Chapter 15
Digester

As you have seen in the previous chapters, we use a Bootstrap class to instantiate a connector, a context, wrappers, and other components. Once you have those objects, you then associate them with each other by calling the `set` methods of various objects. For example, to instantiate a connector and a context, use the following code:

```
Connector connector = new HttpConnector();
Context context = new StandardContext();
```

To associate the connector with the context, you then write the following code:

```
connector.setContainer(context);
```

You can also configure the properties of each object by calling the corresponding `set` methods. For instance, you can set the `path` and `docBase` properties of a `Context` object by calling its `setPath` and `setDocBase` methods:

```
context.setPath("/myApp");
context.setDocBase("myApp");
```

In addition, you can add various components to the `Context` object by instantiating the components and call the corresponding `add` method on the context. For instance, here is how you add a lifecycle listener and a loader to your context object:

```
LifecycleListener listener = new SimpleContextConfig();
((Lifecycle) context).addLifecycleListener(listener);
Loader loader = new WebappLoader();
context.setLoader(loader);
```

Once all necessary associations and additions have been performed, to complete the application start-up you call the `initialize` and `start` methods of the connector and the `start` method of the context:

```
connector.initialize();
((Lifecycle) connector).start();
((Lifecycle) context).start();
```

This approach to application configuration has one apparent drawback: everything is hard-coded. Changing a component or even the value of a property requires the recompilation of the `Bootstrap` class. Fortunately, the Tomcat designer has chosen a more elegant way of configuration, i.e. an XML document named server.xml. Each element in the server.xml file is converted to a Java object and an element's attribute is used to set a property. This way, you can simply edit the server.xml file to change Tomcat settings. For example, a `Context` element in the server.xml file represents a context:

```
<context/>
```

To set the `path` and `docBase` properties you use attributes in the XML element:

```
<context docBase="myApp" path="/myApp"/>
```

Tomcat uses the open source library Digester to convert XML elements into Java objects. Digester is explained in the first section of this chapter.

The next section explains the configuration of a web application. A context represents a Web application, therefore the configuration of a web application is achieved through configuring the instantiated `Context` instance. The XML file used for configuring a web application is named web.xml. This file must reside in the WEB-INF directory of the application.

Digester

Digester is an open source project under the subproject Commons under the Apache's Jakarta project. You can download Digester it from http://jakarta.apache.org/commons/digester/. The Digester API comes in three packages, which are packaged into the commons-digester.jar file:

- `org.apache.commons.digester`. This package provides for rules-based processing of arbitrary XML documents.
- `org.apache.commons.digester.rss`. Example usage of Digester to parse XML documents compatible with the *Rich Site Summary* format used by many newsfeeds.
- `org.apache.commons.digester.xmlrules`. This package provides for XML-based definition of rules for Digester.

We will not cover all members in the three packages. Instead, we will concentrate on several important types used by Tomcat. We will start this

section by presenting the `Digester` class, the most important type in the Digester library.

The Digester Class

The `org.apache.commons.digester.Digester` class is the main class in the Digester library. You use it to parse an XML document. For each element in the document, the `Digester` object will check if it needs to do something. You, the programmer, decide what the `Digester` instance must do before you call its the parse method.

How do you tell the Digester object what to do when it encounters an XML element? Easy. You define patterns and associate each pattern with one or more rules. The root element in an XML document has a pattern that is the same as the name of the element. For example, consider the XML document in Listing 13.1.

Listing 13.1: The example.xml file

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<employee firstName="Brian" lastName="May">
  <office>
    <address streeName="Wellington Street" streetNumber="110"/>
  </office>
</employee>
```

The root document of the XML document is `employee`. The `employee` element has the pattern **employee**. The `office` element is a subelement of `<employee>`. The pattern of a subelement is the name of the subelement prefixed by the pattern of its containing element plus /. Therefore, the pattern for the `office` element is `employee/office`. The pattern for the `address` element is equal to:

```
the parent element's pattern + "/" + the name of the element
```

The parent of the address element is `<office>`, and the pattern for `<office>` is **employee/office**. Therefore, the pattern for `<address>` is **employee/office/address**.

Now that you understand how a pattern derives from an XML element, let's talk about rules.

A rule specifies an action or a couple of actions that the Digester must do upon encountering a particular pattern. A rule is represented by the `org.apache.commons.digester.Rule` class. The `Digester` class contains zero or more `Rule` objects. Inside a `Digester` instance, these rules

and their corresponding patterns are stored in a type of storage represented by the `org.apache.commons.digester.Rules` interface. Every time you add a rule to a `Digester` instance, the `Rule` object is added to the `Rules` object.

Among others, the `Rule` class has the `begin` and `end` methods. When parsing an XML document, a `Digester` instance calls the `begin` method of the `Rule` object(s) added to it when it encounters the start element with a matching pattern. The `end` method of the `Rule` object is called when the `Digester` sees an end element.

When parsing the example.xml document in Listing 13.1, here is what the `Digester` object does:

- It first encounters the `employee` start element, therefore it checks if there is a rule (rules) for the pattern **employee**. If there is, the Digester executes the `begin` method of the `Rule` object(s), starting from the `begin` method of the first rule added to the Digester.
- It then sees the `office` start element, so the `Digester` object checks if there is a rule (rules) for the pattern **employee/office**. If there is, it executes the `begin` method(s) implemented by the rule(s).
- Next, the `Digester` instance encounters the `address` start element. This makes it check if there is a rule (rules) for the pattern **employee/office/address**. If one or more rule is found, execute the `begin` method(s) of the rule(s).
- Next, the `Digester` encounters the `address` end element, causing the end method(s) of the matching rules to be executed.
- Next, the `Digester` encounters the `office` end element, causing the end method(s) of the matching rules to be run.
- Finally, the `Digester` encounters the `employee` end element, causing the end method(s) of the matching rules to be executed.

What rules can you use? Digester has predefined a number of rules. You can use these rules without even having to understand the `Rule` class. However, if these rules are not sufficient, you can make your own rules. The predefined rules include the rules for creating objects, setting the value of a property, etc.

Creating Objects

If you want your `Digester` instance to create an object upon seeing a particular pattern, call its `addObjectCreate` method. This method has four overloads. The signatures of the two more frequently used overloads are as follows:

```
public void addObjectCreate(java.lang.String pattern,
  java.lang.Class clazz)
```

```
public void addObjectCreate(java.lang.String pattern,
  java.lang.String className)
```

You pass the pattern and a `Class` object or a class name. For instance, if you want the `Digester` to create an `Employee` object (whose class is `ex15.pyrmont.digestertest.Employee`) upon encountering the pattern `employee`, you call one of the following lines of code:

```
digester.addObjectCreate("employee",
  ex15.pyrmont.digestertest.Employee.class);
```

or

```
digester.addObjectCreate("employee",
  "ex15.pyrmont.digestertest.Employee");
```

The other two overloads of `addObjectCreate` allow you to define the name of the class in the XML element, instead of as an argument to the method. This is a very powerful feature because the class name can be determined at runtime. Here are the signatures of those method overloads:

```
public void addObjectCreate(java.lang.String pattern,
  java.lang.String className, java.lang.String attributeName)

public void addObjectCreate(java.lang.String pattern,
  java.lang.String attributeName, java.lang.Class clazz)
```

In these two overloads, the `attributeName` argument specifies the name of the attribute of the XML element that contains the name of the class to be instantiated. For example, you can use the following line of code to add a rule for creating an object:

```
digester.addObjectCreate("employee", null, "className");
```

where the attribute name is `className`.

You then pass the class name in the XML element.

```
<employee firstName="Brian" lastName="May"
  className="ex15.pyrmont.digestertest.Employee">
```

Or, you can define the default class name in the `addObjectCreate` method as follows:

```
digester.addObjectCreate("employee",
  "ex15.pyrmont.digestertest.Employee", "className");
```

If the `employee` element contains a `className` attribute, the value specified by this attribute will be used as the name of the class to instantiate. If no `className` attribute is found, the default class name is used.

The object created by `addObjectCreate` is pushed to an internal stack. A number of methods are also provided for you to peek, push, and pop the created objects.

Setting Properties

Another useful method is `addSetProperties`, which you can use to make the `Digester` object set object properties. One of the overloads of this method has the following signature:

```
public void addSetProperties(java.lang.String pattern)
```

You pass a pattern to this method. For example, consider the following code:

```
digester.addObjectCreate("employee",
  "ex15.pyrmont.digestertest.Employee");
digester.addSetProperties("employee");
```

The `Digester` instance above has two rules, object create and set properties. Both are set to be triggered by the pattern **employee**. The rules are executed in the order they are added to the `Digester` instance. For the following `employee` element in an XML document (which corresponds to the pattern employee):

```
<employee firstName="Brian" lastName="May">
```

The `Digester` instance first creates an instance of `ex15.pyrmont.digestertest.Employee`, thanks to the first rule added to it. The `Digester` instance then responds to the second rule for the pattern **employee** by calling the `setFirstName` and `setLastName` properties of the instantiated `Employee` object, passing `Brian` and `May`, respectively. An attribute in the `employee` element corresponds to a property in the `Employee` object. An error will occur if the `Employee` class does not define any one of the properties.

Calling Methods

The `Digester` class allows you to add a rule that calls a method on the topmost object in the stack upon seeing a corresponding pattern. This method is `addCallMethod`. One of its overloads has the following signature:

```
public void addCallMethod(java.lang.String pattern,
  java.lang.String methodName)
```

Creating Relationships between Objects

A `Digester` instance has an internal stack for storing objects temporarily. When the `addObjectCreate` method instantiates a class, the result is pushed into this stack. Imagine the stack as a well. To push an object into the stack is like dropping a round object having the same diameter as the well into it. To pop an object means to lift the top most object from the well.

When two `addObjectCreate` methods are invoked, the first object is dropped to the well first, followed by the second object. The `addSetNext` method is used to create a relationship between the first and the second object by calling the specified method on the first object and passing the second object as an argument to the method. Here is the signature of the `addSetNext` method:

```
public void addSetNext(java.lang.String pattern,
  java.lang.String methodName)
```

The `pattern` argument specifies the pattern that triggers this rule, the `methodName` argument is the name of the method on the first object that will be called. The pattern should be of the form **firstObject/secondObject**.

For example, an employee can have an office. To create a relationship between an employee and his/her office, you will first need to use two `addObjectCreate` methods, such as the following:

```
digester.addObjectCreate("employee",
  "ex15.pyrmont.digestertest.Employee");
digester.addObjectCreate("employee/office",
  "ex15.pyrmont.digestertest.Office");
```

The first `addObjectCreate` method creates an instance of the `Employee` class upon seeing an `employee` element. The second `addObjectCreate` method creates an instance of `Office` on seeing `<office>` under `<employee>`.

The two `addObjectCreate` methods push two objects to the stack. Now, the `Employee` object is at the bottom and the `Office` object on top. To create a relationship between them, you define another rule using the `addSetNext` method, such as the following:

```
digester.addSetNext("employee/office", "addOffice");
```

in which `addOffice` is a method in the `Employee` class. This method must accept an `Office` object as an argument. The second `Digester` example in this section will clarify the use of `addSetNext`.

Validating the XML Document

The XML document a `Digester` parses can be validated against a schema. Whether or not the XML document will be validated is determined by the validating property of the `Digester`. By default, the value of this property is `false`.

The `setValidating` method is used to indicate if you want validation to be performed. The `setValidating` method has the following signature:

```
public void setValidating(boolean validating)
```

If you want the well-formedness of your XML document to be validated, pass `true` to the `setValidating` method.

Digester Example 1

The first example explains how to use Digester to create an object dynamically and set its properties. Consider the `Employee` class in Listing 15.2 that we will instantiate using Digester.

Listing 15.2: The Employee Class

```
package ex15.pyrmont.digestertest;

import java.util.ArrayList;

public class Employee {
  private String firstName;
  private String lastName;
  private ArrayList offices = new ArrayList();

  public Employee() {
    System.out.println("Creating Employee");
  }
  public String getFirstName() {
    return firstName;
  }
  public void setFirstName(String firstName) {
    System.out.println("Setting firstName : " + firstName);
    this.firstName = firstName;
  }
  public String getLastName() {
    return lastName;
  }
  public void setLastName(String lastName) {
    System.out.println("Setting lastName : " + lastName);
    this.lastName = lastName;
  }
  public void addOffice(Office office) {
```

```
    System.out.println("Adding Office to this employee");
    offices.add(office);
  }
  public ArrayList getOffices() {
    return offices;
  }
  public void printName() {
    System.out.println("My name is " + firstName + " " + lastName);
  }
}
```

The `Employee` class has three properties: `firstName`, `lastName`, and `office`. The `firstName` and `lastName` properties are strings, and office is of type `ex15.pyrmont.digester.Office`. The `office` property will be used in the second example of Digester.

The Employee class also has one method: `printName` that simply prints the first name and last name properties to the console.

We will now write a test class that uses a `Digester` and adds rules for creating an `Employee` object and setting its properties. The Test01 class in Listing 15.3 can be used for this purpose.

Listing 15.3: The Test01 Class

```
package ex15.pyrmont.digestertest;

import java.io.File;
import org.apache.commons.digester.Digester;

public class Test01 {

  public static void main(String[] args) {
    String path = System.getProperty("user.dir") + File.separator  +
      "etc";
    File file = new File(path, "employee1.xml");
    Digester digester = new Digester();
    // add rules
    digester.addObjectCreate("employee",
      "ex15.pyrmont.digestertest.Employee");
    digester.addSetProperties("employee");
    digester.addCallMethod("employee", "printName");

    try {
      Employee employee = (Employee) digester.parse(file);
      System.out.println("First name : " + employee.getFirstName());
      System.out.println("Last name : " + employee.getLastName());
    }
    catch(Exception e) {
      e.printStackTrace();
    }
  }
}
```

You first define the path containing the location of your XML document and pass it the `File` class's constructor. You then create a `Digester` object and add three rules having the pattern **employee**:

```
digester.addObjectCreate("employee",
  "ex15.pyrmont.digestertest.Employee");
digester.addSetProperties("employee");
digester.addCallMethod("employee", "printName");
```

Next, you call the `parse` method on the `Digester` object passing the `File` object referencing the XML document. The return value of the `parse` method is the first object in the `Digester's` internal stack:

```
Employee employee = (Employee) digester.parse(file);
```

This gives you an `Employee` object instantiated by the `Digester`. To see if the `Employee` object's properties have been set, call the `getFirstName` and `getLastName` methods of the `Employee` object:

```
System.out.println("First name : " + employee.getFirstName());
System.out.println("Last name : " + employee.getLastName());
```

Now, Listing 15.4 offers the employee1.xml document with the root element `employee`. The element has two attributes, `firstName` and `lastName`.

Listing 15.4: The employee1.xml file

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<employee firstName="Brian" lastName="May">
</employee>
```

The result of running the `Test01` class is as follows:

```
Creating Employee
Setting firstName : Brian
Setting lastName : May
My name is Brian May
First name : Brian
Last name : May
```

Here is what happened.

When you call the `parse` method on the `Digester` object, it opens the XML document and starts parsing it. First, the `Digester` sees the `employee` start element. This triggers the three rules for the pattern **employee** in the order the rules were added. The first rule is for creating an object. Therefore, the Digester instantiates the `Employee` class, resulting the calling of the `Employee` class's constructor. This constructor prints the string `Creating Employee`.

The second rule sets the attribute of the `Employee` object. There are two attributes in the `employee` element, `firstName` and `lastName`. This rule causes the set methods of the `firstName` and `lastName` properties to be invoked. The set methods print the following strings:

```
Setting firstName : Brian
Setting lastName : May
```

The third rule calls the `printName` method, which prints the following text:

```
My name is Brian May
```

Then, the last two lines are the result of calling the `getFirstName` and `getLastName` methods on the `Employee` object:

```
First name : Brian
Last name : May
```

Digester Example 2

The second Digester example demonstrates how to create two objects and create a relationship between them. You define the type of relationship created. For example, an employee works in one or more office. An office is represented by the `Office` class. You can create an `Employee` and an `Office` object, and create a relationship between the Employee and Office objects. The `Office` class is given in Listing 15.5.

Listing 15.5: The Office Class

```
package ex15.pyrmont.digestertest;

public class Office {
  private Address address;
  private String description;
  public Office() {
    System.out.println("..Creating Office");
  }
  public String getDescription() {
    return description;
  }
  public void setDescription(String description) {
    System.out.println("..Setting office description : " +
description);
    this.description = description;
  }
  public Address getAddress() {
    return address;
  }
  public void setAddress(Address address) {
    System.out.println("..Setting office address : " + address);
```

```
    this.address = address;
  }
}
```

You create a relationship by calling a method on the parent object. Note that this example uses the `Employee` class in Listing 15.2. The `Employee` class has the `addOffice` method to add an `Office` object to its `offices` collection.

Without the Digester, your Java code would look like this:

```
Employee employee = new Employee();
Office office = new Office();
employee.addOffice(office);
```

An office has an address and an address is represented by the `Address` class, given in Listing 15.6.

Listing 15.6: The Address Class

```
package ex15.pyrmont.digestertest;

public class Address {
  private String streetName;
  private String streetNumber;
  public Address() {
    System.out.println("....Creating Address");
  }
  public String getStreetName() {
    return streetName;
  }
  public void setStreetName(String streetName) {
    System.out.println("....Setting streetName : " + streetName);
    this.streetName = streetName;
  }
  public String getStreetNumber() {
    return streetNumber;
  }
  public void setStreetNumber(String streetNumber) {
    System.out.println("....Setting streetNumber : " + streetNumber);
    this.streetNumber = streetNumber;
  }
  public String toString() {
    return "...." + streetNumber + " " + streetName;
  }
}
```

To assign an address to an office, you can call the `setAddress` method of the `Office` class. With no help from Digester, you would have the following code:

```
Office office = new Office();
Address address = new Address();
office.setAddress(address);
```

The second Digester example shows how you can create objects and create relationships between them. We will use the `Employee`, `Office`, and `Address` classes. The `Test02` class (in Listing 15.7) uses a `Digester` and adds rules to it.

Listing 15.7: The Test02 Class

```
package ex15.pyrmont.digestertest;

import java.io.File;
import java.util.*;
import org.apache.commons.digester.Digester;

public class Test02 {

  public static void main(String[] args) {
    String path = System.getProperty("user.dir") + File.separator  +
      "etc";
    File file = new File(path, "employee2.xml");
    Digester digester = new Digester();
    // add rules
    digester.addObjectCreate("employee",
      "ex15.pyrmont.digestertest.Employee");
    digester.addSetProperties("employee");
    digester.addObjectCreate("employee/office",
      "ex15.pyrmont.digestertest.Office");
    digester.addSetProperties("employee/office");
    digester.addSetNext("employee/office", "addOffice");
    digester.addObjectCreate("employee/office/address",
      "ex15.pyrmont.digestertest.Address");
    digester.addSetProperties("employee/office/address");
    digester.addSetNext("employee/office/address", "setAddress");
    try {
      Employee employee = (Employee) digester.parse(file);
      ArrayList offices = employee.getOffices();
      Iterator iterator = offices.iterator();
      System.out.println(
        "-------------------------------------------------");
      while (iterator.hasNext()) {
        Office office = (Office) iterator.next();
        Address address = office.getAddress();
        System.out.println(office.getDescription());
        System.out.println("Address : " +
          address.getStreetNumber() + " " + address.getStreetName());
        System.out.println("---------------------------------");
      }
    }
    catch(Exception e) {
      e.printStackTrace();
    }
  }
}
```

To see the Digester in action, you can use the XML document employee2.xml in Listing 15.8.

Listing 15.8: The employee2.xml file

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<employee firstName="Freddie" lastName="Mercury">
  <office description="Headquarters">
    <address streetName="Wellington Avenue" streetNumber="223"/>
  </office>
  <office description="Client site">
    <address streetName="Downing Street" streetNumber="10"/>
  </office>
</employee>
```

The result when the Test02 class is run is as follows:

```
Creating Employee
Setting firstName : Freddie
Setting lastName : Mercury
..Creating Office
..Setting office description : Headquarters
....Creating Address
....Setting streetName : Wellington Avenue
....Setting streetNumber : 223
..Setting office address : ....223 Wellington Avenue
Adding Office to this employee
..Creating Office
..Setting office description : Client site
....Creating Address
....Setting streetName : Downing Street
....Setting streetNumber : 10
..Setting office address : ....10 Downing Street
Adding Office to this employee
-----------------------------------------------
Headquarters
Address : 223 Wellington Avenue
--------------------------------
Client site
Address : 10 Downing Street
--------------------------------
```

The Rule Class

The `Rule` class has several methods, the two most important of which are `begin` and `end`. When a `Digester` instance encounters the beginning of an XML element, it calls the `begin` method of all matching `Rule` objects it contains. The `begin` method of the *Rule* class has the following signature:

```
public void begin(org.xml.sax.Attributes attributes)
  throws java.lang.Exception
```

When the `Digester` instance encounters the end of an XML element, it calls the `end` method of all matching `Rule` instances it contains. The signature of the `end` method of the `Rule` class is as follows.

```
public void end() throws java.lang.Exception
```

How do the `Digester` objects in the preceding examples do the wonder? Every time you call the `addObjectCreate`, `addCallMethod`, `addSetNext`, and other methods of the `Digester`, you indirectly invoke the `addRule` method of the `Digester` class, which adds a `Rule` object and its matching pattern to the `Rules` collection inside the `Digester`.

The signature of the `addRule` method is as follows:

```
public void addRule(java.lang.String pattern, Rule rule)
```

The implementation of the `addRule` method in the `Digester` class is as follows:

```
public void addRule(String pattern, Rule rule) {
  rule.setDigester(this);
  getRules().add(pattern, rule);
}
```

Take a look at the `Digester` class source code for the `addObjectCreate` method overloads:

```
public void addObjectCreate(String pattern, String className) {
  addRule(pattern, new ObjectCreateRule(className));
}
public void addObjectCreate(String pattern, Class clazz) {
  addRule(pattern,  new ObjectCreateRule(clazz));
}
public void addObjectCreate(String pattern, String className,
  String attributeName) {
  addRule(pattern, new ObjectCreateRule(className, attributeName));
}
public void addObjectCreate(String pattern,
  String attributeName, Class clazz) {
  addRule(pattern, new ObjectCreateRule(attributeName, clazz));
}
```

The four overloads call the `addRule` method. The `ObjectCreateRule` class--whose instance gets created as the second argument to the `addRule` method--is a subclass of the `Rule` class. You may be interested in the `begin` and `end` method implementations in the `ObjectCreateRule` class:

```
public void begin(Attributes attributes) throws Exception {
  // Identify the name of the class to instantiate
  String realClassName = className;
  if (attributeName != null) {
    String value = attributes.getValue(attributeName);
```

```
    if (value != null) {
      realClassName = value;
    }
  }
  if (digester.log.isDebugEnabled()) {
    digester.log.debug("[ObjectCreateRule]{" + digester.match +
    "}New " + realClassName);
  }

  // Instantiate the new object and push it on the context stack
  Class clazz = digester.getClassLoader().loadClass(realClassName);
  Object instance = clazz.newInstance();
  digester.push(instance);
}

public void end() throws Exception {
  Object top = digester.pop();
  if (digester.log.isDebugEnabled()) {
    digester.log.debug("[ObjectCreateRule]{" + digester.match +
      "} Pop " + top.getClass().getName());
   }
}
```

The last three lines in the `begin` method creates an instance of the object and then pushes it to the internal stack inside the `Digester`. The `end` method pops the object from the stack.

The other subclass of the `Rule` class works similarly. You can open the source code if you are keen to know what is behind each rule.

Digester Example 3: Using RuleSet

Another way of adding rules to a Digester instance is by calling its `addRuleSet` method. The signature of this method is as follows:

```
public void addRuleSet(RuleSet ruleSet)
```

The `org.apache.commons.digester.RuleSet` interface represents a set of `Rule` objects. This interface defines two methods, `addRuleInstance` and `getNamespaceURI`. The signature of the `addRuleInstance` is as follows:

```
public void addRuleInstance(Digester digester)
```

The `addRuleInstance` method adds the set of `Rule` objects defined in the current `RuleSet` to the `Digester` instance passed as the argument to this method.

The `getNamespaceURI` returns the namespace URI that will be applied to all `Rule` objects created in this `RuleSet`. Its signature is as follows:

```
public java.lang.String getNamespaceURI()
```

Therefore, after you create a `Digester` object, you can create a `RuleSet` object and pass the `RuleSet` object to the `addRuleSet` method on the `Digester`.

A convenience base class, `RuleSetBase`, implements `RuleSet`. `RuleSetBase` is an abstract class that provides the implementation of the `getNamespaceURI`. You only need to provide the implementation of the `addRuleInstances` method.

As an example, let's modify the `Test02` class in the previous example by introducing the `EmployeeRuleSet` class in Listing 15.9.

Listing 15.9: The EmployeeRuleSet Class

```
package ex15.pyrmont.digestertest;

import org.apache.commons.digester.Digester;
import org.apache.commons.digester.RuleSetBase;

public class EmployeeRuleSet extends RuleSetBase  {
  public void addRuleInstances(Digester digester) {
    // add rules
    digester.addObjectCreate("employee",
      "ex15.pyrmont.digestertest.Employee");
    digester.addSetProperties("employee");
    digester.addObjectCreate("employee/office",
      "ex15.pyrmont.digestertest.Office");
    digester.addSetProperties("employee/office");
    digester.addSetNext("employee/office", "addOffice");
    digester.addObjectCreate("employee/office/address",
      "ex15.pyrmont.digestertest.Address");
    digester.addSetProperties("employee/office/address");
    digester.addSetNext("employee/office/address", "setAddress");
  }
}
```

Notice that the implementation of the `addRuleInstances` method in the `EmployeeRuleSet` class adds the same rules to the `Digester` as the `Test02` class does. The `Test03` class in Listing 15.10 creates an instance of the `EmployeeRuleSet` and then adds it to the `Digester` it created earlier.

Listing 15.10: The Test03 Class

```
package ex15.pyrmont.digestertest;

import java.io.File;
import java.util.ArrayList;
import java.util.Iterator;
import org.apache.commons.digester.Digester;

public class Test03 {
```

```
  public static void main(String[] args) {
    String path = System.getProperty("user.dir") +
      File.separator  + "etc";
    File file = new File(path, "employee2.xml");
    Digester digester = new Digester();
    digester.addRuleSet(new EmployeeRuleSet());
    try {
      Employee employee = (Employee) digester.parse(file);
      ArrayList offices = employee.getOffices();
      Iterator iterator = offices.iterator();
      System.out.println(
        "-------------------------------------------------");
      while (iterator.hasNext()) {
        Office office = (Office) iterator.next();
        Address address = office.getAddress();
        System.out.println(office.getDescription());
        System.out.println("Address : " +
          address.getStreetNumber() + " " + address.getStreetName());
        System.out.println("--------------------------------");
      }
    }
    catch(Exception e) {
      e.printStackTrace();
    }
  }
}
```

When run, the `Test03` class produces the same output as the `Test02` class. Note however, that the `Test03` is shorter because the code for adding `Rule` objects is now hidden inside the `EmployeeRuleSet` class.

As you will see later, Catalina uses subclasses of `RuleSetBase` for initializing its server and other components. In the next sections, you will see how Digester plays a very important role in Catalina.

ContextConfig

Unlike other types of containers, a `StandardContext` must have a listener. This listener configures the `StandardContext` instance and upon successfully doing so sets the `StandardContext's configured` variable to `true`. In previous chapters, we used the `SimpleContextConfig` class as the `StandardContext's` listener. This class was a very simple one whose sole purpose is to set the `configured` variable so that the `start` method of `StandardContext` can continue.

In a real Tomcat deployment, the standard listener for `StandardContext` is an instance of `org.apache.catalina.startup.ContextConfig` class.

Unlike our humble SimpleContextConfig class, ContextConfig does a lot of useful stuff that the StandardContext instance cannot live without it. For example, a ContextConfig instance associated with a StandardContext installs an authenticator valve in the StandardContext's pipeline. It also adds a certificate valve (of type org.apache.catalina.valves.CertificateValve) to the pipeline.

More importantly, however, the ContextConfig instance also reads and parses the default web.xml file and the application web.xml file and convert the XML elements to Java objects. The default web.xml file is located in the conf directory of CATALINE_HOME. It defines and maps default servlets, maps file extensions with MIME types, defines the default session timeout, and list welcome files. You should open the file now to see its contents.

The application web.xml file is the application configuration file, located in the WEB-INF directory of an application. Both files are not required. ContextConfig will continue even if none of these files is found.

The ContextConfig creates a StandardWrapper instance for each servlet element. Therefore, as you can see in the application accompanying this chapter, configuration is made easy. You are no longer required to instantiate a wrapper anymore.

Therefore, somewhere in your bootstrap class, you must instantiate the ContextConfig class and add it to the StandardContext by calling the addLifecycleListener method of the org.apache.catalina.Lifecycle interface.

```
LifecycleListener listener = new ContextConfig();
((Lifecycle) context).addLifecycleListener(listener);
```

The StandardContext fires the following events when it is started:

- BEFORE_START_EVENT
- START_EVENT
- AFTER_START_EVENT

When stopped, the StandardContext fires the following events:

- BEFORE_STOP_EVENT
- STOP_EVENT
- AFTER_STOP_EVENT

The ContextConfig class responds to two events: START_EVENT and STOP_EVENT. The lifecycleEvent method is invoked every time the StandardContext triggers an event. This method is given in Listing 15.11. I

have added comments to Listing 15.11 so that the `stop` method is easier to understand.

Listing 15.11: The `lifecycleEvent` method of `ContextConfig`

```
public void lifecycleEvent(LifecycleEvent event) {
  // Identify the context we are associated with
  try {
    context = (Context) event.getLifecycle();
    if (context instanceof StandardContext) {
      int contextDebug = ((StandardContext) context).getDebug();
      if (contextDebug > this.debug)
        this.debug = contextDebug;
    }
  }
  catch (ClassCastException e) {
    log(sm.getString("contextConfig.cce", event.getLifecycle()), e);
    return;
  }
  // Process the event that has occurred
  if (event.getType().equals(Lifecycle.START_EVENT))
    start();
  else if (event.getType().equals(Lifecycle.STOP_EVENT))
    stop();
}
```

As you can see in the end of the `lifecycleEvent` method, it calls either its `start` method or its `stop` method. The `start` method is given in Listing 15.12. Notice that somewhere in its body the `start` method calls the `defaultConfig` and `applicationConfig` methods. Both are explained in the sections after this.

Listing 15.12: The `start` method of `ContextConfig`

```
private synchronized void start() {
  if (debug > 0)
    log(sm.getString("contextConfig.start"));
  // reset the configured boolean
  context.setConfigured(false);
  // a flag that indicates whether the process is still
  // going smoothly
  ok = true;
  // Set properties based on DefaultContext
  Container container = context.getParent();
  if( !context.getOverride() ) {
    if( container instanceof Host ) {
      ((Host)container).importDefaultContext(context);
      container = container.getParent();
    }
    if( container instanceof Engine ) {
      ((Engine)container).importDefaultContext(context);
    }
  }
```

```
  // Process the default and application web.xml files
  defaultConfig();
  applicationConfig();
  if (ok) {
    validateSecurityRoles();
  }

  // Scan tag library descriptor files for additional listener classes
  if (ok) {
    try {
      tldScan();
    }
    catch (Exception e) {
      log(e.getMessage(), e);
      ok = false;
    }
  }
  // Configure a certificates exposer valve, if required
  if (ok)
    certificatesConfig();

  // Configure an authenticator if we need one
  if (ok)
    authenticatorConfig();
  // Dump the contents of this pipeline if requested
  if ((debug >= 1) && (context instanceof ContainerBase)) {
    log("Pipline Configuration:");
    Pipeline pipeline = ((ContainerBase) context).getPipeline();
    Valve valves[] = null;
    if (pipeline != null)
      valves = pipeline.getValves();
    if (valves != null) {
      for (int i = 0; i < valves.length; i++) {
        log("  " + valves[i].getInfo());
      }
    }
    log("======================");
  }
  // Make our application available if no problems were encountered
  if (ok)
    context.setConfigured(true);
  else {
    log(sm.getString("contextConfig.unavailable"));
    context.setConfigured(false);
  }
}
```

The defaultConfig Method

The `defaultConfig` method reads and parses the default web.xml file in the %CATALINA_HOME%/conf directory. The `defaultConfig` method is presented in Listing 15.13.

Listing 15.13: The defaultConfig method

```
private void defaultConfig() {
  // Open the default web.xml file, if it exists
  File file = new File(Constants.DefaultWebXml);
  if (!file.isAbsolute())
    file = new File(System.getProperty("catalina.base"),
      Constants.DefaultWebXml);
  FileInputStream stream = null;
  try {
    stream = new FileInputStream(file.getCanonicalPath());
    stream.close();
    stream = null;
  }
  catch (FileNotFoundException e) {
    log(sm.getString("contextConfig.defaultMissing"));
    return;
  }
  catch (IOException e) {
    log(sm.getString("contextConfig.defaultMissing"), e);
    return;
  }
  // Process the default web.xml file
  synchronized (webDigester) {
    try {
      InputSource is =
        new InputSource("file://" + file.getAbsolutePath());
      stream = new FileInputStream(file);
      is.setByteStream(stream);
      webDigester.setDebug(getDebug());
      if (context instanceof StandardContext)
        ((StandardContext) context).setReplaceWelcomeFiles(true);
      webDigester.clear();
      webDigester.push(context);
      webDigester.parse(is);
    }
    catch (SAXParseException e) {
      log(sm.getString("contextConfig.defaultParse"), e);
      log(sm.getString("contextConfig.defaultPosition",
        "" + e.getLineNumber(), "" + e.getColumnNumber()));
      ok = false;
    }
    catch (Exception e) {
      log(sm.getString("contextConfig.defaultParse"), e);
      ok = false;
    }
    finally {
      try {
        if (stream != null) {
          stream.close();
        }
      }
      catch (IOException e) {
        log(sm.getString("contextConfig.defaultClose"), e);
      }
    }
```

```
  }
}
```

The `defaultConfig` method begins by creating a `File` object that references the default web.xml file.

```
File file = new File(Constants.DefaultWebXml);
```

The value of `DefaultWebXML` can be found in the `org.apache.catalina.startup.Constants` class as follows:

```
public static final String DefaultWebXml = "conf/web.xml";
```

The `defaultConfig` method then processes the web.xml file. It locks the `webDigester` object variable, then parses the file.

```
  synchronized (webDigester) {
    try {
      InputSource is =
        new InputSource("file://" + file.getAbsolutePath());
      stream = new FileInputStream(file);
      is.setByteStream(stream);
      webDigester.setDebug(getDebug());
      if (context instanceof StandardContext)
        ((StandardContext) context).setReplaceWelcomeFiles(true);
      webDigester.clear();
      webDigester.push(context);
      webDigester.parse(is);
```

The `webDigester` object variable references a `Digester` instance that have been populated with rules for processing a web.xml file. It is discussed in the subsection, "Creating Web Digester" later in this section.

The applicationConfig Method

The `applicationConfig` method is similar to the `defaultConfig` method, except that it processes the application deployment descriptor. A deployment descriptor resides in the WEB-INF directory of the application directory.

The `applicationConfig` method is given in Listing 15.14.

Listing 15.14: The `applicationConfig` method of `ContextConfig`

```
private void applicationConfig() {
  // Open the application web.xml file, if it exists
  InputStream stream = null;
  ServletContext servletContext = context.getServletContext();
  if (servletContext != null)
    stream = servletContext.getResourceAsStream
  (Constants.ApplicationWebXml);
  if (stream == null) {
```

```
    log(sm.getString("contextConfig.applicationMissing"));
    return;
  }

  // Process the application web.xml file
  synchronized (webDigester) {
    try {
      URL url =
        servletContext.getResource(Constants.ApplicationWebXml);

      InputSource is = new InputSource(url.toExternalForm());
      is.setByteStream(stream);
      webDigester.setDebug(getDebug());
      if (context instanceof StandardContext) {
        ((StandardContext) context).setReplaceWelcomeFiles(true);
      }
      webDigester.clear();
      webDigester.push(context);
      webDigester.parse(is);
    }
    catch (SAXParseException e) {
      log(sm.getString("contextConfig.applicationParse"), e);
      log(sm.getString("contextConfig.applicationPosition",
        "" + e.getLineNumber(),
        "" + e.getColumnNumber()));
      ok = false;
    }
    catch (Exception e) {
      log(sm.getString("contextConfig.applicationParse"), e);
      ok = false;
    }
    finally {
      try {
        if (stream != null) {
          stream.close();
        }
      }
      catch (IOException e) {
        log(sm.getString("contextConfig.applicationClose"), e);
      }
    }
  }
}
```

Creating Web Digester

A `Digester` object reference called `webDigester` exists in the `ContextConfig` class:

```
private static Digester webDigester = createWebDigester();
```

This `Digester` is used to parse the default web.xml and application web.xml files. The rules for processing the web.xml file are added when the

createWebDigester method is invoked. The createWebDigester method is given in Listing 15.15.

Listing 15.15: The createWebDigester method

```
private static Digester createWebDigester() {
  URL url = null;
  Digester webDigester = new Digester();
  webDigester.setValidating(true);
  url =  ContextConfig.class.getResource(
    Constants.WebDtdResourcePath_22);
  webDigester.register(Constants.WebDtdPublicId_22,
    url.toString());
  url = ContextConfig.class.getResource(
    Constants.WebDtdResourcePath_23);
  webDigester.register(Constants.WebDtdPublicId_23,
  url.toString());
  webDigester.addRuleSet(new WebRuleSet());
  return (webDigester);
}
```

Notice that createWebDigester method calls the addRuleSet on webDigester by passing an instance of org.apache.catalina.startup.WebRuleSet. The WebRuleSet is a subclass of the org.apache.commons.digester.RuleSetBase class. If you are familiar with the syntax of a servlet application deployment descriptor and you have read the Digester section at the beginning of this chapter, you sure can understand how it works.

The WebRuleSet class is given in Listing 15.16. Note that I have removed some parts of the addRuleInstances method to save space.

Listing 15.16: The WebRuleSet class

```
package org.apache.catalina.startup;

import java.lang.reflect.Method;
import org.apache.catalina.Context;
import org.apache.catalina.Wrapper;
import org.apache.catalina.deploy.SecurityConstraint;
import org.apache.commons.digester.Digester;
import org.apache.commons.digester.Rule;
import org.apache.commons.digester.RuleSetBase;
import org.xml.sax.Attributes;

/**
 * <p><strong>RuleSet</strong> for processing the contents of a web
application
 * deployment descriptor (<code>/WEB-INF/web.xml</code>) resource.</p>
 *
 * @author Craig R. McClanahan
 * @version $Revision: 1.1 $ $Date: 2001/10/17 00:44:02 $
 */
```

```
public class WebRuleSet extends RuleSetBase {
  // ----------------------------------- Instance Variables
  /**
   * The matching pattern prefix to use for recognizing our elements.
   */
  protected String prefix = null;

  // ------------------------------------------- Constructor
  /**
   * Construct an instance of this <code>RuleSet</code> with
   * the default matching pattern prefix.
   */
  public WebRuleSet() {
    this("");
  }

  /**
   * Construct an instance of this <code>RuleSet</code> with
   * the specified matching pattern prefix.
   *
   * @param prefix Prefix for matching pattern rules (including the
   *  trailing slash character)
   */
  public WebRuleSet(String prefix) {
    super();
    this.namespaceURI = null;
    this.prefix = prefix;
  }

  // ------------------------------------------- Public Methods
  /**
   * <p>Add the set of Rule instances defined in this RuleSet to the
   * specified <code>Digester</code> instance, associating them with
   * our namespace URI (if any).  This method should only be called
   * by a Digester instance.</p>
   *
   * @param digester Digester instance to which the new Rule instances
   *  should be added.
   */
  public void addRuleInstances(Digester digester) {
    digester.addRule(prefix + "web-app",
      new SetPublicIdRule(digester, "setPublicId"));
    digester.addCallMethod(prefix + "web-app/context-param",
      "addParameter", 2);
    digester.addCallParam(prefix +
      "web-app/context-param/param-name", 0);
    digester.addCallParam(prefix +
      "web-app/context-param/param-value", 1);
    digester.addCallMethod(prefix + "web-app/display-name",
      "setDisplayName", 0);
    digester.addRule(prefix + "web-app/distributable",
      new SetDistributableRule(digester));
    ...
    digester.addObjectCreate(prefix + "web-app/filter",
      "org.apache.catalina.deploy.FilterDef");
    digester.addSetNext(prefix + "web-app/filter", "addFilterDef",
```

```
  "org.apache.catalina.deploy.FilterDef");
digester.addCallMethod(prefix + "web-app/filter/description",
  "setDescription", 0);
digester.addCallMethod(prefix + "web-app/filter/display-name",
  "setDisplayName", 0);
digester.addCallMethod(prefix + "web-app/filter/filter-class",
  "setFilterClass", 0);
digester.addCallMethod(prefix + "web-app/filter/filter-name",
  "setFilterName", 0);
digester.addCallMethod(prefix + "web-app/filter/large-icon",
  "setLargeIcon", 0);
digester.addCallMethod(prefix + "web-app/filter/small-icon",
  "setSmallIcon", 0);
digester.addCallMethod(prefix + "web-app/filter/init-param",
  "addInitParameter", 2);
digester.addCallParam(prefix +
  "web-app/filter/init-param/param-name", 0);
digester.addCallParam(prefix +
  "web-app/filter/init-param/param-value", 1);
digester.addObjectCreate(prefix + "web-app/filter-mapping",
  "org.apache.catalina.deploy.FilterMap");
digester.addSetNext(prefix + "web-app/filter-mapping",
  "addFilterMap", "org.apache.catalina.deploy.FilterMap");
digester.addCallMethod(prefix +
  "web-app/filter-mapping/filter-name", "setFilterName", 0);
digester.addCallMethod(prefix +
  "web-app/filter-mapping/servlet-name", "setServletName", 0);
digester.addCallMethod(prefix +
  "web-app/filter-mapping/url-pattern", "setURLPattern", 0);
digester.addCallMethod(prefix +
  "web-app/listener/listener-class", "addApplicationListener", 0);
...
digester.addRule(prefix + "web-app/servlet",
  new WrapperCreateRule(digester));
digester.addSetNext(prefix + "web-app/servlet",
  "addChild", "org.apache.catalina.Container");
digester.addCallMethod(prefix + "web-app/servlet/init-param",
  "addInitParameter", 2);
digester.addCallParam(prefix +
  "web-app/servlet/init-param/param-name", 0);
digester.addCallParam(prefix +
  "web-app/servlet/init-param/param-value", 1);
digester.addCallMethod(prefix + "web-app/servlet/jsp-file",
  "setJspFile", 0);
digester.addCallMethod(prefix +
  "web-app/servlet/load-on-startup", "setLoadOnStartupString", 0);
digester.addCallMethod(prefix +
  "web-app/servlet/run-as/role-name", "setRunAs", 0);
digester.addCallMethod(prefix +
  "web-app/servlet/security-role-ref", "addSecurityReference", 2);
digester.addCallParam(prefix +
  "web-app/servlet/security-role-ref/role-link", 1);
digester.addCallParam(prefix +
  "web-app/servlet/security-role-ref/role-name", 0);
digester.addCallMethod(prefix + "web-app/servlet/servlet-class",
  "setServletClass", 0);
```

```
    digester.addCallMethod(prefix + "web-app/servlet/servlet-name",
      "setName", 0);
    digester.addCallMethod(prefix + "web-app/servlet-mapping",
      "addServletMapping", 2);
    digester.addCallParam(prefix +
      "web-app/servlet-mapping/servlet-name", 1);
    digester.addCallParam(prefix +
      "web-app/servlet-mapping/url-pattern", 0);
    digester.addCallMethod(prefix +
      "web-app/session-config/session-timeout", "setSessionTimeout", 1,
      new Class[] { Integer.TYPE });
    digester.addCallParam(prefix +
      "web-app/session-config/session-timeout", 0);
    digester.addCallMethod(prefix + "web-app/taglib",
      "addTaglib", 2);
    digester.addCallParam(prefix + "web-app/taglib/taglib-location",
1);
    digester.addCallParam(prefix + "web-app/taglib/taglib-uri", 0);
    digester.addCallMethod(prefix +
      "web-app/welcome-file-list/welcome-file", "addWelcomeFile", 0);
  }
}
// ------------------------------------------- Private Classes

/**
 * A Rule that calls the <code>setAuthConstraint(true)</code> method of
 * the top item on the stack, which must be of type
 * <code>org.apache.catalina.deploy.SecurityConstraint</code>.
 */
final class SetAuthConstraintRule extends Rule {
  public SetAuthConstraintRule(Digester digester) {
    super(digester);
  }
  public void begin(Attributes attributes) throws Exception {
    SecurityConstraint securityConstraint =
     (SecurityConstraint) digester.peek();
    securityConstraint.setAuthConstraint(true);
    if (digester.getDebug() > 0)
      digester.log("Calling
SecurityConstraint.setAuthConstraint(true)");
  }
}

...
final class WrapperCreateRule extends Rule {
  public WrapperCreateRule(Digester digester) {
    super(digester);
  }
  public void begin(Attributes attributes) throws Exception {
    Context context =
      (Context) digester.peek(digester.getCount() - 1);
    Wrapper wrapper = context.createWrapper();
      digester.push(wrapper);
    if (digester.getDebug() > 0)
      digester.log("new " + wrapper.getClass().getName());
  }
```

```
  public void end() throws Exception {
    Wrapper wrapper = (Wrapper) digester.pop();
    if (digester.getDebug() > 0)
      digester.log("pop " + wrapper.getClass().getName());
  }
}
```

The Application

This chapter's application shows how to use a `ContextConfig` instance as a listener to configure the `StandardContext` object. It consists of only one class, `Bootstrap`, which is presented in Listing 15.17.

Listing 15.17: The `Bootstrap` class

```
package ex15.pyrmont.startup;

import org.apache.catalina.Connector;
import org.apache.catalina.Container;
import org.apache.catalina.Context;
import org.apache.catalina.Host;
import org.apache.catalina.Lifecycle;
import org.apache.catalina.LifecycleListener;
import org.apache.catalina.Loader;
import org.apache.catalina.connector.http.HttpConnector;
import org.apache.catalina.core.StandardContext;
import org.apache.catalina.core.StandardHost;
import org.apache.catalina.loader.WebappLoader;
import org.apache.catalina.startup.ContextConfig;


public final class Bootstrap {

  // invoke: http://localhost:8080/app1/Modern or
  // http://localhost:8080/app2/Primitive
  // note that we don't instantiate a Wrapper here,
  // ContextConfig reads the WEB-INF/classes dir and loads all
  // servlets.
  public static void main(String[] args) {
    System.setProperty("catalina.base",
      System.getProperty("user.dir"));
    Connector connector = new HttpConnector();
    Context context = new StandardContext();
    // StandardContext's start method adds a default mapper
    context.setPath("/app1");
    context.setDocBase("app1");
    LifecycleListener listener = new ContextConfig();
    ((Lifecycle) context).addLifecycleListener(listener);
    Host host = new StandardHost();
    host.addChild(context);
    host.setName("localhost");
    host.setAppBase("webapps");
```

```
    Loader loader = new WebappLoader();
    context.setLoader(loader);
    connector.setContainer(host);
    try {
      connector.initialize();
      ((Lifecycle) connector).start();
      ((Lifecycle) host).start();
      Container[] c = context.findChildren();
      int length = c.length;
      for (int i=0; i<length; i++) {
        Container child = c[i];
        System.out.println(child.getName());
      }
      // make the application wait until we press a key.
      System.in.read();
      ((Lifecycle) host).stop();
    }
    catch (Exception e) {
      e.printStackTrace();
    }
  }
}
```

Running the Applications

To run the application in Windows, from the working directory, type the following:

```
java -classpath ./lib/servlet.jar;./lib/commons-
collections.jar;./lib/commons-digester.jar;./lib/commons-
logging.jar;./lib/commons-beanutils.jar;./
ex15.pyrmont.startup.Bootstrap
```

In Linux, you use a colon to separate two libraries.

```
java -classpath ./lib/servlet.jar:./lib/commons-
collections.jar:./lib/commons-digester.jar:./lib/commons-
logging.jar:./lib/commons-beanutils.jar:./
ex15.pyrmont.startup.Bootstrap
```

To invoke `PrimitiveServlet`, use the following URL in your browser.

```
http://localhost:8080/app1/Primitive
```

To invoke `ModernServlet`, use the following URL.

```
http://localhost:8080/app1/Modern
```

Summary

Tomcat is used in different configurations. Easy configuration using a server.xml file is achieved through the use of `Digester` objects that converts XML elements to Java objects. In addition, a web.xml document is used to configure a servlet/JSP application. Tomcat must be able to parse this web.xml document and configure a `Context` object based on the elements in the XML document. Again, Digester solves this problem elegantly.