WA1787 Designing and Developing
Higher Performance Web Services

Student Labs

Web Age Solutions Inc.

Table of Contents

Lab 1 - Responsiveness, Scalability, and Reliability....3
Lab 2 - Parser Performance....5
Lab 3 - Controlling Element Names Using JAXB....19
Lab 4 - Message Transmission Optimization Mechanism....36
Lab 5 - Selecting Performance Strategies....62
Lab 6 - Performance Testing....65
Lab 7 - REST vs. SOAP....72

Lab 1 - Responsiveness, Scalability, and Reliability

This exercise will be a group discussion regarding the tradeoffs and relative merits of tracking responsiveness, scalability, and reliability of a system or business process.

Part 1 - Form a Team

__1. Form a group of two to four people.

__2. Select a scribe and a presenter from the group.

Part 2 - Discuss the similarities and differences between responsiveness, scalability, and reliability

__1. When you are given a requirement to provide a certain degree of performance for a system, what sort of metrics do you measure?

__2. What relationship, if any, do you see between responsiveness and scalability? Does one trade off with the other?

__3. Is there ever a confusion in terms coming from the business team when they pass down a performance requirement? Do they use "performance" when they really mean either scalability or reliability?

__4. Assuming the previous question is true, what steps could be taken to clarify the process of gathering requirements to avoid this sort of confusion?

Part 3 - Discuss the correlation between these non-functional requirements

__1. Does an improvement in responsiveness necessitate an improvement in scalability? Said another way, is it possible to make something perform well but not scale well? Is the opposite possible (i.e. something could scale, but not perform)?

__2. Does the reliability of a system improve or decline when its responsiveness improves?

__3. Does the reliability of a system improve or decline when it is scaled up? What about when it is scaled out?

Part 4 - Discuss the Results

__1. Have the speaker review your analysis and be prepared to share with the group.

__2. Listen to the other groups describe their findings and make any notes that you find particularly interesting or insightful.

Lab 2 - Parser Performance

Any application dealing with XML must eventually face the issue of *parsing*. To parse f XML [from code] means to take the raw XML data (e.g. characters, whitespaces, etc), read it and turn it into something that the code can make use of.

A developer needing to read (or write) XML could create their own parser; this means, code would have to be written to take a stream containing XML, figure out what to do with the raw characters and make something meaningful out of them. This can be very time-consuming to develop.

Fortunately, a few XML parsers already exist and are available to us – right as a part of the core Java libraries! This means that in order to work with XML, all a developer has to do is choose a parser and use its provided API to handle the low-level raw XML operations.

Parsers come in three main categories, based on how they are engineered. One cateogry is DOM; another is SAX, and the third is STaX. They all perform the same basic operations (i.e. parse an XML file allowing a developer to perform operations like getting an element name and data), but do so in very different manners.

The choice of a parser really depends on a variety of factors; e.g. if code is going to be navigating back and forwards through an XML tree, then DOM would be a better choice than SAX. Alternatively, if memory foot print is a concern, than STaX might be a viable option.

In any application that is XML intensive, the choice of XML parser can have a significant impact on performance. In this lab exercise, we will examine that impact.

Using the SAX, DOM and STaX parser, we will perform a simple XML read test, and see which one returns the best performance.

Note
This lab is purely focused on the performance details of each of the parsers; it is by no means meant to be a lesson on how to code using STaX, DOM or SAX. In each of the code samples below, the code does nothing more than a simple "run" through the XML source files, neglecting any details such as obtaining data, or even creating new XML. This is all we are concerned with for now. For more details on how to actually use these parsers to do actual XML work, consult your instructor or other reference material.

Part 1 - DOM Parser

The DOM parser works by reading the entire XML file into memory first, and then constructing a tree. The tree is then treated as a Java object; code can make use of the DOM tree's API to navigate through the tree, getting (and even creating) data as required.

We will now write some simple Java code that uses the DOM parser to simply read in the XML file. We will not do anything with the parsed XML; we will simply see how long it takes to read in an parse the entire XML file.

We have provided a sample XML file that the application will read. It is located in **C:\LabFiles**. You may open it with a text editor if you wish; the contents are just garbage. It is approximately 10 megabytes in size and so will be a reasonably lengthy parsing job.

First lets start RAD 7.5.

__1. From the **Start** menu, select **All Programs > IBM Software Delivery Platform > IBM Rational Application Developer 7.5 > IBM Rational Application Developer**.

__2. The *Workspace Launcher* window will appear. Accept the default that should be **C:\workspace** and click **OK**. If it is not, then change it.

RAD will launch.

__3. Click **Ignore** in the Warning that will open related with the help content.

__4. Close the *Welcome* view by clicking the **X** button in its tab if it is displayed.

__5. Switch to the Java Perspective by selecting from the menu, **Window > Open Perspective > Java**.

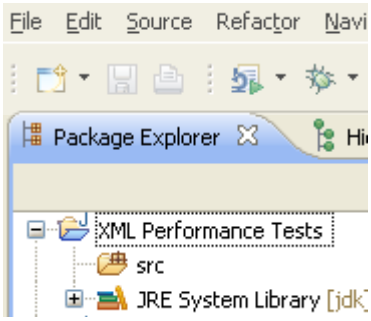
We will need to create a new Java project.

__6. From the menu, select **File > New > Project**...

__7. The *New Project* window will appear. Select **Java Project** and click **Next**.

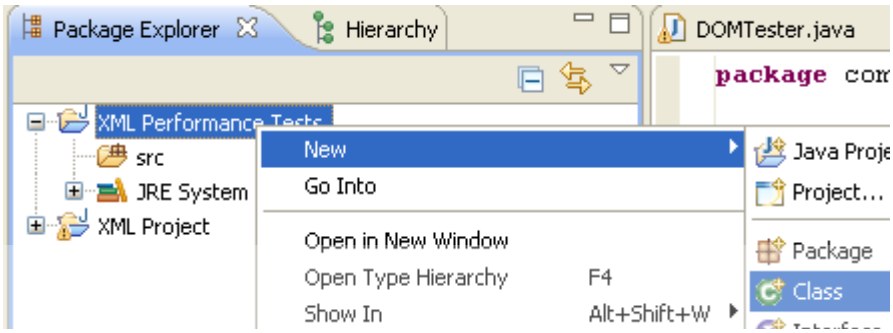
__8. For the *Project name*, enter **XML Performance Tests** and click **Finish**.

The project will be created, and should be shown in the *Package Explorer* view.

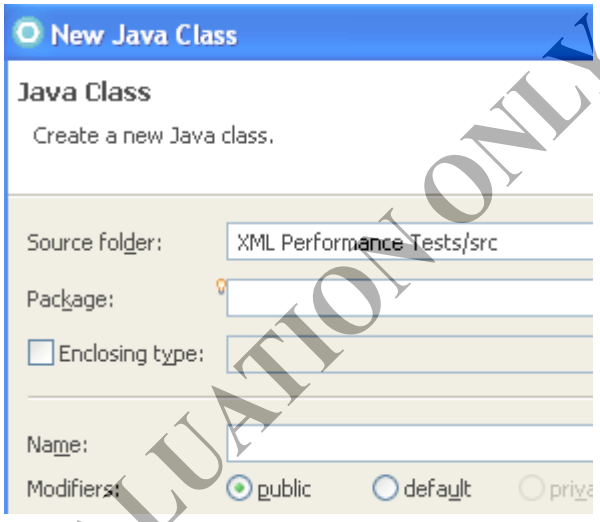


We can now create our DOM parser class.

__9. In the *Package Explorer*, right-click on **XML Performance Tests** and select **New > Class**.



The *New Java Class* wizard will appear.



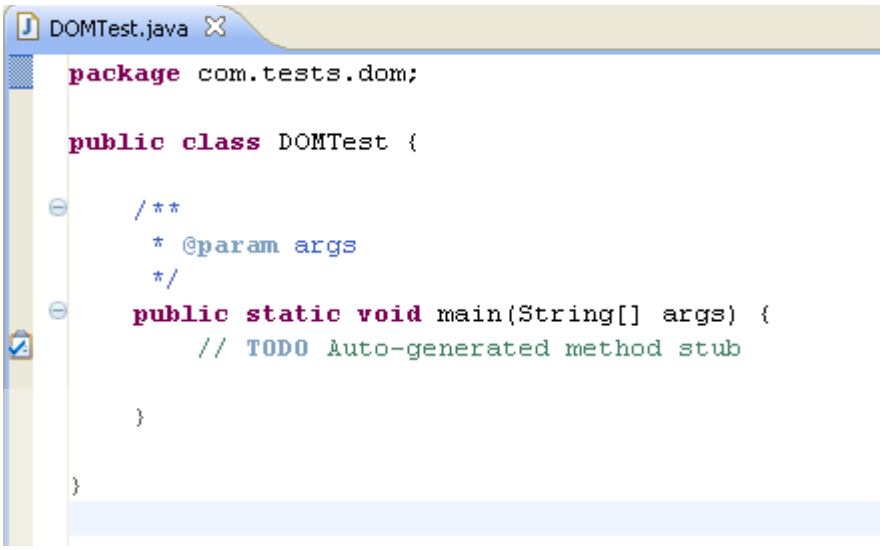
__10. Set the *Package* to be **com.tests.dom**

__11. Set the *Name* to be **DOMTest**

__12. Check the box for **public static void main(String[] args)**

__13. Click **Finish**.

An editor will open on the newly created class.



We can now start writing our code.

__14. In the **main** method, add the following code:

```
String fileName = "C:\\LabFiles\\reallybig.xml";
```

This will set up a reference to XML file that we will read.

__15. Add the following code to the **main** method:

```
DOMParser parser = new DOMParser();
```

This creates an instance of the parser which we will need.

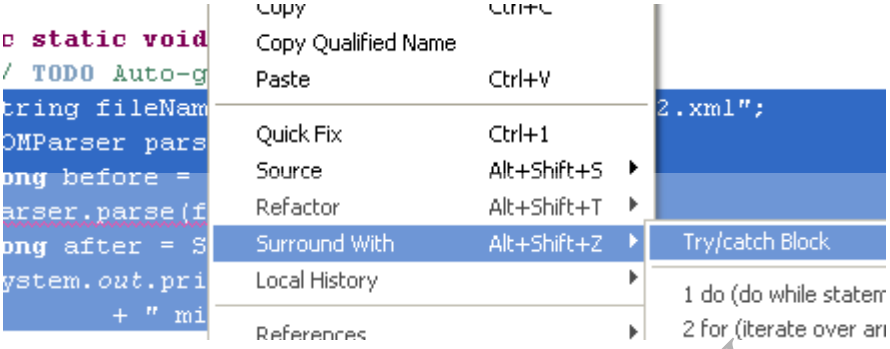
__16. An error will appear. Hit **Ctrl-Shift-O** to *Organize Imports* and the error will go away. Note that this imports **org.apache.xercse.parsers.DOMParser**, which is the DOM parser that ships with the core Java Development Kit (JDK).

__17. Continue adding the following code:

```
long before = System.currentTimeMillis();
parser.parse(fileName);
long after = System.currentTimeMillis();
System.out.println("Using DOM, the time taken was " + (after - before)
                   + " milliseconds");
```

Here we actually perform the parse. We take a timestamp before we perform the actual parse, and then again right after the parse. Finally, we print the difference in timestamp which will show us the elapsed time.

__18. We have an error to fix. Select all the source code within the **main** method. Right-click on the selected source and select **Surround With > Try/catch Block**.



RAD will generate the try/catch code as needed. The errors should go away.

Your source should now look like the following:

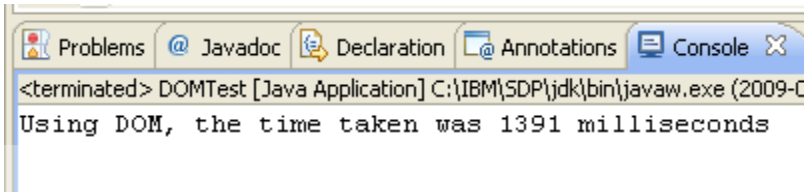
```
public static void main(String[] args) {
    // TODO Auto-generated method stub
    try {
        String fileName = "C:\\LabFiles\\reallybig.xml";
        DOMParser parser = new DOMParser();
        long before = System.currentTimeMillis();
        parser.parse(fileName);
        long after = System.currentTimeMillis();
        System.out.println("Using DOM, the time taken was "
                + (after - before) + " milliseconds");
    } catch (SAXException e) {
        // TODO Auto-generated catch block
        e.printStackTrace();
    } catch (IOException e) {
        // TODO Auto-generated catch block
        e.printStackTrace();
    }
```

There should be no errors.

__19. Save the code (**Ctrl-S**).

__20. Run the code by right-clicking anywhere on the editor and selecting **Run As > Java Application**.

The code will execute, and the *Console* view at the bottom will show you the result.



Specifically, parsing the XML file using the DOM parser has taken this much time.

__21. Run the test 3 more times and record the times for each run below:

Run	Time (ms)
1	
2	
3	

__22. Calculate the average time. (i.e. Add all 3 times and divide by 3) Record it here:

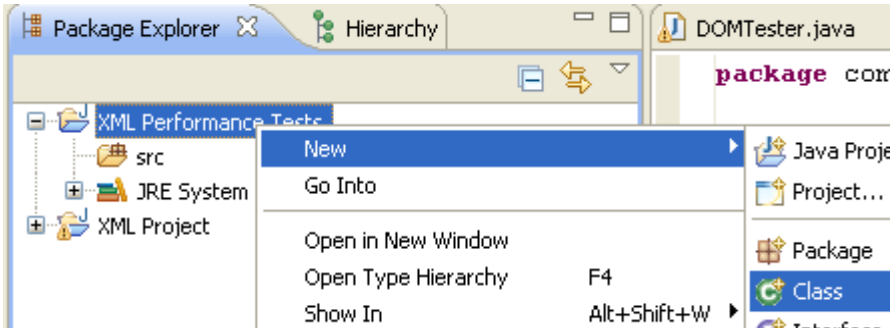
AVERAGE DOM PARSING TIME: _____ ms

We have successfully completed our DOM performance test! We have an idea of how long it takes DOM to parse the large XML file. The next step is to draw a comparison with another parser – the SAX parser.

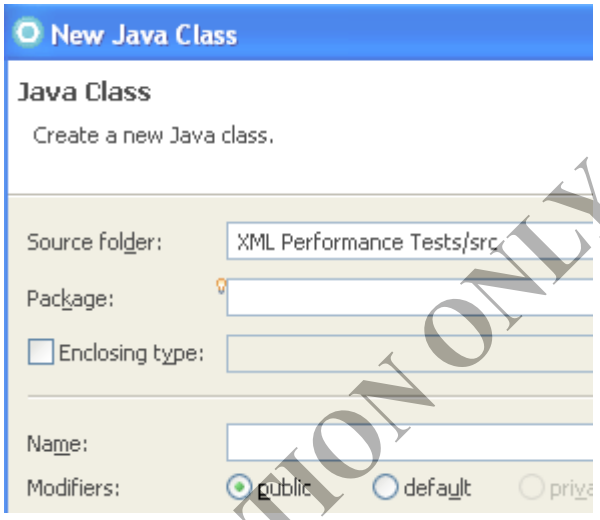
Part 2 - SAX Parser

As before, we will create a simple test class that will read the same large XML file, except that this time we will make use of the SAX parser.

__1. In the *Package Explorer*, right-click on **XML Performance Tests** and select **New > Class**.



The *New Java Class* wizard will appear.



__2. Set the *Package* to be **com.tests.sax**

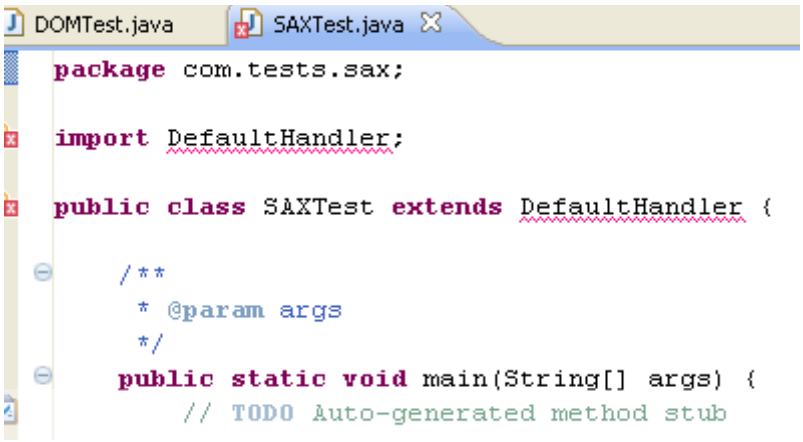
__3. Set the *Name* to be **SAXTest**

__4. In the *Superclass* field, enter **DefaultHandler**

__5. Check the box for **public static void main(String[] args)**

__6. Click **Finish**.

An editor will open on the newly created class.



__7. Errors will immediately be visible. Fix them by performing an *Organize Imports* (**Ctrl-Shift-O**).

The errors will disappear.

__8. Add the following code to the **main** method.

```
String fileName = "C:\\LabFiles\\reallybig.xml";
```

As before, this creates a reference to the XML file to be parsed.

__9. Add the following code to the **main** method:

```
SAXParserFactory factory = SAXParserFactory.newInstance();
```

This obtains an instance of the parsing factory.

__10. Add the following code:

```
SAXParser parser = factory.newSAXParser();
SAXTest sax = new SAXTest();
long before = System.currentTimeMillis();
parser.parse(new File(fileName), sax);
long after = System.currentTimeMillis();
System.out.println("Using SAX, the time taken was " + (after - before)
+ " milliseconds");
```

This creates an instance of a SAX handler (which is conveniently what the **SAXTest** class is) and passes it to the **parse(..)** method.

As before, we take a timestamp before and after the parse call and then print out the difference.

__11. Perform an *Organize Imports* (**Ctrl-Shift-O**). The *Organize Imports* window will appear.

__12. Select **javax.xml.parsers.SAXParser** when prompted and click **Next**.

__13. Select **java.io.File** when prompted and click **Finish**.

__14. Select all the code within the **main** method. Right-click on it and select **Surround With > Try/catch Block**.

The necessary try/catch code will be generated.

Your code should now look like the following.

```
public class SAXTest extends DefaultHandler {

    /**
     * @param args
     */
    public static void main(String[] args) {
        // TODO Auto-generated method stub
        try {
            String fileName = "C:\\LabFiles\\reallybig.xml";
            SAXParserFactory factory = SAXParserFactory.newInstance();
            SAXParser parser = factory.newSAXParser();
            SAXTest sax = new SAXTest();
            long before = System.currentTimeMillis();
            parser.parse(new File(fileName), sax);
            long after = System.currentTimeMillis();
            System.out.println("Using SAX, the time taken was "
                    + (after - before) + " milliseconds");
        } catch (ParserConfigurationException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        } catch (SAXException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        } catch (IOException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }

    }
```

There should be no errors.

__15. Save the code (**Ctrl-S**).

__16. We can now run the code. As you did before with the other class, run this class three times and take the average.

Run	Time (ms)
1	
2	
3	

__17. Calculate the average time. (i.e. Add all 3 times and divide by 3) Record it here:

AVERAGE SAX PARSING TIME: _____ ms

You should see there is a significant difference between SAX and DOM parse times. SAX should be significantly (25% - 40%) faster than DOM.

On top of that, actually working with DOM involves reading the tree into memory first, and then navigating through the tree using DOM API. With SAX, navigation through the tree is done at parse time – which means only one pass through the XML is required.

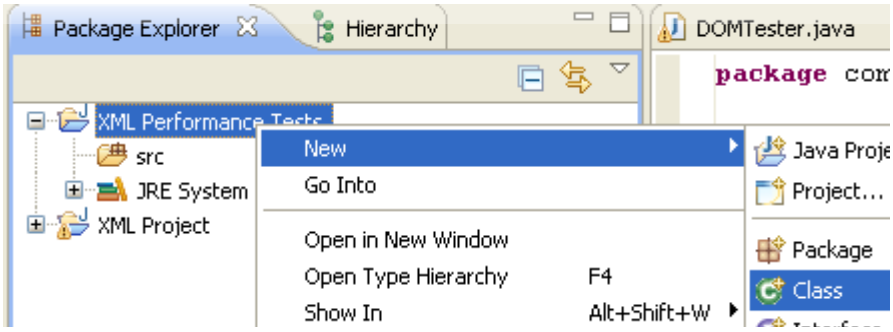
This means that if we wanted to do any actual "real" work with the parsed XML (instead of just reading it as our examples do), the difference between SAX and DOM performance would become much greater, with DOM lagging further behind.

Part 3 - STaX Parser

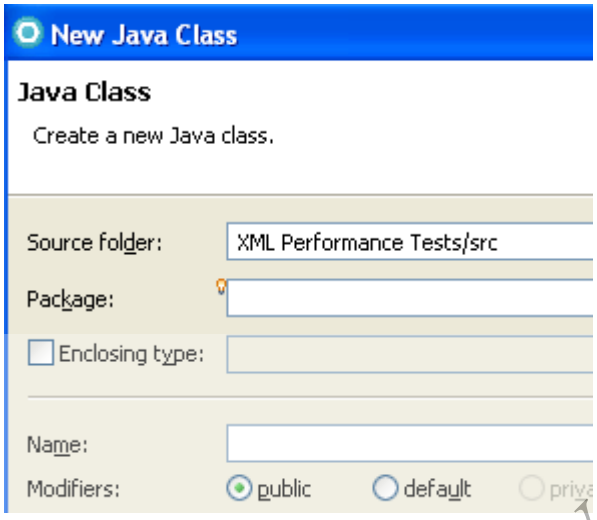
A third type of parser is the STaX parser. STaX is closer to SAX than DOM in that it represents an event-driven linear pass through the tree. However, if operates on a "pull" model, as opposed to the "push" model present in SAX.

As we have done with the two other parsers, we will write code that uses the STaX parser to read the sample XML file.

__1. In the *Package Explorer*, right-click on **XML Performance Tests** and select **New > Class**.



The *New Java Class* wizard will appear.



__2. Set the *Package* to be **com.tests.stax**

__3. Set the *Name* to be **STaXTest**

__4. Check the box for **public static void main(String[] args)**

__5. Click **Finish**.

An editor will open on the class.

```
package com.tests.stax;

public class STaxTest {

    /**
     * @param args
     */
    public static void main(String[] args) {
        // TODO Auto-generated method stub

    }

}
```


__6. Add the following code to the **main** method:

```
FileInputStream fis = new
          FileInputStream("c:\\LabFiles\\reallybig.xml");
XMLEventReader reader = XMLInputFactory.newInstance()
                    .createXMLEventReader(fis);
```

As always, we declare a reference to the file.

Then we create an instance of an **XMLEventReader** – which is the class that will perform the actual STaX parsing.

__7. Organize Imports. (**Ctrl-Shift-O**)

Some errors will remain after doing this; we will fix those later.

__8. Add the following code to the **main** method:

```
long before = System.currentTimeMillis();
while (reader.hasNext()) {
          reader.nextEvent();
          }
long after = System.currentTimeMillis();
System.out.println("Using STaX, the time taken was "
          + (after - before) + " milliseconds");
```

__9. As before, select all the code in the *main* method, right-click on it, and select **Surround With > Try/catch Block**.

The code should now look like the following:

```
public class STaxTest {

    /**
     * @param args
     */
    public static void main(String[] args) {
        // TODO Auto-generated method stub
        try {
            FileInputStream fis = new
                FileInputStream("c:\\LabFiles\\reallybig.xml");
            XMLEventReader reader = XMLInputFactory.newInstance()
                    .createXMLEventReader(fis);
            long before = System.currentTimeMillis();
            while (reader.hasNext()) {
                reader.nextEvent();
            }
            long after = System.currentTimeMillis();
            System.out.println("Using STaX, the time taken was " +
                    (after - before) + " milliseconds");
        } catch (FileNotFoundException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        } catch (XMLStreamException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        } catch (FactoryConfigurationError e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }
    }

}
```


There should be no errors.

__10. Save the file.

__11. Run the code 3 times and fill out the following table:

Run	Time (ms)
1	
2	
3	

__12. Calculate the average time. (i.e. Add all 3 times and divide by 3) Record it here:

AVERAGE STaX PARSING TIME: _____ ms

You should find that STaX performance falls somewhere in between SAX and DOM.

Congratulations! You have successfully used 3 different parsers to examine their raw performance.

__13. Close all open files.

Part 4 - Review

In this lab exercise, you wrote simple code that tested the **DOM**, **SAX** and **STaX** parsers to see how each one handled a raw XML read, performance wise.

You saw that SAX was the fastest, followed by STaX, and trailed by DOM.

Naturally, when it comes to dealing with XML, speed is not necessarily the only factor involved in choosing a parser. Different parsers offer different features, and choices should be made accordingly.

If, however, raw performance is the most important principle, then SAX may be your best choice.