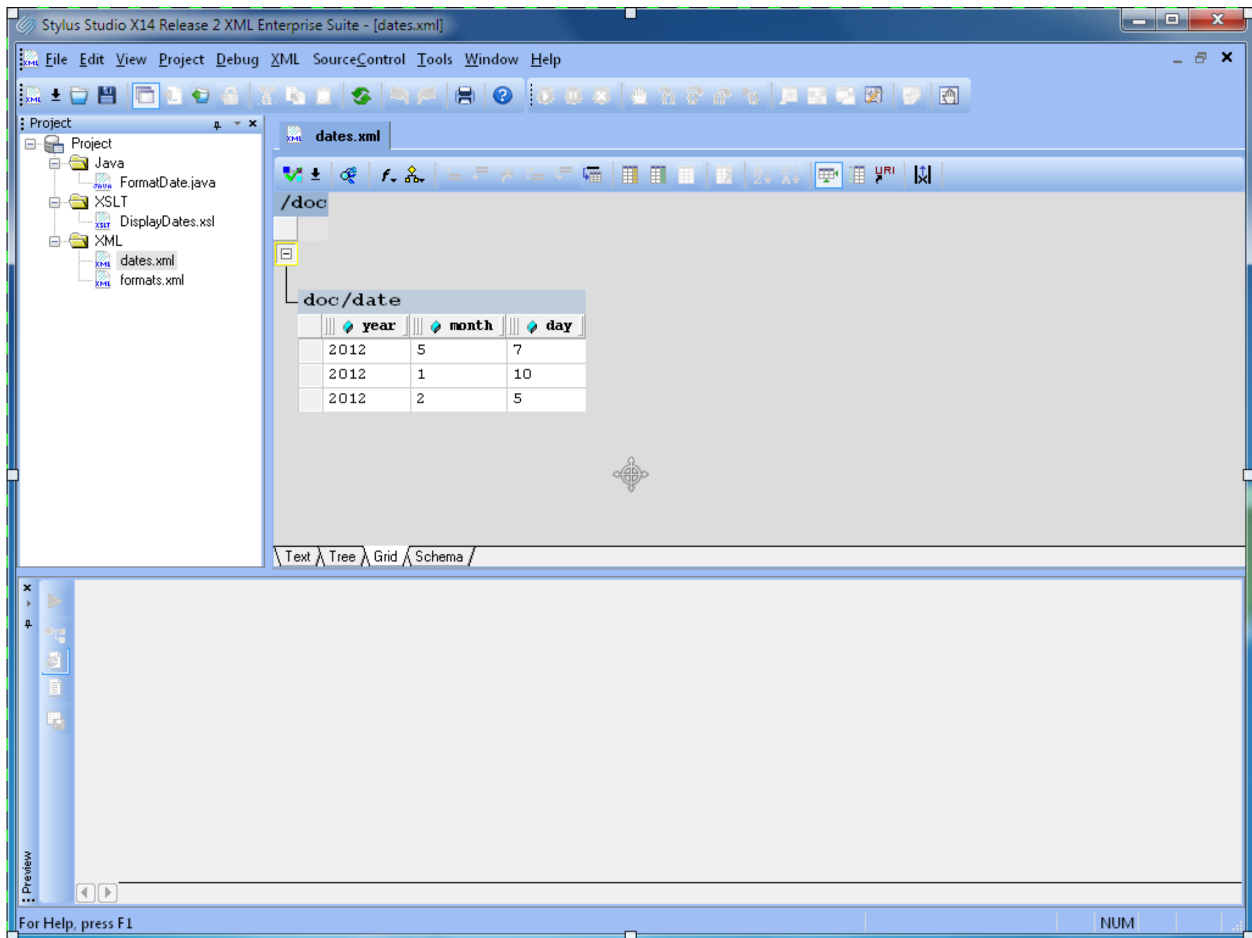
# Extending XSLT with Java and C#

The world is not perfect.  If it were, all data you have to process would be in XML and the only transformation language you would have to learn would XSLT. Because the world is not perfect, sometimes you have to find ways to bridge different systems that were not designed to work together.

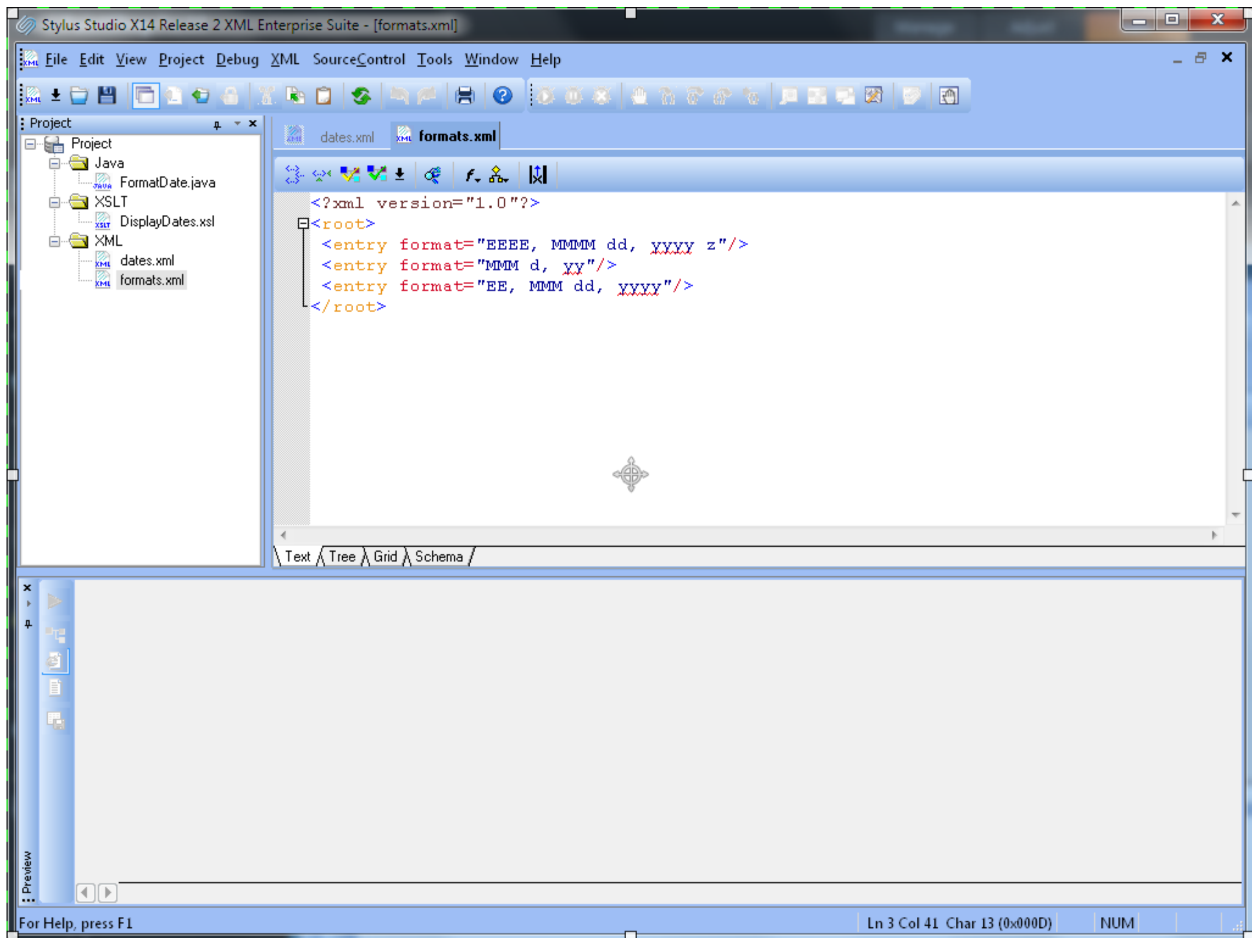
The most popular XSLT processors have been designed, from the very first release, to take advantage of the framework on which they run; for example Apache Xalan-J and Saxon allow calling Java functions.

In this article, we will explore a variety of techniques for invoking native code from XSLT to extend the language beyond its capabilities.

The first example demonstrates how to leverage the Date and Time formatting capabilities available in the Java platform. Imagine you have a list of dates in an XML file that you have to display in a HTML page using different formats. The following screenshot shows a simple XML file with three repeating elements called “date”, each has three attributes: “year”, “month” and “day”.

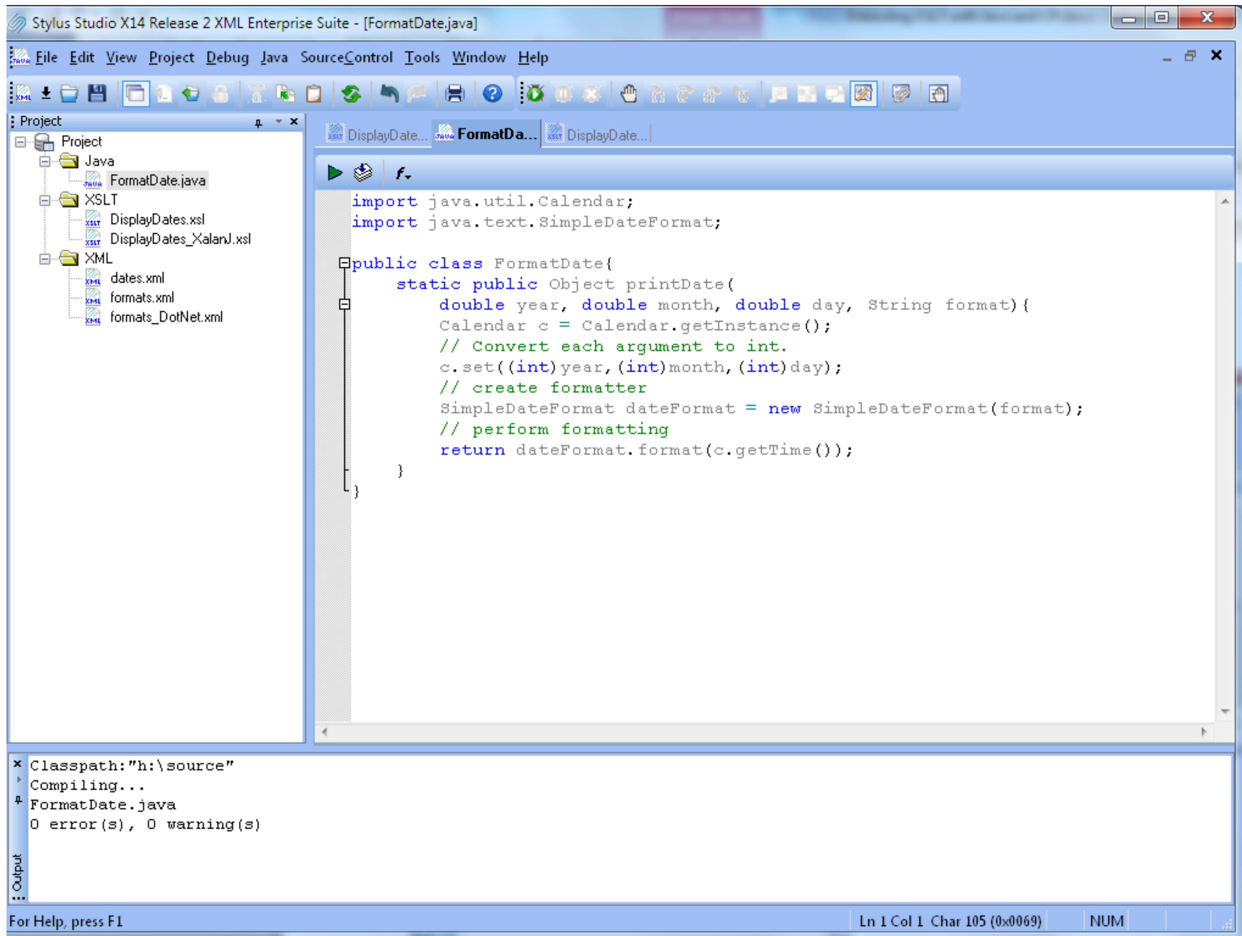



A separate XML document has the date formats. Each element “entry” has an attribute format with the “picture string” which describes how and which part of the date should be displayed.




Our goal is to merge the information from the two XML documents into a simple HTML page which will display each date in multiple formats.

XSLT 1.0 lacks date and time formatting functions but, Java provides two classes: java.util.Calendar and java.text.SimpleDateFormat, which solve our problem. We just need to create a Java class with a single public static method that will be called from our XSLT transformation. In the following screenshot, we see the Stylus Studio Java extension editor which features syntax coloring, background syntax checking and integrated Java compiler invocation.



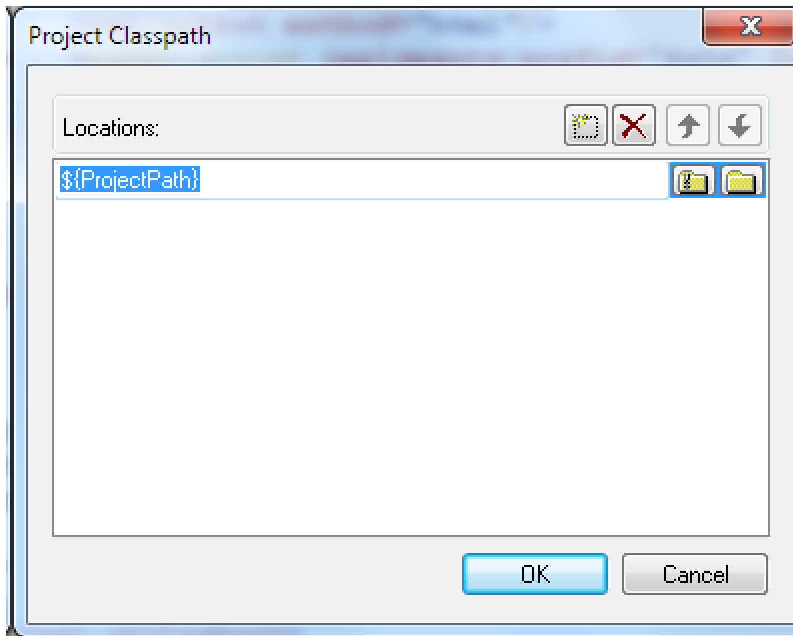

When designing Java extension functions for XSLT, it is important to remember that the function parameter type has to be compatible with the processor type mapping. Apache XalanJ defines the following type mapping between XSLT and Java.

| XSLT Type | Java Type |
|---|---|
| Node-Set | org.w3c.dom.traversal.NodeIterator |
| String | java.lang.String |
| Boolean | java.lang.Boolean |
| Number | java.lang.Double |
| Result Tree Fragment | org.w3c.dom.DocumentFragment |

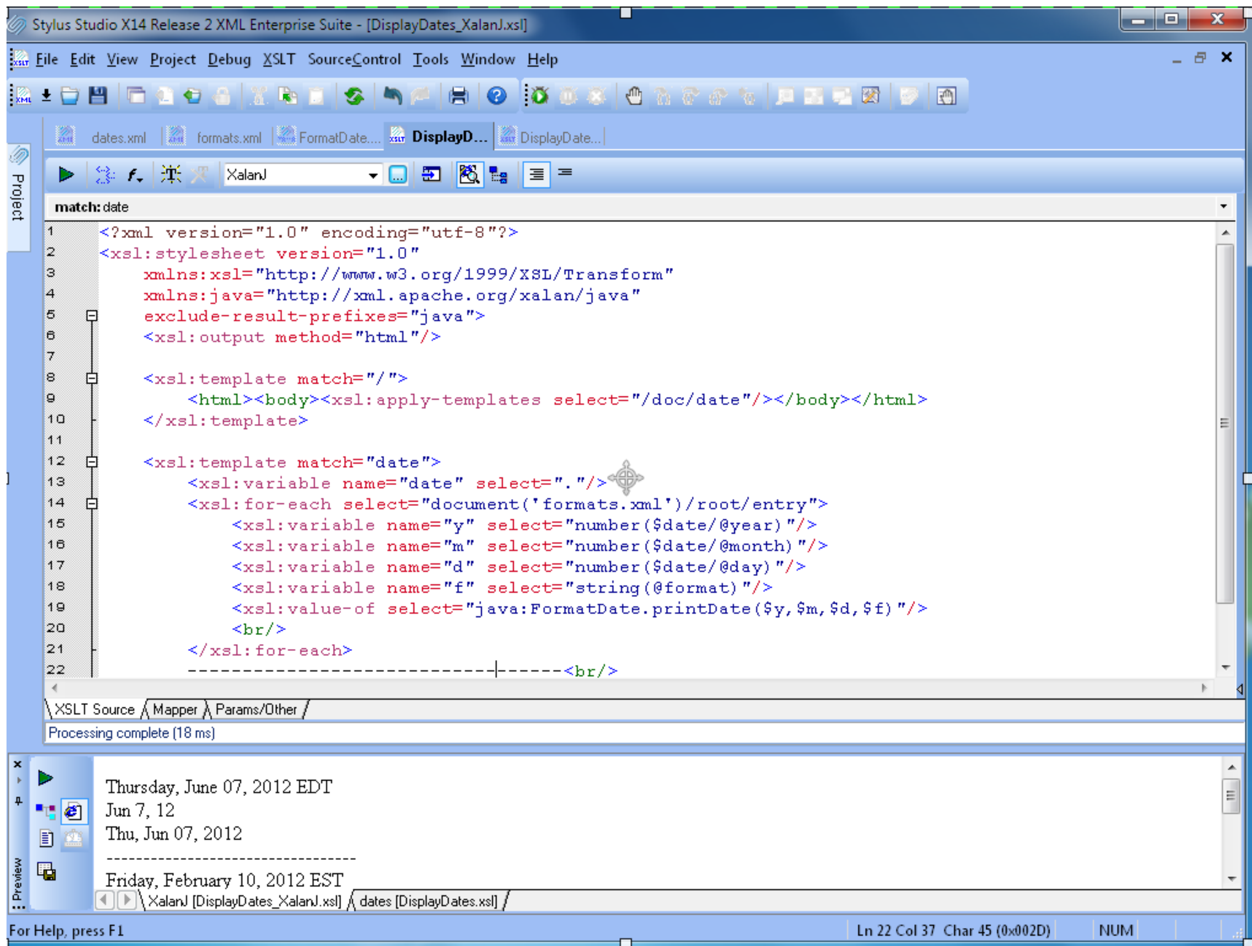
Extension function support is implemented differently on each XSLT processor which makes it difficult to port XSLT code from one processor to another.

In order to run a transformation that makes use of Java code, you have to ensure that the compiled Java code (.class) is reachable from the CLASSPATH. This is a pesky setting which requires changing the

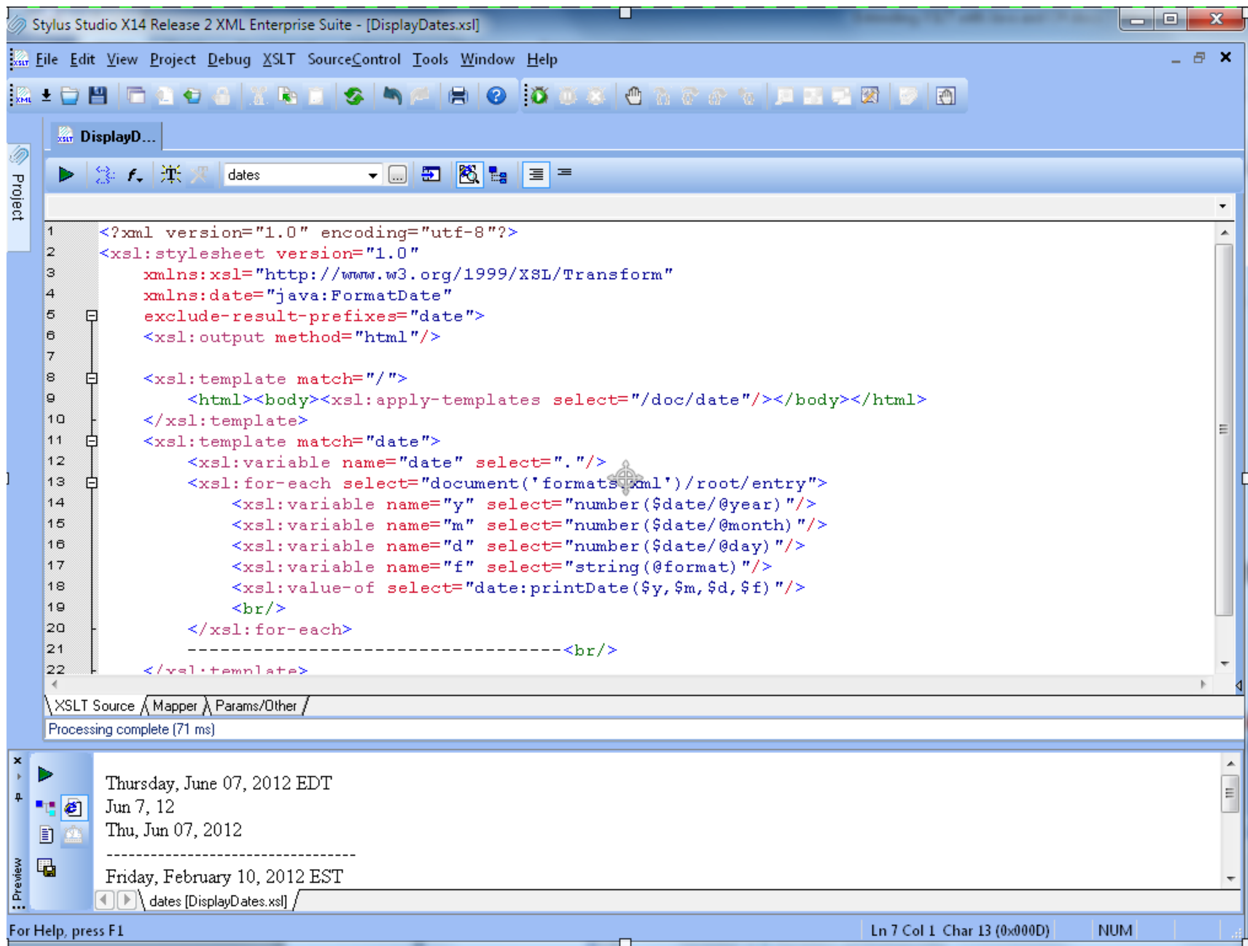
environment variable called CLASSPATH. Fortunately Stylus Studio provides a flexible mechanism to include the Java compiled code (directories or Jar files) at the project level and, if the code is located under the project, Stylus Studio saves the path using a relative form. Therefore, you can move your project to a different location without fear of breaking the link between your XSLT and your Java code.




In the following screenshot, we see how to bind a Java class using XalanJ. The Xalan Java namespace declaration is at line 4, the function invocation is at line 19. Notice that the function name is formed with the prefix java: then the full Java class name the "." and the function name. The Preview shows the transformation result.

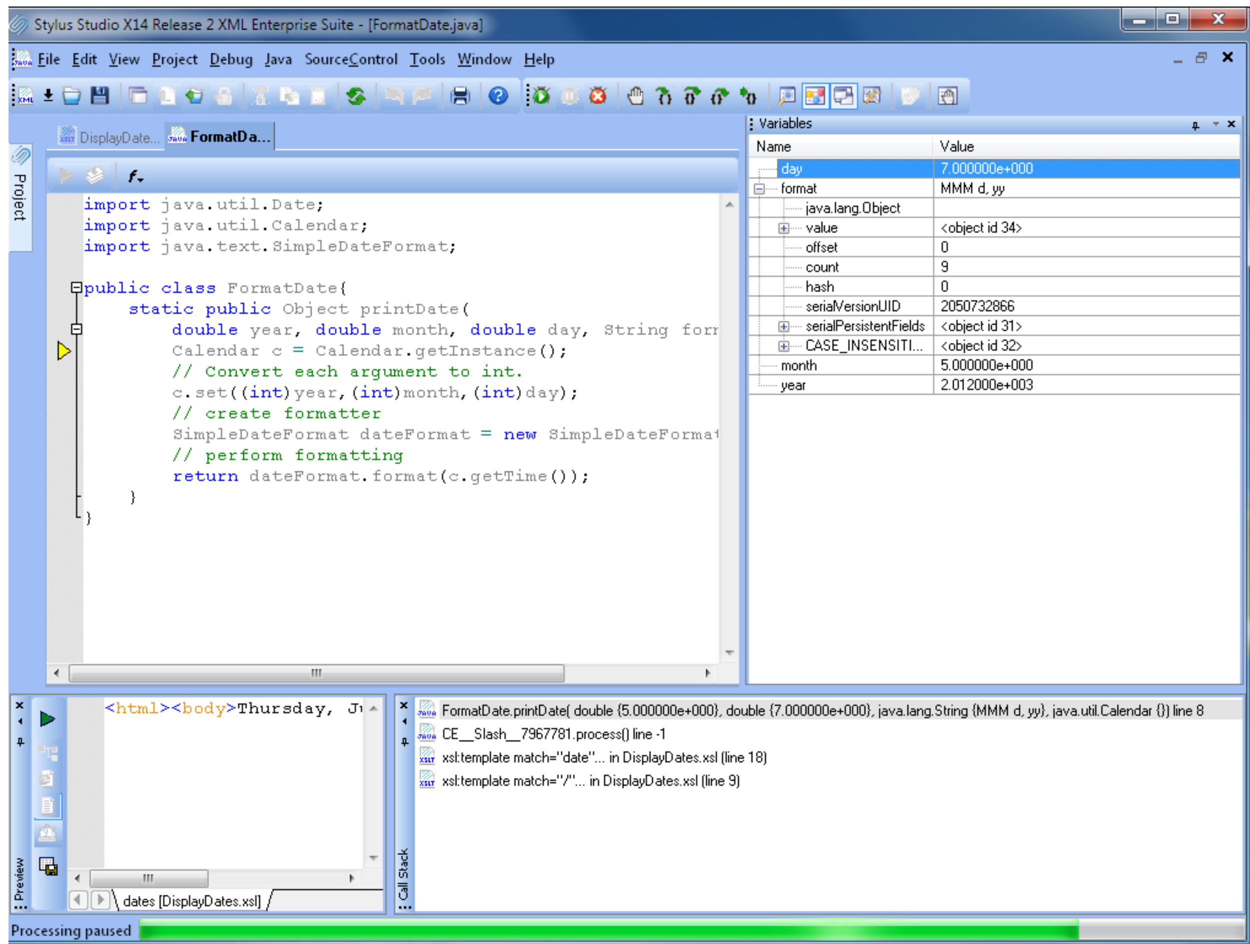



Running the same transformation with Saxon requires a small change. The Java class binding is at line 4; the namespace URI is composed of the prefix “java:” and the full Java class name. The function invocation uses the namespace prefix “date:” and the Java method name. The same result is generated in the Preview window.

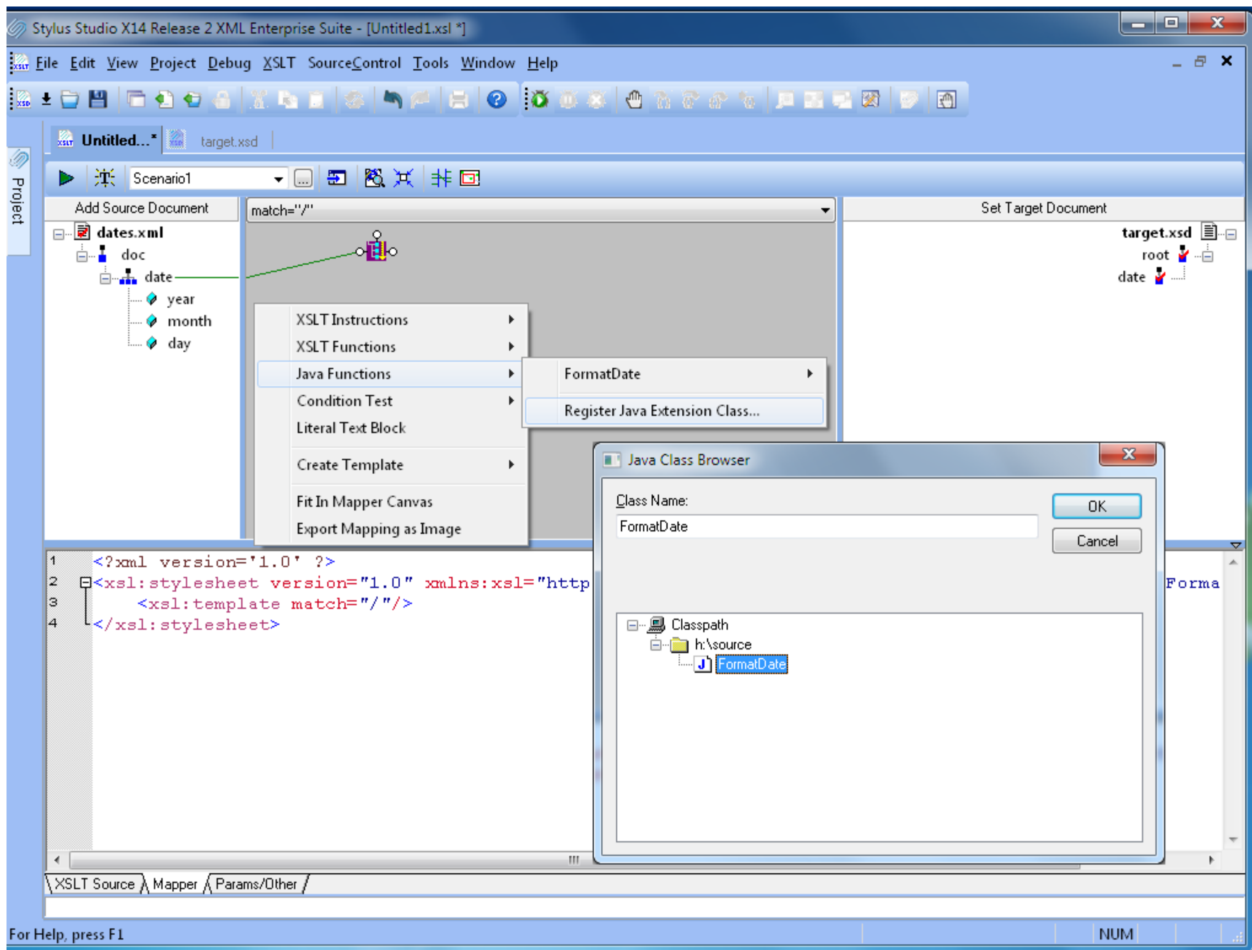



One major advantage in testing the code with Saxon in Stylus Studio is the ability to run the transformation in the XSLT debugger and step into the Java code to debug the Java extension, which is unique to Stylus Studio. In the following screenshot we see the execution suspended inside the extension function “printDate”. The Call-stack window shows from which XSLT template we came in and which parameter values were passed. The Variables window shows all variables in scope with their values. This is an unparalleled experience for the developers who usually have to write hundreds of trace messages in a log file in order to debug their code.

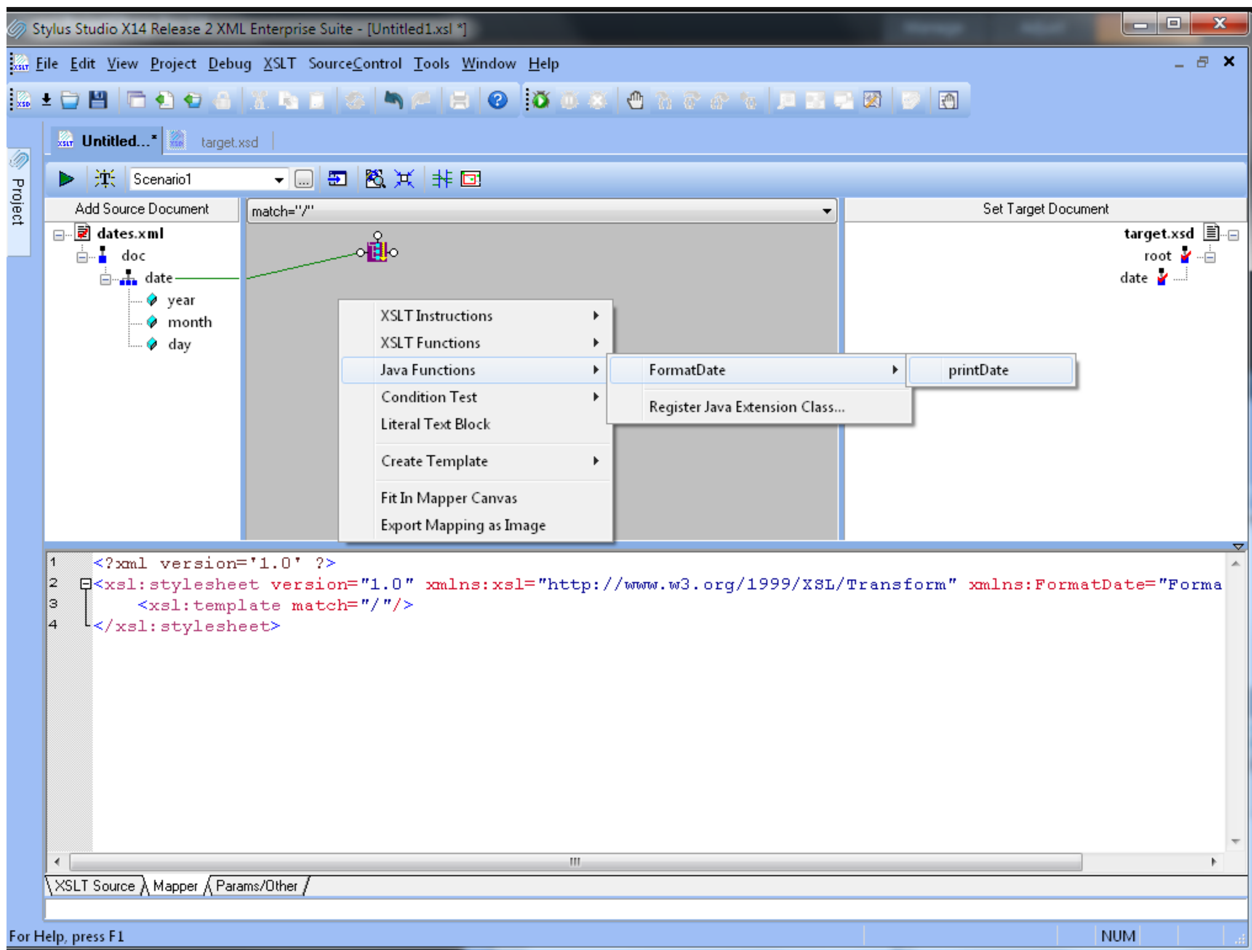
A side note: notice the second item from the top, in the Call-stack window. The new Saxon Just in Time XSLT to Java compiler generates Java code on the fly!



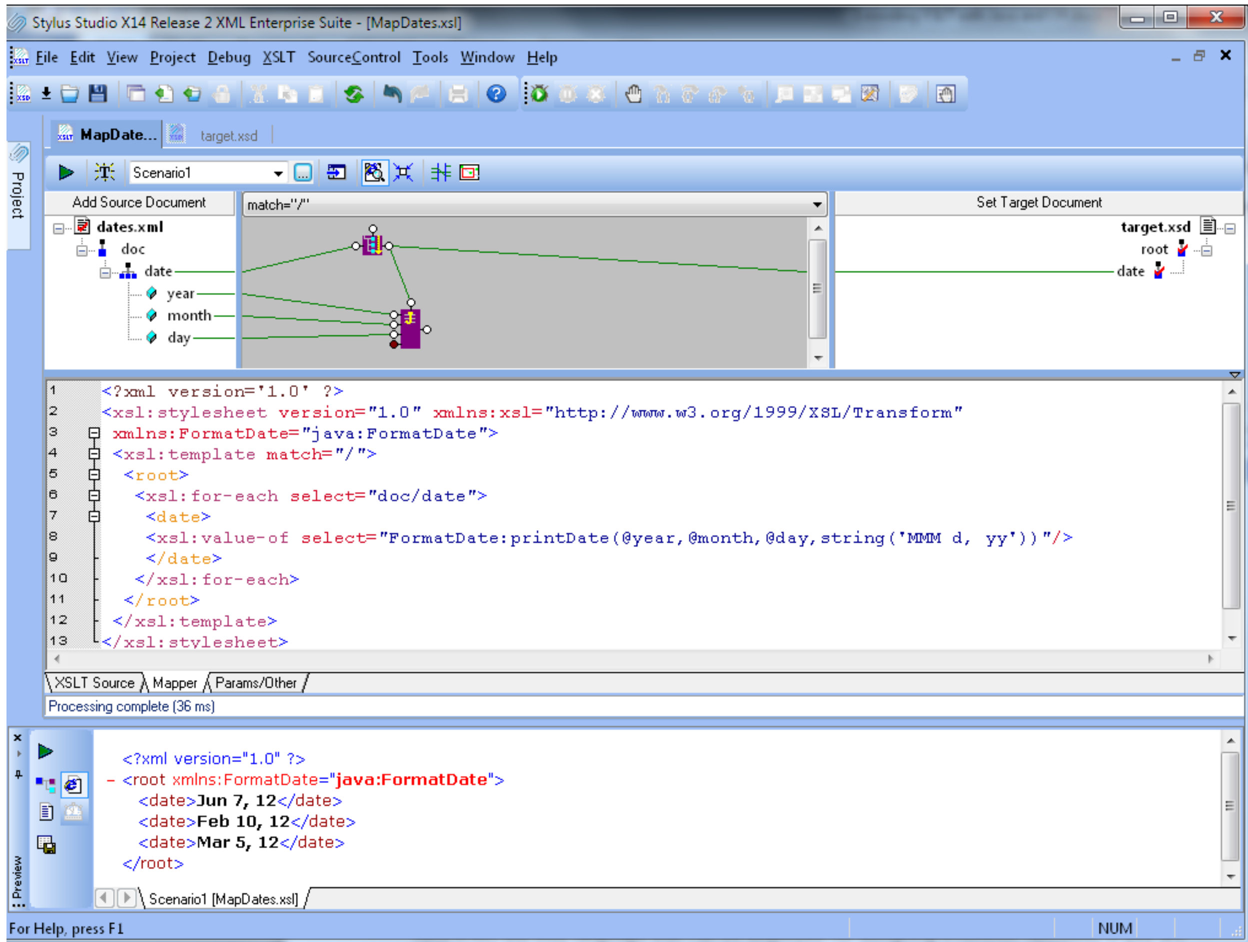

Stylus Studio mapping tool also provides full support for Java extension functions. In the following screenshot you see how to register a Java extension class, browsing the project CLASSPATH.




Once a Java extension class is registered, all of its functions are exposed. To add a new java function call, simply click on the Java Functions menu item.

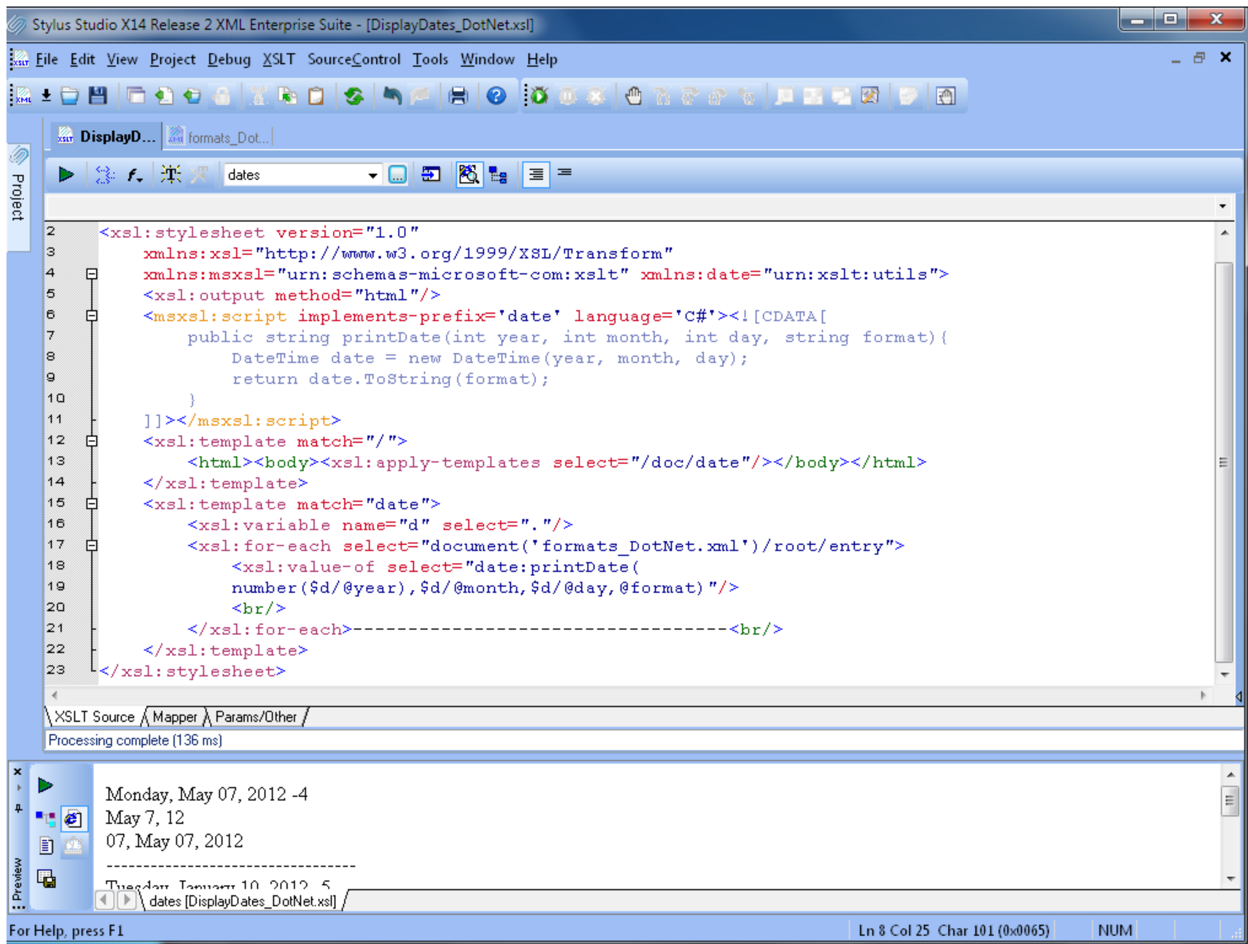



With a few additional links, the mapping is complete. The XSLT visual mapping tool allows XSLT developers to take advantage of Java libraries developed by others without the need to know the underlying technical details.

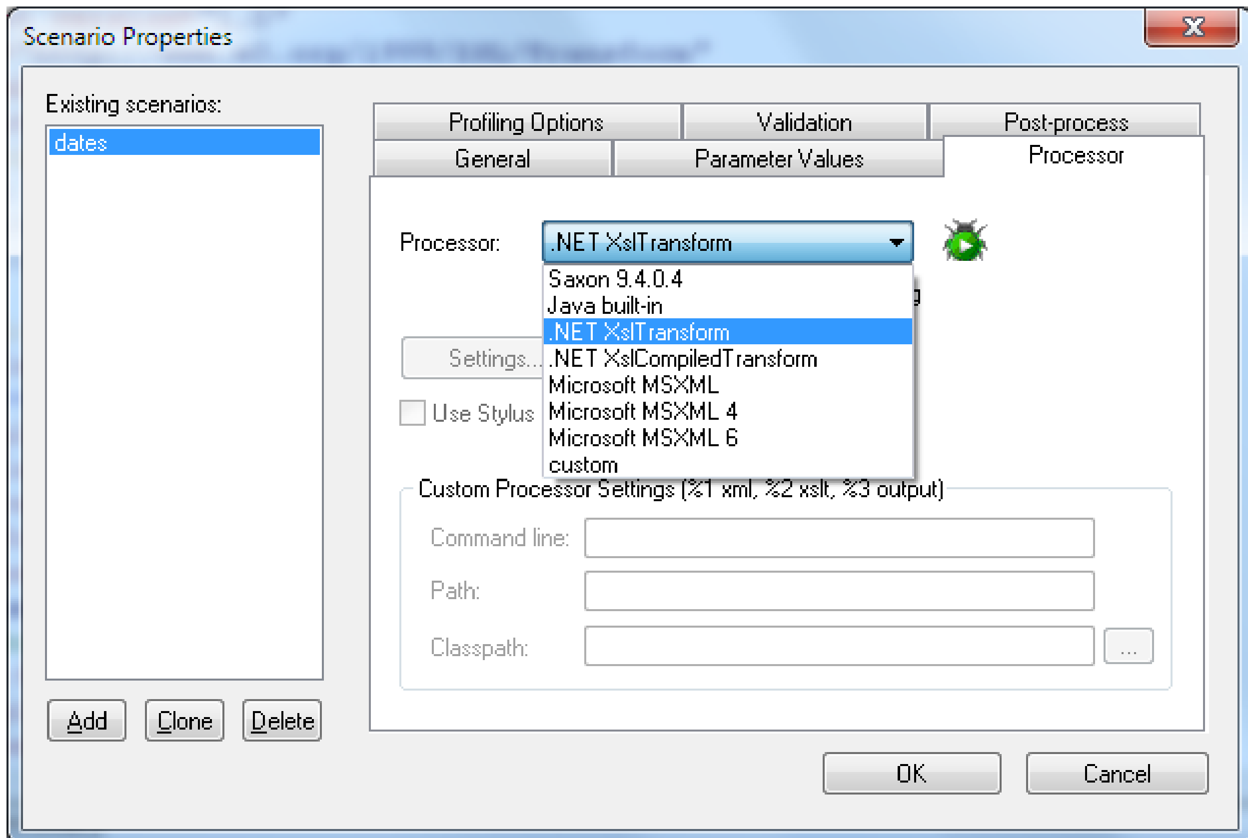



Java is not the only language that can be employed for designing extension functions. If you are developing on Microsoft .NET framework and make use of XslCompiledTransform XSLT processor, you have access to the entire framework API. The following screenshot shows how to implement the date formatter in C# but we could have used JScript as well. The inline code embedded in the XSLT transformation is compiled into MSIL (Microsoft Intermediate Language) by the Just in Time C# compiler.

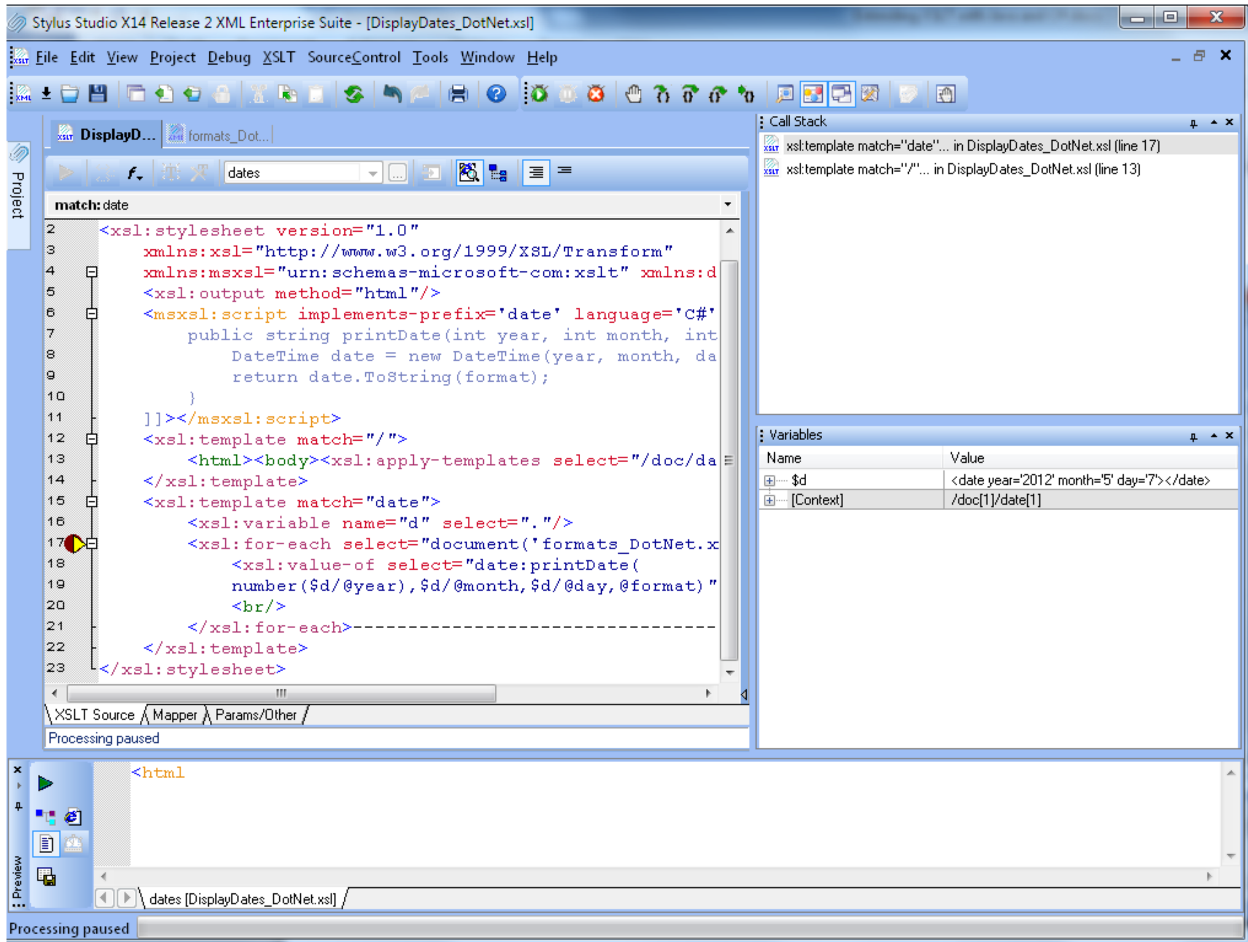
Inline extension functions have several logistic benefits: you don't need to compile a separate module and you don't need to maintain your logic in a different file.



If you need to debug such a transformation, Stylus Studio comes to the rescue Just switch the processor to XslTransform in the XSLT editor scenario dialog and you will be able to debug your code step by step.




In the following screenshot you see the execution suspended inside XSLT match template “date”. The Call-stack window shows the current stack and the variable window shows all variables in scope with their values and XSLT context which represent the XML node currently processed.




The msxsl:script block allows you to import third party .NET libraries which open the door to virtually infinite possibilities. In the following XSLT code fragment, an extension function called “fromEDI” makes use of XML Converters for .NET to parse an EDI file and to return an instance of XPathNavigator which can be manipulated in the XSLT as an XML node.

```
<msxsl:script implements-prefix='ut' language='C#'>
    <msxsl:assembly href="c:\Program Files (x86)\XML Converters for .NET\bin\XmlConverters.dll"/>
    <msxsl:using namespace="DDTek.XmlConverter" />
    <msxsl:using namespace="System.IO" />
<![CDATA[
    public XPathNavigator fromEDI(string ediPath)
    {
        ConverterFactory factory = new ConverterFactory();
        string url = "converter:EDI?" + ediPath;
        XmlReaderSettings settings = new XmlReaderSettings();
        settings.XmlResolver = factory.CreateResolver();
        XmlReader reader = XmlReader.Create(url, settings);
        XPathDocument doc = new XPathDocument(reader);
        return doc.CreateNavigator();
    }
    ]]>
</msxsl:script>
 <xsl:template match="/">
    <xsl:variable name="EDIXML" select="ut:fromEDI($EDI)"/>
```

We hope you enjoyed reading this article. If you have any questions, do not hesitate to contact us.

You can download the Project Zip file by clicking here.

- Stylus Studio Team