

JXPath Tutorial

A Tutorial on the Use of the Jakarta Commons JXPath Component



Bart van Riel

Capgemini Netherlands - Sector Public - Practice D75 "MOJO"

December 2006

Table of Contents

| | | |
|-----------------------------|--|----|
| 1 | About XPath and this tutorial..... | 5 |
| 2 | XPath Quickref..... | 7 |
| 2.1 | Before you read on..... | 7 |
| 2.2 | XPath Quickref..... | 7 |
| 2.2.a | Nodes..... | 7 |
| 2.2.b | Selectors..... | 7 |
| 2.2.c | Predicates..... | 7 |
| 3 | Some Simple Examples..... | 9 |
| 3.1 | Example code..... | 9 |
| 3.2 | Retrieving a single object..... | 11 |
| 3.2.a | Create the XPathContext with the starting point..... | 11 |
| 3.2.b | Run the XPath query and get the result..... | 11 |
| 3.3 | Iterating over a collection of objects..... | 12 |
| 3.4 | Using a predicate..... | 12 |
| 3.5 | Combing several predicates..... | 13 |
| 3.6 | Using variables in predicates..... | 14 |
| 4 | JXPath Nifties..... | 17 |
| 4.1 | Example code..... | 17 |
| 4.2 | Pointers and Relative Contexts..... | 19 |
| Pointers..... | 19 | |
| Relative context..... | 20 | |
| Selecting parent nodes..... | 20 | |
| Iterating Pointers | 21 | |
| 4.3 | Precompiled Queries..... | 21 |
| 4.4 | Extension Functions | 22 |
| 4.4.a | Extension Functions..... | 22 |
| Single Function..... | 22 | |
| Multiple Functions..... | 22 | |
| 4.4.b | Extension functions as predicates..... | 23 |
| 5 | Resources..... | 25 |
| 5.1 | On the web:..... | 25 |
| 5.2 | Sources and Eclipse project..... | 25 |

1 About XPath and this tutorial

On one of my latest projects, I needed a tool to be able to perform queries on object trees. Since I have been advocating the use of Jakarta Commons components for years, practicing what I preach took me to the Jakarta Commons XPath project. This less known Commons component implements (part of the) W3C XPath XML query language in Java. XPath is a very effective query spec to iterate over data elements, using conditions to narrow down results. In short, just what the doctor ordered.

However, there was a catch. As with many of the less known Commons components, documentation is scarce. In fact, only two descriptions of XPath usage are available on the net. One is the User Guide on the XPath website itself (<http://jakarta.apache.org/commons/xpath/users-guide.htm>), the other is a short tutorial on today.java.net (<http://today.java.net/pub/a/today/2006/08/03/java-object-querying-using-xpath.html>). The relative shortage of documentation did not stop me from implementing XPath in the project though, and usually I write down my experiences with a new component as a tutorial. So, here we are!

Now I have divided this tutorial in roughly four parts:

1. A quick grinder of the most important XPath concepts you need to have a working knowledge of to use XPath;
2. A simple example to illustrate those concepts;
3. Some elaboration on XPath's more advanced capabilities;
4. A more elaborate example putting everything together.

At the end of the story, there is a short summary of all the resources you need to try out things yourself.

Enjoy!

2 XPath Quickref

2.1 Before you read on...

I am not going to give you a complete XPath tutorial here. If you need one, I strongly recommend the excellent tutorial on the w3schools website: <http://www.w3schools.com/xpath/default.asp>. Read it. Now!

2.2 XPath Quickref

Now that you know all about XPath (and if you do not, read the previous paragraph!), I will repeat some of the essentials you need to get working with JXPath. I will explain some more on nodes, selectors and predicates.

2.2.a Nodes

An XML structure is comprised of nodes. A node is one item in the XML. XML usually contains lots of info (name spaces, elements, properties, text data) and all those distinct items in the XML are considered to be nodes. In XML, all nodes can be individually referenced (you can specifically retrieve namespace entries, for example). In Java however, there are only objects with properties and those properties have values.

Namespaces, comments and the like do not really have meaning in a Java object tree, so we will not have to consider them here. In fact, in JXPath we only use *element nodes* (being the properties of objects and the objects themselves) and *attribute nodes* (as an alternative way of using object properties). *Child nodes* in JXPath are the same as element nodes in that they are regular properties, with the only difference that these properties are not primitive types but references to other objects ('child objects').

One extra note on the 'root node': JXPath does not really have one. XML has the concept of a 'document root' which is merely a delimiter of the XML data structure and does not really hold any data. The root object of a Java object tree however, is a fully operational class instance which can have all kinds of data and operations associated with it. JXPath does have the notion of a 'root' but we will see in some later examples exactly how this works.

2.2.b Selectors

Selectors are expressions with which you can select a certain node or a group of nodes. The w3schools tutorial talks about 'expressions', but since you use them to select stuff 'selectors' just seems a bit more natural to me. JXPath supports all the selectors defined ('.', '..', '/', '// and even '@'). Note however, that since there is no real root node in a Java object tree, the starting point of a selector (like '/myObject') may return something different than you might expect.

2.2.c Predicates

Predicates are condition clauses, used to limit the results for a XPath query (very much like SQL 'Where' clauses do). In JXPath you use a predicate (or several predicates) to extract a specific object instance from the tree or to get a collection of similar objects.

3 Some Simple Examples

3.1 Example code

So now we have laid some elementary groundwork for XPath, let us do some simple examples. We will be using a rudimentary set of classes to demonstrate getting collections of objects, single objects and on using predicates. I have come up with two classes for this: `SimpleValueHolder` to hold some values and `SimpleValueLister` to hold a list of `SimpleValueHolder`'s.

Here is the code for `SimpleValueHolder`:

```
package test;
import java.util.Map;
import org.apache.commons.beanutils.BeanUtils;

public class SimpleValueHolder {
    private String someValue = "yes";
    private String someOtherValue = "red";
    private int seq = 0;

    public SimpleValueHolder(int seq){
        this.seq = seq;
    }
    public SimpleValueHolder(){
    }
    public String getSomeValue() {
        return someValue;
    }
    public void setSomeValue(String someValue) {
        this.someValue = someValue;
    }
    public int getSeq() {
        return seq;
    }
    public void setSeq(int seq) {
        this.seq = seq;
    }
    public String getSomeOtherValue() {
        return someOtherValue;
    }
    public void setSomeOtherValue(String someOtherValue) {
        this.someOtherValue = someOtherValue;
    }
    public String toString(){
        String myInternals = "";
        try{
            Map internals = BeanUtils.describe(this);
            myInternals = internals.toString();
        } catch(Exception e){
            e.printStackTrace();
        }
        return myInternals;
    }
}
```

And here it is for SimpleValueListner:

```
package test;
import java.util.ArrayList;
import java.util.List;
import java.util.Map;
import org.apache.commons.beanutils.BeanUtils;

public class SimpleValueListner {
    private List valueList = new ArrayList();
    private String checkValue = "Johnny";

    public void addValue(Object o){
        this.valueList.add(o);
    }
    public List getValueList() {
        return valueList;
    }
    public void setValueList(List valueList) {
        this.valueList = valueList;
    }
    public String getCheckValue() {
        return checkValue;
    }
    public void setCheckValue(String checkValue) {
        this.checkValue = checkValue;
    }
    public String toString(){
        String myInternals = "";
        try{
            Map internals = BeanUtils.describe(this);
            myInternals = internals.toString();
        } catch(Exception e){
            e.printStackTrace();
        }
        return myInternals;
    }
}
```

And of course we need some 'main' code to test our classes, so here that one is:

```
package test;
import org.apache.commons.jxpath.JXPathContext;
public class JXPathTester {
    public static void main(String[] args) {
        //Create the ValueHolder lister object
        SimpleValueListner holderList = new SimpleValueListner();
        for(int j = 0; j < 5; j++){
            SimpleValueHolder holder = new SimpleValueHolder();
            //Every other instance gets different values
            if((j%2)==0) {
                holder.setSomeValue("no");
                holder.setSomeOtherValue("blue");
            }
            if((j%4)==0) {
                //one instance gets totally different values
                holder.setSomeOtherValue("green");
            }
            holderList.addValue(holder);
        }
        //Create a 'object tree root object'
        JXPathContext context = JXPathContext.newContext(/*What to put here?*/);
    }
}
```

3.2 Retrieving a single object

Let us start with the simplest of queries: retrieve one object. The process is pretty straightforward:

1. Create the `JXPathContext` with the starting point;
2. Run the XPath query and get the result.

3.2.a Create the `JXPathContext` with the starting point

You may have noticed this line in the source:

```
JXPathContext context = JXPathContext.newContext(/*What to put here?*/);
```

Remember what I told you about XPath and 'root elements'? There is something very important about XPath you need to know: *you always need a root object*. The reason for this is the way XPath works: an XML document always has a root element, the so called 'document root'. XPath queries are always interpreted relative to the document root, so when querying Java objects we also need that kind of a starting point.

So we need to do the following: we supply an instance of some holder object to the `JXPathContext` instance. When getting a single element, simply passing our `SimpleValueLister` instance will do nicely:

```
JXPathContext context = JXPathContext.newContext(holderList);
```

3.2.b Run the XPath query and get the result

Now that we have the context initialized, we can start executing some XPath queries. The simplest query is, of course, retrieving the context root object. The query will be:

```
Object o = context.getValue(".");
```

The XPath selector `'.'` will retrieve the current node, which should be our `SimpleValueLister` instance. Let us put some tracing in the code to validate this, so that our main class will be:

```
package test;
import org.apache.commons.jxpath.JXPathContext;
public class JXPathTester {
    public static void main(String[] args) {
        //Create the ValueHolder lister object
        SimpleValueLister holderList = new SimpleValueLister();
        for(int j = 0; j < 5; j++){
            SimpleValueHolder holder = new SimpleValueHolder();
            //Every other instance gets different values
            if((j%2)==0) {
                holder.setSomeValue("no");
                holder.setSomeOtherValue("blue");
            }
            if((j%4)==0) {
                //one instance gets totally different values
                holder.setSomeOtherValue("green");
            }
            holderList.addValue(holder);
        }
        //Create a 'object tree root object'
        JXPathContext context =
            JXPathContext.newContext(holderList);
        Object o = context.getValue(".");
        System.out.println(((SimpleValueLister)o).getCheckValue());
    }
}
```

Which outputs when running (as we would expect):

Johnny

If we want to retrieve the list of `SimpleValueHolder` objects in the `holderList`, we expand the XPath query to retrieve the list encapsulated by `SimpleValueList`, through its `valueList` property:

```
List list = (List)context.getValue("./valueList");
```

Of course, simple getting the `List` itself is not very useful, you probably want the objects in the list. So the next paragraphs deal with iterating over object collections and using predicates.

3.3 Iterating over a collection of objects

Just as easily as fetching a single object, we can iterate over a collection of objects, by calling the `iterate()` method. Now `JXPath` is intelligent when it comes to handling collections: if the result of the query is a single object, it will be automatically wrapped in a one-item `Iterator`. If the result is a `Collection`, the `Iterator` returned will be an iterator over that collection.

With that in mind, the following code:

```
JXPathContext context = JXPathContext.newContext(holderList);
for(Iterator iter = context.iterate("."); iter.hasNext();){
    System.out.println(((SimpleValueList)iter.next()).getCheckValue());
}
```

will output the value of the only item in the resulting `Iterator`:

Johnny

Whereas the following code:

```
JXPathContext context = JXPathContext.newContext(holderList);
for(Iterator iter = context.iterate("/valueList"); iter.hasNext();){
    System.out.println(((SimpleValueHolder)iter.next()).getSomeValue());
}
```

will output the values of the `SimpleValueHolders` in the list:

```
no
yes
no
yes
no
```

Note that in the second code snippet, I omitted the `'.'` before `'/valueList'`. In `JXPath`, both selectors can be considered identical. Since in `JXPath` the root object is a selectable node in itself, `'.'` (“the current element”) and `'/'` (“the first element starting from the root”) return the same object. This is different from standard XPath over XML.

3.4 Using a predicate

Remember that a predicate is like an SQL WHERE clause? We can use predicates to refine an XPath query to only return elements which conform to the predicate value.

Let us look at an example. If we want to retrieve only the SimpleValueHolder objects which have the value 'blue', the code would look something like:

```
JXPathContext context = JXPathContext.newContext(holderList);
for(Iterator iter = context.iterate("/valueList[someValue='no']"); iter.hasNext();){
    System.out.println(((SimpleValueHolder)iter.next()).getSomeOtherValue());
}
```

All SimpleValueHolders with the property someValue set to 'no' have their someOtherValue property set to 'blue' or 'green', and presto:

```
green
blue
green
```

Predicates support AND like definitions, so the following is possible:

```
JXPathContext context = JXPathContext.newContext(holderList);
for(Iterator iter = context.iterate
    ("/valueList[someValue='no' and someOtherValue='green']"); iter.hasNext();){
    System.out.println(((SimpleValueHolder)iter.next()).getSomeOtherValue());
}
```

Which outputs:

```
green
green
```

Predicates can also be used as a nifty shorthand retrieve an indexed object in a list:

```
for(Iterator iter = context.iterate("/valueList[2]"); iter.hasNext();){
    System.out.println(((SimpleValueHolder)iter.next()).getSomeOtherValue());
}
```

Outputs:

```
red
```

Note that in XPath, collections are *one-based* and not zero-based, like in Java. Therefore, the lowest index in a list is always [1]. In the code snippets above, the indexes would run from [1] to [5].

3.5 Combing several predicates

Predicates are obviously very useful, but so far we have only seen single predicates. XPath allows for multiple predicates in one query, so we can make selections from multilevel object trees, something like: "From a list of lists, select the 3rd list and from that list, select the objects where property X has the value 'AB'".

To illustrate this, let us make the code example a bit more advanced. We define 10 SimpleValueLister objects and put them in yet another list, in this case an array. The main class will look like this (adjustments highlighted):

```
public class JXPathTester {
    public static void main(String[] args) {
        SimpleValueLister[] listerArr = new SimpleValueLister[10];
        for(int i = 0; i < 10; i++){
            //Create the ValueHolder lister object
            SimpleValueLister holderList = new SimpleValueLister();
            //Every other holderList gets value "Billy"
            if(i%2 == 0) holderList.setCheckValue("Billy");
            for(int j = 0; j < 5; j++){
                SimpleValueHolder holder = new SimpleValueHolder();
                //Every other instance gets different values
                if((j%2)==0) {
                    holder.setSomeValue("no");
                    holder.setSomeOtherValue("blue");
                }
            }
            listerArr[i] = holderList;
        }
    }
}
```

```
        }
        if((j%4)==0) {
            //one instance gets totally different values
            holder.setSomeOtherValue("green");
        }
        holderList.addValue(holder);
    }
    listerArr[i] = holderList;
}
//Create a 'object tree root object'
XPathContext context = XPathContext.newContext(listerArr);
//queries here....
}
```

Now suppose we do the following query:

```
XPathContext context = XPathContext.newContext(listerArr);
for(Iterator iter =
    context.iterate(". [checkValue='Billy']/valueList[someValue='no']");
    iter.hasNext();){
System.out.println(((SimpleValueHolder)iter.next()).getSomeOtherValue());
}
```

This outputs (put down the dots there for brevity):

```
green
blue
....
blue
green
```

Notice that the query starts with '.' in stead of '/' or './'. If you put one of the latter two in there, the query will fail with an exception. Why? Remember that the selector '.' means 'the current node'. Also remember that XPath handles lists intelligently, iterating automatically when it encounters a Collection of some sort. The consequence of this is that, when fed with a Collection as the starting point, the XPathContext will always iterate over this starting point. So in this case, '.' selects each element in the array, not the array itself!

3.6 Using variables in predicates

So far we have only seen 'hard coded' XPath queries, in the sense that the predicate values are literals in the query String. In XPath you can define variables, which you can then assign any value to. For example, the code to get only SimpleValueHolders with the value 'green' would be as follows:

```
XPathContext context = XPathContext.newContext(listerArr);
context.getVariables().declareVariable("searchVal", "green");
for(Iterator iter = context.iterate("/valueList[someOtherValue=$searchVal]");
    iter.hasNext();){
    System.out.println(((SimpleValueHolder)iter.next()).getSomeOtherValue());
}
```

This outputs (put down the dots there for brevity again):

```
green
green
...
green
green
```

Using variables can be extremely useful if you want to reuse the same search routing for different combinations of search parameters. For example, the combined-predicate query from earlier could be put in a generic method and then called at will, like this:

```
//...
Iterator iter = performQuery(listerArr, "Billy", "no");
while(iter != null && iter.hasNext()){
    System.out.println(((SimpleValueHolder)iter.next()).getSomeOtherValue());
}
//...

private static Iterator performQuery(Object objectTree, String searchParam1, String
searchParam2){
    JXPathContext context = JXPathContext.newContext(objectTree);
    context.getVariables().declareVariable("searchVal1", searchParam1);
    context.getVariables().declareVariable("searchVal2", searchParam2);
    return context.iterate(".[checkValue=$searchVal1]/valueList[someValue=$searchVal2]");
}
```

Which, when executed, results in the same output as earlier (put down the dots there for brevity):

```
green
blue
....
blue
green
```


4 XPath Nifties

Now we have covered the basics of XPath it is time to take a look at some of the more advanced features. We will get to Pointers and relative contexts, precompiled queries and custom extension functions. But first, to illustrate those advanced features, we need a slightly more elaborate example to work with.

4.1 Example code

We will be using the highly original concept of a company with departments and employees. Check out the code below:

Company.java

```
package test;

import java.util.List;
import java.util.Map;
import org.apache.commons.beanutils.BeanUtils;

public class Company {
    private String name;
    private List departmentList;

    public Company(String name){
        this.name = name;
    }
    public List getDepartmentList() {
        return departmentList;
    }
    public void setDepartmentList(List departmentList) {
        this.departmentList = departmentList;
    }
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    public String toString(){
        String myInternals = "";
        try{
            Map internals = BeanUtils.describe(this);
            myInternals = internals.toString();
        } catch(Exception e){
            e.printStackTrace();
        }
        return myInternals;
    }
}
```

Department.java

```
package test;

import java.util.List;
import java.util.Map;
import org.apache.commons.beanutils.BeanUtils;

public class Department {
    private List employees;
    private String name;
```

```

    public Department(String name){
        this.name = name;
    }
    public List getEmployees() {
        return employees;
    }
    public void setEmployees(List employees) {
        this.employees = employees;
    }
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    public String toString(){
        String myInternals = "";
        try{
            Map internals = BeanUtils.describe(this);
            myInternals = internals.toString();
        } catch(Exception e){
            e.printStackTrace();
        }
        return myInternals;
    }
}

```

Employee.java

```

package test;

import java.util.Map;
import org.apache.commons.beanutils.BeanUtils;

public class Employee {
    private String name;
    private String jobTitle;
    private int age;

    public Employee(String name, String jobTitle, int age){
        this.name = name;
        this.jobTitle = jobTitle;
        this.age = age;
    }
    public int getAge() {
        return age;
    }
    public void setAge(int age) {
        this.age = age;
    }
    public String getJobTitle() {
        return jobTitle;
    }
    public void setJobTitle(String jobTitle) {
        this.jobTitle = jobTitle;
    }
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    public String toString(){
        String myInternals = "";
        try{
            Map internals = BeanUtils.describe(this);

```

```

        myInternals = internals.toString();
    } catch (Exception e) {
        e.printStackTrace();
    }
    return myInternals;
}
}

```

JXPathCompanyTester.java (Main class)

```

package test;

import java.util.ArrayList;
import java.util.List;
import org.apache.commons.jxpath.JXPathContext;

public class JXPathCompanyTester {
    public static void main(String[] args) {
        Company company = new Company("Acme Inc.");
        List departments = new ArrayList();
        departments.add(new Department("Sales"));
        departments.add(new Department("Accounting"));

        List employees = new ArrayList();
        employees.add(new Employee("Johnny", "Sales rep", 45));
        employees.add(new Employee("Sarah", "Sales rep", 33));
        employees.add(new Employee("Magda", "Office assistant", 27));
        ((Department)departments.get(0)).setEmployees(employees);

        employees = new ArrayList();
        employees.add(new Employee("Steve", "Head Controller", 51));
        employees.add(new Employee("Peter", "Assistant Controller", 31));
        employees.add(new Employee("Susan", "Office assistant", 27));
        ((Department)departments.get(1)).setEmployees(employees);

        company.setDepartmentList(departments);

        JXPathContext context = JXPathContext.newContext(company);
        Object o = context.getValue(".");
        System.out.println(o);
    }
}

```

When you run the Main class, the output should be something like:

```

{class=class test.Company, departmentList=[{employees=[{age=45, jobTitle=Sales rep, class=class test.Employee, name=Johnny}, {age=33, jobTitle=Sales rep, class=class test.Employee, name=Sarah}, {age=27, jobTitle=Office assistant, class=class test.Employee, name=Magda}], class=class test.Department, name=Sales}, {employees=[{age=51, jobTitle=Head Controller, class=class test.Employee, name=Steve}, {age=31, jobTitle=Assistant Controller, class=class test.Employee, name=Peter}, {age=27, jobTitle=Office assistant, class=class test.Employee, name=Susan}], class=class test.Department, name=Accounting}], name=Acme Inc.}

```

A.k.a. the fully nested contents of “Acme Inc”.

4.2 Pointers and Relative Contexts

Pointers

A *Pointer* in JXPath is an object describing a specific node in the object tree. In effect, by holding

information of the exact location of an object in the tree (“the 'adress' property of the second 'employee' of the third 'department' in this 'company' root object”), a Pointer is very much optimized to reach objects directly. If you need to work on a specific object in the tree, retrieve a Pointer to it and work from there.

Pointers are particularly useful when using a *relative context*, which is a JXPathContext where the starting point is a node somewhere halfway into the object tree. Use a relative context when you need to perform multiple queries on a part of the object tree starting somewhere in the middle (e.g. queries on individual employees but not the departments they belong to and not the whole company). It is time for an example.

Using Pointers involves the following steps:

1. Create the JXPathContext in the usual way;
2. Use the `getPointer()` or `iteratePointers()` method to retrieve Pointers;
3. Create a new JXPathContext, using the Pointer as the starting point;
4. Perform XPath queries on this new *relative context* in the usual way.

The following snippet illustrates step 1 and 2. It shows you what location in the object tree the Pointer points to and how you can retrieve the object through the Pointer.

```
JXPathContext context = JXPathContext.newContext(company);
Pointer p = context.getPointer("./departmentList[name='Accounting']/employees");
System.out.println("Pointer location: " + p);
System.out.println("Object at Pointer location: " + p.getValue());
```

Which outputs:

```
Pointer location: /departmentList[2]/employees
Object at Pointer location: [{age=51, jobTitle=Head Controller, class=class
test.Employee, name=Steve}, {age=31, jobTitle=Assistant Controller, class=class
test.Employee, name=Peter}, {age=27, jobTitle=Office assistant, class=class
test.Employee, name=Susan}]
```

So the Pointer points directly to the list of employees of the second department. The `getValue()` method returns the actual List instance encapsulated in the `employees` attribute of the department.

Relative context

We now have direct access to the list of employees. This enables us to perform queries on the employees without walking the full object tree every time. To set a Pointer as the starting point, we can use the *relative context*, which is essentially a 'sub'-JXPathContext linked to the context we started with:

```
Pointer p = context.getPointer("./departmentList[name='Accounting']/employees");
//...
JXPathContext relativeContext = context.getRelativeContext(p);
System.out.println(relativeContext.getValue(". [name=' Steve' ]"));
```

Which should output the result from “select from the current root element (which is the List) the object with the name property set to the value 'Steve’”. And it does:

```
{age=51, jobTitle=Head Controller, class=class test.Employee, name=Steve}
```

Selecting parent nodes

Aside from faster and more scoped access to the object tree, there is another advantage in using Pointers: they allow you to walk the tree backwards by selecting parent nodes. From the w3schools tutorial you may remember that the XPath selector for a parent node is `'.. /'`. The following code snippet illustrates its use:

```

Pointer employeePointer = relativeContext.getPointer(".[name='Steve']");
System.out.println("Pointer location: " + employeePointer);
System.out.println("Object at Pointer location: " + employeePointer.getValue());
JXPathContext employeeContext = relativeContext.getRelativeContext(employeePointer);
System.out.println(employeeContext.getValue("../name")); //department name
System.out.println(employeeContext.getValue("../..name")); //company name

```

First we create a new relative context, setting the starting point to employee 'Steve'. From there we can reach all enclosing nodes by using the parent node XPath selectors. Check out the output:

```

Pointer location: /departmentList[2]/employees[1]
Object at Pointer location: {age=51, jobTitle=Head Controller, class=class
test.Employee, name=Steve}
Accounting
Acme Inc.

```

Iterating Pointers

An even more useful use of Pointers is to *iterate* them. Suppose we want a list of all employees and their associated departments and companies. The intuitive way would be to retrieve a collection of all employees and to query each employee for his or her department and company. Pointer iterations combined with relative context enables us to do just that!

```

JXPathContext context = JXPathContext.newContext(company);
for(Iterator iter = context.iteratePointers("/departmentList/employees");
iter.hasNext();){
    Pointer ptr = (Pointer)iter.next();
    System.out.println(ptr.getValue()); //employee object
    JXPathContext employeeContext = context.getRelativeContext(ptr);
    System.out.println(employeeContext.getValue("../name")); //department name
    System.out.println(employeeContext.getValue("../..name")); //company name
}

```

Which outputs (dots added for brevity):

```

{age=45, jobTitle=Sales rep, class=class test.Employee, name=Johnny}
Sales
Acme Inc.
{age=33, jobTitle=Sales rep, class=class test.Employee, name=Sarah}
Sales
Acme Inc.
...
{age=27, jobTitle=Office assistant, class=class test.Employee, name=Susan}
Accounting
Acme Inc.

```

4.3 Precompiled Queries

You may have wondered how much overhead is associated with all this expression interpretation and evaluation. The answer is that although JXPath performs some expression caching in the JXPathContext, it is sometimes feasible to be able to precompile frequently used queries for speed. This is actually quite simple to do: first precompile a query and then perform it, passing in the applicable JXPathContext instance:

```

CompiledExpression allEmps = JXPathContext.compile("/departmentList/employees");
JXPathContext context = JXPathContext.newContext(company);
for(Iterator iter = allEmps.iteratePointers(context); iter.hasNext();){
    Pointer ptr = (Pointer)iter.next();
    System.out.println(ptr.getValue()); //employee object
}

```

Which outputs all employees:

```
{age=45, jobTitle=Sales rep, class=class test.Employee, name=Johnny}
{age=33, jobTitle=Sales rep, class=class test.Employee, name=Sarah}
{age=27, jobTitle=Office assistant, class=class test.Employee, name=Magda}
{age=51, jobTitle=Head Controller, class=class test.Employee, name=Steve}
{age=31, jobTitle=Assistant Controller, class=class test.Employee, name=Peter}
{age=27, jobTitle=Office assistant, class=class test.Employee, name=Susan}
```

Use precompiled queries whenever you need to need to perform the same query many times.

4.4 Extension Functions

4.4.a Extension Functions

Single Function

JXPath allows you to link classes to the `JXPathContext` and call their methods in an XPath like manner. This functionality is called *Extension Functions* and to use it follow these steps:

1. Write a class with methods;
2. Register the class with the context using the `ClassFunctions` wrapper class, linking it to a prefix;
3. Call the method using the prefix.

The following snippets illustrate this:

```
public class CustomDateFormatter {
    public static String formatDate(java.util.Date date, String pattern){
        return new SimpleDateFormat(pattern).format(date);
    }
}
```

Main class:

```
context.setFunctions(new ClassFunctions(CustomDateFormatter.class, "dateformat"));
context.getVariables().declareVariable("date", new Date());
String formatted =
    (String)context.getValue("dateformat:formatDate($date, 'dd/MM/yyyy')");
System.out.println(formatted);
```

Which outputs:

03/11/2006

Multiple Functions

You register the utility class with a prefix, which provides scoped access to the utility methods. This scoping is useful when you want to register not one, but several classes with the context. To register more than one class, you wrap the `ClassFunctions` instances in a `FunctionLibrary` instance which you then register with the context. Example:

```
public class CustomNumberFormatter {
    public static String asPercentage(double number){
        return NumberFormat.getPercentInstance().format(number);
    }
}
```

```

ClassFunctions dateFunctions =
    new ClassFunctions(CustomDateFormatter.class, "dateformat");
ClassFunctions numberFunctions =
    new ClassFunctions(CustomNumberFormatter.class, "numberformat");
FunctionLibrary funcLib = new FunctionLibrary();
funcLib.addFunctions(dateFunctions);
funcLib.addFunctions(numberFunctions);
XPathContext context = XPathContext.newContext(company);
context.setFunctions(funcLib);

context.getVariables().declareVariable("date", new Date());
String formattedDate =
    (String)context.getValue("dateformat:formatDate($date, 'dd/MM/yyyy')");
System.out.println(formattedDate);

context.getVariables().declareVariable("number", new Double(34.5669));
String formattedNumber = (String)context.getValue("numberformat:asPercentage($number)");
System.out.println(formattedNumber);

```

Which outputs:

03/11/2006

3.457%

4.4.b Extension functions as predicates

Apart from calling extension functions for explicit tasks, XPath allows you to write extension functions to use as predicates. Suppose we want to fire all employees over 40 years old. You could write an XPath query to extract only those employees (with something like '[age > 40]' in it). But that is no fun at all. You could also write a function like so:

```

public class CompanyReorganizer {
    public static boolean isFirable(ExpressionContext context) {
        Pointer pointer = context.getContextNodePointer();
        if (pointer == null) {
            return false;
        } else if (pointer.getValue() instanceof Employee) {
            if (((Employee)pointer.getValue()).getAge() > 40) {
                return true; //older than 40? You're fired!
            }
        }
        return false;
    }
}

```

And call it like this:

```

XPathContext context = XPathContext.newContext(company);
ClassFunctions dateFunctions =
    new ClassFunctions(CompanyReorganizer.class, "reorganize");
context.setFunctions(dateFunctions);
for(Iterator iter = context.iterate("//.[reorganize:isFirable()]"); iter.hasNext();){
    System.out.println(iter.next());
}

```

Which outputs:

{age=45, jobTitle=Sales rep, class=class test.Employee, name=Johnny}

{age=51, jobTitle=Head Controller, class=class test.Employee, name=Steve}

Of course, in the above example, we use an extension function for a query we could easily have done with standard XPath. The real use of using extensions as predicates comes in when we need to use non-XPath types as predicates, like dates or valid creditcardnumbers or the like. To find only the nodes which are valid dates, you might use the following:

```
public class MyExtensionFunctions {
    public static boolean isDate(ExpressionContext context){
        Pointer pointer = context.getContextNodePointer();
        if (pointer == null){
            return false;
        }
        return pointer.getValue() instanceof Date;
    }
}
```

And call it like this:

```
for(Iterator iter = context.iterate("//.[myext:isDate()]"); iter.hasNext();){
.....
}
```

Which would only return objects from the tree which are valid Date instances.

5 Resources

5.1 *On the web:*

The first and foremost resource would be the XPath homepage (especially check out the “Users Guide”):

<http://jakarta.apache.org/commons/jxpath>

A very good (although somewhat condensed) XPath tutorial can be found on the w3schools website:

<http://www.w3schools.com/xpath/default.asp>

Another XPath tutorial (apart from this one ;-), I used it for a first introduction to XPath can be found at:

<http://today.java.net/pub/a/today/2006/08/03/java-object-querying-using-jxpath.html>

5.2 *Sources and Eclipse project*

Both the sources for this tutorial (“jxpath-tutorial-src.zip”) and an eclipse project with all JARs and the sources included (“jxpath-tutorial-eclipseproject.zip”) can be downloaded from my website:

<http://www.tutorials.tfo-eservices.eu>