Chapter 6
Lifecycles

Catalina consists of many components. When Catalina is started, these components need to be started as well. When Catalina is stopped, these components must also be given a chance to do a clean-up. For example, when the container is stopped, it must invoke the `destroy` method of all loaded servlets and the session manager must save the session objects to secondary storage. A consistent mechanism for starting and stopping components is achieved by implementing the `org.apache.catalina.Lifecycle` interface.

A component implementing the `Lifecycle` interface can also trigger one or many of the following events: `BEFORE_START_EVENT`, `START_EVENT`, `AFTER_START_EVENT`, `BEFORE_STOP_EVENT`, `STOP_EVENT`, and `AFTER_STOP_EVENT`. The first three events are normally fired when the component is started and the last three when the component is stopped. An event is represented by the org.apache.catalina.LifecycleEvent class. And, of course, if a Catalina component can trigger events, there must be event listeners that you can write to respond to those events. A listener is represented by the `org.apache.catalina.LifecycleListener` interface.

This chapter will discuss these three types `Lifecycle`, `LifecycleEvent`, and `LifecycleListener`. In addition, it will also explain a utility class called `LifecycleSupport` that provides an easy way for a component to fire lifecycle events and deal with lifecycle listeners. In this chapter, you will build a project with classes that implement the `Lifecycle` interface. The application is based on the application in Chapter 5.

The Lifecycle Interface

The design of Catalina allows a component to contain other components. For example, a container can contain components such as a loader, a manager, etc. A parent component is responsible for starting and stopping its child components. The design of Catalina is such that all components but one are put "in custody"

of a parent component so that a bootstrap class needs only start one single component. This single start/stop mechanism is made possible through the `Lifecycle` interface. Take a look at the `Lifecycle` interface in Listing 6.1.

Listing 6.1: The Lifecycle interface

```
package org.apache.catalina;
public interface Lifecycle {
  public static final String START_EVENT = "start";
  public static final String BEFORE_START_EVENT = "before_start";
  public static final String AFTER_START_EVENT = "after_start";
  public static final String STOP_EVENT = "stop";
  public static final String BEFORE_STOP_EVENT = "before_stop";
  public static final String AFTER_STOP_EVENT = "after_stop";

  public void addLifecycleListener(LifecycleListener listener);
  public LifecycleListener[] findLifecycleListeners();
  public void removeLifecycleListener(LifecycleListener listener);
  public void start() throws LifecycleException;
  public void stop() throws LifecycleException;
}
```

The most important methods in `Lifecycle` are start and stop. A component provides implementations of these methods so that its parent component can start and stop it. The other three methods—`addLifecycleListener`, `findLifecycleListeners`, and `removeLifecycleListener`—are related to listeners. A component can have listeners that are interested in an event that occurs in that component. When an event occurs, the listener interested in that event will be notified. The names of the six events that can be triggered by a `Lifecycle` instance are defined in public static final `Strings` of the interface.

The LifecycleEvent Class

The `org.apache.catalina.LifecycleEvent` class represents a lifecycle event and is presented in Listing 6.2.

Listing 6.2: The org.apache.catalinaLifecycleEvent interface

```
package org.apache.catalina;
import java.util.EventObject;

public final class LifecycleEvent extends EventObject {
  public LifecycleEvent(Lifecycle lifecycle, String type) {
    this(lifecycle, type, null);
  }
  public LifecycleEvent(Lifecycle lifecycle, String type,
    Object data) {
    super(lifecycle);
```

```
    this.lifecycle = lifecycle;
    this.type = type;
    this.data = data;
  }
  private Object data = null;
  private Lifecycle lifecycle = null;
  private String type = null;

  public Object getData() {
    return (this.data);
  }
  public Lifecycle getLifecycle() {
    return (this.lifecycle);
  }
  public String getType() {
    return (this.type);
  }
}
```

The LifecycleListener Interface

The `org.apache.catalina.LifecycleListener` interface represents a lifecycle listener and is given in Listing 6.3.

Listing 6.3: The org.apache.catalina.LifecycleListener interface

```
package org.apache.catalina;
import java.util.EventObject;
public interface LifecycleListener {
  public void lifecycleEvent(LifecycleEvent event);
}
```

There is only one method in this interface, `lifecycleEvent`. This method is invoked when an event the listener is interested in fires.

The LifecycleSupport Class

A component implementing `Lifecycle` and allowing a listener to register its interest in its events must provide code for the three listener-related methods in the `Lifecycle` interface (`addLifecycleListener`, `findLifecycleListeners`, and `removeLifecycleListener`). That component then has to store all listeners added to it in an array or `ArrayList` or a similar object. Catalina provides a utility class for making it easy for a component to deal with listeners and fire lifecycle events: `org.apache.catalina.util.LifecycleSupport`. The `LifecycleSupport` class is given in Listing 6.4.

Listing 6.4: The LifecycleSupport class

```
package org.apache.catalina.util;
import org.apache.catalina.Lifecycle;
import org.apache.catalina.LifecycleEvent;
import org.apache.catalina.LifecycleListener;

public final class LifecycleSupport {
  public LifecycleSupport(Lifecycle lifecycle) {
    super();
    this.lifecycle = lifecycle;
  }

  private Lifecycle lifecycle = null;
  private LifecycleListener listeners[] = new LifecycleListener[0];
  public void addLifecycleListener(LifecycleListener listener) {
    synchronized (listeners) {
      LifecycleListener results[] =
        new LifecycleListener[listeners.length + 1];
      for (int i = 0; i < listeners.length; i++)
        results[i] = listeners[i];
      results[listeners.length] = listener;
        listeners = results;
    }
  }

  public LifecycleListener[] findLifecycleListeners() {
    return listeners;
  }

  public void fireLifecycleEvent(String type, Object data) {
    LifecycleEvent event = new LifecycleEvent(lifecycle, type, data);
    LifecycleListener interested[] = null;
    synchronized (listeners) {
      interested = (LifecycleListener[]) listeners.clone();
    }
    for (int i = 0; i < interested.length; i++)
      interested[i].lifecycleEvent(event);
  }

  public void removeLifecycleListener(LifecycleListener listener) {
    synchronized (listeners) {
      int n = -1;
      for (int i = 0; i < listeners.length; i++) {
        if (listeners[i] == listener) {
          n = i;
          break;
        }
      }
      if (n < 0)
        return;
      LifecycleListener results[] =
        new LifecycleListener[listeners.length - 1];
      int j = 0;
      for (int i = 0; i < listeners.length; i++) {
        if (i != n)
```

```
        results[j++] = listeners[i];
      }
      listeners = results;
    }
  }
}
```

As you can see in Listing 6.4, the `LifecycleSupport` class stores all lifecycle listeners in an array called `listeners` that initially has no member.

```
private LifecycleListener listeners[] = new LifecycleListener[0];
```

When a listener is added in the `addLifecycleListener` method, a new array is created with the size of the number of elements in the old array plus one. Then, all elements from the old array are copied to the new array and the new listener is added. When a listener is removed in the `removeLifecycleListener` method, a new array is also created with the size of the number of elements in the old array minus one. Then, all elements except the one removed are copied to the new array.

The `fireLifecycleEvent` method fires a lifecycle event. First, it clones the `listeners` array. Then, it calls the `lifecycleEvent` method of each member, passing the triggered event.

A component implementing `Lifecycle` can use the `LifecycleSupport` class. For example, the `SimpleContext` class in the application accompanying this chapter declares the following variable:

```
protected LifecycleSupport lifecycle = new LifecycleSupport(this);
```

To add a lifecycle listener, the `SimpleContext` class calls the `addLifecycleListener` method of the `LifecycleSupport` class:

```
public void addLifecycleListener(LifecycleListener listener) {
  lifecycle.addLifecycleListener(listener);
}
```

To remove a lifecycle listener, the `SimpleContext` class calls the `removeLifecycleListener` method of the `LifecycleSupport` class.

```
public void removeLifecycleListener(LifecycleListener listener) {
  lifecycle.removeLifecycleListener(listener);
}
```

To fire an event, the `SimpleContext` class calls the `fireLifecycleEvent` method of the `LifecycleSupport` class, such as in the following line of code:

```
lifecycle.fireLifecycleEvent(START_EVENT, null);
```

The Application

The application accompanying this chapter builds upon the application in Chapter 5 and illustrates the use of the Lifecycle interface and lifecycle-related types. It contains one context and two wrappers as well as a loader and a mapper. The components in this application implement the Lifecycle interface and a listener is used for the context. The two valves in Chapter 5 are not used here to make the application simpler. The class diagram of the application is shown in Figure 6.1. Note that a number of interfaces (Container, Wrapper, Context, Loader, Mapper) and classes (SimpleContextValve, SimpleContextMapper, and SimpleWrapperValve) are not included in the diagram.

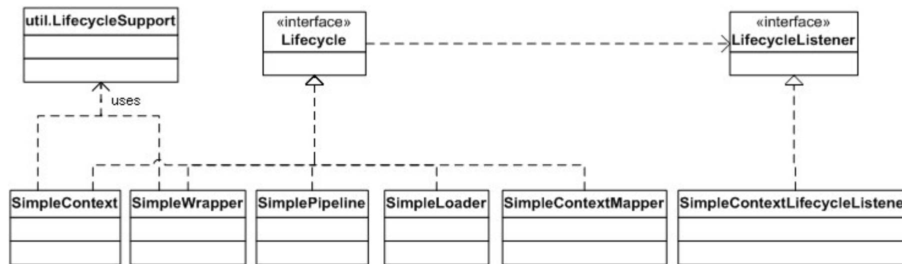

Figure 6.1 The class diagram of the accompanying application

Note that the SimpleContextLifecycleListener class represents a listener class for the SimpleContext class. The SimpleContextValve, SimpleContextMapper, and SimpleWrapperValve classes are the same as the ones in Chapter 5 and will not be discussed again here.

ex06.pyrmont.core.SimpleContext

The SimpleContext class in this application is similar to the one in Chapter 5, except that it now implements the Lifecycle interface. The SimpleContext class uses the following variable to reference a LifecycleSupport instance.

```
protected LifecycleSupport lifecycle = new LifecycleSupport(this);
```

It also uses a boolean named started to indicate if the SimpleContext instance has been started. The SimpleContext class provides implementation of the methods from the Lifecycle interface. Listing 6.5 presents these methods.

Listing 6.5: Methods from the Lifecycle interface.

```
public void addLifecycleListener(LifecycleListener listener) {
  lifecycle.addLifecycleListener(listener);
}
public LifecycleListener[] findLifecycleListeners() {
  return null;
}
public void removeLifecycleListener(LifecycleListener listener) {
  lifecycle.removeLifecycleListener(listener);
}
public synchronized void start() throws LifecycleException {
  if (started)
    throw new LifecycleException("SimpleContext has already started");
  // Notify our interested LifecycleListeners
  lifecycle.fireLifecycleEvent(BEFORE_START_EVENT, null);
  started = true;
  try {
    // Start our subordinate components, if any
    if ((loader != null) && (loader instanceof Lifecycle))
      ((Lifecycle) loader).start();

    // Start our child containers, if any
    Container children[] = findChildren();
    for (int i = 0; i < children.length; i++) {
      if (children[i] instanceof Lifecycle)
        ((Lifecycle) children[i]).start();
    }

    // Start the Valves in our pipeline (including the basic),
    // if any
    if (pipeline instanceof Lifecycle)
      ((Lifecycle) pipeline).start();
    // Notify our interested LifecycleListeners
    lifecycle.fireLifecycleEvent(START_EVENT, null);
  }
  catch (Exception e) {
    e.printStackTrace();
  }
  // Notify our interested LifecycleListeners
  lifecycle.fireLifecycleEvent(AFTER_START_EVENT, null);
}

public void stop() throws LifecycleException {
  if (!started)
    throw new LifecycleException("SimpleContext has not been started");
  // Notify our interested LifecycleListeners
  lifecycle.fireLifecycleEvent(BEFORE_STOP_EVENT, null);
  lifecycle.fireLifecycleEvent(STOP_EVENT, null);
  started = false;
  try {
    // Stop the Valves in our pipeline (including the basic), if any
    if (pipeline instanceof Lifecycle) {
      ((Lifecycle) pipeline).stop();
    }
```

```
    // Stop our child containers, if any
    Container children[] = findChildren();
    for (int i = 0; i < children.length; i++) {
      if (children[i] instanceof Lifecycle)
        ((Lifecycle) children[i]).stop();
    }
    if ((loader != null) && (loader instanceof Lifecycle)) {
      ((Lifecycle) loader).stop();
    }
  }
  catch (Exception e) {
    e.printStackTrace();
  }
  // Notify our interested LifecycleListeners
  lifecycle.fireLifecycleEvent(AFTER_STOP_EVENT, null);
}
```

Notice how the `start` method starts all child containers and associated components such as the `Loader`, `Pipeline`, and `Mapper`, and how the `stop` method stops them? Using this mechanism, to start all the components in the container module, you just need to start the highest component in the hierarchy (in this case the `SimpleContext` instance). To stop them, you only have to stop the same single component.

The `start` method in `SimpleContext` begins by checking if the component has been started previously. If it has, it throws a `LifecycleException`.

```
if (started)
  throw new LifecycleException(
    "SimpleContext has already started");
```

It then raises the `BEFORE_START_EVENT` event.

```
// Notify our interested LifecycleListeners
lifecycle.fireLifecycleEvent(BEFORE_START_EVENT, null);
```

As a result, every listener that has registered its interest in the `SimpleContext` instance's events will be notified. In this application, one listener of type `SimpleContextLifecycleListener` registers its interest. We will see what happens with this listener when we discuss the `SimpleContextLifecycleListener` class.

Next, the `start` method sets the `started` boolean to `true` to indicate that the component has been started.

```
started = true;
```

The `start` method then starts all the components and its child container. Currently there are two components that implement the `Lifecycle` interface,

SimpleLoader and SimplePipeline. The SimpleContext has two wrappers as its children. These wrappers are of type SimpleWrapper that also implements the Lifecycle interface.

```
try {
  // Start our subordinate components, if any
  if ((loader != null) && (loader instanceof Lifecycle))
    ((Lifecycle) loader).start();

  // Start our child containers, if any
  Container children[] = findChildren();
  for (int i = 0; i < children.length; i++) {
    if (children[i] instanceof Lifecycle)
      ((Lifecycle) children[i]).start();
  }

  // Start the Valves in our pipeline (including the basic),
  // if any
  if (pipeline instanceof Lifecycle)
    ((Lifecycle) pipeline).start();
```

After the components and children are started, the start method raises two events: START_EVENT and AFTER_START_EVENT.

```
// Notify our interested LifecycleListeners
lifecycle.fireLifecycleEvent(START_EVENT, null);
.
.
.
// Notify our interested LifecycleListeners
lifecycle.fireLifecycleEvent(AFTER_START_EVENT, null);
```

The stop method first checks if the instance has been started. If not, it throws a LifecycleException.

```
if (!started)
  throw new LifecycleException(
    "SimpleContext has not been started");
```

It then generates the BEFORE_STOP_EVENT and STOP_EVENT events, and reset the started boolean.

```
// Notify our interested LifecycleListeners
lifecycle.fireLifecycleEvent(BEFORE_STOP_EVENT, null);
lifecycle.fireLifecycleEvent(STOP_EVENT, null);
started = false;
```

Next, the stop method stops all components associated with it and the SimpleContext instance's child containers.

```
try {
  // Stop the Valves in our pipeline (including the basic), if any
  if (pipeline instanceof Lifecycle) {
    ((Lifecycle) pipeline).stop();
```

```
    }

    // Stop our child containers, if any
    Container children[] = findChildren();
    for (int i = 0; i < children.length; i++) {
      if (children[i] instanceof Lifecycle)
        ((Lifecycle) children[i]).stop();
    }
    if ((loader != null) && (loader instanceof Lifecycle)) {
      ((Lifecycle) loader).stop();
    }
  }
```

Finally, it raises the `AFTER_STOP_EVENT` event.

```
  // Notify our interested LifecycleListeners
  lifecycle.fireLifecycleEvent(AFTER_STOP_EVENT, null);
```

ex06.pyrmont.core.SimpleContextLifecycleListener

The `SimpleContextLifecycleListener` class represents a listener for a `SimpleContext` instance. It is given in Listing 6.6.

Listing 6.6: The SimpleContextLifecycleListener class

```
package ex06.pyrmont.core;
import org.apache.catalina.Context;
import org.apache.catalina.Lifecycle;
import org.apache.catalina.LifecycleEvent;
import org.apache.catalina.LifecycleListener;

public class SimpleContextLifecycleListener implements
LifecycleListener {

  public void lifecycleEvent(LifecycleEvent event) {
    Lifecycle lifecycle = event.getLifecycle();
    System.out.println("SimpleContextLifecycleListener's event " +
      event.getType().toString());
    if (Lifecycle.START_EVENT.equals(event.getType())) {
      System.out.println("Starting context.");
    }
    else if (Lifecycle.STOP_EVENT.equals(event.getType())) {
      System.out.println("Stopping context.");
    }
  }
}
```

The implementation of the `lifecycleEvent` method in the `SimpleContextLifecycleListener` class is simple. It simply prints the type of the event raised. If it is a `START_EVENT`, the `lifecycleEvent` method prints `Starting context`. If the event is a `STOP_EVENT`, then it prints `Stopping context`.

ex06.pyrmont.core.SimpleLoader

The `SimpleLoader` class is similar to the one in Chapter 5, except that in this application the class implements the `Lifecycle` interface. The method implementation for the `Lifecycle` interface for this class does not do anything other than printing a string to the console. More importantly, however, by implementing the `Lifecycle` interface, a `SimpleLoader` instance can be started by its associated container.

The methods from the `Lifecycle` interface in `SimpleLoader` are given in Listing 6.7.

Listing 6.7: The methods from Lifecycle in the SimpleLoader class

```
public void addLifecycleListener(LifecycleListener listener) { }
public LifecycleListener[] findLifecycleListeners() {
  return null;
}
public void removeLifecycleListener(LifecycleListener listener) { }
public synchronized void start() throws LifecycleException {
  System.out.println("Starting SimpleLoader");
}
public void stop() throws LifecycleException { }
```

ex06.pyrmont.core.SimplePipeline

In addition to the `Pipeline` interface, the `SimplePipeline` class implements the `Lifecycle` interface. The methods from the `Lifecycle` interface are left blank but now an instance of this class can be started from its associated container. The rest of the class is similar to the `SimplePipeline` class in Chapter 5.

ex06.pyrmont.core.SimpleWrapper

This class is similar to the `ex05.pyrmont.core.SimpleWrapper` class. In this application, it implements the `Lifecycle` interface so that it can be started from its parent container. In this application most of the methods from the `Lifecycle` interface are left blank except the `start` and `stop` methods. Listing 6.8 presents the method implementation from the `Lifecycle` interface.

Listing 6.8: The methods from the Lifecycle interface

```
public void addLifecycleListener(LifecycleListener listener) {  }
public LifecycleListener[] findLifecycleListeners() {
  return null;
```

```
}
public void removeLifecycleListener(LifecycleListener listener) { }
public synchronized void start() throws LifecycleException {
  System.out.println("Starting Wrapper " + name);
  if (started)
    throw new LifecycleException("Wrapper already started");
  // Notify our interested LifecycleListeners
  lifecycle.fireLifecycleEvent(BEFORE_START_EVENT, null);
  started = true;

  // Start our subordinate components, if any
  if ((loader != null) && (loader instanceof Lifecycle))
    ((Lifecycle) loader).start();
  // Start the Valves in our pipeline (including the basic), if any
  if (pipeline instanceof Lifecycle)
    ((Lifecycle) pipeline).start();
  // Notify our interested LifecycleListeners
  lifecycle.fireLifecycleEvent(START_EVENT, null);
  // Notify our interested LifecycleListeners
  lifecycle.fireLifecycleEvent(AFTER_START_EVENT, null);
}

public void stop() throws LifecycleException {
  System.out.println("Stopping wrapper " + name);
  // Shut down our servlet instance (if it has been initialized)
  try {
    instance.destroy();
  }
  catch (Throwable t) {
  }
  instance = null;
  if (!started)
    throw new LifecycleException("Wrapper " + name + " not started");
  // Notify our interested LifecycleListeners
  lifecycle.fireLifecycleEvent(BEFORE_STOP_EVENT, null);
  // Notify our interested LifecycleListeners
  lifecycle.fireLifecycleEvent(STOP_EVENT, null);
  started = false;

  // Stop the Valves in our pipeline (including the basic), if any
  if (pipeline instanceof Lifecycle) {
    ((Lifecycle) pipeline).stop();
  }
  // Stop our subordinate components, if any
  if ((loader != null) && (loader instanceof Lifecycle)) {
    ((Lifecycle) loader).stop();
  }
  // Notify our interested LifecycleListeners
  lifecycle.fireLifecycleEvent(AFTER_STOP_EVENT, null);
}
```

The `start` method in `SimpleWrapper` is similar to the `start` method in the `SimpleContext` class. It starts any components added to it (currently, there is none) and triggers the `BEFORE_START_EVENT`, `START_EVENT`, and `AFTER_START_EVENT` events.

The `stop` method in `SimpleWrapper` is even more interesting. After printing a simple string, it invokes the `destroy` method of the servlet instance.

```
System.out.println("Stopping wrapper " + name);
// Shut down our servlet instance (if it has been initialized)
try {
  instance.destroy();
}
catch (Throwable t) {
}
instance = null;
```

Then, it checks if the wrapper has been started. If not, it throws a `LifecycleException`.

```
if (!started)
  throw new LifecycleException("Wrapper " + name + " not started");
```

Next, it triggers the `BEFORE_STOP_EVENT` and `STOP_EVENT` events and reset the `started` boolean.

```
// Notify our interested LifecycleListeners
lifecycle.fireLifecycleEvent(BEFORE_STOP_EVENT, null);
// Notify our interested LifecycleListeners
lifecycle.fireLifecycleEvent(STOP_EVENT, null);
started = false;
```

Next, it stops the loader and pipeline components associated with it. In this application, the `SimpleWrapper` instances do not have a loader.

```
// Stop the Valves in our pipeline (including the basic), if any
if (pipeline instanceof Lifecycle) {
  ((Lifecycle) pipeline).stop();
}
// Stop our subordinate components, if any
if ((loader != null) && (loader instanceof Lifecycle)) {
  ((Lifecycle) loader).stop();
}
```

Finally, it triggers the `AFTER_STOP_EVENT` event.

```
// Notify our interested LifecycleListeners
lifecycle.fireLifecycleEvent(AFTER_STOP_EVENT, null);
```

Running the Application

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

```
java –classpath ./lib/servlet.jar;./ ex06.pyrmont.startup.Bootstrap
```

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

```
java -classpath ./lib/servlet.jar:./ ex06.pyrmont.startup.Bootstrap
```

You will see the following message on the console. Notice how the various events are fired?

```
HttpConnector Opening server socket on all host IP addresses
HttpConnector[8080] Starting background thread
SimpleContextLifecycleListener's event before_start
Starting SimpleLoader
Starting Wrapper Modern
Starting Wrapper Primitive
SimpleContextLifecycleListener's event start
Starting context.
SimpleContextLifecycleListener's event after_start
```

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

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

To invoke `ModernServlet`, you use the following URL.

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

Summary

In this chapter you have learned how to work with the `Lifecycle` interface. This interface defines the lifecycle for a component and provides an elegant way to send events to other components. In addition, the Lifecycle interface also makes it possible to start and stop all the components in Catalina by one single start/stop.