BEA WebLogic Server

Using WebLogic Workspaces (Deprecated)

WebLogic Server 6.0
Document Edition 1.0
December 2000

Contents

Using WebLogic Workspaces (Deprecated)

Document Edition	Date	Software Version
6.0	December 2000	BEA WebLogic Server 6.0

1 Using WebLogic Workspaces (Deprecated)

Other related documents

- Installing WebLogic (non-Windows)
- Installing WebLogic (Windows)
- Writing a WebLogic client application
- Developers Guides
- API Reference Manual
- Glossary
- Code Examples

Note: The documentation for WebLogic Server has been revised for this release. The documents listed have been superseded by the revised documentation.

Deprecation of Workspaces API

The WebLogic Workspaces API has been deprecated for this release.

See the “Data Caching Design Pattern” topic in Progamming with WebLogic JNDI for alternatives to workspaces for temporary storage:

Introduction

Overview of Workspaces

WebLogic provides many different services and facilities to its applications. Facilities, like common logs, instrumentation, configuration, and management, are available to all applications that operate within the WebLogic framework.

One of these facilities is the Workspace. The WebLogic Server hosts a set of hierarchically arranged, threadsafe Workspaces that are assigned to clients, groups of clients, and the WebLogic Server itself. A Workspace can contain any arbitrary objects, and Workspaces can be named, saved, and reentered over several sessions. Things inside a Workspace can be monitored so that an application executes certain methods before destroying or saving Workspace contents.

A T3Client can create subWorkspaces inside its system-assigned client Workspace and can leave and reenter the same work area again and again, depending upon how the WebLogic Server is configured to clean up a T3Client's resources (see Timeouts for more information on T3Client lifetime).

Various information about the Client—its context or state—is stored in its Workspace, including information associated with its T3User object, JDBC connections, etc.

Currently there are two levels of Workspaces, at the client level (defined by the final static int `SCOPE_CLIENT` in WorkspaceDef) and at the server level (defined by the final static int `SCOPE_SERVER` in WorkspaceDef).

Workspaces also offer a way to share objects. An object stored in a client's Workspace can be accessed by other clients. Objects may also be stored in the server Workspace, to which all clients have access. The WebLogic Server itself can store an object (like a database result set) in the server Workspace, and all of the WebLogic Server's clients can be granted access to the result set.

The API

WebLogic Workspace API reference

For a detailed reference on the WebLogic Workspace API, see the following packages in the WebLogic Server API Reference.

- `weblogic.common`
- `weblogic.workspace.common`

Overview of the WebLogic Workspace API

Workspaces are one of the powerful tools that a T3Client has in the WebLogic framework. The API for Workspaces is closely intertwined with that for the T3Client; many of those interfaces and classes are found in the `weblogic.common` package. Here we overview how the Workspace-related classes in the `weblogic.common` package and the `WorkspaceDef` interface in `weblogic.workspace.common` are related and interoperate.

Implementing with WebLogic Workspaces

Getting access to Workspaces

The currently available services and facilities, includingWorkspaces, are accessed through factory methods. Conceptually, a factory method allows the allocation of resources inside the WebLogic Server in a configurable, controllable way. Factory methods take the place of constructors.

A T3Client gains access to the Workspace facility through its **services** object. When you create a T3Client, a default Workspace is automatically created for the T3Client. The code for requesting access to the default Workspace follows this pattern:

```
T3Client t3 = new T3Client("t3://localhost:7001");
t3.connect();
WorkspaceDef defaultWS =
  t3.services.workspace().getWorkspace();
```

The WorkspaceDef object provides access to all of the Workspace functionality available to a T3Client. Before you do any work with Workspaces in a T3Client application, you must get a reference to the WorkspaceDef object that defines the default T3Client Workspace.

Creating, leaving, and reentering Workspaces

`weblogic.workspace.common.WorkspaceDef`

`weblogic.common.WorkspaceServicesDef.getWorkspace()`

Using a Workspace ID to create and reenter a Workspace

You create a client Workspace by default when you create a T3Client. All Workspaces, including the default Workspace, are always identifiable by the unique ID that the WebLogic Server assigns to the Workspace when it is created. By retrieving and saving the ID of the Workspace, you can then use the Workspace ID to reenter the same Workspace with a different T3Client, as shown here:

```
T3Client t3 = new T3Client("t3://localhost:7001");
t3.connect();
// Set the Workspace timeout so the Workspace
// will hang around even after the client disconnects
t3.setSoftDisconnectTimeoutMins(T3Client.DISCONNECT_TIMEOUT_NEVER
);

// Retrieves the default T3Client Workspace
WorkspaceDef defaultWS =
  t3.services.workspace().getWorkspace();
String wsid = defaultWS.getID();
// . . . do some work . . .
t3.disconnect();

// Reconnect, passing the workspace ID wsid
T3Client newt3 = new T3Client("t3://localhost:7001", wsid);
newt3.connect();
// . . . finish up work . . .
// When work is finished, set the Workspace for immediate
// cleanup and disconnect
newt3.setSoftDisconnectTimeoutMins(0);
newt3.disconnect();
```

Using a Workspace name to create and reenter a Workspace

You can also set the name of the default Workspace by supplying a name for the Workspace as an argument when the T3Client is created. The Workspace name can be used later to reenter the Workspace, as in this example:

```
T3Client t3 = new T3Client("t3://localhost:7001",
                           "MY_CLIENT_WS");
t3.connect();
// Set the Workspace timeout so the Workspace
// will hang around even after the client disconnects
t3.setSoftDisconnectTimeoutMins(T3Client.DISCONNECT_TIMEOUT_NEVER
);
// . . . do some work . . .
t3.disconnect();

T3Client newt3 = new T3Client("t3://localhost:7001",
                              "MY_CLIENT_WS");
newt3.connect();
// . . . finish up work . . .
// When work is finished, set the Workspace for immediate
// cleanup and disconnect
newt3.setSoftDisconnectTimeoutMins(0);
newt3.disconnect();
```

Setting the soft disconnect timeout to `NEVER` means that the WebLogic Server will not cleanup the client's serverside resources, even though the client disconnects and goes away. Technically, you can delay the cleanup of client workspace resources forever (or as long as the WebLogic Server is running), but practically, for performance and efficiency, you will want to clean up resources as clients finish with them. Once you have finished the client's work, you can set the soft disconnect timeout to zero, which forces the WebLogic Server to release the T3Client's serverside resources as soon as the client disconnects.

You can also create and name a subWorkspace that exists as a child of a certain Workspace (by default, as a child of the default T3Client Workspace). Note that Workspaces that you create are always (technically) created as a subWorkspace; for example, if you create a Workspace with `SCOPE_SERVER`, it will be created as a subWorkspace of the server Workspace that was created at server startup. You also can't create mis-matched subWorkspaces; for example, you can't create a subWorkspace of the server Workspace that has a scope of `SCOPE_CLIENT`.

SubWorkspaces operate exactly as any other Workspaces; they allow the developer to more finely control how and where application or client objects are stored. The methods in WorkspaceDef operate on any Workspace, no matter its scope, except that (by definition) you can only create subWorkspaces, and you can only destroy those subWorkspaces that you have created. In these docs, we do not differentiate between subWorkspaces and Workspaces.

This example illustrates how to name a subWorkspace and use it for storing and retrieving Objects. To use a named subWorkspace, first get access to the T3Client's default Workspace as shown here:

```
T3Client t3 = new T3Client("t3://localhost:7001");
t3.connect();
WorkspaceDef defaultWS =
  t3.services.workspace().getWorkspace();
```

Then create a subWorkspace as a child of the T3Client's default Workspace. (Note that all child Workspaces are destroyed when the parent is destroyed, which means that when you destroy the T3Client's default Workspace, any named Workspaces that you have created will be destroyed as well.)

```
WorkspaceDef subWS = defaultWS.getWorkspace("DATA_STORE");
```

You can specify a **scope** when you create the Workspace that controls where the Workspace is created. You can create a subWorkspace that is scoped at the following levels, all of which are defined as final static ints in the WorkspaceDef interface:

- `SCOPE_CLIENT` is for creating subWorkspaces of the T3Client workspace. If you do not specify scope when getting a WorkspaceDef, the default sets the scope to `SCOPE_CLIENT`.
- `SCOPE_SERVER` creates a subWorkspace at the WebLogic Server system level. All T3Clients with access to the WebLogic Server have access to objects stored within the server scope. You will probably use this scope for storing Objects for general client use created by startup classes. Note that for security reasons, the contents of workspaces of scope server are not shown in either the WebLogic Console or the Admin servlets.

You can also specify a **mode** when you create the Workspace that controls how the Workspace is created. There are three modes for creating or reentering a Workspace that are defined as final static ints in the `WorkspaceDef` interface:

- `WorkspaceDef.CREATE`, which creates a new Workspace only if an existing Workspace that matches it does not already exist.

- `WorkspaceDef.ATTACH`, which attaches to an existing Workspace only if an existing Workspace that matches it already exists.
- `WorkspaceDef.OPEN`, which creates a new Workspace if a matching one does not exist, or attaches to an existing Workspace. This mode is used as default if you do not supply a mode.

With **mode**, you can be very selective about the conditions under which you want to enter a Workspace. If, for example, you have created a Workspace named `DATASTORE_SPACE` at the system scope and stored an object in it, you can make certain that you return to that Workspace by using the `ATTACH` mode when you get the Workspace. If you attempt to `ATTACH` to a Workspace that doesn't exist, your application will throw an Exception that you can catch and act upon.

In the same way, you can be certain that you get a completely new Workspace by using the `CREATE` mode. If you call the `getWorkspace()` method in the `CREATE` mode and supply a Workspace name or ID to create that already exists, your application will throw an Exception.

Here is the example we used above, except here we create a subWorkspace with the mode set to `WorkspaceDef.CREATE`:

```
T3Client t3 = new T3Client("t3://localhost:7001");
t3.connect();
WorkspaceDef defaultWS =
  t3.services.workspace().getWorkspace();
WorkspaceDef subWS =
  defaultWS.getWorkspace("DATA_STORE",
                         WorkspaceDef.CREATE);
```

You can create subWorkspaces of subWorkspaces to organize objects in the way that makes the most sense for your application.

You can get an Enumeration of the names of all the subWorkspaces of a Workspace with the `WorkspaceDef.subspaces()` method.

Using named Workspaces to store and fetch objects

Workspaces are useful to store objects in. You store and fetch objects as name-value pairs. You can manipulate arbitrary Objects, with the following restrictions:

- The Object is a `java.lang` or `java.util` Object.

- Or, the Object is `java.io.serializable`.

There are three methods for managing objects in Workspaces. All of them use a String **key** to refer to the object.

- `WorkspaceDef.store(String key, Object p)` stores an Object.
- `WorkspaceDef.fetch(String key)` retrieves a stored Object non-destructively.
- `WorkspaceDef.remove(String key)` retrieves a stored Object and removes it from the Workspace.

You can get an Enumeration of all the keys in a particular Workspace with the method `WorkspaceDef.keys()`.

Here is an example of storing a WebLogic JDBC ResultSet in a Workspace that we create as `DATA_SPACE`. `DATA_SPACE` is a subWorkspace of the system Workspace.

```
T3Client t3 = new T3Client("t3://toyboat.toybox.com:7001");
t3.connect();
// Get the default T3Client Workspace

WorkspaceDef defaultWS =
  t3.services.workspace().getWorkspace();
WorkspaceDef dataWS =
  defaultWS.getWorkspace("DATA_WORKSPACE",
                         WorkspaceDef.CREATE,
 WorkspaceDef.SCOPE_SERVER);
// . . . Connect to the DB and get a ResultSet rs . . .
// Then store it in the system subWorkspace we created
dataWS.store("MyResults", rs);
t3.disconnect();
```

Here is the reverse: fetching the ResultSet. In this example, we attach to the already-existing subWorkspace `DATA_SPACE`. Since we want to leave the ResultSet intact in the Workspace for use by other clients, we will use the `fetch()` method to get the ResultSet, rather than the `remove()` method, which is a destructive fetch.

```
T3Client t3 = new T3Client("t3://toyboat.toybox.com:7001");
t3.connect();
// Get the default T3Client Workspace
WorkspaceDef defaultWS =
  t3.services.workspace().getWorkspace();
// Attach to the system subWorkspace already created
WorkspaceDef myDataWS =
  defaultWS.getWorkspace("DATA_WORKSPACE",
                                WorkspaceDef.ATTACH,
 WorkspaceDef.SCOPE_SERVER);
// Get the ResultSet and clean up
ResultSet rs = (ResultSet) myDataWS.fetch("MyResults");
t3.disconnect();
```

Converting from the old model to the new

If you are moving from release 2.4, you will find that the major difference in coding usage with the new Workspace object is accessing the Workspace facility.

The methods that allow you to get access to and operate on Workspaces were originally in the T3Client class. Here is how you used to access the facility:

```
T3Client t3 = new T3Client("t3://localhost:7001");
t3.connect();
String wsid = t3.getID();
```

With the new model, you access the Workspace facility through the T3Client's services stub and the `weblogic.common.workspace.WorkspaceDef` interface, as shown here:

```
T3Client t3 = new T3Client("t3://localhost:7001");
t3.connect();
WorkspaceDef defaultWS =
  t3.services.workspace().getWorkspace();
```

Then you use methods in that interface to operate on the WorkspaceDef. The methods `getID()` and `getName()` are identical to the same methods that were formerly in the T3Client class. The methods for storing in and fetching from workspaces are similar to the same methods that were formerly in `weblogic.common.WorkspaceServicesDef`. These have been deprecated in favor of the more powerful model in `WorkspaceDef`.

Using Workspace Monitors

The new interface WorkspaceDef also gives you access to Monitor functionality, which allows you to call user-written methods on Workspace contents before and after certain operations, like save and destroy. Monitors are useful for implementing business rules, where workflow and data validation are enforced by a global policy.

Classes and interfaces of interest in working with Monitors:

Setting up Monitors

```
    weblogic.common.SetMonitor
    weblogic.common.GetMonitor
    weblogic.common.DestroyMonitor
    weblogic.common.MonitorException
```

Activating Monitors

```
    weblogic.common.GetMonitor
    weblogic.common.Monitor
    weblogic.common.MonitorDef
```

For example, here are some operations for which you might use monitors. We'll use these examples later in this section to demonstrate how to write and use monitors.

- Restricting a workspace value to a particular range. Your application can ensure that values fall into a particular range by checking the value prior to an operation and throwing a MonitorException if a value is invalid.
- Triggering an event when a Workspace value is destroyed. Because your application is operating in the WebLogic framework, you have access to other WebLogic APIs, such as events. You can use Events and Monitors together.
- Mirroring a Workspace value in another Workspace. You can use Monitors to mirror changes in one Workspace to another Workspace.
- Altering/encoding a Workspace value when set. With a Master Monitor it is possible to capture any attempt to set the value and substitute an encoded value, which can be used for data compression and encryption of sensitive information.

Types of Monitors

There are two types of Monitors: regular Monitors, which we refer to by default with the term Monitor; and Master Monitors. There can be only a **single** Master Monitor for each monitored object; there may be many regular Monitors on a single Object.

Regular Monitors are using for merely watching values and perhaps reporting on their changes, or even vetoing an operation on those values. Regular Monitors can't actually affect the value itself. There is no guaranteed order of monitor operations if there are multiple monitors on a value.

A Master Monitor is identical to a regular monitor except for two characteristics:

- A Master Monitor is guaranteed to be the first monitor executed before a monitored operation is performed, and the first to be executed after a monitored operation is performed. That is, its *pre* and *post* operations preempt the same operations of regular Monitors.
- Master Monitors can alter the state of a monitored object. While a Master Monitor is executing, the target object is unlocked and can be altered by the Master Monitor. Regular monitors can only observe (and potentially veto) alterations to an object's state. Here is an example of how a Master Monitor might change a value of an Object target, an operation that a regular Monitor couldn't complete:

```
public void preSet(Setable target, ParamSet callbackData)
    throws MonitorException {
  target.newValue("ALTERED Value");
}
```

A Master Monitor is created from a regular Monitor by calling the Monitor's setMaster() method with the argument true. If there already exists a Master Monitor for the target object, a MonitorException will be thrown. Here is a code excerpt that creates a Monitor and then sets it to be a Master Monitor:

```
Monitor monitor = new Monitor("mycode.MyMonitor", ps);
monitor.setMaster(true);

try {
  workspace.addMonitor(key, monitor);
}
catch (Exception e) {
  inform("addMonitor failed: a Master Monitor " +
         "may already be installed.");
}
```

How Monitors work

What a Monitor monitors depends on the interfaces that it implements and on what operations the monitored object supports. Monitorable objects must implement Setable, Getable, and or Destroyable. Currently, only Workspace implements these interfaces, so only Workspace values can be monitored.

Workspaces support monitoring of the following operations:

- Set
- Get

- Destroy

To monitor an object in a Workspace, you write a class that does something—anything you want—whenever a Workspace object that implements Destroyable, Setable, and/or Getable has one of its `*xxx*Value()` methods called.

Monitors allows your user-written code to be executed whenever a particular Workspace object is queried, modified, or destroyed.

There are six points in at which a Monitor can intervene in a Workspace object's operations:

1. *preSet* and *postSet*, called before or after a Workspace object's `setValue()`, `newValue()`, `oldValue()`, etc., operation has completed. A preSet operation is blocked by throwing a MonitorException.
2. *preGet* and *postGet*, called before or after a get operation is performed. A preGet operation can be blocked by throwing a MonitorException.
3. *preDestroy* and *postDestroy*, called before or after a destroy operation has completed. A preDestroy operation can be blocked by throwing a MonitorException.

Any Monitor can veto an operation (in the *pre* stage) by throwing a MonitorException. When an operation is vetoed, all Monitors are notified of the veto. A Master Monitor can override the veto.

In order to monitor set, get, or destroy operations, your Monitor must implement one or more of the interfaces:

- `SetMonitor`
- `GetMonitor`
- `DestroyMonitor`

Within your class, you can set up tasks to be called before and after set, get, and destroy operations are called on a specific Workspace object. A class that implements all three interfaces will have the appropriate methods for get, set, and destroy operations, which are implemented by the developer.

Practically, there are two parts of monitoring:

- Write the class that does the monitoring—that carries out some operation before or after set, get, and destroy operations are performed on Workspace objects.

This class must implement one or more of the Monitor interfaces, `SetMonitor`, `GetMonitor`, or `DestroyMonitor`.

- Write the class that adds the Monitor to a Workspace. The Monitor that you write has to be instantiated by WebLogic, which is done by adding the Monitor to the Workspace in which it will operate.

First we provide a simple example of each step. Then we explicate the process with four different examples.

Writing a Monitor

Your Monitor must implement one or more of the `SetMonitor`, `GetMonitor`, or `DestroyMonitor` interfaces; and another class that adds the Monitor to the Workspace so that it operates on the Workspace's values.

The operations that a Monitor monitors depends upon which interfaces are implemented by the objects being monitored. Here is an example of a very simple Monitor that monitors set operations.

```
package mycode;

import weblogic.common.*;

public class MyMonitor implements SetMonitor {
  public T3ServicesDef services;
  public void monitorInit(ParamSet params, boolean isMaster) {}

  public void setServices(T3ServicesDef services) {
    this.services = services;
  }

  public void preSet(Setable target,
                     ParamSet callbackData)
    throws MonitorException
  {
    System.out.println("preSet called");
  }

  public void postSet(Setable target,
                      ParamSet callbackData,
      Exception e) {
    System.out.println("postSet called");
  }
}
```

Adding a Monitor to a Workspace

`weblogic.workspace.common.WorkspaceDef.addMonitor()`

To put a Monitor to work, you need to add the Monitor to a Workspace, where it then monitors the values in the Workspace. You do this by calling the `WorkspaceDef.addMonitor()` method.

The `addMonitor()` method takes two parameters, a name by which you can identify it later, and a Monitor object. The Monitor object is essentially a wrapper for a user-written MonitorDef—that is, a class you have written that implements `SetMonitor`, `GetMonitor`, or `DestroyMonitor`, each of which implement MonitorDef.

Here is an example of how you construct the Monitor that you pass to the `addMonitor()` method. Potentially you can use an assortment of these arguments in the Monitor constructor, depending upon whether the Monitor needs initialization parameters and whether it is constructed by a client or server:

- The name of the MonitorDef class (that is, a class that implements `SetMonitor`, `GetMonitor`, or `DestroyMonitor`). The class must be in the CLASSPATH of your WebLogic Server. If you are adding a Monitor from a WebLogic Server, you can alternatively pass an instance of the MonitorDef class to the `addMonitor()` method instead.
- Optionally, a ParamSet object that is a set of initialization parameters. Since the default constructor—without any arguments—must be used to instantiate a class remotely, we pass a set of initialization parameters which are evaluated by the `monitorInit()` method as soon as the Monitor is instantiated.
- Optionally, a ParamSet object that is a set of callback parameters. If you add a Monitor from a client, you might want to set some callback parameters. Callback parameters are passed to the pre and post methods defined by the Monitors.

Here is an example of the code you might use to add a Monitor to the default client Workspace:

```
T3Client t3 = new T3Client("t3://localhost:7001");
t3.connect();
ParamSet initps = new ParamSet();
ParamSet cbps   = new ParamSet();

WorkspaceDef defaultWS =
  t3.services.workspace().getWorkspace();
initps.setParam("topic", "newBooks");
Monitor myMon =
  new Monitor("mycode.MyMonitor", initps, cbps);
defaultWS.addMonitor("topic", myMon);
```

Monitor Examples

In these examples, we show the code that adds the Monitor, and in more detail, the code for the Monitor itself. There are four examples. Each is detailed fully in the code examples in the examples/workspace/monitor directory in the distribution.

Note: The workspace code examples are not packaged with this release.

- Example 1. Restricting the range of a Workspace value.
- Example 2. Triggering an event with a Monitor.
- Example 3. Mirroring Workspace values in another Workspace.
- Example 4. Altering a Workspace value with a Master Monitor.

Example 1. Restricting the range of a Workspace value

`examples.workspace.monitor.RangeMonitor`

This example of a Monitor restricts a Workspace value to a particular range. As a client changes the Workspace value, the Monitor checks the range before the value is set and throws a MonitorException if the value is out of the prescribed range.

How addMonitor() is called

```
  T3Client t3 = new T3Client("t3://localhost:7001");
  t3.connect();
  // Create 2 ParamSets to be used with the pre-
  // and post- operations.
  // Since we do not really need params, we'll leave them empty.
  ParamSet initPS = new ParamSet();
  ParamSet callbackPS = new ParamSet();

  // Get the Workspace
  WorkspaceDef defaultWS =
    t3.services.workspace().getWorkspace();
  WorkspaceDef workspace = defaultWS;
  try {
    Monitor rangeMonitor =
      new Monitor("examples.workspace.monitor.RangeMonitor",
                  initPS, callbackPS);

    workspace.addMonitor("key3", rangeMonitor);
    System.out.println("Setting Value within range 0-100");
    workspace.store("key3", new Integer(50));
    try {
      System.out.println("Setting Value outside range 0-100");
      workspace.store("key3", new Integer(150));
    }
    catch (T3Exception ex) {
      System.out.println("Received Exception for " +
                         "out-of-range value");
    }
  catch (Exception e) {
    e.printStackTrace();
  }
  t3.disconnect();
```

The Monitor itself

In this example (the full example is in `examples/workspace/monitor/RangeMonitor.java`) we show just the implementation of the `preSet()` and `postSet()` methods—although all the work is done in the `preSet()` method. The class implements the `SetMonitor` interface and is monitoring a Workspace object passed in as target.

Note: The workspace code examples are not packaged with this release.

```
  public void preSet(Setable target,
                     ParamSet callbackData)
    throws MonitorException
  {
    if (!target.newValue() instanceof Integer) {
      throw MonitorException("Value must be of type Integer");
    }
    integer newValue = ((Integer) target.newValue()).intValue();
    if (newValue < 0 || newValue > 100) {
      throw MonitorException(newValue +
                             " must be between 0 and 100");
  }

  public void postSet(Setable target,
                      ParamSet callbackData,
      Exception e) {}
```

Example 2. Triggering an event with a Monitor

This example of a Monitor triggers an event when a Workspace value is destroyed. Notice that the class in which the Monitor is added must implement `weblogic.event.actions.ActionDef`.

How addMonitor() is called

Here we send an event when a Workspace value is removed. For more information on submitting events, check the Developers Guide Using WebLogic Events.

In this example, for the sake of brevity, we skip the code fragment for creating and connecting the T3Client (t3), creating the initialization and callback parameters (initPS and callbackPS), and getting the Workspace (workspace). Check the first example for details, or see the full code example in `examples/workspace/monitor/MonitorDemo.java`.

Note: The workspace code examples are not packaged with this release.

DestructionMonitor sends an event when a Workspace value is removed. The lock is used to force the thread to wait until the message notification arrives.

```
initPS.setParam("topic", "destroyTopic");
Monitor destuctionMonitor =
  new Monitor("examples.workspace.monitor.DestructionMonitor",
              initPS, callbackPS);

Evaluate eval =
  new Evaluate("weblogic.event.evaluators.EvaluateTrue");
Action action = new Action(this);

EventRegistrationDef destroyReg =
  t3.services.events().getEventRegistration("destroyTopic",
                                            eval, action);
destroyReg.register();

workspace.addMonitor("key1", destuctionMonitor);
System.out.println("Storing value under key1");
workspace.store("key1", "testValue");
System.out.println("Removing key1 from workspace " +
                   "(expect message notification)");
synchronized(md.lock) {
  workspace.remove("key1");
  md.lock.wait();
}
```

```
// This implements ActionDef, which this class must
// do in order to submit an EventRegistration
public void setServices(T3ServicesDef services) {}

public void registerInit(ParamSet ps) {}

public void action(EventMessageDef message) {
  System.out.println("Message Received: " + message.getTopic());
  synchronized(lock) {
    lock.notify();
  }
}
```

The Monitor itself

In this example, we show only the `postDestroy()` method, which is where the work is done. The class implements the DestroyMonitor interface and passes in a Destroyable Workspace object called target.

```
  public void postDestroy(Destroyable target,
                          ParamSet callbackData, Exception e) {
    if (e == null) {
      ParamSet ps = new ParamSet();
      try {
        if (target instanceof WorkspaceValue) {
  ps.setParam("key", ((WorkspaceValue) target).getKey());
}
EventMessageDef em =
  services.events().getEventMessage(topic, ps);
em.submit();
      }
      catch (ParamSetException pse) {
      }
      catch (EventGenerationException ege) {}
    }
  }
```

Example 3. Mirroring Workspace values in another Workspace

How addMonitor() is called

In this example, for the sake of brevity, we skip the code fragment for creating and connecting the T3Client (t3), creating the initialization and callback parameters (initPS and callbackPS), and getting the Workspace (workspace). Check the first example for details, or see the full code example in `examples/workspace/monitor/MonitorDemo.java`.

Note: The workspace code examples are not packaged with this release.

```
try {
  // MirrorMonitor uses a SetMonitor and a DestroyMonitor
  // to mirror the state of a workspace value in
  // another workspace.
  initPS.setParam("mirror", "mirroredWorkspace");
  Monitor mirrorMonitor =
    new Monitor("examples.workspace.monitor.MirrorMonitor",
                initPS, callbackPS);

  workspace.addMonitor("key2", mirrorMonitor);
  String mirrorVal = "mirror this";
  workspace.store("key2", mirrorVal);
  System.out.println("Set key2 = " + mirrorVal +
                     ", in default workspace");
  WorkspaceDef mirror =
    defaultWS.getWorkspace("mirroredWorkspace",
                           WorkspaceDef.OPEN,
                           WorkspaceDef.SCOPE_SERVER);
  String mirroredValue =
    (String) mirror.fetch("key2");
  System.out.println("Got key2 = " + mirrorVal +
                     ", in workspace " + mirror.getName());

  }
  catch (Exception e) {
    e.printStackTrace();
  }

  // Disconnect the client. Since the soft disconnect
  // timeout is NEVER, the WebLogic Server will
  // preserve the session.
  t3.disconnect();
}
```

The Monitor itself

This Monitor mirrors a Workspace value in another Workspace; each time the value in the first Workspace changes, the Monitor causes the mirrored value to change.

Here we implement both the `SetMonitor` and the `DestroyMonitor` interfaces, since the Monitor operates both set and destroy operations. The Workspace object being monitored (it implements both Setable and Destroyable) is passed in as target.

We also take advantage of the monitorInit() method in this Monitor class and its ParamSet. In this case, we use the ParamSet to pass the name of the mirrored Workspace to the monitoring class.

```
public class MirrorMonitor implements SetMonitor,
                                      DestroyMonitor {
  public T3ServicesDef services;
  WorkspaceDef mirror = null;

  public void setServices(T3ServicesDef services) {
    this.services = services;
  }

  public void monitorInit(ParamSet params, boolean isMaster)
    throws ParamSetException
  {
    String mirrorName = params.getParam("mirror").asString();
    mirror =
      services.workspace().getWorkspace(mirrorName,
                            WorkspaceDef.OPEN,
                    WorkspaceDef.SCOPE_SERVER);
  }

  public void preSet(Setable target, ParamSet callbackData)
    throws MonitorException {}

  // Implement this to mirror changes
  public void postSet(Setable target,
                     ParamSet callbackData,
     Exception e)
  {
    if (e == null && target instanceof WorkspaceValue) {
      try {
        mirror.store(((WorkspaceValue)target).getKey(),
               target.newValue());
      }
      catch (T3Exception t3e) {}
    }
  }

  public void preDestroy(Destroyable target,
                         ParamSet callbackData) {}

  // Implement this to mirror deletes
  public void postDestroy(Destroyable target,
                          ParamSet callbackData,
  Exception e) {
    try {
      mirror.destroy(((WorkspaceValue)target).getKey());
    }
    catch (T3Exception t3e) {}
  }
}
```

Example 4. Altering a Workspace value with a Master Monitor

A monitored object may have at most one Master Monitor. The Master Monitor is guaranteed to be the first Monitor executed before or after a monitored operation can be performed. Unlike a Monitor that is not the master, a Master Monitor can also modify the target object's state; that is, the target object is unlocked while the Master Monitor is running.

You make a Monitor into a master by calling `Monitor.setMaster(true)` on the Monitor itself. If a Master Monitor has already been set, a MonitorException is thrown.

Here is an example of a Master Monitor. In this case, we want to obscure (encode or encrypt) the value that is stored in the Workspace object for security reasons.

How addMonitor() is called and the Monitor is set to Master

```
  try {
    // XXXMonitor is an example of a MasterMonitor, which
    // has the ability to alter the value being monitored.
    // The monitor replaces the supplied value with "XXX".
    Monitor xxxMonitor =
      new Monitor("examples.workspace.monitor.XXXMonitor",
                  initPS, callbackPS);
    xxxMonitor.setMaster(true);

    String val = "hello world";
    workspace.addMonitor("key4", xxxMonitor);
    workspace.store("key4", val);
    System.out.println("Set Value: "+val);
    val = (String) workspace.fetch("key4");
    System.out.println("Got Value: "+val);
  }
  catch (Exception e) {
    e.printStackTrace();
  }
```

The Monitor itself

In this simple example, we simply replace the value with `XXX`; you might write the `preSet()` method instead to use a form of encryption so that the value would also be decodable when retrieved.

```
public class XXXMonitor implements SetMonitor {
  T3ServicesDef services;
  boolean isMaster;

  public void monitorInit(ParamSet params, boolean isMaster) {
    this.isMaster = isMaster;
  }

  public void setServices(T3ServicesDef services) {
    this.services = services;
  }

  public void preSet(Setable target, ParamSet callbackData)
    throws MonitorException
  {
    if (isMaster) {
      target.newValue("XXX");
    }
  }

  public void postSet(Setable target,
                      ParamSet callbackData,
      Exception e) {}
}
```

Removing a Monitor from a Workspace

A Monitor can be removed by calling the `removeMonitor()` method on the Workspace object, as shown here:

```
  workspace.removeMonitor(key, monitor);
```

removes the Monitor for the Workspace value specified by *`key`*. You can also call the method with a single argument, the Monitor itself.

Setting up ACLs for Workspaces in the WebLogic Realm

`weblogic.workspace`

`weblogic.workspace.`*`namedWorkspace`*

WebLogic controls access to internal resources like Workspaces through ACLs set up in the WebLogic Realm Entries for ACLs in the WebLogic Realm are listed as properties in the weblogic.properties file.

Note: This release of WebLogic Server provides a Web-based Administration Console for all configuration and administration tasks. The weblogic.properties file is no longer used.

You can set the Permissions `read` and `write` for Workspaces and named Workspaces by entering a property in the properties file. The ACL name `weblogic.workspace` controls access to all Workspaces created. If you leave the ACL for a user's named Workspace unset, an ACL is created dynamically for the user at login by copying the ACL for `weblogic.workspace` and adding read and write Permissions which allows each user to use its own Workspace without an explicit ACL, but allows users to be given explicit read and write Permission for other workspaces.

Here are some examples. The first ACL (the first pair of ACL entries) allows four users to read and write objects in the T3UserSales named Workspace, as well as in the Workspace created for each user at connect. The second ACL gives the T3User `sysMonitor` read and write access for every workspace; as each Workspace is created, this ACL will be copied and read and write Permissions for the user logging in will be added to it.

Example:
```
weblogic.allow.read.weblogic.workspace.T3UserSales=karl,michael,s
kip,msmithweblogic.allow.write.weblogic.workspace.T3UserSales=kar
l,michael,skip,msmithweblogic.allow.read.weblogic.workspace=sysMo
nitorweblogic.allow.write.weblogic.workspace=sysMonitor
```