Spring Framework Tutorial

Isabelle Muszynski

15 April 2003

Chapter 1

Introduction

This tutorial covers the main packages in the Spring Framework. For full details, we refer you to Rod Johnson's book, Expert One-on-One J2EE Design and Development, published by Wrox Press in 2002. The book's ISBN number is 1-86100-784-1.

The code for the Spring Framework is contained in package com.interface21. We provide UML diagrams for the subpackages, as well as code samples.

1.1 Setting up for the samples

The samples have been written using the MySQL database (www.mysql.com), Tomcat (jakarta.apache.org/tomcat) and JBoss(www.jboss.org). We refer you to these web sites for setting up the environment correctly. In MySQL, create a database named test, a user test with password test, and grant user test all rights:

```
grant all on test.* to test@localhost identified by 'test';
```

If you use a different database, read the comments at the top of the samples files, they will direct you to the spots in the code you will have to change for your database.

1.2 Building the tutorials

The tutorials can be built using the provided Ant script (you can get ant at http://jakarta.apache.org/ant). You may need to edit the build.properties file to set paths correctly for your system.

1.3 Running the tutorials

We provide a shell script, runtest.sh, and a batch file, runtest.bat, for running a single tutorial. You may need to edit these files to set the paths correctly.

Chapter 2

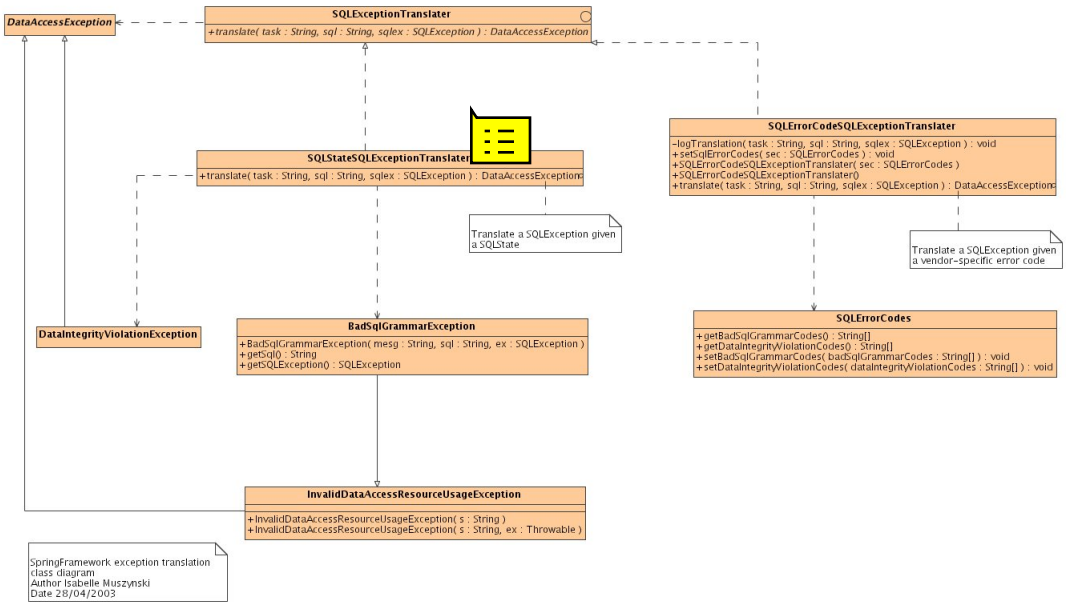
The JDBC Package

Data access is taken care of in the com.interface21.jdbc package and its subpackages. These packages are non-intrusive, and so you can use them separately from the rest of the framework to implement persistence in your application.

There are two main packages, com.interface21.jdbc.core, which implements low-level persistence, and the high-level com.interface21.jdbc.object, which presents an object-oriented view of the low-level package. 

2.1 Exception Handling

Clients of the package - i.e. you, as the developer - should not have to concern themselves with SQLExceptions, as these will not be very informative to your users. Also, because SQLExceptions are checked exceptions, they have to be caught. The Spring Framework takes a different approach : checked SQLExceptions are converted to unchecked DataAccessException and its subclasses. A number of pathological conditions are recognized, through the SQLExceptionTranslater interface and its implementing classes, and will result in subclasses of DataAccessException being thrown. For example, a unique key integrity violation will cause a DataIntegrityViolationException to be thrown. It is up to you, as the developer, to catch this exception – if your code can recover from it – or not, thereby letting it percolate upwards.



2.2 The com.interface21.jdbc.object Package

The com.interface21.jdbc.object package provides a hig-level, JDO-like environment for persistence. Queries, updates and stored procedures are modeled as reusable, threadsafe objects.

The abstract RdbmsOperation class is the root of the hierarchy. It holds a javax.sql.DataSource and a SQL string as instance variables, and allows bind parameters to be declared. Before being used, a RdbmsOperation must be compiled. The exact meaning of the compilation operation varies between subclasses. After compilation, no more parameters can be added, but the operation can be performed repeatedly. Parameters are declared using the declareParameter() method.

The abstract SqlOperation class extends RdbmsOperation and is used as a base class for queries and updates. The compilation process checks that the number of parameters matches the number of parameter placeholders in the SQL string.

The SqlQuery class is used for queries; it requires the user to implement method newResultReader, which should return a ResultReader for the returned ResultSet. The MappingSqlQuery class is a subclass of SqlQuery which provides a default ResultReader implementation, and is therefore easier to use. When using MappingSqlQuery, you must implement the mapRow() method which is called once by the framework for each row in the ResultSet. The SqlUpdate class is used for inserts, updates and deletes.

2.2.1 Code Sample

The code sample (tutorial.jdbc.JdbcObjectPackage.java) assumes the presence of the following table:

```
CREATE TABLE specialty (
```

```
  sp_code VARCHAR(10) NOT NULL PRIMARY KEY,
  sp_name VARCHAR(30)
);
```

The class has the following data members :

```
private DataSource ds;                              // the data source
private String dbUrl = ``jdbc:mysql:///test'';   // the JDBC URL
```

In the constructor, the data source is intialized :

```
public JdbcObjectPackage() {
  ds = new SingleConnectionDataSource(``com.mysql.jdbc.Driver'',
      dbUrl, ``test'', ``test'', false);
}
```

SingleConnectionDataSource is a simple wrapper which provides a fake data source in the absence of a real one. The JdbcTemplate is passed the datasource in its constructor, so that it will be able to hand out connections later.

We start by deleting all rows from the table :

```
public void setUp() {

  // Delete existing rows
  SqlUpdate upd = new SqlUpdate(ds, ``delete from specialty'');
  upd.compile();
  upd.update();
}
```

We then insert a number of rows :

```
public void insert() {

  SqlUpdate upd = new SqlUpdate(ds,
    ``insert into specialty values (?, ?)'');
  upd.declareParameter(new SqlParameter(Types.VARCHAR));
  upd.declareParameter(new SqlParameter(Types.VARCHAR));
  upd.compile();
  upd.update(new Object[] { new String(``RESP''),
                            new String(``Respiratory Diseasese'') } );
  upd.update(new Object[] { new String(``HOMEO''),
                            new String(``Homeopathy'' } );
  upd.update(new Object[] { new String(``INTERN''),
                            new String(``Internal Medicine'') } );
}
```

Our insert statement has 2 parameters of String type, which are added to the SqlUpdate using the declareParameter() method. The update() call takes the actual values of the parameters as arguments.

Similarly, we update one row :

```
public void update() {

  SqlUpdate upd = new SqlUpdate(ds,
    ‘‘update specialty set sp_code = ‘RESPIR’ where sp_code = ‘RESP’\,’’);
  upd.compile();
  upd.update();
}
```

In order to list all results, we use a helper class to represent one row of data, class Specialty :

```
public static class Specialty {
  public Specialty() {
  }
  public Specialty(String code, String descr) {
    this.code = code;
    this.description = descr;
  }
  public String getCode() {
    return code;
  }
  public void setCode(String code) {
    this.code = code;
  }
  public String getDescription() {
    return description;
  }
  public void setDescription(String desc) {
    this.description = desc;
  }
  private String code;
  private String description;
}
```

We will now use a MappingSqlQuery to list all rows with code = “HOMEO”. First we create a custom MappingSqlQuery, MyMappingQuery :

```
public static class MyMappingQuery extends MappingSqlQuery {
  MyMappingQuery(DataSource ds) {
    super(ds, "select * from specialty where sp_code = ?");
    declareParameter(new SqlParameter(Types.VARCHAR));
    compile();
  }
  // Must implement mapRow() method
  protected Object mapRow(ResultSet rs, int rownum) throws SQLException {
    Specialty spec = new Specialty();
    spec.setCode(rs.getString(1));
```

```
    spec.setDescription(rs.getString(2));
    return spec;
  }
}
```

The query takes one parameter, as indicated by the question mark placeholder. The extract method creates an instance of Specialty with the data in the returned row.

The method to read the results follows :

```
public void readOneRow() {
  MyMappingQuery query = new MyMappingQuery(ds);
  List list = query.execute(new Object[] { new String("HOMEO") } );
  System.out.println("Listing all homeopathy rows");
  ListIterator iter = list.listIterator();
  while (iter.hasNext()) {
    Specialty spec = (Specialty)(iter.next());
    System.out.println("Specialty code = " + spec.getCode() +
      ", specialty name = " + spec.getDescription());
  }
}
```

We next use a SqlQuery to read all rows. We start by creating an application-specific subclass of SqlQuery, as SqlQuery is abstract and requires implementation of the newResultReader() method.

```
public static class MyQuery extends SqlQuery {
  public MyQuery(DataSource ds) {
    super(ds, ‘‘select sp_code, sp_name from specialty’’);
  }
  protected ResultReader newResultReader(int rowsExpected, Object[] parameters) {
    return new ResultReader() {
      private List list = new LinkedList();
      public List getResults() {
        return list;
      }
      public void processRow(ResultSet rs) throws SQLException {
        list.add(new Specialty(rs.getString(1), rs.getString(2)));
      }
    };
  }
}
```

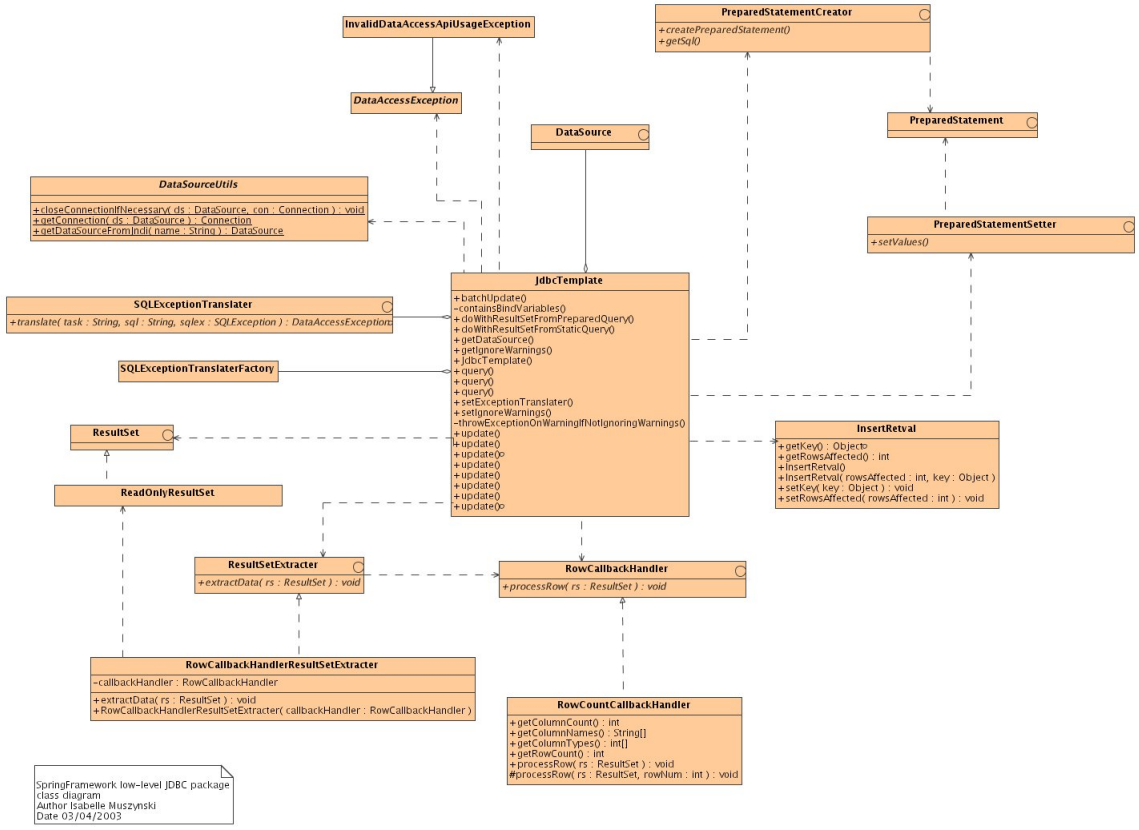
We can then use our query object and its ResultReader to read all rows in the table :

```
public void readAllRows() {
  MyQuery query = new MyQuery(ds);
  query.compile();
  List list = query.execute();
```

```
  System.out.println(‘‘Listing all rows’’);
  ListIterator iter = list.listIterator();
  while (iter.hasNext()) {
    Specialty spec = (Specialty)(iter.next());
    System.out.println(‘‘Specialty code = ’’ + spec.getCode() +
      ‘‘, specialty name = ’’ + spec.getDescription());
  }
}
```

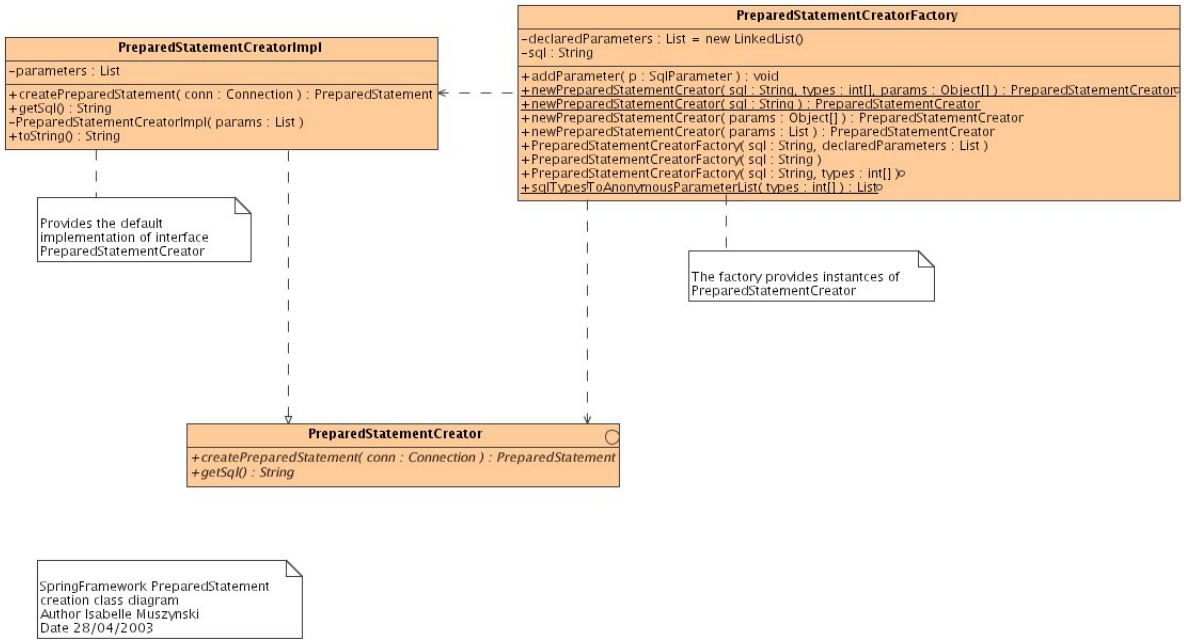
2.3 The com.interface21.jdbc.core Package

The core package is centered around class JdbcTemplate. This class takes care of the workflow for JDBC access, and calls back to the developer's code for aspects such as parameter binding and treatment of the returned rows in a query's resultset. JdbcTemplate takes care of resource management and exception translation. JdbcTemplate can be used for running queries, update, insert and delete statements.



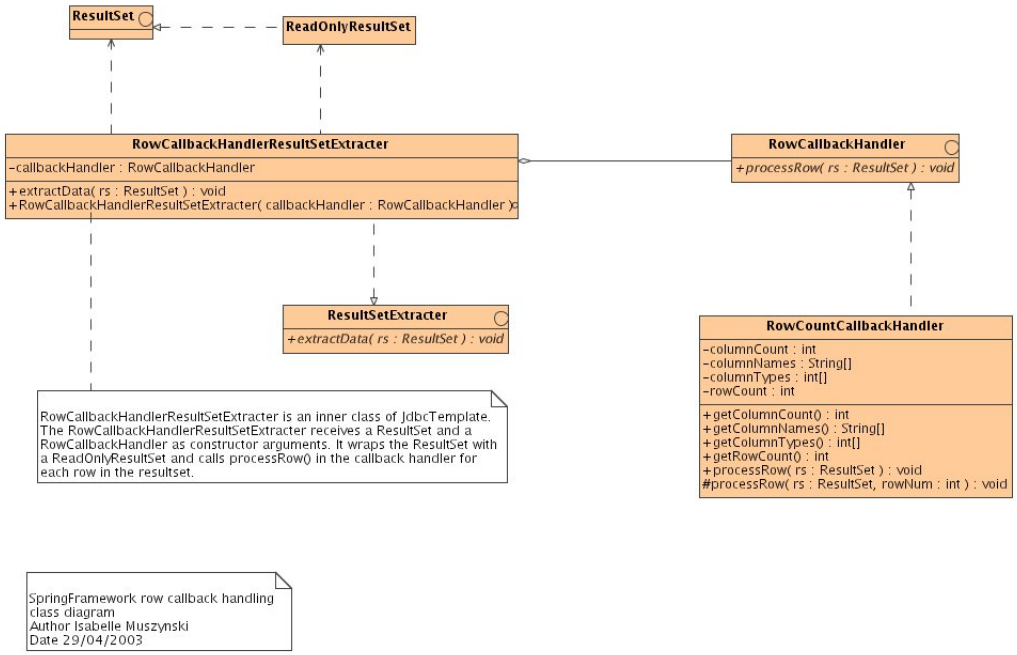
2.3.1 The PreparedStatementCreator Interface

JdbcTemplate calls back to client code to create a prepared statement. To this end, the client code must implement the PreparedStatementCreator interface to return a java.sql.PreparedStatement with correctly bound parameters.



2.3.2 The RowCallbackHandler Interface

The client code must implement the RowCallBackHandler interface to extract results from each row of the resultset returned by a query.



2.3.3 Code Sample

The code sample (tutorial.jdbc.JdbcCorePackage.java) assumes the presence of the following tables:

```
  CREATE TABLE specialty (
  sp_code VARCHAR(10) NOT NULL PRIMARY KEY,
  sp_name VARCHAR(30)
);
  CREATE TABLE species (
  spc_id INT UNSIGNED NOT NULL PRIMARY KEY,
  spc_name VARCHAR(30)
);
  CREATE TABLE species_seq (
  seq INT UNSIGNED NOT NULL AUTO_INCREMENT PRIMARY KEY
);
```

You should also insert a starting value of zero into the sequence table :

```
INSERT INTO species_seq VALUES(0);
```

The class has the following data members :

```
private JdbcTemplate tpl;                       // the JdbcTemplate instance
private DataSource ds;                          // the data source
private String dbUrl = ``jdbc:mysql:///test'';   // the JDBC URL
```

In the constructor, the data source and JdbcTemplate are intialized :

```
public JdbcCorePackage() {
  ds = new SingleConnectionDataSource(``com.mysql.jdbc.Driver'',
        dbUrl, ``test'', ``test'', false);
  tpl = new JdbcTemplate(ds);
}
```

SingleConnectionDataSource is a simple wrapper which provides a fake data source in the absence of a real one. The JdbcTemplate is passed the datasource in its constructor, so that it will be able to hand out connections later.

We start by deleting all rows from the table :

```
public void setUp() {

  // Delete existing rows
  tpl.update(``delete from specialty'');
}
```

A simple insert is performed as follows :

```
public void staticInsert() {

  // Insert one row
  int numRows = tpl.update(
    ``insert into specialty values('HOMEO', 'Homeopathy')'');
  if (1 == numRows) {
    System.out.println(``One row was inserted. Rows in table are:'');
    // Need a RowCallbackHandler
    tpl.query(``select * from specialty'',
      new RowCallbackHandler() {
        public void processRow(ResultSet rs) throws SQLException {
          System.out.println(``Specialty code = '' + rs.getString(1) +
            '', specialty name = '' + rs.getString(2));
        }
      }
    );
  }
}
```

The update() method of JdbcTemplate returns the number of rows affected by the operation. Here, we check that one row was inserted. We then select all rows from the table using the quey() method of JdbcTemplate. This method requires a RowCallbackHandler, whose processRow() method will be called once per row. In our implementation, we simply print the row of data.

In order to use a prepared statement, we must pass the JdbcTemplate a PreparedStatementCreator which prepares the statement and binds the parameters:

```
public void dynamicInsert() {

  // Insert one row

  PreparedStatementCreator psc = new PreparedStatementCreator() {
    private String sql = ‘‘insert into specialty values(?, ?)’’;
    public String getSql() { return sql; }
    public PreparedStatement createPreparedStatement(Connection conn)
        throws SQLException {
      PreparedStatement ps = conn.prepareStatement(sql);
      ps.setString(1, ‘‘INTERN’’);
      ps.setString(2, ‘‘Internal Medicine’’);
      return ps;
    }
  };
  int numRows = tpl.update(psc);
  if (1 == numRows) {
    System.out.println(‘‘One row was inserted. Rows in table are:’’);
    // Need a RowCallbackHandler
    tpl.query(‘‘select * from specialty’’,
      new RowCallbackHandler() {
        public void processRow(ResultSet rs) throws SQLException {
          System.out.println(‘‘Specialty code = ’’ +
          rs.getString(1) + ‘‘, specialty name = ’’ + rs.getString(2));
        }
      }
    );
  }
```

We can also use the PreparedStatementCreatorFactory to prepare our statement:

```
public void dynamicInsertUsingPSFactory() {

  // Insert one row
  int[] types = new int[] { Types.VARCHAR, Types.VARCHAR };
  Object[] params = new Object[] { new String(‘‘RESP’’),
    new String(‘‘Respiratory Diseases’’) };

  PreparedStatementCreator psc =
  PreparedStatementCreatorFactory.newPreparedStatementCreator(
    ‘‘insert into specialty values(?, ?)’’, types, params);
  int numRows = tpl.update(psc);
  if (1 == numRows) {
```

```
  System.out.println(``One row was inserted.Rows in table are:'');
  // Need a RowCallbackHandler
  tpl.query(``select * from specialty'',
    new RowCallbackHandler() {
      public void processRow(ResultSet rs) throws SQLException {
        System.out.println(``Specialty code = '' +
          rs.getString(1) + ``, specialty name = '' +
          rs.getString(2));
    }
  }
  );
}
```

The PreparedStatementCreatorFactory needs to know the number and SQL types of the parameters, as well as the values of the paramaeters to be bound.

Next, we list all results. We use a helper class to represent one row of data, class Specialty :

```
public static class Specialty {
  public Specialty(String code, String descr) {
    this.code = code;
    this.description = descr;
  }
  public String getCode() {
    return code;
  }
  public String getDescription() {
    return description;
  }
  private String code;
  private String description;
}
```

We pass a ResultReader to the query method of JdbcTemplate. The ResultReader collects all rows in a list, which we can access later.

```
public void readAllRows() {
  ResultReader reader = new ResultReader() {
    private List list = new LinkedList();
    public List getResults() {
      return list;
    }
    public void processRow(ResultSet rs) throws SQLException {
      list.add(new Specialty(rs.getString(1), rs.getString(2)));
    }
  };
```

```
  tpl.query(‘‘select * from specialty’’, reader);

  System.out.println(‘‘Listing all rows’’);
  List list = reader.getResults();
  ListIterator iter = list.listIterator();
  while (iter.hasNext()) {
    Specialty spec = (Specialty)(iter.next());
    System.out.println(‘‘Specialty code = ’’ +
    spec.getCode() + ‘‘, specialty name = ’’ +
    spec.getDescription());
  }
}
```

Finally, we test the insertion of new rows using a key generator. Table species_seq contains the sequence values. We retrieve the keys by bunches of 2, which is more efficient than retrieving key values one at a time, since this way we only need to go to the database once for every two insertions.

```
public void insertWithSequence() throws DataAccessException {
  int[] types = new int[] { Types.INTEGER, Types.VARCHAR };
  String[] species = new String[] {"Cat", "Dog", "Horse"};

  JdbcTemplate tpl = new JdbcTemplate(ds);
  // First clean up the table
  tpl.update("delete from species");
  // Read keys by bunches of 2 values
  MySQLMaxValueIncrementer incr = new MySQLMaxValueIncrementer(ds, "species_seq", "seq", 2);
  // Insert 3 records
  for (int i = 0; i < 3; i++) {
    Object[] params = new Object[] { new Integer(incr.nextIntValue()), species[i] };

    PreparedStatementCreator psc =
      PreparedStatementCreatorFactory.newPreparedStatementCreator(
        "insert into species values(?, ?)", types, params);

    tpl.update(psc);
    System.out.println("Inserted species with id = " + params[0]);
  }
  // List all rows
  tpl.query("select * from species",
    new RowCallbackHandler() {
      public void processRow(ResultSet rs) throws SQLException {
        System.out.println("Species id = " + rs.getInt(1) +
          ", species name = " + rs.getString(2));
      }
    }
  );
```

```
}
```

The MySQLMaxValueIncrementer reads keys from the database in bunches of 2 values, as specified by the last argument to the constructor. In each iteration of the insertion loop, we bind a key value and a species to the prepared statement.

Chapter 3

The Beans Package

The com.interface21.beans package and its subpackages provide for specifying the infrastructure of your project using Java Beans. Beans are used extensively throughout the framework, as will be demonstrated in following tutorials.

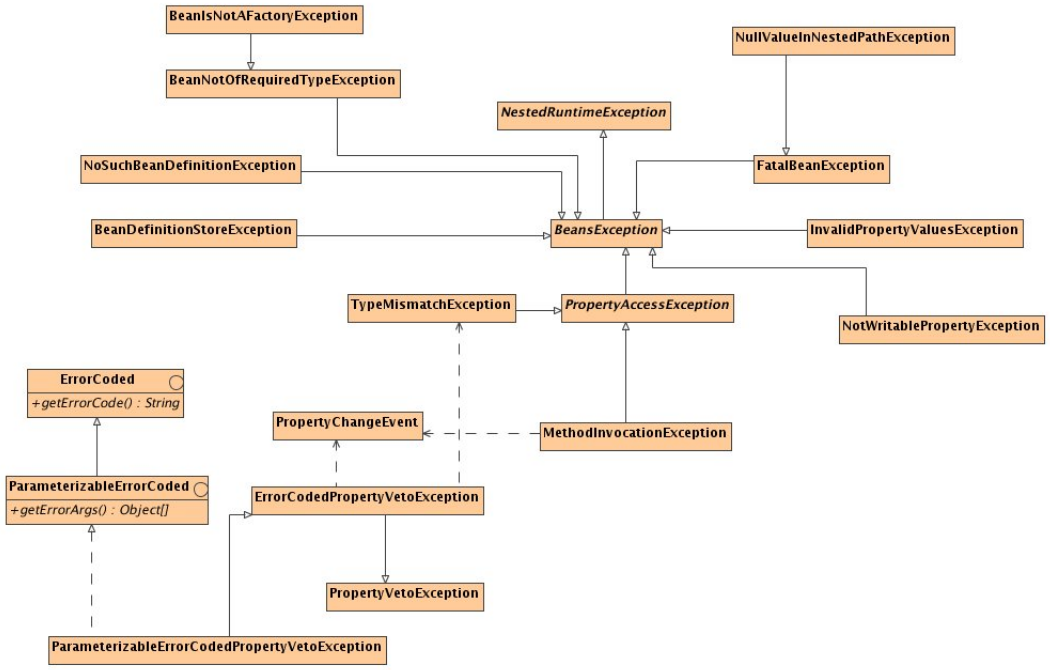
There are three main packages, com.interface21.beans, com.interface21.beans.factory which defines bean factories, and com.interface21.beans.propertyeditors which defines a number of property editors (we will discuss property editors later in this tutorial).

A Java Bean is simply a class with a default constructor, and which follows a naming convention where a property named "prop" has a setter setProp and a getter getProp. We refer you to the Sun tutorials for more information :

- http://developer.java.sun.com/developer/onlineTraining/Beans/bean01/
- http://developer.java.sun.com/developer/onlineTraining/Beans/bean02/
- http://developer.java.sun.com/developer/onlineTraining/Beans/bean03/

3.1 Exception Handling

The base class of all bean exceptions is BeansException, which is a runtime exception, thereby freeing you of the need to explicitly catch it.



3.2 The sample classes

The example in this tutorial uses the following interfaces and implementation classes :

3.2.1 The ISpecies interface

```
public interface ISpecies {
  String getName();
  void setName(String name);
}
```

3.2.2 The IPet interface

```
public interface IPet {
  void setName(String name);
  String getName();
  int getAge();
  void setAge(int age);
  void setSpecies(ISpecies sp);
  ISpecies getSpecies();
}
```

3.2.3 The IOwner interface

```
public interface IOwner {
  void setName(String name);
  String getName();
  void setPets(List pets);
  List getPets();
}
```

3.2.4 The Species class

```
public class Species implements ISpecies {
  public Species() {
  }
  public Species(String name) {
    this.name = name;
  }
  public String getName() {
    return name;
  }
  public void setName(String name) {
    this.name = name;
  }
  private String name;
  }
```

3.2.5 The Pet class

```
public class Pet implements IPet {
  public Pet() {
  }
  public Pet(String name, int age, ISpecies sp) {
    this.name = name;
    this.age = age;
    species = sp;
  }
  public void setName(String name) {
    this.name = name;
  }
  public String getName() {
    return name;
  }
  public int getAge() {
    return age;
  }
  public void setAge(int age) {
```

```
    this.age = age;
  }
  public void setSpecies(ISpecies sp) {
    species = sp;
  }
  public ISpecies getSpecies() {
    return species;
  }
  private String name;
  private int age;
  private ISpecies species;
}
```

3.2.6 The Owner class

```
public class Owner implements IOwner {
  public Owner() {
  }
  public Owner(String name, List pets) {
    this.name = name;
    setPets(pets);
  }
  public void setName(String name) {
    this.name = name;
  }
  public String getName() {
    return name;
  }
  public void setPets(List pets) {
    this.pets.clear();
    this.pets.addAll(pets);
  }
  public List getPets() {
    return pets;
  }
  private String name;
  private ArrayList pets = new ArrayList();
}
```

3.3 The com.interface21.beans Package

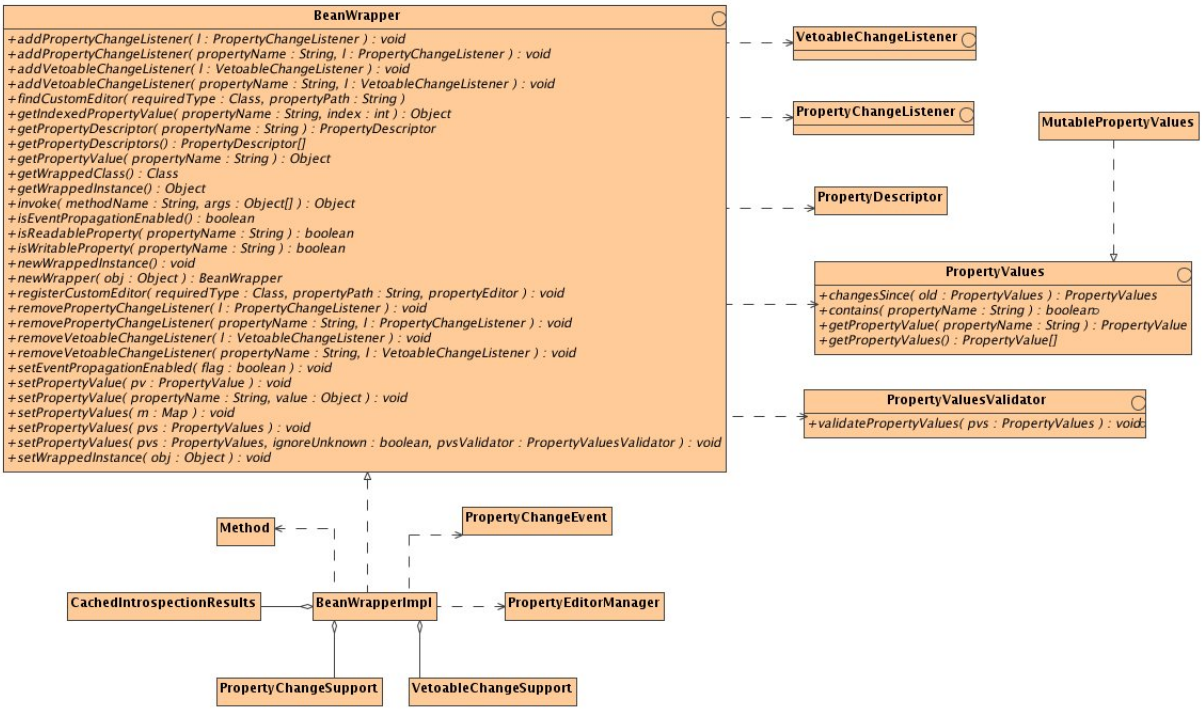
The com.interface21.beans package provides facilities for manipulating beans, in addition to those provided by the java.beans package :

1. Setting multiple properties in a single operation

2. Getting or setting nested properties

3. Providing a standard method to perform initialization after all bean properties have been set

The main classes in the package is the BeanWrapper interface and its default implementation, BeanWrapperImpl. A BeanWrapper wraps a Java Bean and provides facilities for manipulating it.



3.3.1 Code Sample

The code sample (tutorial.beans.TestBeans.java) starts out by manipulating some beans through bean wrappers.

```
public void simpleTest() {
  Species cat = new Species();
  Species dog = new Species();
  Owner owner = new Owner();
  Pet bodo = new Pet();
  Pet pixel = new Pet();
  ArrayList list = new ArrayList();
  list.add(bodo);
  list.add(pixel);
  BeanWrapper bwbodo = new BeanWrapperImpl(bodo);
  BeanWrapper bwpixel = new BeanWrapperImpl(pixel);
  BeanWrapper bwo = new BeanWrapperImpl(owner);
  try {
    bwbodo.setPropertyValue("name", "Bodo");
    bwbodo.setPropertyValue("age", new Integer(4));
    bwbodo.setPropertyValue("species", new Species());
    bwbodo.setPropertyValue("species.name", "Cat");
    bwpixel.setPropertyValue("name", "Pixel");
    bwpixel.setPropertyValue("age", new Integer(11));
    bwpixel.setPropertyValue("species", new Species());
    bwpixel.setPropertyValue("species.name", "Dog");
    bwo.setPropertyValue("name", "Isabelle");
    bwo.setPropertyValue("pets", list);
  } catch (PropertyVetoException e) {
    e.printStackTrace();
  }
  String isabelle = (String)bwo.getPropertyValue("name");
  System.out.println("Bodo's owner is " + isabelle);
  List pets = (List)bwo.getPropertyValue("pets");
  System.out.println(isabelle + "'s pets are: ");
  ListIterator iter = pets.listIterator();
  while (iter.hasNext()) {
    IPet pet = (IPet)iter.next();
    ISpecies sp = pet.getSpecies();
    System.out.println(pet.getName() + " of species " + sp.getName());
  }
}
```

We start by creating some beans, using their default constructors. We then use bean wrappers to set their properties. Of special note is the setting of the nested Species property of the pets, bodo and pixel. The notation used to denote nesting is the dot, i.e. "species.name".

We then print out all of the owner's pets.

3.4 The com.interface21.beans.propertyeditors Package

Property editors allow the conversion of strings into whatever format is required for the bean. For example, imagine we have to specify a list of pets as a parameter in a HTTP GET request. We can provide a property editor which will convert the string into a List of Pet objects.

3.4.1 Code Sample

A list of pets will be defined as a repetition of the string name/age/species, once for each pet, with a comma as separator. We start by defining a property editor that will parse this string and create a list of pets (file OwnerEditor.java). It then sets the pets property to the newly created list.

```
public class OwnerEditor extends PropertyEditorSupport {
  public void setAsText(String text)  throws IllegalArgumentException {
    // text contains a sequence of petname/age/species, separated by comma's
    ArrayList pets = new ArrayList();
    StringTokenizer t = new StringTokenizer(text, ",");
    while (t.hasMoreTokens()) {
      String name_owner_species = t.nextToken();
      StringTokenizer nos = new StringTokenizer(name_owner_species, "/");
      Pet pet = new Pet();
      pet.setName(nos.nextToken());
      try {
        pet.setAge(Integer.parseInt(nos.nextToken()));
      } catch(NumberFormatException e) {
        pet.setAge(0);
      }
      pet.setSpecies(new Species(nos.nextToken()));
      pets.add(pet);
    }
    setValue(pets);
  }
}
```

This is the code that uses the property editor :

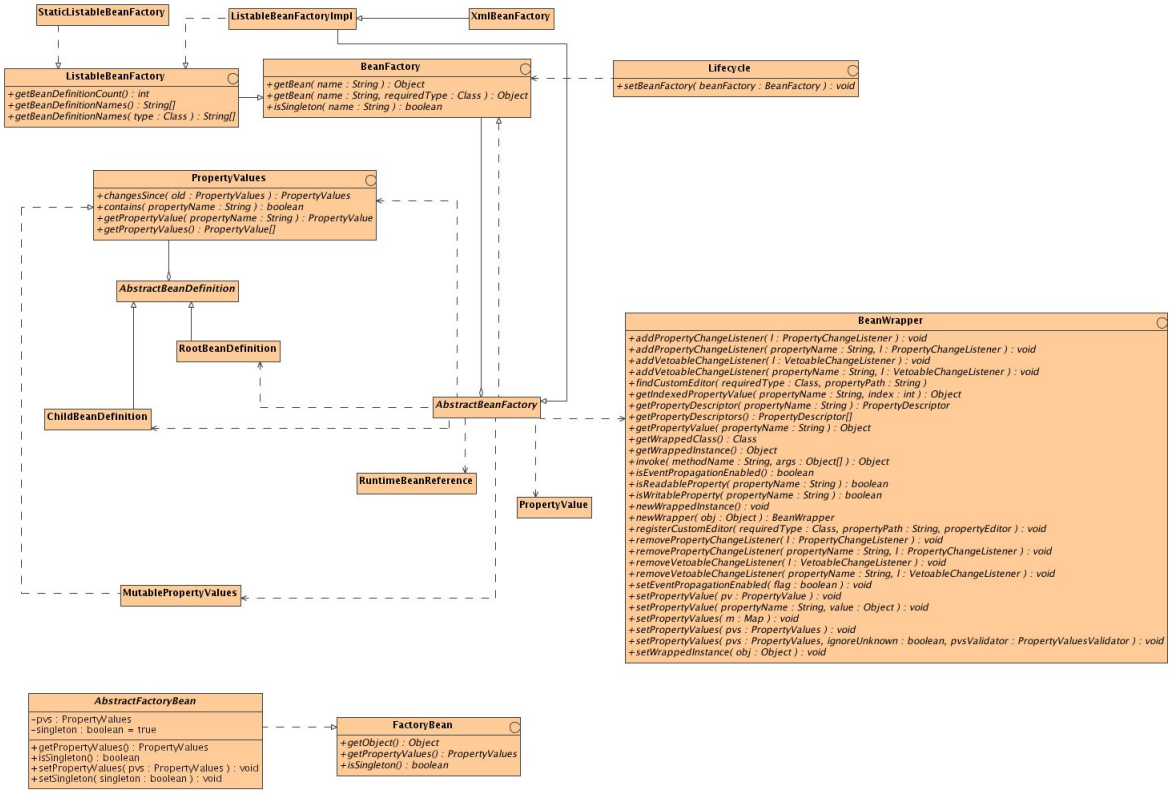
```
public void editorTest() {
  PropertyEditorManager.registerEditor(List.class, OwnerEditor.class);
  Owner isabelle = new Owner();
  BeanWrapper bw = new BeanWrapperImpl(isabelle);
  try {
    bw.setPropertyValue("name", "Isabelle");
    bw.setPropertyValue("pets", "Bodo/4/Cat,Pixel/11/Dog,Raphael/6/Cat");
```

```
  } catch (PropertyVetoException e) {
    e.printStackTrace();
  }
  System.out.println("Property editor : " + isabelle.getName() + "'s pets are: ");
  ListIterator iter = isabelle.getPets().listIterator();
  while (iter.hasNext()) {
    IPet pet = (IPet)iter.next();
    ISpecies sp = pet.getSpecies();
    System.out.println(pet.getName() + " of species " + sp.getName());
  }
}
```

Now, instead of specifying all the pet properties separately, we provide them as one string.

3.5 The com.interface21.beans.factory Package

Bean factories provide facilities for instantiating beans based on definitions in an XML file, a properties file or a resource bundle.



We will demonstrate factories using the following XML file (pets.xml):

```
<?xml version="1.0"?>

<beans>

  <bean name="cat" singleton="true" class="tutorial.beans.Species">
    <property name="name">Cat</property>
  </bean>

  <bean name="dog" singleton="true" class="tutorial.beans.Species">
    <property name="name">Dog</property>
  </bean>

  <bean name="bodo" singleton="true" class="tutorial.beans.Pet">
    <property name="name">Bodo</property>
    <property name="age">4</property>
    <property name="species" beanRef="true">cat</property>
  </bean>

  <bean name="pixel" singleton="true" class="tutorial.beans.Pet">
    <property name="name">Pixel</property>
    <property name="age">11</property>
    <property name="species" beanRef="true">dog</property>
  </bean>

  <bean name="raphael" singleton="true" class="tutorial.beans.Pet">
    <property name="name">Raphael</property>
    <property name="age">6</property>
    <property name="species" beanRef="true">cat</property>
  </bean>

  <bean name="isabelle" singleton="true" class="tutorial.beans.Owner">
    <property name="name">Isabelle</property>
    <property name="pets">Bodo/4/Cat,Pixel/11/Dog,Raphael/6/Cat</property>
  </bean>

</beans>
```

Note the beanRef attribute on the species property of the pets. It signifies that the property refers to a bean definition somewhere in the file.

3.5.1 Code Sample

We read in the bean definitions from the XML file using an XmlBeanFactory :

```
public void useFactory(String file) {
  XmlBeanFactory fac = new XmlBeanFactory(file, null);
```

```
  // Find bean isabelle
  Owner isabelle = (Owner)fac.getBean("isabelle");
  System.out.println("Found bean: " + isabelle.getName() + " with pets: ");
  ListIterator iter = isabelle.getPets().listIterator();
  while (iter.hasNext()) {
    IPet pet = (IPet)iter.next();
    ISpecies sp = pet.getSpecies();
    System.out.println(pet.getName() + " of species " + sp.getName());
  }
}
```

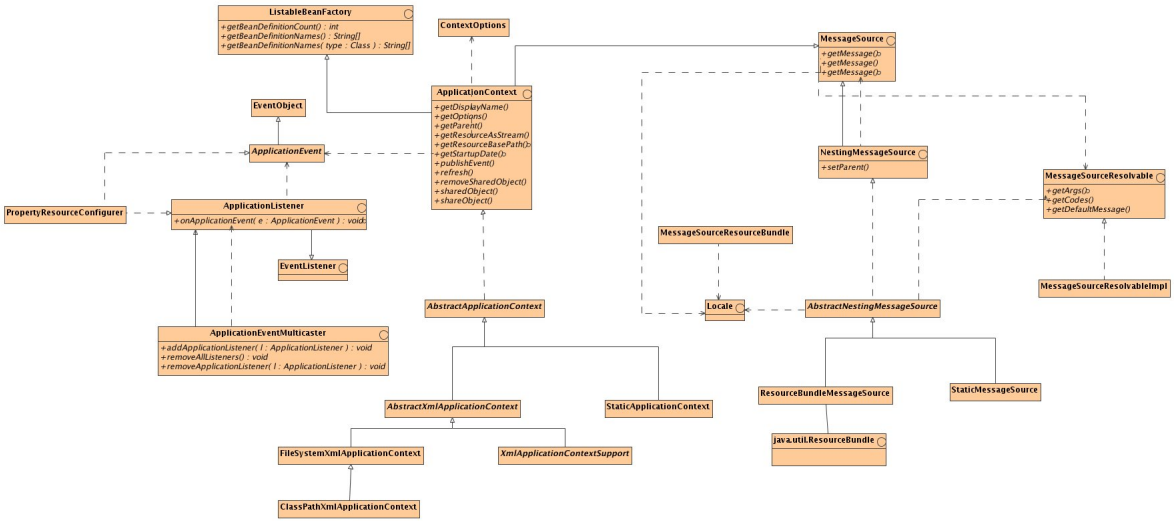
Chapter 4

The Context Package

The com.interface21.context package and its subpackages build on the beans package, allowing beans to be stored in a standard place which is accessible from your application. The application context also allows storage of objects for later retrieval, and lookup of messages in message files. This is a very important feature for internationalizing an application. Application contexts are hierarchical, i.e. a context can have a parent context. When looking up an object, if it isn't found in the current context the parent contexts will be searched until the root of the tree is reached.

Application contexts also allow for the publication of events to registered listeners.

The com.interface21.context.support package contains implementations of the interfaces defined in the com.interface21.context package.



4.1 The sample bean definition and message files

The example in this tutorial uses two bean definition files and one message file.

4.1.1 The root application context, appContext.xml

```
<?xml version="1.0"?>

<beans>

<bean name="cat" singleton="true" class="tutorial.beans.Species">
  <property name="name">Cat</property>
</bean>

<bean name="dog" singleton="true" class="tutorial.beans.Species">
  <property name="name">Dog</property>
</bean>

<bean name="isabelle" singleton="true" class="tutorial.context.Owner">
  <property name="name">Isabelle</property>
  <property name="pets">Bodo/4/Cat,Pixel/11/Dog,Raphael/6/Cat</property>
</bean>

</beans>
```

4.1.2 The child application context, babyContext.xml

```
<?xml version="1.0"?>

<beans>

<bean name="messageSource"
    class="com.interface21.context.support.ResourceBundleMessageSource">
  <property name="basename">messages</property>
</bean>

<bean name="bodo" singleton="true" class="tutorial.context.Pet">
  <property name="name">Bodo</property>
  <property name="age">4</property>
  <property name="species" beanRef="true">cat</property>
</bean>

<bean name="pixel" singleton="true" class="tutorial.context.Pet">
  <property name="name">Pixel</property>
  <property name="age">11</property>
  <property name="species" beanRef="true">dog</property>
</bean>

<bean name="raphael" singleton="true" class="tutorial.context.Pet">
```

```
  <property name="name">Raphael</property>
  <property name="age">6</property>
  <property name="species" beanRef="true">cat</property>
</bean>

</beans>
```

Note that the bean references to cat and dog refer to beans found in the parent context. This file also contains a message source bean; the name property refers to the name of the message file.

4.1.3 The message file, message.properties

```
tutorial.context.defaultmsg=I am a message
tutorial.context.othermsg=I am another message, dated at {0,date}
```

The file is located in the classes directory, so that it can be found at runtime.

The second message contains an argument, which should be formatted as a date. For details on the supported syntax, see the documentation of class java.text.MessageFormat.

4.2 Working with an application context

The tutorial sample is located in tutorial.context.AppContext.java.

We start off by constructing the application context in the constructor :

```
public AppContext() {
  PropertyEditorManager.registerEditor(List.class, OwnerEditor.class);
  String sep = System.getProperty("file.separator");
  // The parent context
  StringBuffer buff1 = new StringBuffer();
  buff1.append("src").append(sep).append("tutorial").append(sep).
  append("context").append(sep).append("appContext.xml");
  // The child context
  StringBuffer buff2 = new StringBuffer();
  buff2.append("src").append(sep).append("tutorial").append(sep).
  append("context").append(sep).append("babyContext.xml");
  try {
    ctx = new FileSystemXmlApplicationContext(new String[]
                      {buff1.toString(), buff2.toString()});
  } catch (IOException e) {
    e.printStackTrace();
  }
}
```

Since we have 2 contexts, we pass a string array containing the parent and child contexts to the constructor of the application context.

We then check that our beans have been correctly loaded into the application context :

```
public void listBeans() {
  try {
    IPet pet = (IPet)ctx.getBean("bodo");
    if (null != pet)
      System.out.println("Found " + pet.getName() + " of species " +
        pet.getSpecies().getName());
    pet = (IPet)ctx.getBean("raphael");
    if (null != pet)
      System.out.println("Found " + pet.getName() + " of species " +
        pet.getSpecies().getName());
    pet = (IPet)ctx.getBean("pixel");
    if (null != pet)
      System.out.println("Found " + pet.getName() + " of species " +
        pet.getSpecies().getName());

    ApplicationContext parent = ctx.getParent();
    if (null != parent) {
      IOwner o = (IOwner)parent.getBean("isabelle");
      if (null != o)
        System.out.println("Found " + o.getName());
    }
  } catch (BeansException e) {
    e.printStackTrace();
  }
}
```

We next try to store a custom object into the application context, and check that we can retrieve it :

```
public void testSharedObject() {
  class Foo {
    void foo() {
      System.out.println("I am a Foo shared object");
    }
  }
  ctx.shareObject("foo", new Foo());
  Foo foo = (Foo)ctx.sharedObject("foo");
  if (null != foo)
    foo.foo();
}
```

Next, we try retrieving some messages :

```
public void testMessages() {
  try {
    System.out.println(ctx.getMessage("tutorial.context.defaultmsg",
                        null, Locale.getDefault()));
```

```
    Object[] arguments = {new Date(System.currentTimeMillis())};
    System.out.println(ctx.getMessage("tutorial.context.othermsg",
                        arguments, Locale.getDefault()));
  } catch(NoSuchMessageException e) {
    e.printStackTrace();
  }
}
```

Remember that othermsg has an argument, which will be formatted as a date.

4.3 Working with events

In order to demonstrate events, we first introduce an event class, FoodEvent, which is derived from the framework's ApplicationEvent class.

```
public class FoodEvent extends ApplicationEvent {
  public static final int BREAKFAST = 0;
  public static final int DINNER = 1;

  public FoodEvent(Object source, int t) {
    super(source);
    if (t < 0 || t > 1)
      t = 0;
    type = t;
  }
  public void setType(int t) {
    type = t;
  }
  public int getType() {
    return type;
  }

  private int type;
}
```

The source parameter to the constructor is the object which caused the event to be published. In our case, it will be an Owner.

We modify the Pet class from the beans tutorial to be an event listener :

```
public class Pet implements IPet, ApplicationListener {
```

and to implement the onApplicationEvent() method :

```
public void onApplicationEvent(ApplicationEvent e) {
  if (e instanceof FoodEvent) {
    FoodEvent ev = (FoodEvent) e;
```

```
    Owner owner = (Owner) ev.getSource();
    String name = owner.getName();
    switch(ev.getType()) {
    case FoodEvent.BREAKFAST:
      System.out.println("Hurrah, " + name +
            " says it's breakfast time");
      break;
    case FoodEvent.DINNER:
      System.out.println("Hurrah, " + name +
            " says  it's dinner time");
      break;
    }
  }
}
```

Class Owner needs to become "application context aware", so that it can cause an event to be published. This involves implementation of interface ApplicationContextAware :

```
public class Owner implements IOwner, ApplicationContextAware {
```

and

```
  public void setApplicationContext(ApplicationContext ctx)
      throws ApplicationContextException {
    this.ctx = ctx;
  }
```

where the ctx member variable is of type ApplicationContext. The Owner class will then use this context to publish events.

We next add a feedPets() method to Owner :

```
  public void feedPets() {
    double d = Math.random();
    FoodEvent ev = new FoodEvent(this, d < 0.5 ? FoodEvent.BREAKFAST : FoodEvent.DINNER);
    ctx.publishEvent(ev);
  }
```

The method creates a FoodEvent and publishes it through the application context.

In order to test our event mechanism, we add the following method to AppContext.java :

```
  public void testEvents() {
    try {
      Owner isabelle = (Owner)ctx.getBean("isabelle");
      if (null != isabelle) {
        isabelle.feedPets();
      }
    } catch (BeansException e) {
```

```
    e.printStackTrace();
  }
}
```

Chapter 5

Building web applications with the Spring Framework

This chapter of the tutorial introduces you to building web applications with the Spring framework. The application we develop in this chapter doesn't do anything useful, but shows you how to set up the environment correctly and can serve as a skeleton for your own applications.

5.1 Setting up Tomcat

The first step is to install Tomcat, which you can download from http://jakarta.apache.org/tomcat. After installation, edit tomcat-users.xml in the conf directory and create a user with roles admin and manager:

```
<?xml version='1.0' encoding='utf-8'?>
<tomcat-users>
  <role rolename="tomcat"/>
  <role rolename="manager"/>
  <role rolename="admin"/>
  <user username="tomcat" password="tomcat" roles="tomcat,admin,manager"/>
</tomcat-users>
```

If you now open the URL http://localhost:8080/manager/html/ in your browser, you should see the list of installed applications. Try out the examples to check that your Tomcat installation is functioning correctly.

5.2 The webapp tutorial directory structure

In the root directory of the tutorial, there is a directory called webapps. Under that directory there is a directory called skeleton, which contains the code for this chapter. In the skeleton directory, edit build.properties so that all directory names are correct for your system. The only property you should need to change is tomcat.home.

The build file, build.xml, makes use of Ant's Tomcat tasks :

- deploy
- install
- list
- reload
- remove
- resources
- roles
- start
- stop
- undeploy

You may need to copy Tomcat's server/lib/catalina-ant.jar to ant's lib directory.

All the source files are contained in the src subdirectory and its subdirectories. The web subdirectory contains the web content of the application (html and jsp files). The metadata subdirectory contains various xml and property files, which we will come back to later.

5.3 The metadata files

5.3.1 applicationContext.xml

The applicationContext.xml file defines the root context of the application. This root context is added as an attribute to the ServletContext, making it available to all web-tier objects. In this example, it only contains the definition of a message source, which refers to file messages.properies, also located in the metadata directory.

```
<?xml version="1.0" encoding="UTF-8"?>

<beans>

  <!-- ============================================================-->
  <!-- Global Message source. For all servlets.                    -->
  <!-- ============================================================-->
  <bean name="messageSource"
      class="com.interface21.context.support.ResourceBundleMessageSource">
    <property name="basename">messages</property>
  </bean>

</beans>
```

5.3.2 skeleton-servlet.xml

Each servlet using the framework has its own application context, which is a child of the root context. This context is defined by a file with a name of the form [servletName]-servlet.xml. Since our servlet will be called skeleton, the application context is defined in skeleton-servlet.xml.

The sample file contains the definition of another message source, more-messages.properties, and the definition of the view resolver, which is required by the framework. The view resolver definition refers to views.properties, which contains mappings for views in our application. Finally, we have a section defining the mapping from request URL's to controller beans, and the definition of the controller beans themselves. In this example, we have two controller servlet, SkeletonController and HelloController. HelloController demonstrates the use of Java bean properties.

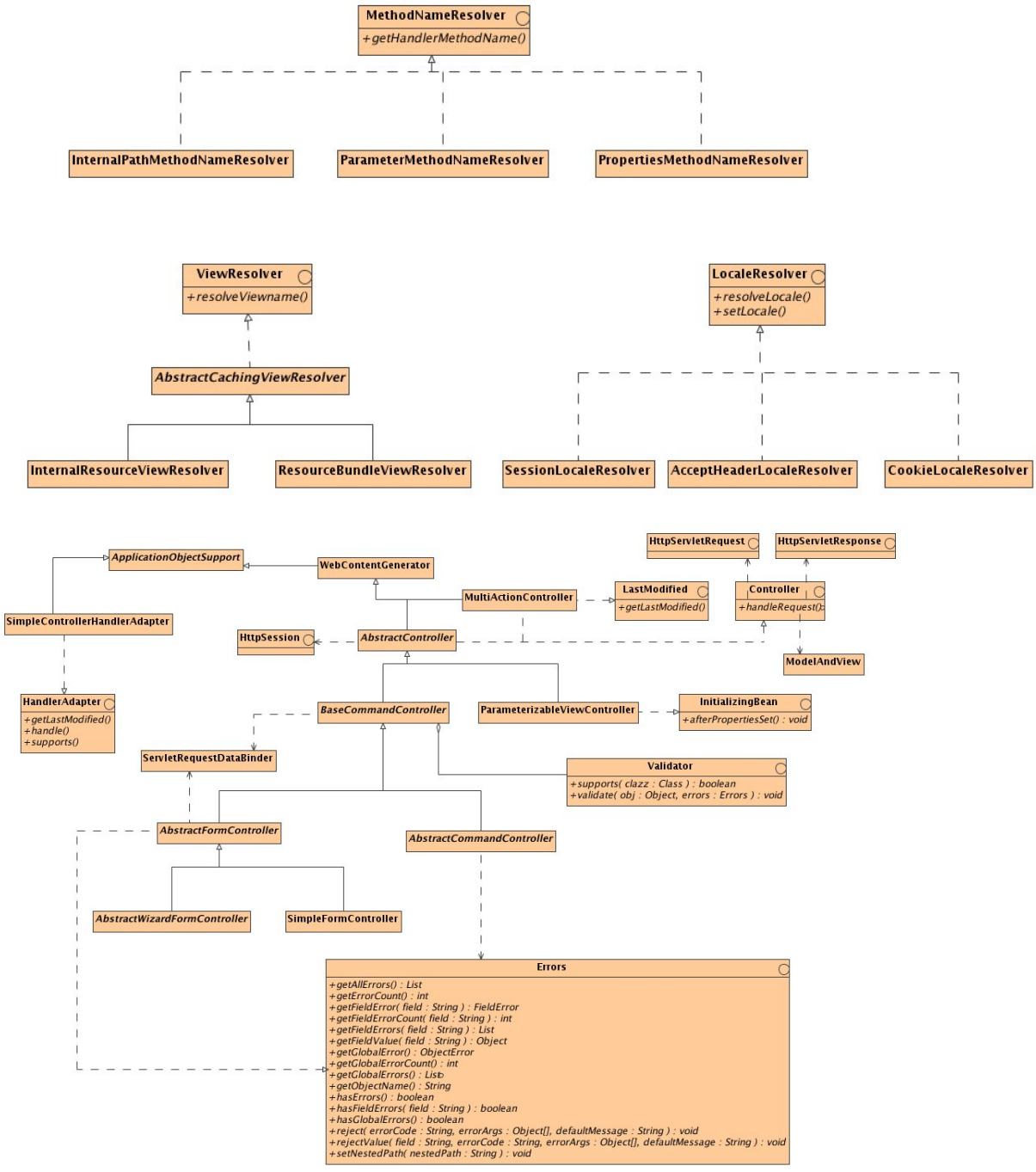
```
<?xml version="1.0" encoding="UTF-8"?>

<beans>

  <!-- ========================================================-->
  <!-- Message source                                          -->
  <!-- ========================================================-->
  <!-- Messages will be taken from classpath.
    Place file under /WEB-INF/classes in the WAR
    CAN OVERRIDE
  -->
  <!--
  <bean name="messageSource"
      class="com.interface21.context.support.ResourceBundleMessageSource">
    <property name="basename">more-messages</property>
  </bean>
  -->

  <!-- ========================================================-->
  <!-- View resolver. Required by web framework.               -->
  <!-- ========================================================-->
  <bean name="viewResolver"
      class="com.interface21.web.servlet.view.ResourceBundleViewResolver">
    <!--
      A false value is for development only:
      REPLACE BY true OR COMMENT OUT IN PRODUCTION OR
      performance will be severely impacted. ***************
    -->
    <property name="cache">true</property>

    <!--
      We can use this to avoid conflict with other command servlets.
      It's the name of the resource bundle loaded from the classpath.
    -->
```

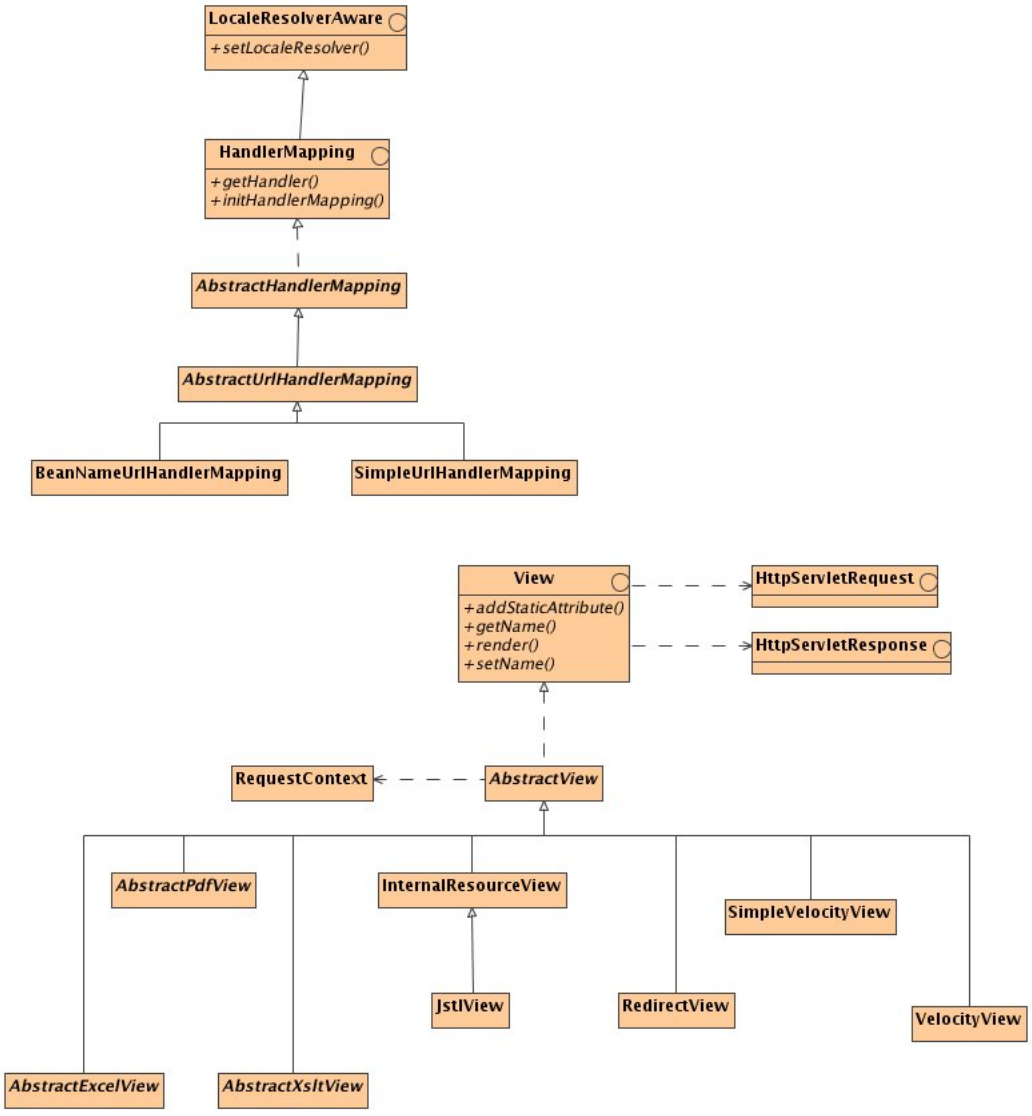
```
  <property name="basename">views</property>
</bean>


<!-- ========================================================-->
<!-- Listeners to this servlet only                          -->
<!-- ========================================================-->

<!--
    Use this listener for debugging only:
    comment out in production.
-->
  <bean name="consoleListener"
    class="com.interface21.context.support.ConsoleListener"
/>

<!-- ========================================================-->
<!-- URL mappings for web framework.                         -->
<-- ========================================================-->
<--
  Simple HandlerMapping implementation that maps from request URL
  to handler bean name in this application context.
  Handlers can have any bean name, and are applied in order of the
  "order" property
  This application uses only one handler map.
-->
<bean name="urlMap"
    class="com.interface21.web.servlet.handler.SimpleUrlHandlerMapping">
  <property name="mappings">
    /skeleton.html=skeletonController
  </property>
</bean>

<!-- ========================================================-->
<!-- Controllers                                             -->
<!-- ========================================================-->

<bean name="skeletonController"
  class="tutorial.web.skeleton.SkeletonController" >
</bean>

<!--
  Illustrates use of a JavaBean property. See page 482.
-->
<bean name="helloController"
```

```
      class="tutorial.web.skeleton.HelloController" >
    <property name="name">The Bean Controller</property>
  </bean>

</beans>
```

Class Diagrams

The controller, resolver and url mapper class diagrams are shown below.


MethodNameResolver
+getHandlerMethodName()
InternalPathMethodNameResolver
ParameterMethodNameResolver
PropertiesMethodNameResolver
ViewResolver
+resolveViewname()
AbstractCachingViewResolver
InternalResourceViewResolver
ResourceBundleViewResolver
LocaleResolver
+resolveLocale()
+setLocale()
SessionLocaleResolver
AcceptHeaderLocaleResolver
CookieLocaleResolver
ApplicationObjectSupport
WebContentGenerator
MultiActionController
LastModified
+getLastModified()
HttpServletRequest
HttpServletResponse
Controller
+handleRequest()
SimpleControllerHandlerAdapter
HttpSession
AbstractController
ModelAndView
HandlerAdapter
+getLastModified()
+handle()
+supports()
BaseCommandController
ParameterizableViewController
InitializingBean
+afterPropertiesSet() : void
ServletRequestDataBinder
Validator
+supports(clazz : Class) : boolean
+validate(obj : Object, errors : Errors) : void
AbstractFormController
AbstractCommandController
AbstractWizardFormController
SimpleFormController
Errors
+getAllErrors() : List
+getErrorCount() : int
+getFieldError(field : String) : FieldError
+getFieldErrorCount(field : String) : int
+getFieldErrors(field : String) : List
+getFieldValue(field : String) : Object
+getGlobalError() : ObjectError
+getGlobalErrorCount() : int
+getGlobalErrors() : List
+getObjectName() : String
+hasErrors() : boolean
+hasFieldErrors(field : String) : boolean
+hasGlobalErrors() : boolean
+reject(errorCode : String, errorArgs : Object[], defaultMessage : String) : void
+rejectValue(field : String, errorCode : String, errorArgs : Object[], defaultMessage : String) : void
+setNestedPath(nestedPath : String) : void



5.3.3 web.xml

The web.xml file is a required part of every application. It contains definitions of initial parameters to pass to the web context, listener definitions and servlet definitions.

```
<?xml version="1.0" encoding="UTF-8"?>
```

```
<!DOCTYPE web-app PUBLIC
  '-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN'
  'http://java.sun.com/dtd/web-app_2_3.dtd'>

<web-app>
  <display-name>Skeleton</display-name>
  <description>Skeleton web application for Spring tutorial</description>

  <!-- Argument for XmlWebApplicationContext -->
  <context-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>/WEB-INF/myApplicationContext.xml</param-value>
  </context-param>

  <!-- Argument for log4j -->
  <context-param>
    <param-name>webAppRootKey</param-name>
    <param-value>skeleton.root</param-value>
  </context-param>
  <!-- Can omit log4jConfiguration context-param if using
       the default location of WEB-INF/classes/log4j.properties.
  -->
  <context-param>
    <param-name>log4jConfigLocation</param-name>
    <param-value>WEB-INF/log4j.properties</param-value>
  </context-param>

  <listener>
    <listener-class>com.interface21.web.context.ContextLoaderListener</listener-class>
  </listener>

  <!--
    This is the controller servlet that handles all incoming requests,
    delegating work to Controller implementations according to mappings in the
    skeleton-servlet.xml file.
  -->
  <servlet>
    <servlet-name>skeleton</servlet-name>
    <servlet-class>com.interface21.web.servlet.ControllerServlet</servlet-class>
    <load-on-startup>1</load-on-startup>
  </servlet>


  <!--
```

```
    Mapping of URLs onto the controller servlet.
  -->
  <servlet-mapping>
    <servlet-name>skeleton</servlet-name>
      <url-pattern>*.i21</url-pattern>
  </servlet-mapping>

  <session-config>
    <session-timeout>60</session-timeout>
  </session-config>

  <welcome-file-list>
    <welcome-file>index.jsp</welcome-file>
  </welcome-file-list>


    <!--
      Display this page to handle unexpected exceptions gracefully.
      Otherwise we get whatever error page the container chooses.
      -->
  <error-page>
    <exception-type>java.lang.Exception</exception-type>
    <location>/jsp/uncaughtException.jsp</location>
  </error-page>

</web-app>
```

The framework depends on a context listener, the ContextLoaderListener, which will load the application context from the context-param URL specified at the top of the file, and the controller servlet that handles all incoming requests. Our controller servlet has a servlet-name of skeleton. In the servlet-mapping section, we define that all requests for pages with an extension of i21 will be handled by our controller servlet. We could also have specified an extension of html, but then requests for static html pages would also have been handed out to our controller. A way out of this is to use a different extension for static pages, such as htm.

5.3.4 messages.properties

The format of this file is very simple. An identifier for a message is followed by an equals sign and its definition.

```
tutorial.webapps.skeleton.greeting=I am a skeleton
```

We do not use messages in this chapter of the tutorial, but at least now you know the format of the message file.

5.3.5 views.properties

This file defines the mappings from URL's to views.

```
# Format is

#viewName.class=<fully qualified class name>
#viewName.<propertyname>=<property value>
#...more property settings

skeletonView.class=com.interface21.web.servlet.view.InternalResourceView
skeletonView.url=/skeleton.html

greetingView.class=com.interface21.web.servlet.view.InternalResourceView
greetingView.url=/greeting.jsp

enterNameView.class=com.interface21.web.servlet.view.InternalResourceView
enterNameView.url=/enterName.jsp
```

5.3.6 log4j.properties

In the log4j property file, we can refer to the context-param we set in web.xml to configure the location of the log file :

```
log4j.rootCategory=info, stdout

log4j.appender.R=org.apache.log4j.RollingFileAppender
log4j.appender.R.File=${skeleton.root}/WEB-INF/skeleton.log

log4j.appender.R.MaxFileSize=100KB
# Keep one backup file
log4j.appender.R.MaxBackupIndex=1

log4j.appender.R.layout=org.apache.log4j.PatternLayout
log4j.appender.R.layout.ConversionPattern=***%C : %M%n
```

5.3.7 The controllers

SkeletonController.java

```
public class SkeletonController implements Controller {
  public ModelAndView handleRequest(HttpServletRequest request,
      HttpServletResponse response) throws ServletException {
    return new ModelAndView("skeletonView");
  }
}
```

This controller is extremely simple: it passes control to the view mapping defined by “skeletonView” in views.properties, i.e. skeleton.html.

HelloController.java

```
public class HelloController
  extends AbstractController {

  private String name;

  /**
   * JavaBean property, set through XML configuration descriptor
   */
  public void setName(String name) {
    this.name = name;
  }

  /**
   * This is the main request handling method
   */
  protected ModelAndView handleRequestInternal(
      HttpServletRequest request,
      HttpServletResponse response)
        throws ServletException, IOException {
    String pname = request.getParameter("name");
    if (pname == null) {
      return new ModelAndView("enterNameView");
    }
    else {
      return new ModelAndView("greetingView", "greeting",
        "Hello " + pname + ", my name is " + this.name);
    }
  }

  /**
   * Overridden method that is invoked after bean properties
   * are set, but before we begin handling requests.
   */
  protected void init() throws ApplicationContextException {
    if (this.name ==  null)
      throw new ApplicationContextException(
        "name property must be set on beans of class " + getClass().getName());
  }
}
```

This controller dispatches to one of two views, depending on whether the name property is set. Of particular interest is the call to the "greetingView" view. The second parameter, "greeting", is the name of the model passed to the view. The third parameter is an Object representing the value of the model. This model is used in greeting.jsp.

5.4 greeting.jsp

```
<jsp:useBean id="greeting" type="java.lang.String" scope="request" />
<%=greeting%>
```

5.5 Testing the application

Point your browser to http://localhost:8080/Skeleton/index.jsp and you should see a simple html page offering you the options of calling the SkeletonController and the HelloController.

Chapter 6

The MultiAction controller

In the previous chapter, we used a separate controller per request. Sometimes, however, it is more convenient to use a single controller for multiple requests. This is the job of the MultiActionController.

This chapter presents a simple example of one controller, MultiController, handling two requests. The second request also shows you how to return output from the controller itself, instead of dispatching to a view. This is not something you would usually do, but the facility is there if you ever need it.

The source code for this chapter is in webapps/multi_action.

6.1 web.xml

The only relevant changes to web.xml are

```
<!--
  This is the controller servlet that handles all incoming requests,
  delegating work to Controller implementations according to mappings in the
  multi-action-servlet.xml file.
-->
<servlet>
  <servlet-name>multi-action</servlet-name>
  <servlet-class>
    com.interface21.web.servlet.ControllerServlet
  </servlet-class>
  <load-on-startup>1</load-on-startup>
</servlet>
```

6.2 multi-action-servlet.xml

The relevant changes to multi-action-servlet.xml are the url mappings and the definition of the multiple actions.

```
<!-- ============================================================-->
<!-- URL mappings for web framework.        -->
<!-- ============================================================-->
<!--
  Simple HandlerMapping implementation that maps from request URL
  to handler bean name in this application context.
  Handlers can have any bean name, and are applied in order of
  their "order" property
  This application uses only one handler map.
-->
<bean name="urlMap"
  class="com.interface21.web.servlet.handler.SimpleUrlHandlerMapping">
  <property name="mappings">
    /debug.i21=multiActionController
    /build.i21=multiActionController
  </property>
</bean>


<!-- ============================================================-->
<!-- Controllers                                                 -->
<!-- ============================================================-->

<!--
  Map from application URLs to method names in the MultiActionController class.
  Analogous to Struts LookupDispatchAction approach.
  See page 485 for discussion of this framework capability.
-->
<bean name="multiActionMethodNameResolver"
    class="com.interface21.web.servlet.mvc.multiaction.PropertiesMethodNameResolver">
  <property name="mappings">
    /something.i21=doSomething
    /build.i21=buildResponse
  </property>
</bean>

<!--
  Controller bean that handles multiple request types.
  See page 484 for discussion of this framework capability.
-->
<bean name="multiActionController"
    class="tutorial.web.multi_action.MultiController" >
  <!--
    Reference to another bean in this factory.
    We can use this syntax to define any references
```

```
      among our application objects.
    -->
    <property name="methodNameResolver"
      beanRef="true">multiActionMethodNameResolver</property>
  </bean>
```

The multiActionMethodNameResolver bean defines the mapping from requests to methods in the MultiAction controller.

6.3 MultiController.java

```
public class MultiController extends MultiActionController {
  /**
   * Demonstrates a simple request handling method.
   */
  public ModelAndView doSomething(HttpServletRequest request, HttpServletResponse response) {
    return new ModelAndView("someView");
  }

  /**
   * This controller method shows how any controller
   * can build the request itself, if it returns null.
   * Of course, usually it's best to let a view handle
   * result generation--using a view gives us an extra
   * level of indirection that is often useful.
   */
  public ModelAndView buildResponse(HttpServletRequest request,
      HttpServletResponse response) throws IOException {
    ServletOutputStream sos = response.getOutputStream();
    sos.println("This response was generated by " +
      "the controller " + this +
      ".\n As we normally don't want to generate " +
      "content in Java classes, this is an unusual special case. " +
      "It's most appropriate for some binary formats. " +
      "\nWhen we want to build the response in a controller, " +
      "we return null. This works for ordinary controllers and" +
      "MultiActionControllers.");
    sos.close();

    // Return value of null will let the controller know
    // that this controller built the response itself
    return null;
  }
}
```

Note that you return null from the controller to let the framework know that the controller built the response itself.

6.4 Testing the application

Point your browser to http://localhost:8080/multi-action/index.jsp and you should see a simple html page offering you the options of calling the two methods in the MultiAction controller.