# Advanced Java Programming

Lecture Notes

Nauman
Lecturer Computer Science Department,
FAST National University of
Computer and Emerging Sciences,
Peshawar, Pakistan
nauman@csrdu.org

January 4, 2012*

*Last generated on April 16, 2012

# Contents

CHAPTER 1

# INTRODUCTION AND REVIEW

This course is aimed to give you a chance to experience large-scale industry standard softwares and technologies related to the Java language and platform. The goal is going to be able to form a core group of compentencies that allow you to learn new technologies quickly and apply them in practical situations. The outlines of the course can be seen on the lecture server or on the course web page [1].

The choice of the IDE for development in this course would be Eclipse. However, if you are familiar with (or prefer) any other IDE (such as NetBeans or IntelliJ IDEA), please feel free to use that. It is however, my experience that eclipse has some great features and if you're just starting out, it's a great way to begin Java development.

## 1.1 Eclipse IDE Review

So we begin by exploring some of the features of the Eclipse IDE. This will help us setup our environment and let us work quickly and efficiently on all our tasks.

First, create a new workspace for this course. You can do this using `File` → `Switch Workspace` → `Others`. Now, create a new project and name it `w01-java-review`. Here, we're going to first create a new file to demonstrate one of the most important features of the Eclipse IDE – *Quick Fix*.

*Quick Fix* Suggested fixes in eclipse.

### 1.1.1 Quick Fix

Create a new class `RunConfigsExample` in the default package using `New` → `Class` from the context menu of the project (Right-click on the project name). Add the following package declaration to the top of the class file:

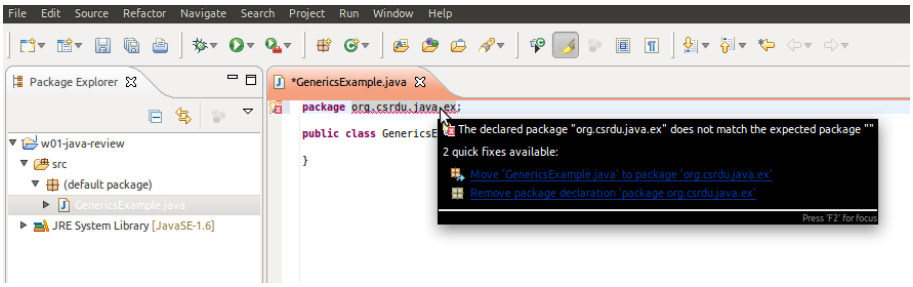
```
1  package org.csrdu.java.ex;
```

Notice that the package name is wavy-underlined by eclipse. Hover the mouse pointer over this error to view the error. The error is:

```
1  The declared pacakge "org.csrdu.java.ex" does not match the
2  expected package "".
```

Whenever you hover over any error, eclipse tries to suggest fixes for the error. This is the quick fix feature. You can also get to the quickfix menu by moving your cursor over to the error and hitting `Ctrl+1`. Figure 1.1 shows one such quickfix popup. Select the option to move the source file to the appropriate package folder (first option).

Figure 1.1: QuickFix in Eclipse




| Shortcut | Description |
|---|---|
| `Ctrl+1` | Quick Fix |
| `Ctrl+3` | Quick Access Window |
| `Shift+Alt+R` | Refactor (Cursor-Context Sensitive) |
| `Ctrl+Shift+F` | Fromat Source |
| `Ctrl+D` | Delete Current Line |
| `Ctrl+E` | Show Editor Selection Popup |
| `Ctrl+I` | Correct Indentation (Current Line/Selection) |
| `Ctrl+/` | Comment/Uncomment |

Table 1.2: List of Useful Eclipse Shortcuts

### 1.1.2 Useful Eclipse Shortcuts

Eclipse comes with a powerful set of shortcuts that can make your coding experience very smooth. Refer to Table 1.2 Learn the shortcuts, use each one at least once. You'll be expected to know them.

### 1.1.3 Run Configurations

*Run Configurations*

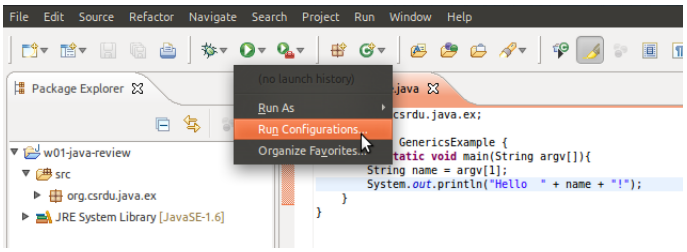
We will now take a look at eclipse *Run Configurations*. Run configurations allow you to define different parameters and options for running your project. This allows you to pass parameters to your function, set JVM params and so on. Let's try passing a name to the hello world program given below:

```
1  package org.csrdu.java.ex;
2
3  public class RunConfigsExample {
4    public static void main(String argv[]){
5      String name = argv[0];
6      System.out.println("Hello  " + name + "!");
7     }
8   }
```

As you can see, this program requires an input parameter. We can pass this parameter using the run configuration dialog box. To do this, open the run configuration dialog box (either by long-clicking on the run menu – cf. Fig 1.2 – or by going to `Run` → `Run Configurations`).

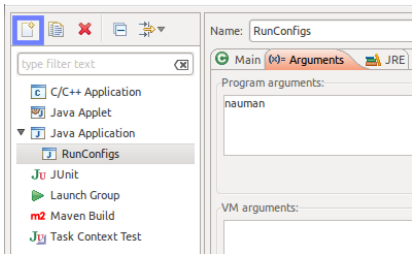
[1] `http://csrdu.org/nauman` ... to be put up.

Figure 1.2: Starting Run Configuration Dialog




Double-click on Java Application or select Java Application and click on the `New` button on the mini-toolbar. This will create a new run configuration. You can give it a descriptive name for future reference. Navigate to the `Arguments` tab and enter your name in the `Program Arguments` textbox. Click `Apply` or `Run` to start the program and view the output in the console. See Fig 1.3 for an example.

Figure 1.3: Run Configurations Dialog



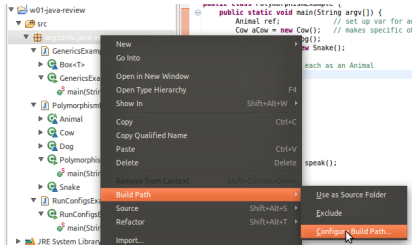

### 1.1.4 Build Paths

When running `javac` and `java` from the command line, we can set the *class path* using the `CLASSPATH` environment variable or by using the `-cp` switch. In eclipse, we can define dependencies in a similar manner using the concept of *build path*. To define the build path for a project, right-click on the project and select `Build Path` → `Configure Build Path`.

*class path*

*build path*

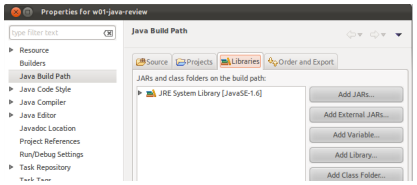
Figure 1.4: Opening the Build Path Dialog




The important things to note here (for now) are the `Projects` and `Libraries` tabs. In projects tab, you can define the projects that *this* project depends on. Similarly, in the libraries tab, you can tell eclipse about the `jar` files that reside inside your project hierarchy (`Add JARs`) or outside it (`Add External JARs`). It's

usually a good idea to keep all relevant JAR files inside the project hierarchy for better portability of the source. Of course, you have to be wary about redistribution and packaging etc.

Figure 1.5: Build Path Dialog




## 1.2 Java Review

Now, we're going to talk about some of the basics of Java that might come in handy during the course of our practical sesesions. We'll begin by covering *generics* and *polymorphism*.

### 1.2.1 Generics

*generics*

If you've worked with Standard Template Library (STL) in C++, you would be very familiar with the concepts of *generics*. Generics are a built-in language feature that will make your software more reliable. [5] They allow you to create a template for a class that can be *instantiated* based on different types and thus allow customization at compile time. Take a look at the following example.

```
1  package org.csrdu.java.ex;
2
3  public class GenericsExample {
4
5    public static void main(String argv[]) {
6      Box<Integer> integerBox = new Box<Integer>();
7      integerBox.add(new Integer(10));
8      Integer someInteger = integerBox.get(); // no cast required!
9      System.out.println(someInteger);
10
11     Box<Float> floatBox = new Box<Float>();
12     floatBox.add(new Float(10.4));
13     Float someFloat = floatBox.get();
14     System.out.println(someFloat);
15   }
16 }
17
18 /**
19 * Generic version of the Box class.
20 *
21 * @param <T>
22 *            the type of value being boxed
23 */
24 class Box<T> { // T stands for "Type"
25   private T t;
26
27   public void add(T t) {
28     this.t = t;
29   }
```

```
30
31    public T get() {
32      return t;
33    }
34  }
```

Line 24 is the most important one in this code. It creates a new *generic class* that takes in one type as input and will be instantiated when you create an object. In order to use this class, we can pass it a class name (e.g. Line 6) and all instances of `T` will be replaced with the class name that you pass. You can imagine how this technique might be useful when you want to, say, create a stack of integers, strings, floats or even custom objects. Java collections library uses this technique extensively.

*generic class*

### 1.2.2 Polymorphism

*Polymorphism* is the capability of an action or method to do different things based on the object that it is acting upon. This is the third basic principle of object oriented programming. Overloading, overriding and dynamic method binding are three types of polymorphism.

*Polymorphism*

*Overloaded methods* are methods with the same name signature but either a different number of parameters or different types in the parameter list. For example 'spinning' a number may mean increase it, 'spinning' an image may mean rotate it by 90 degrees. By defining a method for handling each type of parameter you control the desired effect.

*Overloaded methods*

*Overridden methods* are methods that are redefined within an inherited or subclass. They have the same signature and the subclass definition is used.

*Overridden methods*

*Dynamic (or late) method binding* is the ability of a program to resolve references to subclass methods at runtime. As an example assume that three subclasses (Cow, Dog and Snake) have been created based on the Animal abstract class, each having their own speak() method. Although each method reference is to an Animal (but no animal objects exist), the program is will resolve the correct method reference at runtime. [9]

*Dynamic (or late) method binding*

```
1   package org.csrdu.java.ex;
2
3   public class PolymorphismExample {
4     public static void main(String argv[]) {
5       Animal ref;                          // set up var for an Animal
6       Cow aCow = new Cow();          // makes specific objects
7       Dog aDog = new Dog();
8       Snake aSnake = new Snake();
9
10      // now reference each as an Animal
11      ref = aCow;
12      ref.speak();
13      ref = aDog;
14      ref.speak();
15      ref = aSnake;
16      ref.speak();
17    }
18  }
19
20  abstract class Animal {
21    public abstract void speak();
22  }
```

```
23
24    class Cow extends Animal {
25      @Override
26      public void speak() {
27        System.out.println("Moo.");
28      }
29    }
30
31    class Dog extends Animal {
32      @Override
33      public void speak() {
34        System.out.println("Woof.");
35      }
36    }
37
38    class Snake extends Animal {
39      @Override
40      public void speak() {
41        System.out.println("...");
42      }
43    }
```

CHAPTER 2

# ECLIPSE FEATURES

In this chapter, we're going to talk about some of the more advanced features of eclipse including debugging and version control with EGIT. Let's begin by talking about how we can debug our code using the eclipse debugging perspective.

## 2.1 Debugging with Eclipse

Let's start off by creating a simple sort class that uses the *quick sort* algorithm. This will give us a chance to test two important features of eclipse – *JUnit testing* and *debugging*. The code for the class is as follows:

*JUnit testing*
*debugging*

```
1  package org.csrdu.java.ex;
2
3  public class Quicksort {
4    private Integer[] numbers;
5    private int number;
6
7    public void sort(Integer[] values) {
8      if (values == null || values.length == 0) {
9        return;
10     }
11     this.numbers = values;
12     number = values.length;
13     quicksort(0, number - 1);
14   }
15
16   private void quicksort(int low, int high) {
17     int i = low, j = high;
18     int pivot = numbers[low + (high - low) / 2];
19
20     while (i <= j) {
21       while (numbers[i] < pivot) {
22         i++;
23       }
24       while (numbers[j] > pivot) {
25         j--;
26       }
27       if (i <= j) {
28         exchange(i, j);
29         i++;
30         j--;
31       }
32     }
33     if (low < j)
34       quicksort(low, j);
35     if (i < high)
36       quicksort(i, high);
37   }
38
39   private void exchange(int i, int j) {
40     Integer temp = numbers[i];
41     numbers[i] = numbers[j];
42     numbers[j] = temp;
```

```
43   }
44 }
```

This is the simple quick sort algorithm that you have already studied. You can refer to any textbook for the explanation of the algorithm itself. Now, we will write a test case for our sorting algorithm.

### 2.1.1 JUnit Testing

Let's add a new test case for our class by right-clicking on the project and selecting `New → JUnit Test Case`. Enter the name `QuicksortTest` for the new test case. Use the package `org.csrdu.java.ex.tests` for the test case. The code for the test case is[1]:

```
1  package org.csrdu.java.ex.tests;
2
3  import static org.junit.Assert.*;
4  import org.junit.Test;
5
6  public class QuicksortTest {
7    private Integer[] numbers;
8    private final static int SIZE = 7;
9    private final static int MAX = 20;
10
11   @Before public void setUp() throws Exception {
12     numbers = new Integer[SIZE];
13     Random generator = new Random();
14     for (int i = 0; i < numbers.length; i++) {
15       numbers[i] = generator.nextInt(MAX);
16     }
17   }
18
19   @Test
20   public void testNull() {
21     Quicksort sorter = new Quicksort();
22     sorter.sort(null);
23   }
24
25   @Test
26   public void testEmpty() {
27     Quicksort sorter = new Quicksort();
28     sorter.sort(new Integer[0]);
29   }
30
31   @Test
32   public void testSimpleElement() {
33     Quicksort sorter = new Quicksort();
34     Integer[] test = new Integer[1];
35     test[0] = 5;
36     sorter.sort(test);
37   }
38
39   @Test
40   public void testSpecial() {
41     Quicksort sorter = new Quicksort();
42     Integer[] test = { 5, 5, 6, 6, 4, 4, 5, 5, 4, 4, 6, 6, 5, 5 };
43     sorter.sort(test);
```

[1] You will get quite a few errors on this code. This is intentional. It's an exercise for you to use the quick fix feature of eclipse to resolve these errors.

```
44      if (!validate(test)) {
45        fail("Should not happen");
46      }
47      printResult(test);
48    }
49
50    @Test
51    public void testQuickSort() {
52      for (Integer i : numbers) {
53        System.out.println(i + " ");
54      }
55      long startTime = System.currentTimeMillis();
56
57      Quicksort sorter = new Quicksort();
58      sorter.sort(numbers);
59
60      long stopTime = System.currentTimeMillis();
61      long elapsedTime = stopTime - startTime;
62      System.out.println("Quicksort " + elapsedTime);
63
64      if (!validate(numbers)) {
65        fail("Should not happen");
66      }
67      assertTrue(true);
68    }
69
70    @Test
71    public void testStandardSort() {
72      long startTime = System.currentTimeMillis();
73      Arrays.sort(numbers);
74      long stopTime = System.currentTimeMillis();
75      long elapsedTime = stopTime - startTime;
76      System.out.println("Standard Java sort " + elapsedTime);
77      if (!validate(numbers)) {
78        fail("Should not happen");
79      }
80      assertTrue(true);
81    }
82
83    private boolean validate(Integer[] numbers) {
84      for (int i = 0; i < numbers.length - 1; i++) {
85        if (numbers[i] > numbers[i + 1]) {
86          return false;
87        }
88      }
89      return true;
90    }
91
92    private void printResult(int[] numbers) {
93      for (int i = 0; i < numbers.length; i++) {
94        System.out.print(numbers[i]);
95      }
96      System.out.println();
97    }
98  }
```

The test case is fairly straight-forward. It defines a `setUp` function that initiates the data required for the tests. It tests certain conditions – when the input data is null, when it is empty but not null, when it is a single-element and when it has more then one elements. That covers a broad range of conditions we might want to test.

In each case, the test calls the `Quicksort` class for sorting and then compares the result with expected results. If the computed result matches the expected result, the test passes. Otherwise, it fails.

You can run the test using `Run` → `Run As` → `JUnit Test`. Notice the new view `JUnit` that opens up to show the results. [2] We are now going to try and follow how the code is actually executed using the debugging feature of eclipse.

### 2.1.2 Eclipse Debugger

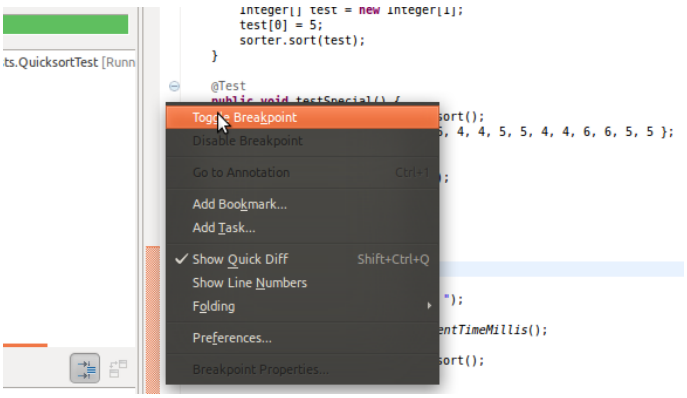
Begin by opening up the code for `QuicksortTest` class and navigating to the `testSpecial()` function. This is the more interesting test case that we have. The objective is to create a *breakpoint* at the beginning of this function so that we can start debugging when our JVM reaches it.

*breakpoint*

*gutter*

To set a breakpoint, right-click on the *gutter* of the editor and select `Toggle Breakpoint`. This should create a little blue disc in the gutter on the line. Our debugger will stop and give us control when it gets to this line.

Figure 2.1: Setting Breakpoints in Eclipse




To debug the project, use `Run` → `Debug As` → `JUnit Test`. Eclipse should start running the debugger and ask you to change to the *debug perspective* as soon as it reaches a breakpoint for the first time. Click on `Yes` and the perspective will change to show views more suitable for the debugging task at hand.

*debug perspective*

There are a couple of things to note here:

**Debug View** shows the call stack. This is very useful in figuring out how you got to the point in code that is shown in the editor view. In this example, the information may not be very useful but in cases of massively recursive algorithms, this comes in very handy.

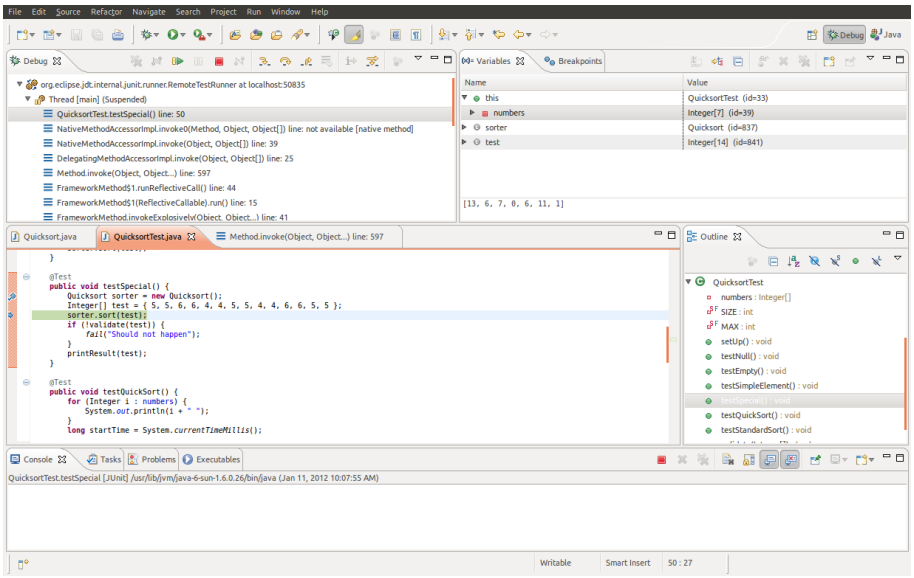
**Editor View** highlights (in light green) the line that is just about to be executed by the JVM. You can scroll around and add/remove breakpoints as usual to help with debugging.

**Debug Toolbar** allows you to `Step Into`, `Step Over` and `Step Return` as well as `Stop` the execution. Usually, you would want to *step* into your own functions that

*step*

[2] All the tests should pass. If you see a failure, go back to the code and fix it.

Figure 2.2: Debug Perspective




you're not sure are working properly and simply step over Java's built-in functions as they tend to get into gory details that are often not that useful during debugging.

Variables View allows you to take a look at the variables available to the currently higlighted line of code. You can drill down into objects to view their fields etc.

Breakpoint View shows the set breakpoints and allows you to disable/enable specific ones.

You should try stepping through the code and taking a look at how the call stack and variables change. Using this method, you can debug your code quickly and efficiently without having to scatter `println` statements all around your code.[3]

## 2.2 Version Control with Eclipse

Version control is an important aspect of any software development process. There are several version control systems available but here we are going to talk about Git.

Eclipse initially came with CVS but support was added for *Subversion* or *SVN*. For a long while, SVN was the recommended version control system. SVN is a centralized version control system – this means that there is going to be a central server managing the revisions and all computers that need to use VCS facilities connect to this server (cf. Fig 2.3). However, this means that all clients need persistent connections to the server for committing/managing changes.[4] To solve these issues, the idea of *distributed version control* systems emerged.

*Subversion*
*SVN*
*distributed version control*

[3]The eclipse debug perspective works in many languages other than Java. It has the same semantics so any experience you gain from this exercise can be useful elsewhere as well.

[4]There were also many issues that you can find out by refering to any distributed version control systems.

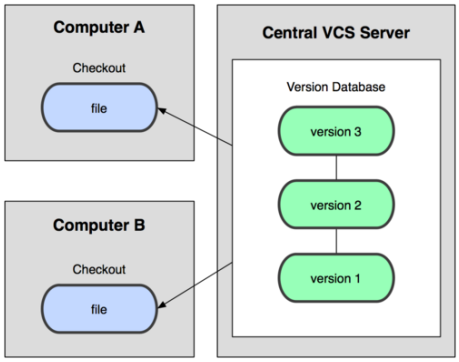



Figure 2.3: Centralized Version Control System

### 2.2.1 Distributed Version Control Systems

Distributed version control systems work differently. There is no "central" server. Whenever a client wants to get something from a DVCS, they get a complete copy of the version control database (cf. Fig 2.4). They can then make any changes in their local databases and then "synchronize" their changes into the *upstream server*. There are many distributed version control systems in use today. These include *Git*. *mercurial*, and *bazaar*. We will be focusing on Git here.

*upstream server*
*Git*
*mercurial*
*bazaar*

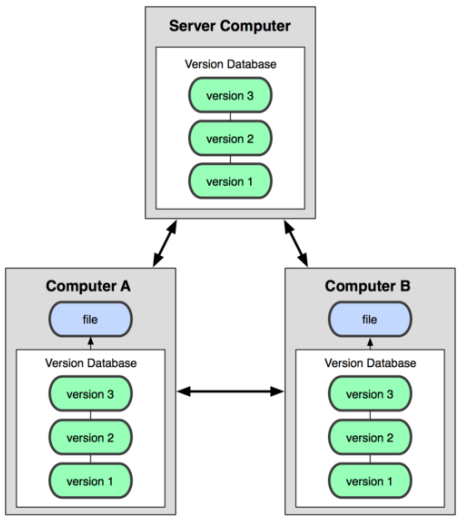



Figure 2.4: Centralized Version Control System

### 2.2.2 GIT

Being a distributed version control system, Git is a little difficult to understand at first. However, knowing the right terminology and a little practice makes this extremely powerful tool available for use in projects.

*revisions*

Let's begin by trying to understand how Git saves *revisions* to a file. Refer to

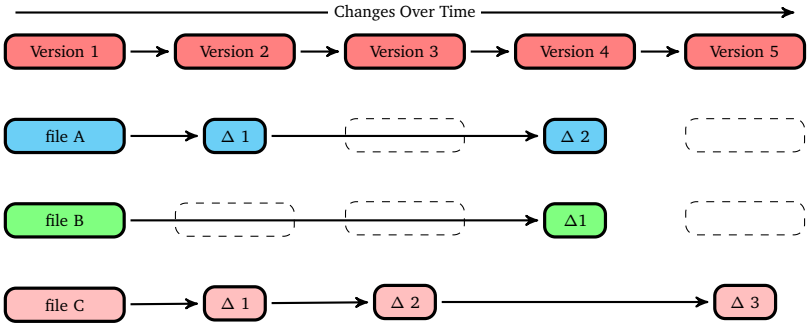



Figure 2.5: Centralized Version Control System

Fig 2.5 for the following explanation. Each revision has different *deltas* associated with it. Deltas are made whenever a file is changed and added[5] to Git.

*deltas*

### Making Changes

Let's take a brief look at the life cycle of how file changes are made in Git (cf. Fig 2.6). When we start with a Git repository, all files are *untracked* – Git will not be keeping track of these files. We can *add files to git* and they will reach the *unmodified* state. We can then change files and they will become *modified*. We can then add these changed files to a special cache location called *index*. Index is a location for temporarily saving changes and collectively committing them to the Git repository. This location is also called the *stage*. Finally, we can *commit* the changes and they will be saved in the Git repository and again reach the modified state. Finally, we may also sometimes want to remove a file from tracking in which

*add files to git*

*index*

*stage*

*commit*

[5]We will cover the semantics of adding in a little while.

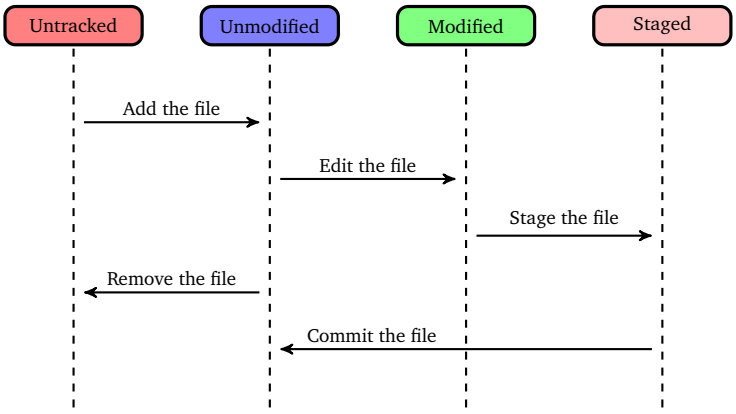



Figure 2.6: Centralized Version Control System

case it will rescend to the untracked state. Now we will take a look at how we can do all this practically.

### Using Git from the Console

First, we will create a new repository.

```
mkdir -p ~/tmp/gittemp
cd ~/tmp/gittemp
git init
```

This command will initialize a git repository in the current directory. Let's create a couple of files and make some changes to it.

```
touch a.txt
touch b.txt
```

Insert the following lines into the files respectively.

```
Lorem Ipsum is simply dummy text of the printing and typesetting
industry. Lorem Ipsum has been the industry's standard dummy text
ever since the 1500s, when an unknown printer took a galley of
type and scrambled it to make a type specimen book. It has
survived not only five centuries, but also the leap into
electronic typesetting, remaining essentially unchanged. It was
popularised in the 1960s with the release of Letraset sheets
containing Lorem Ipsum passages, and more recently with desktop
publishing software like Aldus PageMaker including versions of
Lorem Ipsum.
```

```
It is a long established fact that a reader will be distracted
by the readable content of a page when looking at its layout.
The point of using Lorem Ipsum is that it has a more-or-less normal
distribution of letters, as opposed to using 'Content here, content
here', making it look like readable English. Many desktop
publishing packages and web page editors now use Lorem Ipsum as
their default model text, and a search for 'lorem ipsum' will
uncover many web sites still in their infancy. Various versions
have evolved over the years, sometimes by accident, sometimes on
purpose (injected humour and the like).
```

After inserting this content, if we try to take a look at the status of the current repository, we get the following results:

```
~/tmp/gittemp$ git status
# On branch master
#
# Initial commit
#
# Untracked files:
#   (use "git add <file>..." to include in what will be committed)
#
#       a.txt
#       b.txt
nothing added to commit but untracked files present (use "git add" to track)
```

The first line shows the command `git status`. Then, there is the name of the branch, *master* – which is the default branch. Next is a couple of lines showing which files are untracked – the two files we just created. Finally, it shows you a helpful hint telling you how to add files. Let's do that now.

```
$ git add .
$ git status
# On branch master
#
# Initial commit
#
# Changes to be committed:
#   (use "git rm --cached <file>..." to unstage)
#
#        new file:   a.txt
#        new file:   b.txt
#
```

The command `git add .` adds the current directory (signified by the `.`) to the repository. This also adds any files within this directory. Next, the status shows that these files are no longer untracked. These are now *staged* i.e. they are in the index or staging area. We can now commit the files using the `git commit` command.

```
$ git commit -m "Initial Commit"
[master (root-commit) c6e7a25] Initial Commit
2 files changed, 20 insertions(+), 0 deletions(-)
create mode 100644 a.txt
create mode 100644 b.txt
```

There are a couple of important aspects of this command and output: The most important is the `-m` switch. It allows you to give a helpful short message to this commit. This will be visible to anyone taking a look at the history to see which changes meant what. This message can be anything but should be descriptive.

The next couple of lines show what the commit did to the repository. Now, we make a couple of modifications to the files and see what that does. Open `a.txt` and delete the second line, Delete the last line from `b.txt`. Also add a new line with some text to this file.

Imagine that at this point, we need to take a look at exactly what changes we've made to the files under version control. To do this, we issue the `git diff` command.

```
$ git diff
diff --git a/a.txt b/a.txt
index b90cbbe..fc3ea0c 100644
--- a/a.txt
+++ b/a.txt
@@ -1,5 +1,4 @@
Lorem Ipsum is simply dummy text of the printing and typesetting
-industry. Lorem Ipsum has been the industry's standard dummy text
ever since the 1500s, when an unknown printer took a galley of
type and scrambled it to make a type specimen book. It has
survived not only five centuries, but also the leap into
diff --git a/b.txt b/b.txt
index 3fac899..37b2374 100644
--- a/b.txt
+++ b/b.txt
@@ -7,4 +7,3 @@ publishing packages and web page editors now use Lorem Ipsum as
their default model text, and a search for 'lorem ipsum' will
uncover many web sites still in their infancy. Various versions
have evolved over the years, sometimes by accident, sometimes on
-purpose (injected humour and the like).
+Some NEW TEXT HERE ...
```

The first line show what command was used to generate the difference. Line 8 shows that the line beginning with the word 'industry' was deleted from `a.txt`. Similarly, Line 20 shows the line deleted from `b.txt`. Lastly, see the new line inserted in `b.txt`. Notice that this is the *unified diff syntax* that is used by many open source projects for exchanging patches made to source code.

*unified diff syntax*

Now take a look at git status again:

```
$ git status
# On branch master
# Changes not staged for commit:
#   (use "git add <file>..." to update what will be committed)
#   (use "git checkout -- <file>..." to discard changes in working directory)
#
#       modified:   a.txt
#       modified:   b.txt
#
no changes added to commit (use "git add" and/or "git commit -a")
```

See that the two files are now *modified*. Now, we can add the files to the staging area and commit the changes.

```
$ git add .
$ git commit -m "Changed two files"
[master ccd6d2b] Changed two files
2 files changed, 1 insertions(+), 2 deletions(-)
```

Lastly, you can take a look at the complete history using `gitk`. To install the package, issue the command `sudo apt-get install gitk` or the equivalent installation command for your distro. Now, we will look at how we can use Git from within Eclipse.

### 2.2.3 Eclipse and Git

To use Git with eclipse, we first need to install the EGit plugin (which is based on the JGit project itself). Do this by starting eclipse and navigating to `Help` → `Install New Software`. From here, you can add the update site `http://download.eclipse.org/egit/updates` and give it a name. If you get a duplicate site error, click on the `Available Software Sites` link in the installation window and check the site with the above URL and apply. Now, you can select the egit update site from the dropdown box and install EGit plugin. Click next, accept the agreement and restart eclipse after the installation completes.

#### Placing a Project Under Version Control

Create a project called `w02-egit-usage` and put a hellow world code in it:

```
package org.csrdu.java.ex;

public class GitUsage {
  /**
  * @param args
  */
  public static void main(String[] args) {
    System.out.println("Hello World!");
  }
}
```

Right-click on the project and select `Team` → `Share Project`. Choose to *create a new repository* and then select the current project name.[6] Click on `Create Repository` and then `Finish`.

*create a new repository*

Notice the changed icons in the project explorer. Similar to how we added files to git repository earlier, we can add the `src` folder to our index (or staging area) by right-clicking on the folder name and selecting `Team` → `Add To Index`. Again, notice the changed emblems on the folder and `.java` file icons. We can also commit the file using the `Team` → `Commit` menu. When you commit, you will see the commit dialog which lets you enter a message and select files which you want to commit. Give a descriptive message and click on `Commit`. Once again, notice the changed emblems on file icons.

### Making Changes

Similar to the CLI version of git usage, we can modify the source file, add it to the index and commit to create a revision in the repository. Do this by changing the 'hello world' line to:

```
1   System.out.println("Hello Changed World!");
```

Add the file to index and commit it. You can view a file's history by right-clicking on the filename and selecting `Team` → `Show in History`. You can take a look at the commits for this file and select the filename from the right column to see the diffs for the commit.

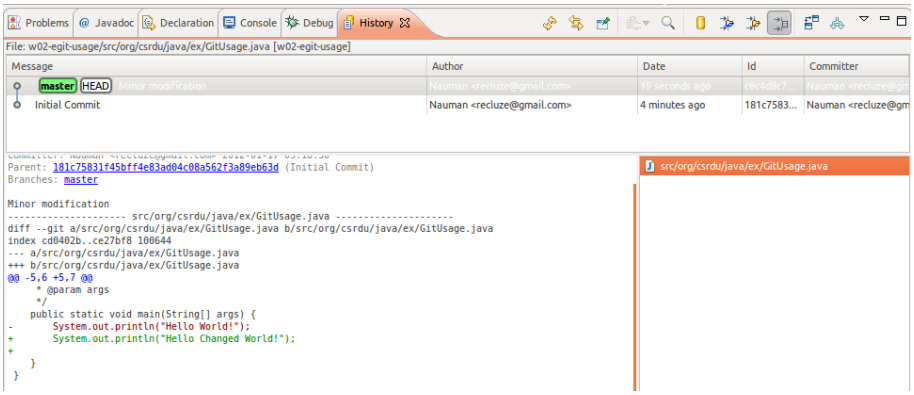



Figure 2.7: EGit History View

[6] Notice the warning eclipse gives you. We are ignoring it for the moment for the sake of learning the use of EGit. You should always create your repositories outside the eclipse workspace.

CHAPTER 3

# JAVA REFLECTION

It is often very useful if a computer program can study its own (or another program's) code and "think" about what it does. It's a feature used by many IDEs include eclipse. Take a look at the *outline* view. It lists all the fields and methods in the class currently open in the editor. There are many other situations in which such information can be useful. In this chapter, we're going to take a look at the basics of reflections and discuss some of the concepts related to this impressive strength of the Java platform. [6]

## 3.1 Class Loaders

Let's begin with a simple introduction to class loaders. In Java, all programs are defined using *classes*. These classes do not have to be sitting on the filesystem. They can come from anywhere – from the filesystem, over the network, even created on the fly. The portions of the Java platform required to load a class when it is required are called *class loaders*. For example, when you read a class present in a jar file[1] a class loader will extract the `.class` file from the jar and make it available for execution. Similarly, there might be a need to load a specific class over the network. [11]

*classes*

*class loaders*

To take a look at how the classes are loaded, you can use the *VM argument* `verbose:class`. VM arguments are similar to program arguments but they are passed to the Java virtual machine instead of to your program. The syntax for VM arguments is generally:

*VM argument*

```
java [vm arguments] program_name [program arguments]
```

Let's write a simple program that loads two classes and see what goes on in the background when it is loaded.

```
package org.csrdu.java.ex;

public class Greeter {

  public Greeter() {
    System.out.println("Main greeter");
    SubGreeter sg = new SubGreeter("sub class loaded");
  }

  public static void main(String[] args) {
    new Greeter();
  }
}

class SubGreeter {
  public SubGreeter (String msg){
    System.out.println(msg);
  }
}
```

[1] In the second half of this chapter, we'll see how to create a jar file.

The program itself is very simple. The main class `Greeter` is instantiated and in turn, it creates an instance of the class `SubGreeter`. We'll see how the classes are loaded when we run the program.

Edit the run configurations and this time, modify the `VM Configurations` to include the option `-verbose:class`.

```
[Opened /usr/lib/jvm/java-6-sun-1.6.0.26/jre/lib/rt.jar]
[Loaded java.lang.Object from /usr/lib/jvm/java-6-sun-1.6.0.26/jre/lib/rt.jar]
[Loaded java.io.Serializable from /usr/lib/jvm/java-6-sun-1.6.0.26/jre/lib/rt.jar]
[Loaded java.lang.Comparable from /usr/lib/jvm/java-6-sun-1.6.0.26/jre/lib/rt.jar]
[Loaded java.lang.CharSequence from /usr/lib/jvm/java-6-sun-1.6.0.26/jre/lib/rt.jar]
[Loaded java.lang.String from /usr/lib/jvm/java-6-sun-1.6.0.26/jre/lib/rt.jar]

...

[Loaded org.csrdu.java.ex.Greeter from file:/home/.../w03-reflection/bin/]
Main greeter
[Loaded org.csrdu.java.ex.SubGreeter from file:/home/.../w03-reflection/bin/]
sub class loaded
[Loaded java.lang.Shutdown from /usr/lib/jvm/java-6-sun-1.6.0.26/jre/lib/rt.jar]
[Loaded java.lang.Shutdown$Lock from /usr/lib/jvm/java-6-sun-1.6.0.26/jre/lib/rt.jar]
```

## 3.2 Reflection

### 3.2.1 Referring to Classes and Their Members

As I mentioned earlier in the chapter, it's possible for a Java program to 'reflect upon' itself and make decisions based on what the program is. To do this, we begin by introducing the concept of reflection by referring to classes and their members. We will first write a program that can look into the bytecode of classes not contained within itself. Then we'll move on to programs that inspect themselves.

Take a look at the following code:

```
1  import java.lang.reflect.*;
2
3  public class DumpMethods {
4    public static void main(String args[]) {
5      try {
6        Class c = Class.forName(args[0]);
7        Method m[] = c.getDeclaredMethods();
8        for (int i = 0; i < m.length; i++)
9        System.out.println(m[i].toString());
10     }
11     catch (Throwable e) {
12       System.err.println(e);
13     }
14   }
15 }
```

The program takes in one argument (that you can pass to it using run configurations from within eclipse) and prints out the methods that are declared by the class. (It does not list methods inherited from the parent classes. To do that, you can use the `getMethods()` function instead of the `getDeclaredMethods()`.)

The most important line to note in the above program is Line 6. This line uses the static `forName()` method of the class named `Class`[2]. This method takes

[2] You should now be getting an idea of why this concept is called reflection.

in a string and gets the class represented by that string. This has to be a *fully-qualified class name* specified using the prefixed package name. An example would be `java.util.Stack`. This method is used extensively for creating pluggable Java programs. You can get this string from any configuration file (or even from a user at runtime) and modify your program's behavior based on the selected class.

*fully-qualified class name*

The output produced by the program is as follows:

```
public synchronized java.lang.Object java.util.Stack.pop()
public java.lang.Object java.util.Stack.push(java.lang.Object)
public boolean java.util.Stack.empty()
public synchronized java.lang.Object java.util.Stack.peek()
public synchronized int java.util.Stack.search(java.lang.Object)
```

### 3.2.2 Making Changes to Objects through Reflection

Reflection is not just about inspecting the code. It also allows you to make changes to your program's behavior at runtime by modifying the state of objects. [12] The following program demonstrates this concepts:

```
package org.csrdu.java.ex;

import java.lang.reflect.Field;

public class ReflectMod {
  public int incrementField(String name, Object obj) {
    Field field;
    int value = 0;

    try {
      // get the field from the object
      field = obj.getClass().getDeclaredField(name);
      // get current value of the field and add 1
      value = field.getInt(obj) + 1;
      // set value explicitly
      field.setInt(obj, value);
    } catch (NoSuchFieldException e) {
      e.printStackTrace();
    } catch (IllegalArgumentException e) {
      e.printStackTrace();
    } catch (IllegalAccessException e) {
      e.printStackTrace();
    }
    return value;
  }

  public static void main(String args[]) {
    ReflectMod rm = new ReflectMod();
    DummyClass dm = new DummyClass();
    dm.x = 100; // initialize value to 100
    rm.incrementField("x", dm); // increment using reflection
    System.out.println("The current value of x is: " + dm.x);
  }
}

class DummyClass {
  public int x;
}
```

The program is fairly straight-forward. It's similar to the previous example with a couple of changes:

- The class is retrieved from an object instead of using the class's fully qualified name.
- It retrieves a specific field based on its name.
- It makes changes to the field using the `setInt` method of the `Field` class.

The semantics of the program are fairly simple. The program starts by creating an instance of the `DummyClass`. It sets the value of the field `x` and then increments it by calling the `incrementField()` method of our main `ReflectMod` class. Notice also the different exceptions that might be thrown when trying to access/modify a field.

### 3.2.3 Security in Reflection

*security restrictions*

Security is an important concern in reflection. Here, we'll take a brief look at how some *security restrictions* apply on some methods in Java.[3]

Let's create two simple classes. [12] The first one is `TwoStrings.java`:

```
1  package org.csrdu.java.ex;
2
3  public class TwoString {
4    private String m_s1, m_s2;
5
6      public TwoString(String s1, String s2) {
7      m_s1 = s1;
8      m_s2 = s2;
9    }
10 }
```

The second file is `ReflectSecurity.java`:

```
1  package org.csrdu.java.ex;
2
3  import java.lang.reflect.Field;
4
5  public class ReflectSecurity {
6    public static void main(String[] args) {
7      try {
8        TwoString ts = new TwoString("m1's string here", "m2's string here");
9        Field field = ts.getClass().getDeclaredField("m_s1");
10       // field.setAccessible(true);
11       Integer inst = 22;
12       System.out.println("Retrieved value is: " + field.get(ts));
13     } catch (Exception ex) {
14       ex.printStackTrace(System.out);
15     }
16   }
17 }
```

Notice the commented out line. By default (i.e. if you don't uncomment Line 10), access to private fields is not allowed. If we run the code as it is, we get the following exception:

```
1  java.lang.IllegalAccessException:
2    Class org.csrdu.java.ex.ReflectSecurity can not access
3    a member of class org.csrdu.java.ex.TwoString with
```

[3] Later on in the course outline, we have more on security.

```
4   modifiers "private"
5 at sun.reflect.Reflection.ensureMemberAccess(Reflection.java:65)
6 at java.lang.reflect.Field.doSecurityCheck(Field.java:960)
7 at java.lang.reflect.Field.getFieldAccessor(Field.java:896)
8 at java.lang.reflect.Field.get(Field.java:358)
9 at org.csrdu.java.ex.ReflectSecurity.main(ReflectSecurity.java:12)
```

The solution, though is fairly straight-forward. We can call the `setAccessible()` function of a `Field` object and that will allow us to access the private members as well. The caveat, though, is that the code calling this method must not be restricted. That, we will leave, for another time.

### 3.2.4 Mini Case Study: A Plugin Manager

Let's finish off this section by looking at a very useful application of reflection. Imagine a software that is based on the *plugin architecture*. You might be familiar with this type of software if you've used Adobe Photoshop or any of the family of image manipulation software that allow manipulation of images through plugins. The way this works is that the main software loads *n* number of plugins at startup. Whenever the user runs a plugin, the image (or selection) is sent to the plugin, which manipulates it and returns the result.

*plugin architecture*

The strength of this technique comes from the fact that the main software does not have to know how the plugin works. It only needs to know how to provide inputs to the plugin and receive outputs from it. This is exactly the purpose of *interfaces* – they define a contract between the caller and the callee.

*interfaces*

We can simulate a similar plugin software using reflection very easily. For simplicity, instead of images, we are going to use strings. The main software will pick a plugin (based on the user's input or any other means) and send a string to the plugin. Depending on the implementation of the plugin, the string will be changed and sent back to the main program.

The code for this program follows:

```
1  package org.csrdu.java.ex;
2
3  public class PluginManager {
4
5    public static void main(String[] args) {
6      try {
7        String pluginClass = "org.csrdu.java.ex.FirstPlugin";
8        // pluginClass =  "org.csrdu.java.ex.SecondPlugin";
9        Plugin pg = (Plugin) Class.forName(pluginClass).newInstance();
10       String out = pg.manipulate("Some string");
11       System.out.println(out);
12     } catch (InstantiationException e) {
13       e.printStackTrace();
14     } catch (IllegalAccessException e) {
15       e.printStackTrace();
16     } catch (ClassNotFoundException e) {
17       e.printStackTrace();
18     }
19   }
20 }
21
22 interface Plugin {
23   public String manipulate(String in);
24 }
```

```
25
26 class FirstPlugin implements Plugin {
27   @Override
28   public String manipulate(String in) {
29     return in + " changed.";
30   }
31 }
32
33 class SecondPlugin implements Plugin {
34   @Override
35   public String manipulate(String in) {
36     return in + " changed by a lot.";
37   }
38 }
```

First, let's see the output:

```
1 Some string changed.
```

If we uncommment Line 8, we get:

```
1 Some string changed by a lot.
```

If you take a look at the code, you will notice that it's very straight-forward. It only uses the reflection concepts we've already studied. The only new method is the `newInstance()` method of the `Class` class. This method essentially calls the default constructor of the class that is returned by the `forName()` method.[4]

We then cast the returned object to the `Plugin` interface. All our main program cares about the new object now, is that it is a `Plugin` – it will implement all the functions specified by the `Plugin` interface. This is a contract between the caller – our main class – and the callee – the plugin we just instantiated.

We can then call the `manipulate()` function of this plugin and it will change the string we pass to it. The actual modification is at the discretion of the plugin.

The important thing to note here is that the plugin is selected based on the value of the variable `pluginClass`. The value of this variable can be retrieved from a database, an XML file or even input from the console. The user can thus select which plugin to run. This is the core technique used behind many contemporary softwares including the *Spring Framework* and even the Eclipse IDE!

*Spring Framework*

Let's do one final exercice to drive this point home. Right-click on the variable `pluginClass` and select `Source` → `Externalize Strings`. Deselect all but the string that is used as the value for `pluginClass` variable. In the `Key` field, set the value to `pluginName`. Click on `Next`, review the changes and hit `Finish`.

Take a look at the newly created files: `Messages.java` and `messages.properties`. The former retrieves the values set for the strings from the latter. You can now modify the value for `pluginName` in this text file and run a different plugin without having to recompile the programs's code.

[4] You can also call the non-default constructors by passing an `Object` array to this method.

CHAPTER 4

# LOGGING

Beginning programmers usually debug all their programs by spraying `print` statements throughout their programs. This works well by providing all sorts of useful information about different variables but simply doesn't work for large systems. One problem is that you get way too many `print` outputs to make sense of which one you want. The second, more important, issue is that during deployment, a lot of effort has to go into removing these print statements (and then putting them back in during maintainance debugging).

When these programmers get a little smarter, they find out about stepping debuggers and work through the watch windows and other tools to debug. These run out of steam though after the program gets very large and there are many asynchronous calls between different components. It becomes very difficult to figure out the trace of exection by stepping through the code. Also, it's usually not possible to step through a program and find out the mistakes – especially if it's running in production environment.

The solution to all these problems is the concept of *logging*. Logging enables you to produce helpful debug/informative output at regular intervals and control which portion of that output you actually see. This separates the aspect of function requirements with those of debugging. *logging*

Logging within the context of program development constitutes inserting statements into the program that provide some kind of output information that is useful to the developer. Examples of logging are trace statements, dumping of structures and the familiar System.out.println or printf debug statements. *log4j* offers a hierarchical way to insert logging statements within a Java program. Multiple output formats and multiple levels of logging information are available. [4] *log4j*

## 4.1 Installation

First, create a new project called `w04-logging`.

Download the log4j binary from `http://logging.apache.org/log4j/1.2/download.html`.

Extract the archive and get the `log4j-*.jar` file. We need to tell eclipse about this jar file because this is a dependency for our code. To do this, we use the same *build path* configuration we studied in Chapter 1. Create a folder called `referenced` in the project and paste the jar file in there. Now, you can use the bulid path configuration editor to add this jar file as a dependency. Alternatively, you can right-click on the jar file in the project explorer and `Build Path` → `Add to Build Path`. *build path*

## 4.2 Usage Example

Let's create a new class within this project. Put the following code in the class. Discussion of the code follows the listing.

```
1  package org.csrdu.java.ex;
2
3  import org.apache.log4j.ConsoleAppender;
4  import org.apache.log4j.Level;
5  import org.apache.log4j.Logger;
6  import org.apache.log4j.SimpleLayout;
7  import org.apache.log4j.PatternLayout;
8  import org.apache.log4j.FileAppender;
9
10 public class LoggingExample {
11   static Logger logger = Logger.getLogger(LoggingExample.class);
12
13   public static void main(String args[]) {
14     boolean outputToConsole = true;
15
16     SimpleLayout layout = new SimpleLayout();
17     PatternLayout layout = new
18     PatternLayout("%-2r [%5.15t] %-5p %30.30c %x - %m%n");
19
20
21     if (outputToConsole) {
22       logger.addAppender(new ConsoleAppender(layout));
23     } else {
24       FileAppender appender = null;
25       try {
26         appender = new FileAppender(layout, "logs/output.log", false);
27       } catch (Exception e) {
28       }
29
30       logger.addAppender(appender);
31     }
32
33     logger.setLevel(Level.ERROR);
34     // logger.setLevel(Level.DEBUG);
35
36     logger.debug("Here is some DEBUG");
37     logger.info("Here is some INFO");
38     logger.warn("Here is some WARN");
39     logger.error("Here is some ERROR");
40     logger.fatal("Here is some FATAL");
41   }
42 }
```

When we run the code, we get the following output:

```
1 ERROR - Here is some ERROR
2 FATAL - Here is some FATAL
```

The code might appear a little complicated at first but it's really quite simple. We first import the required classes and then initiate the logger (Line 11). Notice that we're passing the class to the `getLogger()` function so that the logger will know where the calls are coming from.

Then, we create a `SimpleLayout` (Line 16) which simply outputs the log level and the message. We can also comment out Line 16 and uncomment Lines 17-18 and if we want to use a `PatternLayout` which outputs a lot more information in a specific format.

We define a condition variable `outputToConsole` which, if we to true will enable the use of `ConsoleAppender`. If disabled, it will select the `FileAppender` – which, in

turn, will write all log messages to the file `logs/output.log`[1].

Next, We set the logging level to `ERROR`. This ensures that we only get log messages which are at the `ERROR` level or are more critical than that. The order of the log levels is `FATAL` > `ERROR` > `WARNING` > `INFO` > `DEBUG`. So, if you set the logging level to `DEBUG`, you get all messages.

All this was the set up. Now, we can use the logs as we see fit. We insert some log messages using the different functions corresponding to the different log levels.

For details of the patterns that you can create, refer to the log4j documentation [3].

## 4.3 External Log Configurations

In the previous section, we put the logging configuration in our source code. This is generally not a good idea because to make modifications, we would need to re-compile the source code. The actual practice is to separate the logging configuration in a separate file (either XML-based or text-based). The following code demonstrates this example.

```
1  package org.csrdu.java.ex;
2
3  import org.apache.log4j.Logger;
4  import org.apache.log4j.PropertyConfigurator;
5
6  public class LoggingExampleExtFile {
7
8    static Logger logger = Logger.getLogger(LoggingExampleExtFile.class);
9
10   public static void main(String args[]) {
11     PropertyConfigurator.configure("src/plainlog4jconfig.txt");
12
13     logger.debug("Here is some DEBUG");
14     logger.info("Here is some INFO");
15     logger.warn("Here is some WARN");
16     logger.error("Here is some ERROR");
17     logger.fatal("Here is some FATAL");
18
19     SubClass sc = new SubClass();
20     sc.subClassFunction();
21   }
22 }
23
24 class SubClass {
25   static Logger logger = Logger.getLogger(SubClass.class);
26
27   public void subClassFunction(){
28     logger.debug("Inside sub-class");
29   }
30 }
```

The semantics are fairly simple. We have two classes. The main class creates the logger and sets the configuration file to `src/plainlog4jconfig.txt`. We can use string externalization to move this configuration file name outside the source as well but for demonstration, we leave it like this. The sub class also creates a new

[1] It is a convention to put all log files in a separate `logs` folder. Also, after this file is created, you might want to right click on the project and hit `Refresh` to see the newly created file.

logger but does not have to set the configuration file as that has already been done. The whole configuration detail sits in the text file, which looks like this:

```
1  log4j.rootLogger=DEBUG
2
3  # MainFileAppender - used to log messages in the main.log file.
4  log4j.appender.MainFileAppender=org.apache.log4j.FileAppender
5  log4j.appender.MainFileAppender.File=logs/main.log
6  log4j.appender.MainFileAppender.layout=org.apache.log4j.PatternLayout
7  log4j.appender.MainFileAppender.layout.ConversionPattern= %-4r [%t] %-5p %c %x - %m%n
8
9  # SubClassFileAppender - used to log messages in the subclass.log file.
10 log4j.appender.SubClassFileAppender=org.apache.log4j.FileAppender
11 log4j.appender.SubClassFileAppender.File=logs/subclass.log
12 log4j.appender.SubClassFileAppender.layout=org.apache.log4j.PatternLayout
13 log4j.appender.SubClassFileAppender.layout.ConversionPattern= %-4r [%t] %-5p %c %x - %m%n
14
15 # Define loggers for different classes
16 log4j.logger.org.csrdu.java.ex.LoggingExampleExtFile=WARN,MainFileAppender
17 log4j.logger.org.csrdu.java.ex.SubClass=DEBUG,SubClassFileAppender
```

Again, this might look complicated but is fairly simple when you break it down. Lines 4–7 define one logger which is a `FileAppender` which outputs to `logs/main.log` and is based on the pattern.

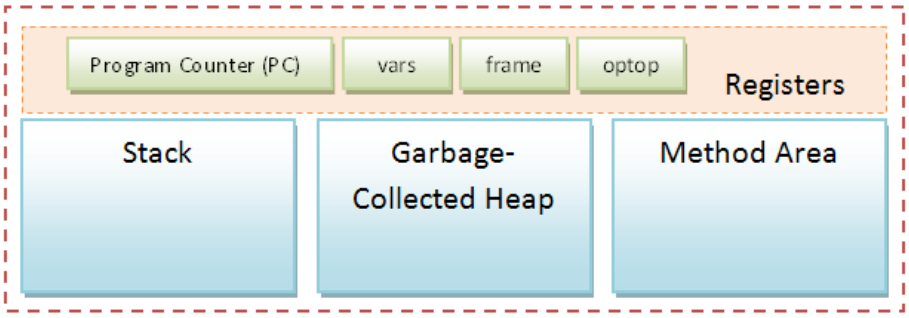
Lines 10–13 is another appender that outputs to a different log file (`logs/subclass.log`). The two appenders are associated with the two classes on Lines 16 and 17 respectively. Each file gets associated with a different appender and thus messages from these end up in different files. You can also define separate appenders for different packages.

Run the file and take a look at the files `logs/main.log` and `logs/subclass.log`. You will notice that each class outputs to their respective files. That allows for very flexible separation of log files based on their source.

## 4.4 Java Memory Optimizing

Java Virtual Machine like its real counter part, executes the program and generate output. To execute any code, JVM utilizes different components.

JVM is divided into several components like the stack, the garbage-collected heap, the registers and the method area. Let us see diagram representation of JVM. [13]



### 4.4.1 The Stack

Stack in Java virtual machine stores various method arguements as well as the local variables of any method. Stack also keep track of each an every method invocation. This is called *Stack Frame*. There are three registers thats help in stack manipulation. They are `vars`, `frame` and *optop*. This registers points to different parts of current Stack.

*Stack Frame*
*optop*

There are three sections in Java stack frame:

### 4.4.2 Local Variables

The local variables section contains all the local variables being used by the current method invocation. It is pointed to by the `vars` register.

### 4.4.3 Execution Environment

The execution environment section is used to maintain the operations of the stack itself. It is pointed to by the `frame` register.

### 4.4.4 Operand Stack

The *operand stack* is used as a work space by bytecode instructions. It is here that the parameters for bytecode instructions are placed, and results of bytecode instructions are found. The top of the operand stack is pointed to by the `optop register`.

*operand stack*

### 4.4.5 Method Area

This is the area where bytecodes reside. The *program counter* points to some byte in the method area. It always keep tracks of the current instruction which is being executed (interpreted). After execution of an instruction, the JVM sets the PC to next instruction. Method area is shared among all the threads of a process. Hence if more then one threads are accessing any specific method or any instructions, synchorization is needed. *Synchronization* in JVM is acheived through *Monitors*.

*program counter*
*Synchronization*
*Monitors*

### 4.4.6 Garbage-collected Heap

The *Garbage-collected Heap* is where the objects in Java programs are stored. Whenever we allocate an object using new operator, the heap comes into picture and memory is allocated from there. Unlike C++, Java does not have the `free` operator to free any previously allocated memory. Java does this automatically using *garbage collection* mechanism. Till Java 6.0, *mark and sweep* algorithm is used as a garbage collection logic. Remember that the local object reference resides on Stack but the actual object resides in Heap only. Also, arrays in Java are objects, hence they also resides in garbage-collected Heap.

*Garbage-collected Heap*
*garbage collection*
*mark and sweep*

### 4.4.7 Memory Optimization Example

We demonstrate how we can manipulate memory allocation and use in Java through the following code:

```java
package org.csrdu.java.ex;
/**
* Options that can be passed to the VM to manipulate memory
* -Xms<size>        set initial Java heap size
* -Xmx<size>        set maximum Java heap size
* -Xss<size>        set java thread stack size
*
* @author Nauman
*
*/
public class GetHeapSize {
  public static void main(String[]args){
    // Get the jvm heap size.
    long heapSize = Runtime.getRuntime().totalMemory();

    //Print the jvm heap size.
    System.out.println("Heap Size = " + heapSize);

    // Print the free memory
    System.out.println("Free Memory = " +
        Runtime.getRuntime().freeMemory());
    System.out.println("Available Processors = " +
        Runtime.getRuntime().availableProcessors());
  }
}
```

First run the program normally. The output looks likes so:

```
Heap Size = 61734912
Free Memory = 61411152
Available Processors = 2
```

*VM Arguments*

Now, we can increase the memory size by going to Run Configurations and changing the *VM Arguments* to `-Xms128m -Xmx256m`. The first param declares the initial heap size and the latter sets the maximum heap size. Run the program with these arguments and you get the following result:

```
Heap Size = 128647168
Free Memory = 127976064
Available Processors = 2
```

CHAPTER 5

# DISTRIBUTED COMPUTING

Some of the most powerful and important concepts of modern computing come from the simple idea of moving computation (or storage) to a remote machine. This simple yet powerful idea of *distributed computing* gives rise to a myriad of new computing paradigms. In this chapter, we take a look at two of these. First, we will talk about *Remote Method Invocation* and then look at an *Apache* project that allows you to write powerful server-side programs with little boilerplate code.

*distributed computing*

*Remote Method Invocation*

*Apache*

## 5.1 Remote Method Invocation

Remote Method Invocation – or *RMI* [7] – enables Java programmers to easily execute code on a remote machine. The basic idea is very simple from a programmer's perspective. We execute a function on a local machine, the actual function call gets transported to a remote machine, executed there and the results returned to our local function as if the function was executed locally. This is depicted in Figure **??**.

*RMI*

This adds immensely to the ease of deploying distributed applications. If you have a processing-intensive task that needs to be run many times, you might want to move it to a more powerful machine. Another use case might be when you have a data store in a central location and want to retrieve data easily without having to worry about writing network programming code.

Below is an example of how to use RMI. First, we will create an interface that acts as a connection that both the server and client will share. This will allow us to write client code that can be compiled without any errors. The following interface goes in a new project `w05-rmi-server`.

```
1 package org.csrdu.java.ex;
2
3 import java.rmi.Remote;
4 import java.rmi.RemoteException;
5
```

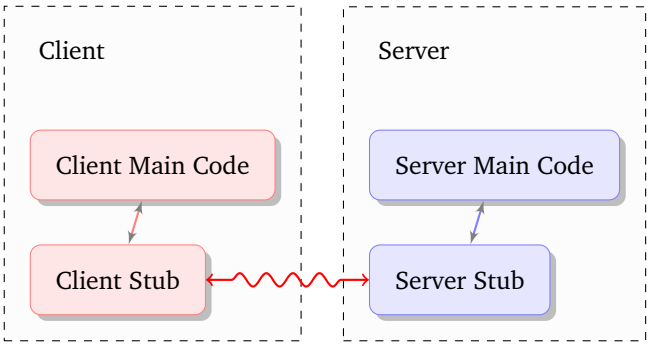



Figure 5.1: Java RMI Architecture

```
6 public interface Hello extends Remote {
7   String sayHello() throws RemoteException;
8   String concatStrings(String s1, String s2) throws RemoteException;
9 }
```

### 5.1.1 Writing the Server

Now, we will create a server that will serve requests coming in for this interface.

```
1 package org.csrdu.java.ex;
2
3 import java.rmi.RemoteException;
4 import java.rmi.registry.Registry;
5 import java.rmi.registry.LocateRegistry;
6 import java.rmi.server.UnicastRemoteObject;
7
8 public class Server implements Hello {
9   public String sayHello() {
10    return "Hello, world!";
11  }
12
13  public static void main(String args[]) {
14    try {
15      Server obj = new Server();
16      Hello stub = (Hello) UnicastRemoteObject.exportObject(obj, 0);
17
18      // Bind the remote object's stub in the registry
19      Registry registry = LocateRegistry.getRegistry();
20      registry.bind("Hello", stub);
21      System.err.println("Server ready");
22    } catch (Exception e) {
23      System.err.println("Server exception: " + e.toString());
24      e.printStackTrace();
25    }
26  }
27
28  @Override
29  public String concatStrings(String s1, String s2)
30    throws RemoteException {
31    return s1.concat(s2);
32  }
33 }
```

There are a couple of things to note here:

*stub*

1. Line 16 exports a *stub* method based on the `Server` class. RMI works around the concept of stubs (see the architecture figure above). This stub had to be created manually prior to Java 5 but this line automates this process.

*bind* *registry*

2. We *bind* our `Hello` server to the local *registry*. This registry acts as a directory listing for the services offered by the current machine.

3. Everything else is taken care of automatically by the stub methods.

### 5.1.2 Writing the Client

To use the server, we need to write a client. For now, we will be running the client on the same machine but we will create a separate project for the client so that it can be moved independently of the server. First, create a new project called

`w05-rmi-client`. Add the same code for the `Hello` interface as the server. Then, add a client class with the following code:

```
1  package org.csrdu.java.ex;
2
3  import java.rmi.registry.LocateRegistry;
4  import java.rmi.registry.Registry;
5
6  public class Client {
7
8    public static void main(String[] args) {
9
10   String host = (args.length < 1) ? null : args[0];
11     try {
12       Registry registry = LocateRegistry.getRegistry(host);
13       Hello stub = (Hello) registry.lookup("Hello");
14       String response = stub.sayHello();
15       System.out.println("response sayHello: " + response);
16       response = stub.concatStrings("First", "Second");
17       System.out.println("response concatStrings: " + response);
18     } catch (Exception e) {
19       System.err.println("Client exception: " + e.toString());
20       e.printStackTrace();
21     }
22   }
23 }
```

As with the server, there are a couple of things to note here:

1. We can provide the `host` of the registry to the client. If our server were on a separate machine, we could provide that as the host and our client code need not be changed at all.
2. We use the registry to get a stub for our client. We can then use this stub for calling any method on the remote machine. It is the responsibility of the stubs and the registry to communicate our calls to the server virtual machine and object.

### 5.1.3 Running the Example

Running the example requires only a little bit of configuration. We need to do a couple of things:

1. **Start the registry:** We need to have the registry service running on our machine so that the server can *register* with it and the client can *look up* services through it. To do this, we can issue the following command on our terminal:

   *register*
   *look up*

   ```
   1  rmiregistry
   ```

2. To start the server, we need to set the *code base location*. Do this by setting a VM argument in your server's run configuration:

   *code base location*

   ```
   -Djava.rmi.server.codebase=file:///[workspace location]/w05-rmi-server/bin/
   ```

   This is the location of your compiled files. When a stub is created, it will be placed automatically in the correct location and this parameter will allow the VM to read it correctly.

   After setting this option, you can start the server.

3. Starting the client is very easy. Just run the client and everything will be taken care of by the code. You should see the following output if everything goes as planned:

```
response sayHello: Hello, World!
response concatStrings: FirstSecond
```

You can now extend the RMI example to pass all sorts of parameters and get all manners of responses from the server. The only issue is that the client needs to be a Java program as well. We'll take care of this limitation in the following section.

## 5.2 Apache Mina

*Apache MINA*

According to the official homepage, "*Apache MINA* is a network application framework which helps users develop high performance and high scalability network applications easily." [2]. In essence, it allows you to write distributed applications that can communicate over the usual *TCP/IP* or *UDP/IP* protocols. You can, for example, write an FTP server using this infrastructure.

*TCP/IP*

*UDP/IP*

We demonstrate the usefulness of MINA through an example. Begin by downloading the binaries for MINA from `http://mina.apache.org/downloads.html`. From the downloaded archive, you need to get two files `dist/mina-core-*.jar` and `lib/slf4j-api-*.jar`. Create a new project and put these in the `referenced` folder of the project. Add both jars to the build path. Now, create the following class that acts as our server:

```
package org.csrdu.java.ex;

import java.io.IOException;

import java.net.InetSocketAddress;
import java.nio.charset.Charset;

import org.apache.mina.core.service.IoAcceptor;
import org.apache.mina.core.session.IdleStatus;
import org.apache.mina.filter.codec.ProtocolCodecFilter;
import org.apache.mina.filter.codec.textline.TextLineCodecFactory;
import org.apache.mina.filter.logging.LoggingFilter;
import org.apache.mina.transport.socket.nio.NioSocketAcceptor;

public class MinaTimeServer {
  private static final int PORT = 9123;

  public static void main(String[] args) throws IOException {
    IoAcceptor acceptor = new NioSocketAcceptor();

    acceptor.getFilterChain().addLast("logger", new LoggingFilter());
    acceptor.getFilterChain().addLast("codec",
      new ProtocolCodecFilter(new TextLineCodecFactory(
        Charset.forName("UTF-8")
      ))
    );

    acceptor.setHandler(new TimeServerHandler());
    acceptor.getSessionConfig().setReadBufferSize(2048);
    acceptor.getSessionConfig().setIdleTime(IdleStatus.BOTH_IDLE, 10);
    acceptor.bind(new InetSocketAddress(PORT));
  }
}
```

The main method begins by creating an `IoAcceptor`. This acceptor receives data from the network interface (on the port specified on Line 31). We set the logger for the acceptor and add a *codec*. This codec will decode all incoming messages. Since we will be using a command-line interface, this is a `TextLineCodecFactory` object which works on *UTF-8* character set. We then set the handler which will be passed all messages that are received – this is a custom class that we define below. Finally, we set the buffer size and idle timeout and start the server.

*codec*

*UTF-8*

The `TimeServerHandle` class used above is defined as:

```
1  package org.csrdu.java.ex;
2
3  import java.util.Date;
4
5  import org.apache.mina.core.session.IdleStatus;
6  import org.apache.mina.core.service.IoHandlerAdapter;
7  import org.apache.mina.core.session.IoSession;
8
9  public class TimeServerHandler extends IoHandlerAdapter {
10   @Override
11   public void exceptionCaught(IoSession session, Throwable cause)
12     throws Exception {
13     cause.printStackTrace();
14   }
15
16   @Override
17   public void messageReceived(IoSession session, Object message)
18   throws Exception {
19     String str = message.toString();
20     if (str.trim().equalsIgnoreCase("quit")) {
21     session.close(true);
22       return;
23     }
24
25     Date date = new Date();
26     session.write(date.toString());
27     System.out.println("Message written...");
28   }
29
30   @Override
31   public void sessionIdle(IoSession session, IdleStatus status)
32   throws Exception {
33     System.out.println("IDLE " + session.getIdleCount(status));
34   }
35 }
```

The code is very straight-forward. Whenever a message is received, it checks whether the command is `quit`. If so, it closes the session. Otherwise, it returns the current date. To run this example, we can initiate a session through telnet on the given port and issue some commands. A sample session follows:

```
1  nam@temp:~$ telnet localhost 9123
2  Trying 127.0.0.1...
3  Connected to localhost.
4  Escape character is '^]'.
5  hello
6  Tue Feb 14 10:29:58 PKT 2012
7  another command
8  Tue Feb 14 10:30:02 PKT 2012
9  quit
10 Connection closed by foreign host.
```

CHAPTER 6

# XML PROCESSING

XML is an important technology by anyone's standards. It powers many of the modern technologies and you have to know how to process XML if you want to work with software of any value in modern industry. In this chapter, we are going to talk about three different methods of parsing XML and turning it into a format that our programs can reason with. We begin with the most rudimentary approach.

## 6.1 Simple API for XML

Simple API for XML (or *SAX*) is the most basic (and oftentimes, most efficient) way of processing XML. It is an *event-based parser* – it starts processing the XML from top-down and generates events as it reads different portions of the file. Examples of events are `START_TAG` and `END_TAG`. Based on the different events (and their sequences), we can decide exactly what data is coming in to us. Let's take a look at an example to understand SAX.

*SAX*

*event-based parser*

### 6.1.1 A SAX Example

Let's consider an XML file that describes a mathematical expression. A basic version of this file (which we will save as `data/expression.xml`) looks like this:

```
<?xml version="1.0"?>
<expression>
  <expr func="+">
    <expr func="*">
      <const val="2" />
      <const val="3" />
    </expr>
    <expr func="-">
      <const val="4" />
      <const val="5" />
    </expr>
    <const val="6" />
  </expr>
</expression>
```

This file corresponds to the expression $(2*3)+(4-5)+6$. It's very verbose and if we want to do something useful with this file (such as calculate the result), we need to be able to represent it in some meaningful Java object format.

The first step in doing that would be to define a class `Expresion` in our project.

```
package org.csrdu.java.ex.xml.expression.model;

import java.util.Vector;

public class Expression {
  public String function = "";
  public Vector operands;

```

```
9   public Expression() {
10    operands = new Vector();
11  }
12
13  public void printExpression(int level) {
14      System.out.println(nSpaces(level * 2) + "Function:  [" + function + "]");
15
16      for (Object i : operands) {
17        if (i instanceof Integer)
18          System.out.println(nSpaces(level * 2 + 2) + i);
19        else
20          ((Expression)i).printExpression(level+1);
21      }
22  }
23
24  public String nSpaces(int length) {
25    StringBuffer outputBuffer = new StringBuffer(length);
26    for (int i = 0; i < length; i++) {
27      outputBuffer.append(" ");
28    }
29    return outputBuffer.toString();
30  }
31 }
```

The class itself is very simple. It has a field called `function` to store the function name of the expression and a `Vector` of `Expression` objects as the operands. This is needed as the operands can be both constants and other expressions. In case one of the operands is a constant, the `function` field is ignored and the only `operand` will be an `Integer`.

The `printExpression()` function simply outputs the expression in a properly indented format (the indentation coming from the `nSpaces()` function.

*model* *controller*

Now that we have a *model*, we can define a *controller* for it that reads the data from the XML file and creates an instance of the expression in Java object code.

```
1  package org.csrdu.java.ex.xml.expression.controller;
2
3  import org.csrdu.java.ex.xml.expression.model.Expression;
4  import org.xml.sax.*;
5  import org.xml.sax.helpers.*;
6  import java.io.*;
7  import java.util.*;
8
9  public class ExpressionParser extends DefaultHandler {
10
11   Stack<Expression> st;
12
13   Expression curExpr;
14
15   public ExpressionParser() {
16     st = new Stack<Expression>();
17   }
18
19   // Override methods of the DefaultHandler class
20   // to gain notification of SAX Events.
21   //
22   // See org.xml.sax.ContentHandler for all available events.
23   //
24   public void startElement(String namespaceURI, String localName,
25   String qName, Attributes attr) throws SAXException {
26     if (localName.equals("expr")) {
```

```
27        // we have a new expression.
28        // push the old one on the stack (if any)
29        if (curExpr != null)
30        st.push(curExpr);
31
32        // We need to create a new expression.
33        curExpr = new Expression();
34
35        // and set the function for this expression
36        curExpr.function = attr.getValue("func");
37
38      } else if (localName.equals("const")) {
39        // we have a constant. Let's add it to the expression
40        // first get the value of the constant
41        Integer val = Integer.parseInt(attr.getValue("val"));
42        curExpr.operands.add(val);
43      }
44    }
45
46    public void endElement(String namespaceURI, String localName, String qName)
47      throws SAXException {
48      if (localName.equals("expr")) {
49        // End of current expression
50        // save the expression just ended
51        Expression endedExpr = curExpr;
52        // pop the current expression from the stack
53        if (!st.empty()) {
54          curExpr = st.pop();
55          // add ended expression to parent
56          curExpr.operands.add(endedExpr);
57        }
58      }
59      // we can ignore the end of const tag
60    }
61
62    public static void main(String[] argv) {
63      try {
64        // Create SAX parser...
65        XMLReader xr = XMLReaderFactory.createXMLReader();
66
67        // Set the ContentHandler...
68        ExpressionParser exprs = new ExpressionParser();
69        xr.setContentHandler(exprs);
70
71        // Parse the file...
72        xr.parse(new InputSource(new FileReader("data/expression.xml")));
73        exprs.curExpr.printExpression(0);
74
75      } catch (Exception e) {
76        e.printStackTrace();
77      }
78    }
79  }
```

Let's begin the explanation by describing the `main()` function. It sets up the `XMLReader` which is simply a SAX *parser*. We then create an instance of our `ExpressionParser` class and parse it to the XMLReader as the *handler*. This means that whenever the parser finds an "important piece" of the XMl, it will generate an event which will be handled by our handler. This allows our code to be notified whenever an XML event (such as start of tag) occurs.

*parser*

*handler*

**Algorithm 1** The `startElement` event

```
1: if event source = expr then
2:     push old expression on stack (if any)
3:     create a new expression object
4:     set function of new expression to attribute 'func'
5: else if event source = const then
6:     add attribute 'val' as operand to current expression
7: end if
```

**Algorithm 2** The `endElement` event

```
1: if event source = expr then
2:     save the expression just ended as endedExpr
3:     if stack is not empty then
4:         pop the expression at the top of stack as curExpr
5:         add endedExpr as an operand of curExpr
6:     end if
7: end if
```

*Plain Old Java Object (POJO)*

We then parse the XML file (`data/expression.xml`). This will convert our XML file into a *Plain Old Java Object (POJO)*. We can then print it out or do whatever we need to do with it.

All we have to understand now is how our `ExpressionHandler` handles the events. This is covered in two functions:

#### startElement()

This function defines the majority of our logic. The algorithm for handling the *startElement* event is given in Algorithm 1. We first check if the event was raised for `expr` tag. If so, we know that we are starting a new expression. So, we push the old one on the stack (if one exists) and create a new expression tag. Any constant we see from now on will be an operand of this new expression. We also get the `func` attribute of the expression and set the function name equal to it.

If, on the other hand, we have this event raised for `const`, we know that we got a constant and we can take it's `val` attribute and set it as an operand of the current expression.

The other half of this logic comes into place when we look at what happens when a tag ends.

#### endElement()

When the `const` tag ends, we don't care because it's can't be a parent tag. What we are interested in is when the `expr` tag ends (cf. Algorithm 2). When that happens, we know that the current expression has ended. We need to pop the (parent) expression that we pushed on the stack and then add the expression that just ended to the parent. And that takes care of the whole logic!

If you have trouble understanding, Figure 6.1 might be of help. Refer to this in conjunction with the algorithm and the code.

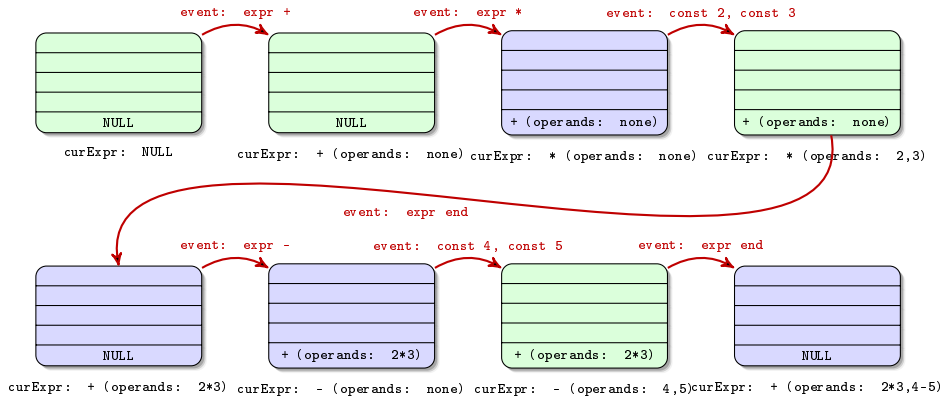



Figure 6.1: SAX Events Dry Run

### 6.1.2 Limitations of SAX

SAX is often very efficient but it has a major limitation. It does not allow random access. You have to parse the whole file to get to a particular piece of information. This becomes very problematic if you have a complicated file and you want to retrieve a particular piece of information from it. One way of solving this problem is through the use of a tree model of XML parsing. We discuss one of these below.

## 6.2 Document Object Model

Document Object Model (more commonly known as *DOM*) is a *tree-based parser* for XML. It allows you to traverse the XML document as if it were a tree (which it really is). The best way to understand DOM is by looking at a simple example. We are going to parse the same expression file that we parsed using SAX. Let's first take a look at the code.

*DOM*
*tree-based parser*

```
1  package org.csrdu.java.ex.xml.expression.controller;
2
3  import javax.xml.parsers.DocumentBuilderFactory;
4  /* some imports hidden */
5
6  public class ExpressionParserDom {
7
8    public static void main(String argv[]) {
9      try {
10       File file = new File("data/expression.xml");
11       DocumentBuilderFactory dbFactory = DocumentBuilderFactory
12       .newInstance();
13       DocumentBuilder dBuilder = dbFactory.newDocumentBuilder();
14       Document doc = dBuilder.parse(file);
15       doc.getDocumentElement().normalize();
16
17       Element rootElement = doc.getDocumentElement();
18       NodeList nodes = rootElement.getChildNodes().item(1).getChildNodes();
19
20       for (int i = 0; i < nodes.getLength(); i++) {
21         Node node = nodes.item(i);
```

```
22
23          if (node instanceof Element) {
24            // a child element to process
25            Element child = (Element) node;
26            if (child.getTagName() == "expr") {
27              String attribute = child.getAttribute("func");
28              System.out.println("Expression with function ID "
29                  + attribute + " found.");
30            } else if (child.getTagName() == "const"){
31              String attribute = child.getAttribute("val");
32              System.out.println("Constant with value "
33                  + attribute + " found.");
34            }
35          }
36        }
37      } catch (Exception e) {
38        e.printStackTrace();
39      }
40    }
41  }
```

*document builder factory*

First, we create what is called a *document builder factory*. This object can build a document by parsing a file (or string). We then normalize the document to get it into a standardized format. After that, it's fairly simple. We can get the root element of the XML file (using the `getDocumentElement()` function). This returns an object of type `Node`. All tags and text in DOM are *nodes*. When we have a node object, we can get its child nodes using the `getChildNodes()` function. This returns an *iterable* collection. We can loop over this collection and use the different functions.

*nodes*

*iterable*

## 6.3 XML Pull Parser

*XML Pull Parser API*

One of the more recent parsers has been proposed in the form of *XML Pull Parser API* [16]. It's a fast implementation that works (at least on the face), similar to SAX. Instead of relying on the push parsing of SAX, the XML Pull Parsers actively poll the document for events. This leads to a faster parsing performance. The XML Pull API is used extensively in the Android framework. However, an implementation of this API is not available in vanilla JREs. In this example, we will use the reference implementation[1] of this API provided by the kXML project [8].

Below, we provide code for parsing our expression XML through this pull parser.

```
1  package org.csrdu.java.ex.xml.expression.controller;
2
3  import java.io.FileReader;
4  import java.io.IOException;
5  import org.xmlpull.v1.XmlPullParser;
6  import org.xmlpull.v1.XmlPullParserException;
7  import org.xmlpull.v1.XmlPullParserFactory;
8
9  /**
10 * An example of an application that uses XMLPULL V1 API.
11 *
12 * @author <a href="http://www.extreme.indiana.edu/~aslom/">Aleksander
13 *         Slominski</a>
```

[1] You can download the required JAR file from the lecture server or from the kXML site given in the references.

```
14 */
15 public class ExpressionParserXMLPull {
16   public static void main(String args[])
17     throws XmlPullParserException, IOException {
18     XmlPullParserFactory factory = XmlPullParserFactory.newInstance();
19     factory.setNamespaceAware(true);
20     XmlPullParser xpp = factory.newPullParser();
21     System.out.println("Parser implementation class is "
22         + xpp.getClass());
23
24     ExpressionParserXMLPull app = new ExpressionParserXMLPull();
25
26     System.out.println("Parsing file. ");
27     xpp.setInput(new FileReader("data/expression.xml"));
28     app.processDocument(xpp);
29   }
30
31   public void processDocument(XmlPullParser xpp)
32     throws XmlPullParserException, IOException {
33     int eventType = xpp.getEventType();
34     do {
35       if (eventType == xpp.START_DOCUMENT) {
36         System.out.println("Start document");
37       } else if (eventType == xpp.END_DOCUMENT) {
38         System.out.println("End document");
39       } else if (eventType == xpp.START_TAG) {
40         processStartElement(xpp);
41       } else if (eventType == xpp.END_TAG) {
42         processEndElement(xpp);
43       } else if (eventType == xpp.TEXT) {
44         processText(xpp);
45       }
46       eventType = xpp.next();
47     } while (eventType != xpp.END_DOCUMENT);
48   }
49
50   public void processStartElement(XmlPullParser xpp) {
51     String name = xpp.getName();
52     System.out.println("Start element: " + name);
53   }
54
55   public void processEndElement(XmlPullParser xpp) {
56     String name = xpp.getName();
57     System.out.println("End element: " + name);
58   }
59
60   int holderForStartAndLength[] = new int[2];
61
62   public void processText(XmlPullParser xpp) throws XmlPullParserException {
63     char ch[] = xpp.getTextCharacters(holderForStartAndLength);
64     int start = holderForStartAndLength[0];
65     int length = holderForStartAndLength[1];
66     System.out.print("Characters:    \"");
67     for (int i = start; i < start + length; i++) {
68       if (ch[i] != '\n' && ch[i] != '\t')
69       System.out.println(ch[i]);
70     }
71     System.out.print("\"\n");
72   }
73 }
```

The developer-facing semantics of XML Pull Parser are approximately the same

*factory design pattern*

as SAX. We first create an instance of the parser. This is done using the *factory design pattern*. We can then use our handler to parse this file. The most important line in our handler is the line `xpp.next()` (Line 46). It pulls the next event (hence the name *pull* parser). We can then decide which even it is and call the relevant function based on that. The only difference here from the SAX parser is that we're testing for basic text as well. This is the portion of characters that fall within the element starting and ending tags. This code can be modified to use the stack-based semantics we used in our SAX example.

CHAPTER 7

# XPATH AND XQUERY

In the previous chapter, we processed XML using parsers. While these are easy to implement (given you know about parsing and transformation etc.), they're very time consuming and bug-prone. Imagine what would happen if you want to extract an attribute of the fifth child of the second sibbling of a node with name `Book` whose author is Ali. This might sound like a contrived example but it's really not. Because XML is so verbose, it's usually necessary to do this kind of parsing to retrieve a single relevant attribute from a small XML document. Now consider how much code will have to be written just to get this one attribute – not a pretty picture if you're dealing with the average programmer. Deadlines will be missed, bugs will be introduced and in general, chaos would ensue!

Of course, there are easier ways to get this attribute from an XML document. It might not be as efficient as SAX but it would require very little coding time and would be fairly bug-free. Given small enough documents, it might even be faster than raw processing. In this chapter, we're going to take a look at two techniques that help us retrieve data from an XML document using high-level languages – *XPath* and *XQuery*.

*XPath*
*XQuery*

## 7.1 XPath

XML Path Language (XPath) is a specificatio nof W3C which provides syntax and semantics for a non-XML language that allows retrieval of data from XML documents [14]. What that means, simply, is that XPath itself is not XML. You don't have to write many tags simply to define what you need – that makes XPath very concise unlike XML itself which is very verbose. For example, the following XPath query retrieves the node corresponding to the second section from the fifth chapter in the book (assuming doc is the root node in the XML document):

```
1 /book/chapter[5]/section[2]
```

So, let's begin by creating an XML document that we want to retrieve the information from. Let's call this document `departments.xml` and place it in the `src` folder of our project. The document is fairly simple: it's a collection of departments each with an id (represented as an attribute), a name and a group name.

```
1  <?xml version="1.0" encoding="ISO-8859-1"?>
2  <Departments>
3          <Department id='1'>
4                  <Name>Engineering</Name>
5                  <GroupName>Research and Development</GroupName>
6          </Department>
7          <Department id='2'>
8                  <Name>Tool Design</Name>
9                  <GroupName>Research and Development</GroupName>
10         </Department>
11         <Department id='3'>
12                 <Name>Sales</Name>
```

```
13                <GroupName>Sales and Marketing</GroupName>
14        </Department>
15        <Department id='5'>
16     <Name>Purchasing</Name>
17     <GroupName>Inventory Management</GroupName>
18   </Department>
19        <Department id='4'>
20                <Name>Marketing</Name>
21                <GroupName>Sales and Marketing</GroupName>
22        </Department>
23 </Departments>
```

### 7.1.1 A Basic XPath Evaluator

In order to use XPath, we will need to have some code that can read XML document and run XPaths on them. Java provides both an XPath API and an implementation in native JRE so we don't have to include any library for this purpose. We can use the classes in `javax.xml.xpath` package for this purpose. Let's write a runner for XPaths.

```
1  package org.csrdu.java.ex;
2
3  import org.w3c.dom.*;
4  import javax.xml.xpath.*;
5  import javax.xml.parsers.*;
6  import java.io.IOException;
7  import org.xml.sax.SAXException;
8
9  public class XPathRunner {
10
11     public static void main(String args[])
12          throws ParserConfigurationException,
13            SAXException, IOException, XPathExpressionException {
14
15         String xpathstr = "---- XPATH HERE ----";
16
17
18         // Create an XML document from a file and load it
19         DocumentBuilderFactory domFactory = DocumentBuilderFactory
20                 .newInstance();
21         domFactory.setNamespaceAware(false);
22         DocumentBuilder builder = domFactory.newDocumentBuilder();
23         Document doc = builder.parse("src/departments.xml");
24         XPath xpath = XPathFactory.newInstance().newXPath();
25
26         // Parse the XPath expression
27         XPathExpression expr = xpath.compile(xpathstr);
28
29         // Get the results and display them
30         Object result = expr.evaluate(doc, XPathConstants.NODESET);
31         NodeList nodes = (NodeList) result;
32         for (int i = 0; i < nodes.getLength(); i++) {
33             System.out.println(nodes.item(i).getNodeValue());
34         }
35     }
36 }
```

First, we create an XML document using `DocumentBuilderFactory`. This is the same class we used back in Section 6.2. We also set the factory to ignore *namespaces*.

*namespaces*

Line 23 defines which file contains the XML we want to read and Line 24 creates an instance of the XPath engine.

Before we can execute an XPath, it needs to be compiled. We do this in Line 27 and then evaluate the given XPath on the document we just read in Line 30. This will return a set of `Node` objects over which we can loop and get the data from the specific nodes using the usual DOM syntax.

Now, we're ready to run some XPaths on our documents using this runner.

### 7.1.2 Basic XPath Example

Let's begin by changing the `xpathstr` to the following:

```
1 //Department[@id = '1']/Name/text()
```

After execution, the runner returns:

```
1 Engineering
```

Let's dissect this expression: The first `//Department` matches a `Department` tag *anywhere* in the document. If we replace that with a single `/`, it would mean that we are looking for an absolute path from the root node of the document. In that, there will be no result as there is no `Department` tag in the root of our XML document.

Next, the square brackets denote a *predicate*. The syntax is `location[ predicate ]` – the predicate acts as a filter an removes any nodes returned that do not satisfy it. The predicate here is `@id='1'`, the `@` denoting the fact that we're looking for the value of an attribute – `id` in this case. Therefore, the predicate denotes that the evaluator should return only those `Department` nodes which have an attribute `id` *and* whose value is exactly the string `'1'`. Note that if we have a predicate `@id=''`, it means match those nodes which have an empty id attribute. `@id` alone would mean that there is an `id` attribute regardless of what value it has.

*predicate*

After that, we have a `/` which essentially goes one level below on all nodes returned by the expression up to this point. It searches for all child nodes of the nodes returned up to this point and returns only those nodes which correspond to `Name` tags. So, the expression now means, get all `Name` nodes inside `Department` nodes anywhere in the document having their `id` attribute equal to `1`. Of course, this returns an XML node but what we need is the text between the tags. To get this text, we execute the `text()` function on this node and that returns just the text contained within the starting and ending tags.

Phew! That was a lot of explanation but if you go back and read the XPath expression, you will see that it makes logical sense and is fairly easy to read (and write). The learning curve might be slightly steep but it gets very easy with very little practice.

### 7.1.3 Returning Multiple Nodes

You can see that there is nothing stopping the XML document from returning more than one `Name` nodes which match the above criteria. It will be up to us to define our XPath expressions carefully to get only the data that we need. Let's take a look at a concrete example now, where want not only the name but also the group name of the department with `id` equal to 1. Here's the XPath for this:

```
1 //Department[@id = '1']/*/text()
```

The only difference between this XPath and the previous one is that now we are retrieving *all* children of the `Department` node. This gets us both `Name` and `GroupName` nodes and the `text()` returns texts within both tags. So, here the evaluator is returning two strings to us. Running this XPath therefore outputs the following:

```
Engineering
Research and Development
```

### 7.1.4 Intermediate XPath Expressions

We are not restricted to creating predicates using attributes alone. We can define predicated based on child nodes and even combine predicates using *boolean operators* (`and` and `or`). Let's take a look at an example:

*boolean operators*

```
//Department[Name/text()='Engineering']/*/text()
```

The predicate now reads, "match all nodes which have a child called `Name` whose text is equal to `'Engineering'`". In our XML document, there is exactly one such department (i.e. the same one returned in the previous XPath expressions). Once this department is returned, we again retrieve all its children and then output their text. Notice the difference between the use of predicates and location paths here. The predicate restricts the given nodes just as a location does but it does not traverse inside the paths i.e. your pointer (the official name is *XPointer*) remains on the current node. Make sure you understand this difference.

*XPointer*

#### Using Axes

It is also possible to retrieve nodes based on tree relationship with the currently selected node. These relationship specifications are called *axes* in XPath. An axis specifies the relationship type. Some of the axes are `child`, `parent`, `ancestor`, `following-sibbling` and `preceding`.

*axes*

Let's explain the concept of axes using an example. Plug the following XPath expression in the runner:

```
//Department[@id = '2']/following-sibling::*/@id
```

Using this expression, we retrieve the `id` attribute of all the *following siblings* of the `Department` with `id='2'`. The `following-sibling::*` means "all the following siblings irrespective of which tag they are (`*`)". This, of course, is going to return:

```
3
5
4
```

If we need only the first following sibling, we can restrict it as usual using the index notation:

```
//Department[@id = '2']/following-sibling::*[1]/@id
```

This will return only `3`.

The last example we have is to retrieve the parent of a node:

```
//Department/Name[text()='Engineering']/parent::*/@id
```

This translates to "get me the id attribute of the parent of the department *name* node the text of which is equal to Engineering". So, it first retrieves the `Name` node (inside of a `Department` node), then filters only those nodes which have the text equal to `Engineering`. It then retrieves the parent of this node (irrespective of what the parent tag is but we already know that it's going to be `Department`). Finally, it retrieves the `id` attribute of this department.

That concludes our discussion about XPath but there a couple of limitations. For example, what if you want to sort the nodes returned based on a specific attribute? This is not possible with XPath but it's possible to do such operations using another technology called *XQuery*.

*XQuery*

## 7.2 XQuery

XQuery [15] is a standard language for intelligently querying XML data sources. It's a flexible, easy-to-read non-XML language. The detailed discussion of the components of XQuery is beyond the scope of these notes but we will include the details of how to set up and use some basic examples in XQuery. For this purpose, we will use two XQuery implementations: *MX Query* and *DataDirect XQuery*. The reason we chose two different implementations of XQuery is that there are no standard ones available from Oracle and the non-standard ones differ in their functionality and use. We therefore provide two flavours of what is possible. Let's begin by talking about MX Query.

*MX Query*

*DataDirect XQuery*

### 7.2.1 MX Query

MX Query is an opensource implementation of a subset of the XQuery standard. You can download it from `mxquery.org`. Add the mxquery.jar to a new project and add the following code to your main class.

```
1  package org.csrdu.java.ex;
2
3  import javax.xml.namespace.QName;
4  import javax.xml.xquery.*;
5
6  import ch.ethz.mxquery.xqj.MXQueryXQDataSource;
7
8  public class XQueryRunner {
9    public static void main(String args[]) throws XQException{
10     String query = "declare variable $ext external; (14 + $ext, $ext * $ext)";
11
12     XQDataSource ds = new MXQueryXQDataSource();
13     XQConnection conn = ds.getConnection();
14
15     System.out.println("Running query: " + query);
16
17     XQPreparedExpression exp = conn.prepareExpression(query);
18     exp.bindInt(new QName("ext"), 7, null);
19     XQResultSequence result = exp.executeQuery();
20     XQSequence sequence = conn.createSequence(result);
21
22     sequence.writeSequence(System.out, null);
23
24     result.close();
25     sequence.close();
26   }
27 }
```

The first thing to note in the code is the actual query. It declares a variable `ext` (the value of which will be set later in the code) and then outputs `14+ext` and `ext*ext` as a *sequence* (which in this case is a pair).

*sequence*

We then create a data source, which is going to be a `MXQueryXQDataSource`. Notice that the implementation of the data source is dependent on MX Query. If we were using another XQuery implementation, this might have been different.

Then we create a connection that will pull the data from the data source. We create a prepared expression from the query we wrote earlier and then bind the `ext` variable to value `7`.

When we execute the query, we get the result in a `XQResultSequence` object which we then use to get a sequence. Output of this sequence is made available through the `writeSequence()`. The result produced by the code is:

```
1 (21, 49)
```

Let's run another example by replacing the query above with the following:

```
1 declare variable $ext external; ($ext , current-time())
```

Here we're using a function to get the system's current time. The result is once again a sequence which has the bound variable and the current time as it's first and second elements. Finally, let's see if we can take the sequence as input and work with it:

```
1 for $seq in (1,2,3,4,5) where $seq mod 2 eq 0 return $seq
```

The query is going to loop over the given sequence, binding `$seq` to each element in turn. It then outputs the element only if the division of element by 2 returns 0 (i.e. if the `mod` operator is *equal to* 0).

When you run this code, you will get an exception. The reason is that our code is trying to bind a variable – `ext` – whereas no such variable exists in our xquery. To fix this problem, comment out the following line from the code:

```
1 exp.bindInt(new QName("ext"), 7, null);
```

### 7.2.2 Manipulating XML Input

Our second example with MXQuery is slightly more involved. It takes as input some XML and returns two copies of the XML enclosed in the root tag `<result>`. Our query would look like so:

```
1 declare variable $input external;
2   <result>{$input,$input}</result>
```

The code required to run this query, however is slightly more detailed than our previous example. Create a new main class with this code:

```
1 package org.csrdu.java.ex;
2
3 import java.io.IOException;
4 import javax.xml.xquery.XQException;
5 import ch.ethz.mxquery.contextConfig.CompilerOptions;
6 import ch.ethz.mxquery.contextConfig.Context;
7 import ch.ethz.mxquery.datamodel.QName;
```

```
8  import ch.ethz.mxquery.exceptions.MXQueryException;
9  import ch.ethz.mxquery.exceptions.QueryLocation;
10 import ch.ethz.mxquery.model.XDMIterator;
11 import ch.ethz.mxquery.query.PreparedStatement;
12 import ch.ethz.mxquery.query.XQCompiler;
13 import ch.ethz.mxquery.query.impl.CompilerImpl;
14 import ch.ethz.mxquery.util.StringReader;
15 import ch.ethz.mxquery.xdmio.XDMInputFactory;
16 import ch.ethz.mxquery.xdmio.XDMSerializer;
17 import ch.ethz.mxquery.xdmio.XMLSource;
18
19 public class XQueryRunner2 {
20   public static void main(String args[])
21     throws XQException, IOException, MXQueryException {
22     String query = "declare variable $input external; <result>{$input,$input}</result>";
23
24     // Create new (unified) Context
25     Context ctx = new Context();
26
27     // create a XQuery compiler
28     XQCompiler compiler = new CompilerImpl();
29     PreparedStatement statement;
30
31     try {
32       statement = compiler.compile(ctx, query, new CompilerOptions());
33       XDMIterator result;
34
35       String strResult = "";
36
37       result = statement.evaluate();
38
39       // Add the contents of the external variable
40       String xml = "<elem id='1'/>";
41       XMLSource xmlIt = XDMInputFactory.createXMLInput(
42       result.getContext(), new StringReader(xml), true,
43       Context.NO_VALIDATION, QueryLocation.OUTSIDE_QUERY_LOC);
44
45       statement.addExternalResource(new QName("input"), xmlIt);
46
47       XDMSerializer ip = new XDMSerializer();
48       strResult = ip.eventsToXML(result);
49       statement.applyPUL();
50       statement.close();
51
52       System.out.println(strResult);
53     } catch (MXQueryException err) {
54       MXQueryException.printErrorPosition(query, err.getLocation());
55       System.err.println("Error:");
56     }
57   }
58 }
```

We first create a context (which essentially describes the environment where the query will be executed – i.e. inputs and bound variables etc.). Next, we create a compile for our query and pass it our prepared statement as before.

Lines 41–47 are new and warrant a discussion. Here's we first create a basic string which has our input XML. Then then create an XML source from it using the input factory provided by MXQuery. Last, we add this output XML source to our prepared statement as input. Our variable `$input` will therefore get bound to the XML `<elem id='1' />`. Lastly, we output the produced XML using a serializer.

One important point to note here is that we evaluate the XQuery expression before we even create the input variable. It is therefore possible to do late binding in XQuery.

## 7.3 DataDirect XQuery

For our second case study, we consider a much more popular (albeit commercial) implementation of XQuery. This implementation comes from DataDirect and is available at `http://xquery.com`. Download the 15-day trial version from the site. To begin installation, simply extracting the JAR doesn't work – it needs to be executed. From a console, issue the following command and follow the on-screen instructions to complete installation.

```
java -jar datadirectxquery.jar
```

During the installation, take note of where this implementation of XQuery is installed. Now, create a new project and add the `lib/ddxqj.jar` to the build path. To do this, right-click on the project, `Edit Build Path` and then in the libraries tab, click on `Add External JAR`.

Now, import the `departments.xml` file created in the previous section into the project. The main processing class looks like the following:

```
package org.csrdu.java.ex;

import javax.xml.namespace.QName;
import java.util.Properties;

import com.ddtek.xquery3.XQConnection;
import com.ddtek.xquery3.XQException;
import com.ddtek.xquery3.XQExpression;
import com.ddtek.xquery3.XQItemType;
import com.ddtek.xquery3.XQSequence;
import com.ddtek.xquery3.xqj.DDXQDataSource;

public class XQueryRunner3 {

  // Filename for XML document to query
  private String filename;

  // Data Source for querying
  private DDXQDataSource dataSource;

  // Connection for querying
  private XQConnection conn;

  public XQueryRunner3(String filename) {
    this.filename = filename;
  }

  public void init() throws XQException {
    dataSource = new DDXQDataSource();
    conn = dataSource.getConnection();
  }

  public String query(String queryString) throws XQException {
    XQExpression expression = conn.createExpression();
    expression.bindString(new QName("docName"), filename,
    conn.createAtomicType(XQItemType.XQBASETYPE_STRING));
```

```
37      XQSequence results = expression.executeQuery(queryString);
38      return results.getSequenceAsString(new Properties());
39    }
40
41    public static void main(String[] args) {
42
43      try {
44        String xmlFilename = "departments.xml";
45        XQueryRunner3 tester = new XQueryRunner3(xmlFilename);
46        tester.init();
47
48        final String sep = System.getProperty("line.separator");
49        String queryString = "declare variable $docName as xs:string external;"
50        + sep
51        + " for $cd in doc($docName)/Departments/Department "
52        + "       where $cd/@id = '4' "
53        + "         or $cd/@id='5'  "
54        // + " order by $cd/@id "
55        + "return "
56        + "<dept><name>{$cd/Name/text()}</name>"
57        + " <group>{$cd/GroupName/text()}</group></dept>";
58        System.out.println(tester.query(queryString));
59
60      } catch (Exception e) {
61        e.printStackTrace(System.err);
62         System.err.println(e.getMessage());
63      }
64    }
65  }
```

Most of the code is self-explanatory. It initializes some objects and sets up the XQuery runner. The important part of the code is the query. Let's separate that out and try to explain it piece-by-piece:

```
1  for $cd in doc($docName)/Departments/Department
2    where $cd/@id = '4'
3      or $cd/@id = '5'
4      order by $cd/@id
5  return
6  <dept><name>{$cd/Name/text()}</name>
7  <group>{$cd/GroupName/text()}</group></dept>
```

We begin by initializing a variable for a nodeset `cd` which is a result of executing the XPath `/Departments/Department` on the document enclosed in the file pointed to by the variable `docName`. (We set this variable in Line 25 of the Java code above.)

Then, this nodeset is filtered based on a *where clause* which requires that the attribute `id` of the node be equal to `4` or `5`. The nodes are then ordered by their IDs. *where clause*

The actual return value of the XQuery is an XML structure which has <dept> as its root node, which has two child nodes, name and group name. This XML is returned for each matching `cd`. You can try changing the where clause and the *order-by clause* to get different results. *order-by clause*

CHAPTER 8

# AUTOMATING TASKS WITH ANT

Compilation of Java programs from within Eclipse is very straight-forward and time-saving. However, there are cases when you need a lot more done than just compilation. Imagine a scenario where after the successful compilation of code, the output classes should be bundled into a jar file and the jar file should be copied over to the production server. This is a very routine task and needs to be somehow automated.

Another situation is one in which the code is not compiled from within Eclipse. If you have a large piece of software that requires some configurations for compilation and deployment. In this case, you need a method of telling the builder/installer where the files should be compiled and where the output binaries should be placed. All this (and more) can be done using *Apache ANT* [1].[1]

*Apache ANT*

Let's begin by looking at a basic ANT script that compiles the project and runs it automatically.

## 8.1 Adding a Basic ANT Script

ANT scripts are defined in XML files. To create an ANT script, create a new project and add some basic source to it. Now, right-click on the project and `New` → `File`. Name the file `build.xml`. Open the newly created empty file and hit the keys `Ctrl+Backspace`. Eclipse helpfully allows you to create a simple buildfile from a template. From the popup menu, select `Buildfile Template`. Eclipse automatically populates the file with the following code:

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <project name="project" default="default">
3    <description>
4      description
5    </description>
6
7    <!-- ==================================
8      target: default
9      ================================== -->
10   <target name="default" depends="depends"  description="description">
11   </target>
12
13   <!-- - - - - - - - - - - - - - - - - - -
14     target: depends
15   - - - - - - - - - - - - - - - - - - - - -->
16   <target name="depends">
17   </target>
18 </project>
```

Most of the file is easy to understand. A couple of things you should note are the following.

[1] If you are familiar with `make` utilities in Linux, you can think of ANT as `make` for Java.

- The project has a name and a description. This is useful when you distribute your code

*ANT targets*

- The buildfile has two *ANT targets*: `default` and `depends`. The `depends` target depends on the `default` target. This means that whenever the `depends` target needs to be executed, `default` will be executed before it. (Notice that the targets themselves do nothing at this point.)

*ANT property*

Let's modify the default target to compile the file and then run the output files. For this, let's first create a *ANT property* that defines where our output is going to be. For that, we only need to add the following line after the code<description> tag.

```
1 <property name="outputdirectory" value="bin" />
```

*ANT task*

Next, let's modify the default target to compile all of our source and put the output classes in the outputdirectory we just described. For this, we can use the built-in *ANT task* `javac`.

```
1 <target name="default" description="Some Assignment Code">
2   <javac srcdir="src" destdir="${outputdirectory}" />
3 </target>
```

When it reads this tag, ANT will start the `javac` compiler on the source directory `src` and put the output in the destination directory defined by the constant `outputdirecoty` i.e. `bin`.

Next, let's have ANT output a helpful message and then run the main class. For this, we use the tasks `echo` and `java` respectively.

```
1 <target name="default" description="Some Assignment Code">
2   <javac srcdir="src" destdir="bin" />
3   <echo>Executing the file now...</echo>
4   <java classname="org.csrdu.java.ex.basicshape1.Runner"
5     classpath="${outputdirectory}" />
6 </target>
```

## 8.2 Running ANT Scripts

To run ANT scripts, we have two options. The first one is from within eclipse and the second one is when you don't have the project loaded in Eclipse and you want to use the script through the command line.

### 8.2.1 Execute ANT from within Eclipse

*ANT View*

Eclipse has excellent support for ANT. The auto-completion works great within ANT scripts and after the creation of the scripts, you can run them using the *ANT View*. Open the ANT view by going to `Window` → `View` → `ANT`.

Click on the `Add Buildfiles` button in the view and select `build.xml`. You can now see the targets in the view.

To run a target, select it and click on the `Run the Selected Target` button. Note the output in the console.

```
Buildfile:  .../w08-ant/build.xml
default:
[javac] .../w08-ant/build.xml:8: warning: 'includeantruntime' was
  not set, defaulting to build.sysclasspath=last; set to false for
  repeatable builds
[echo] Executing the file now...
[java] [Rectangle] Point a: < 1.0 ; 2.0 > ; Point b: < 3.0 ; 3.5 >
[java] Perimeter: 7.0
[java] Surface: 3.0
[java] [Triangle] Point a: < 1.0 ; 1.0 > ; Point b: < 2.0 ; 1.0 > ...
[java] Perimeter: 3.414213562373095
[java] Surface: 0.49999999999999983
[java] [Circle] Center: < 1.0 ; 1.0 > ; Radius: 4.0
[java] Perimeter: 25.132741228718345
[java] Surface: 50.26548245743669
BUILD SUCCESSFUL
Total time: 817 milliseconds
```

Notice how, not only can be build the code but also execute it through ANT.

### 8.2.2 Running ANT from the Command Line

To run ANT from the command line, we first need to install ANT. On Ubuntu, it's simply a matter of executing the following command:

```
sudo apt-get install ant -y
```

Next, change directory to the folder where `build.xml` file was created. Now, to run the `default` target, simply type:

```
ant
```

Notice that we did not have to worry about setting the classpaths or any other configuration. Any required configuration can be set by the developer through ANT.

## 8.3 Bundling a JAR File through ANT

As another usecase, let's see how we can build a jar file from our sourcecode automatically through ANT. Here's the requirement: Our ANT target should be able to compile the whole sourcecode, take all the output classes and put them in a jar file. Then, it should move the newly created jar file to the `dist` folder. Let's create a target for this rule:

```
<target name="createjar" depends="default">
  <jar jarfile="Code.jar" includes="**/*.class" basedir="${outputdirectory}" />
  <move file="Code.jar" todir="dist" />
</target>
```

First, note that we do not have to re-define the compilation aspect of our requirement. That is already taken care of by the `default` target. All we need to do is to have the `createjar` target depend on the `default` target. This way, before `createjar` is executed, the compilation will be completed.

Immediately after compilation, we call the `jar` task and pass it a `jarfile` name. We describe which files to include in the jar through the `includes` attribute. The

`**/*.class` means look through all folders (and sub-folders recursively) and collect all `*.class` files to put in the jar. Then we specify the directory where this search should take place through the `basedir` attribute.

The final thing we need to do is move the file to the `dist` folder using the `move` task. If you want to move the file to a server (for example, if you have to upload your code to a server for review), you can use the `scp` task instead of `move`.

You can run this target using either of the two methods described above. If you are using the commandline method, you need to specify the target (since we are no longer using the default target) through the following command:

```
1 ant createjar
```

Now, refresh the project in eclipse, open the jar file just created and take a look at the compiled code. Notice that the files are arranged properly according to their respective packages.

CHAPTER 9

# JAVA AUTHENTICATION AND AUTHORIZATION SERVICE (JAAS)

Java SE comes with a strong security architecture that includes a cryptography API, public key infrastructure and a pluggable authentication and authorization architecture. [10] In this section, we are going to take a look at the latter of these features and describe how it can help us create secure applications.

## 9.1 Sandboxed Applications

By default, all applications are permitted to perform sensitive operations such as reading the home directory path, creating and deleting files etc. If you want to run an untrusted application in a *sandbox* and restrict its rights, you can use a *Security Manager*. Let's explain this concept of a sandbox through a simple example.

*sandbox*
*Security Manager*

### 9.1.1 Unrestricted Acces

The following applications tries to read some environment variables. In the default configuration, everything is allowed.

```
1  package org.csrdu.java.ex.jaas.basic;
2
3  import java.lang.*;
4  import java.security.*;
5
6  class GetProps {
7    public static void main(String[] args) {
8      String s;
9
10     try {
11       System.out.println("About to get os.name property value");
12
13       s = System.getProperty("os.name", "not specified");
14       System.out.println("  The name of your operating system is: " + s);
15
16       System.out.println("About to get java.version property value");
17
18       s = System.getProperty("java.version", "not specified");
19       System.out.println("  The version of the JVM you are running is: "
20       + s);
21
22       System.out.println("About to get user.home property value");
23
24       s = System.getProperty("user.home", "not specified");
25       System.out.println("  Your user home directory is: " + s);
26
27       System.out.println("About to get java.home property value");
28
29       s = System.getProperty("java.home", "not specified");
30       System.out.println("  Your JRE installation directory is: " + s);
31     } catch (Exception e) {
32       System.err.println("Caught exception " + e.toString());
33     }
```

```
34   }
35 }
```

This produces the expected output:

```
1 About to get os.name property value
2 The name of your operating system is: Linux
3 About to get java.version property value
4 The version of the JVM you are running is: 1.6.0_26
5 About to get user.home property value
6 Your user home directory is: /home/nam
7 About to get java.home property value
8 Your JRE installation directory is: /usr/lib/jvm/java-6-sun-1.6.0.26/jre
```

### 9.1.2 Restricting Access

Now let's see what happens when we sandbox the application without providing it any rights. To do so, we have to pass the VM argument `-Djava.security.manager`. With this VM argument, the output turns to:

```
1 About to get os.name property value
2 The name of your operating system is: Linux
3 About to get java.version property value
4 The version of the JVM you are running is: 1.6.0_26
5 About to get user.home property value
6 Caught exception java.security.AccessControlException:
7    access denied (java.util.PropertyPermission user.home read)
```

*access denial*

The reason for *access denial* is that a security manager is now in effect and the default policy for the sandbox restricts all applications from accessing the system property `user.home`. Let's add a permission for this application to allow it to access this environment variable.

### 9.1.3 Granting Permissions

*security policy*

In order to grant permissions, we need to define a new *security policy*. Security policies in JAAS follow a simple text-based syntax. We will refrain from providing a formal definition of the syntax but explain the policy files through an example for our sandboxed application.

```
1 grant codeBase "file:[workspace-location]/w09-jaas/bin" {
2   permission java.util.PropertyPermission "user.home", "read";
3 };
```

In the policy file, we grant `PropertyPermission` over the variable `user.home` for the action `read`. This will enable our application to access this property. To tell the security manager to use this policy file, we need to append a VM argument to our run configuration. We now run our application with the following VM arguments:

```
1 -Djava.security.manager -Djava.security.policy=src/basicpolicy/examplepolicy
```

assuming that the policy file is saved in `src/basicpolicy/examplepolicy`. Run the application again to see how the output changes. We will leave it as an exercise for you to grant permission to the app for accessing `java.home` property.

## 9.2 Logins, Rights and Custom Permissions

Up till now, we have only used existing permissions to restrict applications from calling privileged JVM operations. In this section, we will describe how we can use the JAAS mechanism to create custom permissions, allow users to login and perform privileged operations defined by our own applications. To do this, we first need to describe login contexts, custom login handlers and permissions. Let's begin by describing the relationship between all these components (cf. Fig 9.1).

The core component of JAAS is the *login context*. It provides access to different *login modules* which can be specified in the *JAAS configuration* file. These can be existing modules such as those that access *Pluggable Authentication Module (PAM)* or they can be custom written as we do in our example below. Each login module is capable of using *callback handlers*. A callback handler is a class that allows login modules to retrieve information from the user. For example, we write a custom username/password handler to retrieve username and password from the user through the console. The login module also communicates with the *security domain* to retrieve security policies. We can hard-code our security policies in code or declare them through a JAAS policy file in the same syntax we used earlier to allow access to property permissions.

*login context*
*login modules*
*JAAS configuration*
*Pluggable Authentication Module (PAM)*
*callback handlers*
*security domain*

After authentication, the login module can create a *subject* and assign *principals* to it. A subject corresponds to a specific logged in user and the principals are the *identities* that this user can assume within the system. For example, a user "Ali" can be given the subject `Ali` in the system. Two principals can be associated with Ali – `faculty` and `finance` to allow him to access both academic and financial records of students.

*subject*
*principals*
*identities*

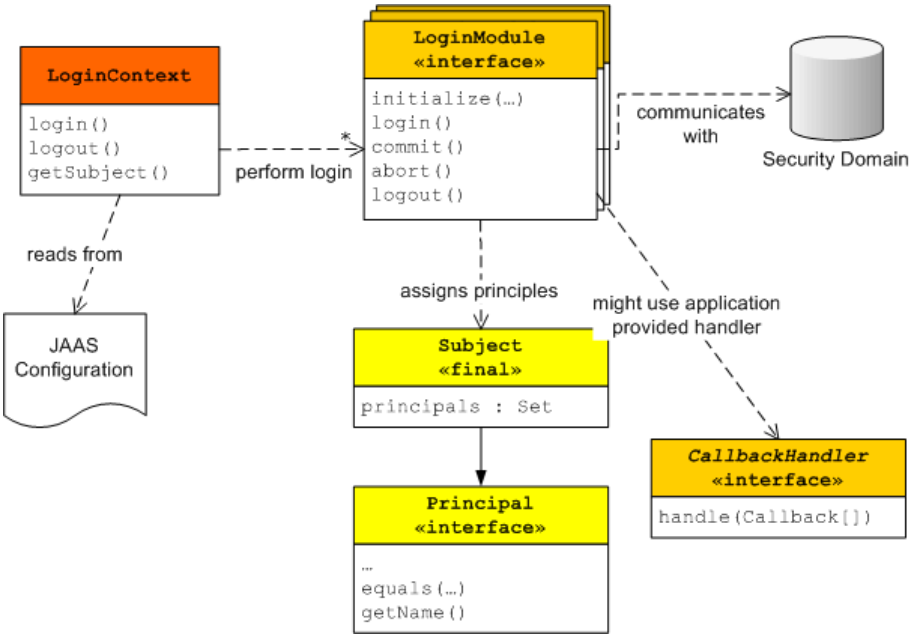



Figure 9.1: JAAS Components

### 9.2.1 JAAS Component Function

#### LoginContext

`LoginContext` provides two basic functions – `login()` and `logout()`. The semantics of login and logout are defined through a `login.config` file along with some specific login modules. Following is an example of our login configuration file:

```
1 JAASExample {
2   org.csrdu.java.ex.jaas.login.AlwaysLoginModule required;
3   org.csrdu.java.ex.jaas.login.PasswordLoginModule optional;
4 };
```

The login context is named `JAASExample` and has two login modules – the first one is required and the second is optional. This means that the first module must succeed for overall login to succeed.

#### LoginModule

The login module has a few important functions:

initalize sets up the login module

login gets the user identification and performs the login authentication for the subject

commit adds a principal to the subject

abort called if overall authentication fails so that login modules can undo changes they made to the security context

logout removes principals from the subject and cleans up the state

As an example, take a look at the `AlwaysLoginModule`. This modules asks for the username through the callback handler but always assumes it's a valid user. It's not very realistic but helps demonstrate the concept.

```
1 package org.csrdu.java.ex.jaas.login;
2
3 public class AlwaysLoginModule implements LoginModule {
4   private Subject subject;
5   private Principal principal;
6   private CallbackHandler callbackHandler;
7   private String username;
8   private boolean loginSuccess;
9
10  public void initialize(Subject sub,
11      CallbackHandler cbh, Map sharedState, Map options) {
12    subject = sub;
13    callbackHandler = cbh;
14    loginSuccess = false;
15  }
16
17
18  public boolean login() throws LoginException {
19    if (callbackHandler == null) {
20      throw new LoginException("No CallbackHandler defined");
21    }
22
```

```
23      Callback[] callbacks = new Callback[1];
24      callbacks[0] = new NameCallback("Username");
25      try {
26        System.out.println("\nAlwaysLoginModule Login");
27        callbackHandler.handle(callbacks);
28        username = ((NameCallback) callbacks[0]).getName();
29      } catch (IOException ioe) {
30        throw new LoginException(ioe.toString());
31      } catch (UnsupportedCallbackException uce) {
32        throw new LoginException(uce.toString());
33      }
34
35      loginSuccess = true;
36      System.out.println();
37      System.out.println("Login: AlwaysLoginModule SUCCESS");
38      return true;
39    }
40
41    public boolean commit() throws LoginException {
42      if (loginSuccess == false) {
43        System.out.println("Commit: AlwaysLoginModule FAIL");
44        return false;
45      }
46      principal = new PrincipalImpl(username);
47      if (!(subject.getPrincipals().contains(principal))) {
48        subject.getPrincipals().add(principal);
49      }
50
51      System.out.println("Commit: AlwaysLoginModule SUCCESS");
52      return true;
53    }
54
55    public boolean abort() throws LoginException {
56      if (loginSuccess == false) {
57        System.out.println("Abort: AlwaysLoginModule FAIL");
58        principal = null;
59        return false;
60      }
61
62      System.out.println("Abort: AlwaysLoginModule SUCCESS");
63      logout();
64      return true;
65    }
66
67    public boolean logout() throws LoginException {
68      subject.getPrincipals().remove(principal);
69      loginSuccess = false;
70      principal = null;
71      System.out.println("Logout: AlwaysLoginModule SUCCESS");
72      return true;
73    }
74  }
```

### Secured Functions

In order to protect some functions through our security context, we can use the principals associated with each function. For this purpose, whenever we want to call a privileged function, we must use the `doAs()` function of the `Subject` class. A snippet of code which uses this function is:

```
try {
  Subject.doAs(lc.getSubject(), new PayrollAction());
} catch (AccessControlException e) {
  System.out.println("Payroll Access DENIED");
}
// new PayrollAction().run();
```

Try uncommenting the last line to directly call this function and notice that you get a `denied` exception. The reason is that `PayrollAction` implements `PrivilegedAction`. Take a look at the complete code in file `org.csrdu.java.ex.jaas.login.JAASExample` around line 40.

This action programmatically checks whether the principal is 'joeuser'.

```
Set principals = subject.getPrincipals();
Iterator iterator = principals.iterator();
while (iterator.hasNext()) {
  PrincipalImpl principal = (PrincipalImpl) iterator.next();
  if (principal.getName().equals("joeuser")) {
    System.out.println("joeuser has Payroll access\n");
    return new Integer(0);
  }
}
throw new AccessControlException("Denied");
```

We can also declare permissions using our policy file. To demonstrate this, we declare another action `PersonnelAction`.

```
class PersonnelAction implements PrivilegedAction {
  public Object run() {
    AccessController.checkPermission(
      new PersonnelPermission("access"));
    System.out.println("Subject has Personnel access\n");
    return new Integer(0);
  }
}
```

*access controller*

This action calls the *access controller* to get the decision regarding the permission. Our policy file for this is given below:

```
grant {
  permission javax.security.auth.AuthPermission "createLoginContext";
  permission javax.security.auth.AuthPermission "doAs";
  permission javax.security.auth.AuthPermission "doAsPrivileged";
  permission javax.security.auth.AuthPermission "modifyPrincipals";
  permission javax.security.auth.AuthPermission "getSubject";
};

grant Principal org.csrdu.java.ex.jaas.login.PrincipalImpl "Nauman" {
  permission org.csrdu.java.ex.jaas.login.PersonnelPermission "access";
};
```

The important lines here are 9–11. They specify that the principal "Nauman" is allowed to "access" code protected by "PersonnelPermission". Related this to the check in the `PersonnelAction` code above. If permission is not granted, `run()` function in `PersonnelAction` will throw an *access denied exception*.

*access denied exception*

### 9.2.2 Running the Code

Run the code with the following VM arguments:

```
1 -Djava.security.manager
2  -Djava.security.auth.login.config=src/loginpolicy/login.config
3  -Djava.security.policy=src/loginpolicy/jaas.policy
```

Provide the username “Nauman” to `AlwaysLoginModule` and username/password paid “joeuser”/“joepw” to `PasswordLoginModule`. Notice the outputs.

Now, modify the policy file and change the permission to grant access rights on personnel actions only to principal “Ali”. Re-run the code with the same usernames to see how the action is denied.

Next, try logging in with an other username/password pair into the `PasswordLoginModule`. Note the output.

Finally, change the `login.config` file to require both login modules to succeed. Then provide incorrect username/password pair to `PasswordLoginModule` and notice the difference in output this time.

# BIBLIOGRAPHY


[1] Apache. Apache ANT. *Available at:* http://ant.apache.org, 2012.

[2] Apache. Apache MINA – Multipurpose Infrastructure for Network Applications. *Available at:* http://mina.apache.org/, 2012.

[3] Apache. PatternLayout – Log4J Documentation. *Available at:* http://logging.apache.org/log4j/1.2/apidocs/org/apache/log4j/PatternLayout.html, 2012.

[4] Ashley J.S Mills. Log4J. *Available at:* http://supportweb.cs.bham.ac.uk/documentation/tutorials/docsystem/build/tutorials/log4j/log4j.html, 2012.

[5] Java. Lesson: Generics. *Available at:* http://docs.oracle.com/javase/tutorial/java/generics/index.html, 2012.

[6] Java. Lesson: The Reflection API. *Available at:* http://docs.oracle.com/javase/tutorial/reflect/, 2012.

[7] Java. Remote Method Invocation Home. *Available at:* http://www.oracle.com/technetwork/java/javase/tech/index-jsp-136424.html, 2012.

[8] kXML. About kXML. *Available at:* http://kxml.sourceforge.net/about.shtml, 2012.

[9] John W. M. Russell. Inheritance & Polymorphism. *Available at:* http://home.cogeco.ca/~ve3ll/jatutor5.htm, 2012.

[10] Java SE Security. Oracle Inc. *Available at:* http://www.oracle.com/technetwork/java/javase/jaas/index.html, 2012.

[11] Dennis Sosnoski. Java Programming Dynamics, Part 1: Java Classes and Class Loading. *Available at:* http://www.ibm.com/developerworks/java/library/j-dyn0429/, 2003.

[12] Dennis Sosnoski. Java Programming Dynamics, Part 2: Introducing Reflection. *Available at:* http://www.ibm.com/developerworks/library/j-dyn0603/, 2003.

[13] Viral Patel. Java Virtual Machine, An inside story. *Available at:* http://viralpatel.net/blogs/2008/12/java-virtual-machine-an-inside-story.html, 2008.

[14] W3C. XML Path Language (XPath) – Version 1.0. *Available at:* http://www.w3.org/TR/xpath/, 1999.

[15] W3C. XQuery 1.0: An XML Query Language. *Available at:* http://www.w3.org/TR/xquery/, 2011.

[16] XMLPull.org. XML pull parsing. *Available at:* `http://www.xmlpull.org/`, 2012.