Observing Data Grid Events

Objectives

After completing this lesson, you should be able to:

- Describe Event concepts, including:
  - Event life cycle
  - Event handler hooks
- Describe, compare, and contrast event handlers types
- Implement and register event interceptors
- Implement and register map listeners
- Filter and transform map listener events
- Implement and register map triggers
- Implement and use continuous query caches

6 - 2

Event Concepts

Applications can listen for events which include:

- Data changes, such as add, update, or delete
- Cache changes, such as backup creation, cluster rebalancing, and data partitioning

Beyond the scope of this course

Events are:

- Application Level: Events can be registered locally and processed within the application that registered for them.
- Remote Level: Events can be registered remotely, and processed where data updates occur.
- Implemented via:
  - Event listeners, implementing the `MapListener`
  - Map triggers, implementing the `MapTrigger`

6 - 3

Coherence applications can take advantage of a rich event model which supports events in a variety of ways.

- **Data Changes:** Coherence provides event hooks for data changes, such as add, delete, and update. These can be pre- or post events, with event hooks being called before or after the event has occurred. Such changes can even be denied.
- **Location:** Events can be signaled to an application, to a handler within a cluster node, or both, or a subset of the cluster.

Application Versus Server Side Events

6 - 4

Coherence supports two event categories and several event mechanisms. Categories of events include application side and server side events. Mechanisms include listeners, triggers.

In all cases, an event is generated and passed to a handler for review. A number of different event types exist, including:

- Partition cache events, raised based on CRUD operations on entity objects
- Service events, including data migration between instances

And

- Cache life cycle, the creation or destruction of caches

In the case of application side events, all registered handlers are called in the order they are registered, often programmatically. Likewise, on the server side, all handlers registered, typically declaratively, are called in the order they are registered. In either case, both pre- and post-operation events are supported via the `MapListener`, `MapTrigger` interfaces.

Event Handler Mechanisms

Coherence supports three distinct event mechanisms:

Name & Mechanism	Description
Live Events `EventInterceptor<T>`	• Server side • Support synchronous-pre and asynchronous-post mutation data events • Support Service and Cache events
Map Trigger Events `Trigger`s	• Application or Server side • Support synchronous-pre data mutation events
Map Events `MapListener`s	• Application or Server side • Support asynchronous-post data mutation events • Can be used with complex filters

6 - 5

With the release of Coherence 12*c*, there are now three distinct event mechanisms. Choosing an event handling mechanism is based on need and personal choice. However, there are some restrictions based on the capabilities of the method as well as the specific requirements.

- `EventInterceptor<T>`s, released in Coherence 12*c*, are the most versatile of all event mechanisms. Interceptors are typed to the variety of the event being intercepted and support three specific event classes, data events, partition events, and service events. However, interceptors can only be registered declaratively via a coherence-cache-config.xml
- `Trigger`s, provide both application and server side event support but are limited to synchronous pre-data mutation (executed based on CRUD operations) only.
- `MapListener`s, provide both application and server side vent support but are limited to asynchronous post-data mutation only. `MapListener`s also support a filtering capability to limit which data elements trigger events.

Live Event Model: Overview

- The Live Event Model is the first step in unifying application and server events.
- The Live Event Model:
  - Simplifies server-side events
  - Is completely asynchronous
  - Is triggered based on pre/post event life cycle such as:
    - Inserting, inserted
    - Updating, updated
    - Removing, removed
  - Supports declarative registration
  - Uses annotation to restrict event delivery
  - Is monitorable via JMX

Declaratively = in coherence-cache-config.xml

6 - 6

Coherence 12*c* now provides a Live Event Model. This framework uses an event-based model where events represent observable occurrences of cluster operations. The events that are currently supported include:

- Partitioned Cache Events – A set of events that represent the operations being performed against a set of entries in a cache. Partitioned cache events include both entry events and entry processor events. Entry events are related to inserting, removing, and updating entries in a cache. Entry processor events are related to the execution of entry processors.
- Partitioned Service Events – A set of events that represent the operations being performed by a partitioned service. Partitioned service events include both partition transfer events and partition transaction events. Partition transfer events are related to the movement of partitions among cluster members. Partition transaction events are related to changes that may span multiple caches and are performed within the context of a single request.
- Lifecycle Events – A set of events that represent the activation and disposal of cache instances

Applications create and register event interceptors to consume events. Event interceptors handle the events and implement any custom logic as required. Events have different rules that govern whether or not mutable actions may be performed after receiving the event.

Map Trigger: Overview

Map Triggers:

- Execute *before* data changes occur
- Execute *synchronously* in response to CRUD operations
- Are typically used for server side operations
- Can be used to validate, modify, or even reject an operation
- Can be registered declaratively or programmatically

Map Trigger Events

6 - 7

Map triggers supplement the standard capabilities of Oracle Coherence to provide a synchronous pre-mutation event system. Map triggers can be used to prevent invalid transactions, enforce complex security authorizations or complex business rules, provide transparent event logging and auditing, and gather statistics on data modifications. Other possible use for triggers include restricting operations against a cache to those issued during application redeployment time.

For example, assume that you have code that is working with a `NamedCache`, and you want to change an entry's behavior or contents before the entry is inserted into the map. The addition of a map trigger will allow you to make this change, without having to modify all the existing code.

Map triggers could also be used as part of an upgrade process. The addition of a map trigger could prompt inserts to be diverted from one cache into another.

Map Triggers are most often used to:

- Validate, modify, or even reject an operation
- Enforce complex business rules or security

Map Listener: Overview

Map Listeners:

- Execute *after* data changes occur
- Execute *asynchronously* in response to CRUD operations
- Can be used with filters to selectively process data
- Are often used for application side processing
- Can be registered declaratively or programmatically

6 - 8

Map Listeners supplement the normal Coherence development model by providing a JavaBeans style asynchronous post mutation event mechanism. Map Listeners differ from map triggers in that they are executed after data changes have completed and been committed. Because map listener events are triggered post data mutation, they are most suitable for use for operations, such as logging, cascading of changes to other systems outside of Coherence. While map triggers can be used to invalidate or change data *before* it is committed, map listeners only have access to data *after* it has been changed. Additionally, Map Listeners can be more selective about the data they process through the use of filters.

Map listeners are most often used to:

- Log event changes
- Trigger data transfer to outside systems
- Alert applications of data changes
- Handle general data post processing

`MapListener` Behavior

- All update events on the grid are guaranteed to be received by registered `MapListener`s, even if a storage JVM fails.
- When a `MapListener` is instantiated, it begins to listen for events, *asynchronously*.
  - `MapListener`s do not receive events that occurred before they were instantiated.
- Ordering of events is not guaranteed.
  - For example: `put(1)`, `put(2)`, and `put(3)` may be received by a listener as 2, 1, and 3.

Storage JVM means local-storage = true

6 - 10

It is important to understand how listeners behave when they are implemented, and how they work in the grid environment.

All update events on the grid are guaranteed to be received by a listener, even if a storage JVM fails. The way that Coherence works with listeners is that when an object is to be inserted into the grid, that object is inserted into the storage JVM that owns the particular object. Therefore, it is this storage JVM that is responsible for sending the event to the client.

What the JVM actually does is it sends the event into a queue. The events are queued up and sent to the client. This queue is backed up along with the data. If a JVM or the server fails in the middle of an update, Coherence automatically processes the update. It sends the update to the new storage JVM, and to the surviving ones that are automatically determined by Coherence. These queues in turn are backed up. The listener is guaranteed to receive the event. Therefore, if an event such as an insert, update, or delete does occur in the grid, the client-side listeners are guaranteed to receive that event at least once.

When a listener is instantiated, it begins to listen for events. However, listeners will not receive events that occur before they begin to listen. They begin to listen when they are instantiated.

Post-events and `MapListener`s

Map Listeners:

- Are developed by implementing `MapListener`
- Support insert, update, or delete events, *asynchronously after* the operation occurs
- Can extend `AbstractMapListener` and selectively implement methods

```
package com.tangosol.util;

public interface MapListener extends java.util.EventListener{
    public void entryInserted (MapEvent event);
    public void entryUpdated (MapEvent event);
    public void entryDeleted (MapEvent event);
}
```

maplistener.java

6 - 12

Map listeners are developed by implementing/extending either the `MapListener` or `AbstractMapListener`. `AbstractMapListener` allows you to selectively decide which of the three events you are interested in tracking. A map listener is called after the operation occurs.

Each of the methods takes a single argument, a `MapEvent`, for which a partial list of methods is shown, which describes the specifics of the event that occurred. Using the map event, you can determine the type of event using the `getId`() method and the old and new values. Depending on the type of event, for example `ENTRY_INSERTED`, there may be only a partial set of data.

You can register `MapListener` for all cache events, or those satisfying a filter or a specific key. You register a `MapListener` using the `addMapListener()` method. You can also have a server-side event listener, which can be set up in the cache configuration XML file. These listeners run on the server, that is, they run on the JVM where the entry that is being updated is actually located.

Contract for writing a `MapListener`

`MapListener`s are implemented by:

1. Extending the `MapListener` interface
2. Implementing the three required methods:

```
import com.tangosol.util.MapListener;

public class SimpleMapListener implements MapListener {

    public void entryInserted(MapEvent event) {
       System.out.println("\n\n+++entryInserted++++\n");
    }

    public void entryUpdated(MapEvent event) {
       System.out.println("\n\n+++entryUpdated++++\n");
    }

    public void entryDeleted(MapEvent event) {
       System.out.println("\n\n+++entryDeleted++++\n");
    }
}
```

Required imports

Implements...

Implement each event method

SimpleMapListener.java

Exceptions thrown within MapListeners are wrapped and rethrown to the application thread that registered the listener.

6 - 13

The map listener interface is straightforward to implement. Simply include the `MapListener` import, and implement the required methods.

Selected `MapEvent` Methods

```
package com.tangosol.util;

public class MapEvent extends EventObject {

    public int getId();

    public Object getKey();

    public Object getOldValue();
    public Object getNewValue();

}
```

Defines:
ENTRY_INSERTED
ENTRY_UPDATED
ENTRY_DELETED

Returns the type of events inserted, updated, or deleted

Returns the key associated with the object for which the event occurred

Returns the prior and new values of the object, respectively

mapevent.java

6 - 14

Some useful `MapEvent` methods:

- `getNewValue()`: Returns a new value associated with this event. The new value represents a new value inserted into or updated in a map. It is always null for “delete” notifications. The `getNewValue()` method that you saw in the preceding code example returns the value in the object that was changed in the grid.
- `getOldValue()`: Returns an old value associated with this event. The old value represents a value deleted from, or updated in, a map. It is always null for “insert” notifications. If it is a nonlite event, the method gets or returns the old value. In a sales order example, there is an update in the quantity of the number of items being ordered. You retrieve the old value of quantity as 5 and insert the new value of quantity as 10. Therefore, in the update event, you can write code to take action based on that change.
- `getKey()`: Returns a key associated with this event. It returns the key of the object that was changed.

`MapListener` Event Contents

Based on the event type, the old and new values of a `MapListener` event may contain the following values:

Event Type	Old Value	New Value
`Insert`	`null`	Inserted object
`Update`	Previously inserted object	Newly inserted object
`Delete`	Previously inserted object	`null`

6 - 15

Map listener events provide both an old and a new value. The values differ depending on the event type.

For example, on an insert, there is no old value because a new value is being added to the cache. Likewise, on a delete, there is no new value because the value is being removed.

Synthetic Events

- Events normally reflect changes such, as insert, update, or delete on a cache.
- Events can originate within the cache itself. Such events are referred to as *synthetic* events.
- Common synthetic event reasons are:
  - When entries automatically expire from a cache
  - When entries are evicted from a cache because the maximum size of the cache has been reached
  - When entries are transparently added to a cache ← As the result of a Read-Through operation
  - When entries in a cache are transparently updated ← As the result of a Read-Ahead operation

6 - 16

Events usually reflect the changes being made to a cache. For example, one server is modifying one entry in a cache, while another server is adding several items to a cache, while a third server is removing an item from the same cache, all while fifty threads on each and every server in the cluster are accessing data from the same cache. All the modifying actions will produce events that any server within the cluster can choose to receive. These actions are referred to as *client actions*, and the events as being *dispatched to clients*, even though the “clients” in this case are actually servers. This is a natural concept in a true peer-to-peer architecture, such as a Coherence cluster: Each peer is both a client and a server, both consuming services from its peers and providing services to its peers. In a typical Java Enterprise application, a “peer” is an application server instance that is acting as a container for the application, and the “client” is that part of the application that is directly accessing and modifying the caches and listening to events from the caches.

Determining if an Event is Synthetic

- `MapEvent`s generated for synthetic reasons are derived from `CacheEvent`.
- `CacheEvent` includes the `isSynthetic` method which returns `true` if an event is synthetic.

```
import com.tangosol.util.MapEvent;
import com.tangosol.net.cache.CacheEvent;

. . .

private boolean isSynthetic(MapEvent event) {
    if ( event instanceof CacheEvent) {
        return ((CacheEvent) event).isSynthetic();
    }
    return false;
}
. . .
```

Likely a method in a MapListener

6 - 18

Synthetic events are generated by the underlying server and derived from `CacheEvent`. However, to be consistent with the `MapListener` and `MapTrigger` interfaces, the event is sent as a `MapEvent`. To determine if an event is synthetic, first check that the class is an instance of `CacheEvent`, then cast it to `CacheEvent` and check the `isSynthetic`() method as shown in the example.

Behind the scenes: ObservableMap

ObservableMap:

- Provides the ability to "observe" changes in cache entries
- Implements the JavaBean event model listener pattern
- Is extended by all NamedCaches including LocalCache, OverflowMap, NearCache, and others

```
package com.tangosol.util;
public interface ObservableMap extends Map {
    public abstract void addMapListener(MapListener listener);
. . .
    public abstract void removeMapListener(MapListener listener);
. . .
}
```

Register for all

Unregister

observablemap.jav
a

6 - 19

ObservableMap and Events

The slide lists some of the ObservableMap interface methods. The methods that are available to create a map listener on the client are:

- addMapListener(MapListener listener) – Adds a standard map listener that receives all events (inserts, updates, and deletes) that occur against the map, along with the key, old value, and new value. The method takes the listener to add as parameter.
- addMapListener(MapListener listener, Filter filter, boolean fLite) – Adds a map listener that receives events based on a filter evaluation
- addMapListener(MapListener listener, Object oKey, boolean fLite) – Adds a map listener for a specific key. This method includes the listener to add the key that identifies the entry for which to raise events, and fLite set to TRUE to indicate that the MapEvent objects need not include the OldValue and NewValue property values to allow optimizations. A Lite event is an event that does not contain the old value of the object. With entryUpdated, you can do a getOldValue() and getNewValue() to prepare the old value and new value of the object. If you do not need that functionality, you pass TRUE to the fLite attribute. If the event is Lite, it does not have to send the old value back to the client. There is less network traffic and also less CPU involved. If you do not need the old value, you can set the Lite attribute to TRUE.

Observable Map Workings

6 - 21

Registering `MapListener`s

Class implementing `MapListener` must be registered.

Registration can be:

- Programmatic, via `ObservableMap.addMapListener()`
- Declarative, via `coherence-cache-config.xml`

Programmatic

```
SimpleMapListener listener  = new SimpleMapListener();
NamedCache cache=CacheFactory.getCache("airports");
cache.addMapListener(listener);
```

application.java

Declarative

```
<cache-config>
. . .
<scheme-type-scheme>
   <scheme-name>. . . </scheme-name>
   <service-name>. . .</scheme-name>
   <listener>
      <class-scheme>
         <class-name>. . . SomeMapListener</class-name>
      </class-scheme>
   </listener>
</scheme-type-scheme>
```

coherence-cache-config.xml

6 - 22

Map Listeners are registered in one of two ways: programmatically, or declaratively via `coherence-cache-config.xml`.

The primary differences are:

- **Programmatically:** The map listeners are registered via one of the `addMapListener` methods exposed by the `ObservableCache` interface. Multiple listeners can be registered programmatically.
- **Declaratively:** A listener element is added to a scheme definition, refer to the `cache-config.dtd` for more information on the elements of schemes such as `distributed-scheme` and others. A single listener only can be registered declaratively.

Map Listeners and Exceptions

- `MapListener`s can be registered programmatically via `addMapListener()`, or declaratively, via a cache configuration.
- Exceptions on `MapListener`s registered programmatically:
  - Are client side exceptions and do not affect the cluster
  - Do not traverse the network and are only thrown to the registering thread
- Exceptions on `MapListener`s registered declaratively:
  - Are server side exceptions and fail silently
  - Are wrapped in `RuntimeException` and logged
  - Are not sent across the networks as there is no client
  - Must be carefully coded to not affect the cluster

6 - 23

Map Listeners are written in Java and can raise exceptions.

Most map listeners are client-side, and exceptions that occur there will be raised in the client application. Such exceptions do not traverse the network. These exceptions are usually due to developer code. Such exceptions are similar to an exception being raised in an event handler, for example, in Swing. Such exceptions should not affect the cluster.

Map listeners registered in the configuration must be very carefully considered. Such an exception can affect the cluster. Because there is no client, such an exception is first wrapped in a `RuntimeException`, and then logged. Extreme care should be taken when raising an exception in such cases.

`MapListener` Registration Variations

Programmatic map listener registration comes in several forms:

1. Simple registration for all events on a cache
2. Register for all events by key
3. Register for events using a Filter

Also supports events "lite," or event notification w/o content

```
package com.tangosol.util;

public interface ObservableMap extends Map {
    public abstract void addMapListener(MapListener listener);
```

Adds the specified listener that will be invoked for all changes, for all elements

```
    public abstract void addMapListener(MapListener maplistener,
                                        Object key, boolean liteFlag);
```

Adds the specified listener that will be invoked for only events for the given key, the event may be also "lite," that is no "content"

```
    public abstract void addMapListener(MapListener listener,
                                        Filter filter, boolean liteFlag);
```

Calls the event listener based on the contents of the provided filter, either lite or not.

```
}
```

6 - 24

Map listeners can be registered using a variety of mechanisms. The slide shows three variations:

- A simple listener that is registered for all changes
- A listener that takes an object as a key only is triggered if the key matches
- A listener that takes a filter to select a set of objects to trigger events on

Additionally, several interfaces provide the "liteFlag," which is covered in later slides.

MapListener Lite

Map Listeners can be registered as *Lite*, meaning that only the key is provided and not the entry.

- Normally, the old and new parameters of the event are `null`, but may be passed if no overhead is incurred.

```
SimpleMapListener listener  = new SimpleMapListener();
NamedCache cache=CacheFactory.getCache("airports");
cache.addMapListener(listener,someKey,true);
```

application.java

```
import com.tangosol.util.MapListener;
public class SimpleMapListener implements MapListener {
. . .
    public void entryInserted(MapEvent event) {
       System.out.println("\n\n+++entryInserted++++\n");
    }
. . .
}
```

May contain null, old, and new values!

SimpleMapListener.java

Lite MapListeners are useful when the underlying values are unimportant. Because they are not sent across the wire, network traffic is minimized.

6 - 25

If you have specified an event as Lite, it means that the old value may or may not be present in these events. If you do not need the old values, you can make it a Lite event. This means that the old value is not sent back to the client and you will have less network traffic and fewer overheads involved with the event listener.

The `MapEvent` object that is passed to `MapListener` carries the necessary information about the event that has occurred, including the source (`ObservableMap`) that raised the event, the identity (key) that the event is related to, the action against that identity (insert, update, or delete), the old value, and the new value.

To sign up for events, pass an object that implements the `MapListener` interface to one of the `addMapListener` methods on `ObservableMap`.

By default, Coherence provides both old and new values as part of an event. When an application does not require the old and new values to be included in the event, it can indicate that by requesting only “lite” events. When you add a listener, you can request lite events by using one of the two `addMapListener` methods that takes an additional `boolean fLite` parameter.

In some cases, a lite event’s old value and new value may be null. In others, even if you request lite events, the old and new values may be included if there is no additional cost to generate and deliver the event. Therefore, when you request that a `MapListener` receive lite events, it is just a hint to the system that `MapListener` does not need to know the old and new values for the event

`MapListener` Implementations

Several `MapListener` helper implementations exist:

- `AbstractMapListener`:
  - Has empty implementations for each `MapListener` signature
  - Simplifies your implementations by overriding default empty implementations
- `MultiplexingMapListener`:
  - Introduces abstract `onMapEvent` method
  - Sets all `MapListener` methods to `onMapEvent`
  - Simplifies your implementations by overriding `onMapEvent`

6 - 26

Anonymous `AbstractMapListener`: Example

Anonymous inner class can be used to register `MapListener` using `AbstractMapListener` and overriding only the methods of interest.

```
cache.addMapListener(
     new AbstractMapListener() {
        public void entryUpdated(MapEvent mapEvent) {
         // Implementation
        }
     });
```

Override only method of interest

Application.java

6 - 27

Examples

The slide shows an example of the `AbstractMapListener`, which provides dummy overrideable implementations of all required methods. An additional class `MultiplexingMapListener`, requires one method called `onMapEvent()` that can handle any event that occurs in the grid. Within the method, you can use the event ID to determine what type of event it is, whether it is an insert, update, or delete.

For example:

```
public static class EventPrinter extends MultiplexingMapListener {
    public void onMapEvent (MapEvent evt) {
        out(evt);
    }
}
```

Because only one method needs to be implemented to capture all the events, `MultiplexingMapListener` can also be very useful when creating an inner class to use as a `MapListener`.

Making Maps Observable

Maps such as `HashMap`, which are not derived from `ObservableMap`, can be made observable using `WrapperObservableMap`.

```
import com.tangosol.util.WrapperObservableMap;
. . .

 Map hashMap = new java.util.HashMap();
 WrapperObservableMap wom = new WrapperObservableMap(hashMap);

 wom.addMapListener(new SimpleMapListener());

 wom.put ("somekey","some object");
```

Application.java

6 - 28

Coherence provides support for making traditional Java maps observable. Any map that is derived from, or implements `java.util.Map`, can be wrapped using `WrapperObservable` map and then used with the Coherence listener infrastructure.
The example shown uses a traditional Java hash map, wraps it, and then instantiates a listener. On each put or remove operation, the listener is then called.

The wrapper works as a write-through map, providing the functionality required, but adding the listener support.

MapListeners and Filters

Filter instances:

- Can be used with MapListeners to limit event scope
- Must extend Filter and implement evaluate, returning true if the event is to be processed

```
import com.tangosol.util.Filter;
public class DeletedFilter implements Filter, Serializable {
    public boolean evaluate(Object o) {
        MapEvent evt = (MapEvent) o;
        return evt.getId() == MapEvent.ENTRY_DELETED;
    }
}
```

Only return true if a deleted event

DeletedFilter.java

```
SimpleMapListener listener = new SimpleMapListener();
NamedCache cache = CacheFactory.getCache("airports");
cache.addMapListener(listener, new DeletedFilter(), false);
```

Application.java

6 - 29

Filters extend the capability of a listener to select which events should be processed. Custom filters can be easily developed by implementing the Filter interface and providing an implementation of the evaluate method as shown in the slide. Evaluate is passed an event object, which can be used to determine how to process the filter. In the example shown, the ID field of the event is used to determine if a DELETE event is being processed.
The filter is then used by passing an instance to the addMapListener method with the appropriate listener.

Provided Filters

Coherence provides a set of filters in the `com.tangosol.util.filter` package:

Filter	Description
`MapEventFilter`	Evaluates the content of a `MapEvent` object according to the specified criteria, includes support for event types, such as insert, update, and delete
`AndFilter`	Provides logical `AND` between two filters
`OrFilter`	Provides logical `OR` between two filters
`ValueChangeEvent Filter`	Evaluates to true only for update events that change the value of an extracted attribute
`ContainsFilter`	Filter that tests a `Collection` or Object array value returned from a method invocation for containment of a given value

6 - 30

Coherence provides a number of filters in the `com.tangosol.util.filter` package, a subset of which is shown in the slide. These filters provide a mechanism for creating very complex filter functionality by providing `AND`, `OR`, `XOR`, `LIST`, and a variety of other filters, available for immediate use without any development effort.

Map Triggers: Review

Map Triggers:

- Execute *before* data changes occur
- Execute *synchronously* in response to a pending change/delete to the corresponding map entry
- Are guaranteed to execute *once* per change
- Can be used to validate, modify, or *reject* an operation
- Can be registered programmatically or declaratively
- Must implement `hashCode()` and `equals()` methods

Conceptually similar to database triggers

```
public class SimpleMapTrigger implements MapTrigger {
   public void process(Entry entry) {. . . }
   public boolean equals(Object o) { . . . }
   public int hashCode() { . . . }
}
```

SimpleMapTrigger.java

6 - 31

MapTrigger Basics

Map triggers are implemented by:

1. Extending the MapTrigger interface and implementing the process method
2. Implementing equals() and hashCode()
3. Registering the trigger

```
import com.tangosol.util.MapTrigger;

public class SimpleMapTrigger implements MapTrigger {

    @Override
    public void process(Entry entry) {
        System.out.println("Called with '"+entry+"'");
    }

    public boolean equals(Object o) {
        return o != null && o.getClass() == this.getClass();
    }

   public int hashCode() {
      return getClass().getName().hashCode();
   }
}
```

Required imports

Implements...

Requires process method

Minimal equals and hashCode

SimpleMapTrigger.java

6 - 32

Maptriggers are simpler to implement than listeners, requiring only a single process method be implemented for user functionality. However, Maptriggers do require hashCode and equals methods for management.

Maptriggers require that you implement hashCode and equals because they are essentially placed in an internal Hashset of Triggers, in which these methods are used to differentiate between instances, especially when adding/removing. This management differs from MapListeners which are managed via references.

Registering MapTriggers Programmatically

Registering a MapTrigger programmatically requires:

- An instance of the class implementing MapTrigger
- A wrapper MapTriggerListener
- Registration of the wrapper using addMapListener(…);

```
NamedCache cache=CacheFactory.getCache("airports");
SimpleMapTrigger trigger = new SimpleMapTrigger();
MapTriggerListener listener = new MapTriggerListener(trigger);
cache.addMapListener(listener);
```

Application.java

6 - 33

MapTriggers are registered programmatically in a three-step process:

1. Create an instance of a class that implements a MapTrigger Listener.
2. Create a com.tangosol.util.MapTriggerListener wrapper instance passing the MapTrigger class.
3. Pass the wrapper instance to the addMapListener method of the cache instance.

MapTrigger: **Example**

```
import com.tangosol.util.MapTrigger;

public class ToUpperMapTrigger implements MapTrigger {


  public void process(Entry entry) {
    System.out.println("\n\nProcess('" + entry.getValue() + "')");

    Object o = entry.getValue();
    AirPortJS ap = (AirPortJS)o;

    String upper = ap.getName().toUpperCase();
    ap.setName(upper);
    System.out.println("Replacing entry with:" + ap);
    entry.setValue(ap);
  }

  public boolean equals(Object o) {
    return o != null && o.getClass() == this.getClass();
  }

  public int hashCode() {
    return getClass().getName().hashCode();
  }
}
```

Implements...

Use entry arg to get current value object

Cast it as required

Change it somehow

Save the change

ToUpperMapTrigger.java

6 - 34

The power of a `MapTrigger` comes from a combination of when it is called in an object's life cycle and its ability to change an object in the background.

In the example shown, a map trigger is used to replace the value of a field with an upper case value. In this particular case, you use the entry arguments `getValue`() method to return the object, then cast it to the appropriate type. Note that production code would likely check the class type using `instanceof`. Then change one of the values in some way and store it back into the entry.

Registering a `MapTrigger` Declaratively

`MapTrigger`s:

- Are registered similarly to map listeners
- Require a `Factory` class to produce an instance
- Factories are registered in `coherence-cache-config.xml`

```
<cache-config>
  ...
    <distributed-scheme>
      ...
      <listener>
        <class-scheme>
          <class-factory-name>[package.]Factory</class-factory-name>
          <method-name>createTriggerListener</method-name>
        </class-scheme>
      </listener>
    <distributed-scheme>
  ...
</cache-config>
```

coherence-cache-config.xml

6 - 35

`MapTriggers` are registered almost identically to `MapListeners`, with the exception of the use of a class factory. In the example shown, a class factory returns, via its `createTriggerListener` method, a `MapTrigger` Listener.

Additionally, such factory methods can take parameters by following the method name element with an `init-params` element. The `init-params` element then contains one or more `init-params` each a `param-type/param-value` set. Currently, supported types are `string`, `int`, `long`, `double`, and `float`.

An example of a method that takes a string might be:

```
<method-name>createTriggerListener</method-name>
 <init-params>
     <init-param>
         <param-type>string</param-type>
         <param-value>value</param-value>
      </init-param>
  </init-params>
```

`MapTrigger`s and Exceptions

Map triggers can be registered programmatically or declaratively, but raise exceptions in the same way in both scenarios. Exceptions:

- Are logged locally on the storage-enabled member in which they occurred
- Are wrapped in a `RuntimeException`
- Are returned to the client which resulted in the trigger

6 - 36

All event processors can raise exceptions, and map triggers are no exception. There are several ways an exception is handled when raised in a map trigger:

- If no client is involved, the exception is logged.
- if a client was involved, the exception is passed to the client who “caused” it.

In all cases, the exceptions are wrapped in a `RuntimeException` and the resulting exception is returned.

Remember: Throwing an exception in a map trigger method causes the mutation of the object to be discarded.

Live Event Interceptors

6 - 37

Coherence supports both application-side and server-side events. In either case, an event is generated based on a variety of events, such as an insert/update/delete map operation. Event listeners are registered declaratively and execute solely on the application side. Event interceptors differ from listeners, and are purely a server-side mechanism. Event interceptors are conceptually similar to map triggers but they cover a much broader range of event types.

Live Event Model Events

The Live Event Model supports the following three classes of events:

- Partitioned Cache Events – Events associated with cache data changes (add, remove, update) or events associated with entry processors
- Partitioned Service Events – Events representing data migration between partitions
- Lifecycle Events – Events representing the life cycle of cache instances

6 - 38

Applications create and register event interceptors to consume events. Event interceptors handle events and implement any custom logic as required. Events have different rules that govern whether or not mutable actions may be performed after receiving the event.

The events that are currently supported include:

- Partitioned Cache Events – A set of events that represent the operations being performed against a set of entries in a cache. Partitioned cache events include both entry events and entry processor events. Entry events are related to inserting, removing, and updating entries in a cache. Entry processor events are related to the execution of entry processors and include executing and executed events.
- Partitioned Service Events – A set of events that represent the operations being performed by a partitioned service. Partitioned service events include both partition transfer events and partition transaction events. Partition transfer events are related to the movement of partitions among cluster members. Partition transaction events are related to changes that may span multiple caches and are performed within the context of a single request.
- Lifecycle Events – A set of events that represent the activation and disposal of a `ConfigurableCacheFactory` instance

Partitioned Cache Events

- They represent operations on entries or execution of event processors.
- Entry events include pre- and post-executing of inserts, updates, and deletes.
- Entry processor events represent pre- and post-execution.

6 - 39

Partitioned cache events fall into two specific classes, data mutation and entry processor events. Data mutation events occur pre/post-insert, update, or remove operations. Both event types can be found in the `com.tangosol.net.events.partition.cache` package. For each of these operations, the following hold true:

- All registered event interceptors are called *synchronously* pre-commit, with a lock held for each object to avoid concurrent updates. Note that pre-commit interceptors may change data.
- Mutation occurs.
- All post-commit operations are executed *asynchronously* after the operation has completed, in the order in which the operation was executed.
- Any exception prevents the operation from being committed.

`EntryEvent`s:

- `INSERTING` indicates that an entry or multiple entries are going to be inserted in the cache.
- `INSERTED` indicates that an entry or multiple entries have been inserted in the cache.
- `UPDATING` indicates that an entry or multiple entries are going to be updated in the cache.

Partitioned Cache Event
Entry Event Behavior

`INSERTING`, `REMOVING`, or `UPDATING` data events are handled *before* the mutation is performed and:

- Interceptors are called synchronously before the event
- Locks are held on entry data to prevent concurrent operations
- Exceptions prevent the change from being committed

`INSERTED`, `REMOVED`, or `UPDATED` data events are handled *after* the mutation is performed and:

- Interceptors are called asynchronously after the event
- Data cannot be modified in any way

6 - 41

Partitioned cache entry event behavior is very well defined.

For events that occur BEFORE a mutation:

- Interceptors are called synchronously in the order registered. (More on event ordering later in this lesson.)
- Hold data locks to prevent concurrent access
- Exceptions can be used to block the operation
- Slow resources should never be accessed or long running operations should never be performed

For events which occur AFTER a mutation:

- Interceptors are called asynchronously
- Data cannot be changed in any way
- Long running operations should be avoided to minimize event backlog
- The Java Executor service can be used to create thread pools to handle processing

Partitioned Cache Event

Entry Processor Event Behavior

`EXECUTING` entry processor events:

- Are handled synchronously before the entry processor is called
- Hold locks for each entry to prevent concurrent access
- Exceptions prevent the entry processor from being executed
- May change their associated entry set

`EXECUTED` entry processor events:

- Are handled asynchronously after the entry processor completes
- Are handled in the same order as the events that caused them
- Cannot change their associated entry set

6 - 42

Entry Processor events represent the executing of entry processors on their associated binary entries. There are two possible entry processor events, `EXECUTING` and `EXECUTED`.

`EXECUTING` events:

- Are handled before the associated entry processor is called. Care should be taken to ensure that all calls are quickly executed.
- Are called synchronously in the order of their associated data
- Hold locks on all associated data to avoid concurrency concerns
- That throw exceptions cause the underlying processor to not be executed
- May be shared and should be thread safe

`EXECUTED` events:

- Are handled after the completion of the entry processor
- Are called asynchronously
- Can read but not modify associated entries

Partitioned Service Events

- Represent operations being performed by the partitioned service
- Include information about the event, such as partition ID, cluster members involved, and map of cache names to entries

6 - 43

Partitioned service events represent those operations being performed by the partitioned service. Such events fall into two classes, events resulting in data being transported between members, (typically as a result of a member joining or leaving the cluster or other event related to rebalancing), and transactions on binary data. The two types, both of which are present in the `com.tangosol.net.events.partition` package include:

- `TransferEvent`s – Transfer events represent partition transfers between storage-enabled members.
- `TransactionEvent`s – Transaction events represent changes to binary entries, possibly from multiple caches, which are made in the context of a single service request.

During processing of the event, a lock is held for each entry to prevent concurrent updates. Throwing an exception prevents the operation from being committed.

In either case, a set of common data is provided:

- Partition ID – A unique identifier representing the partition being impacted
- Cluster members – A list of the cluster members impacted by the change
- Event type – Depending on the handler, either of four possible types
- Map of entries – A map of the binary entries involved with the event

Partitioned Service Event Behavior

DEPARTING and ARRIVED events

- Lock the associated partition
- Contain a map of all entries being moved
- Cannot modify associated values

COMMITTING and COMMITTED events

- Lock associated entries to prevent concurrent updates
- Exceptions prevent their associated change from completing (COMMITTING)
- Cannot modify associated values (COMMITTED)

6 - 44

Partition service events include events associated with moving data between partitions or transactions across multiple caches.

DEPARTING and ARRIVED:

- Represent the moving out of and into one partition to another respectively
- Include a map of all values being moved but do not allow modification of associated values
- Lock the associated partition during the operation

COMMITTING and COMMITTED events represent transactions across multiple entries and possibly across multiple caches. All made in the context of a single request, most likely a partition-level transaction from an entry processor.

COMMITTING events:

- Happen synchronously before the transaction
- Can change associated entries
- Lock all associated entries to prevent concurrent access
- Exceptions prevent the associated change from taking place

COMMITTED events:

- Happen asynchronously after the transaction
- Can only read associated entries

Lifecycle Events

- Represent actions that occur against a `ConfigurableCacheFactory` instance
- Are raised on cache creation and removal

6 - 45

`com.tangosol.net.events.application.LifecycleEvent` represent actions that occur on a `ConfigurableCacheFactory` instance and include cache creation and removal. Events include:

- `ACTIVATED` – Raised after all services that are associated with a cache factory are started. The services are defined in the cache configuration file and must be configured to `autostart`.
- `DISPOSING` – Raised before all services are shut down and any resources are reclaimed. Event interceptors that handle a `DISPOSING` event are notified before the services are shut down.

Note that a `ConfigurableCacheFactory` instance can be activated and disposed of only once.

Developing and Registering Event Handlers

Event Handlers are created by:

1. Developing the handler
   - Implement the `EventIntercepter` interface based on one of the known interceptor types (`EntryEvent`, `LifeCycleEvent`, `TransferEvent`, or `TransactionEvent`)

```
import com.tangosol.net.events.EventInterceptor; . . .
public class EntryEventHandler implements EventInterceptor<T> {…}
```

2. Registering the handler
   - Register the interceptor against a specific service or cache

```
<cache-config>...
  <interceptor>
    . . .
  </interceptor>
</cache-config>
```

6 - 46

Event Interceptors had defined and registered in two distinct steps. First a class is created which implements the generic `EntryInterceptor` class specifying an event type it will handle. Interceptor classes also require an `onEvent`() method to handle the raising of the event. Second, the interceptor must be registered in a class configuration for either a specific cache or a specific cache scheme.

Continuous Query Cache

A Continuous Query Cache, or CQC is:

- Based on an existing cache and a filter
- A local cache that is constantly updated as data changes
- Useful for:
  - Real-time consoles (update a GUI screen immediately if data is updated on the grid)
  - Complex event processing
  - Any application that needs zero latency access to a subset of data in a distributed cache

6 - 47

It would be easy to imagine a cache that is maintained as a result of listener events. Such a cache would be maintained in real time, and always consistent. Coherence provides such a cache in the form of a Continuous Query Cache. Such a cache provides a local, in process view of data that is updated as if a query was constantly run with no latency. Effectively, Coherence provides a feature that combines a query result with a continuous stream of related events in order to maintain up-to-date query results in real-time fashion.

Coherence implements the Continuous Query functionality by materializing the results of the query into a continuous query cache, and then keeping that cache up-to-date in real time using event listeners on the query.

The benefits of a continuous query cache:

- It is an ideal building block for Complex Event Processing (CEP) systems and event correlation engines.
- It is ideal for situations in which an application repeats a particular query, and would benefit from always having instant access to the up-to-date result of that query.

Continuous Query Cache Basics

A continuous query cache:

- Is based on an existing cache
- Uses a filter to keep the cache up to date

```
import com.tangosol.net.NamedCache;
import com.tangosol.net.cache.ContinuousQueryCache;
import com.tangosol.util.Filter;
import com.tangosol.util.filter.*;

NamedCache flights = CacheFactory.getCache("flights");

Filter filter = new AndFilter(
                    new EqualsFilter("state", "flying"),
                    new EqualsFilter("departedFrom", "BOS"));

ContinuousQueryCache inTheAir =
           new ContinuousQueryCache(flights, filter);
```

Required imports

Based on a specific cache

Maintained based on a filter

CQC created from the original cache and filter

6 - 49

Coherence implements continuous query using a combination of its data fabric parallel query capability and its real-time event filtering and streaming. The result is support for thousands of client application instances, such as trading desktops or air traffic controllers.

To create a CQC:

1. Create a normal cache, for example, a set of flights, as show in the slide.
2. Create a filter that represents the query to be executed.
3. Associate the query with the cache, to create a cache that is constantly up to date.

The result of the continuous query is maintained locally, and optionally, all the corresponding data can be cached locally as well. The Coherence implementation of continuous query is found in the `com.tangosol.net.cache.ContinuousQueryCache` class. This class implements the standard `NamedCache` interface.

Note that changes made to an entry in a CQC are reflected in the parent cache.

Summary

In this lesson, you should have learned how to:

- Describe, compare and contrast listener types and life cycle
- Implement and register map listeners
- Filter and transform map listener events
- Implement and register map triggers
- Implement backing map listeners
- Explain and use continuous query caches

6 - 50