# Performing In-place Processing of Data with Entry Processors

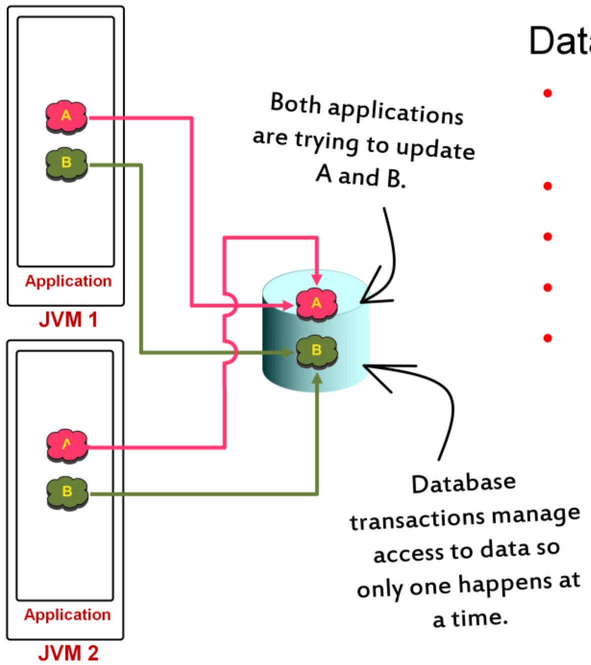
# Objectives

After completing this lesson, you should be able to:

- Manage concurrent access to data
- Implement Entry Processors to process data “in-place” where it is stored in the cache
- Configuring and implementing invocable agents

8 - 2

# Managing Data Consistency and Transactions: Databases




Database Data Consistency:

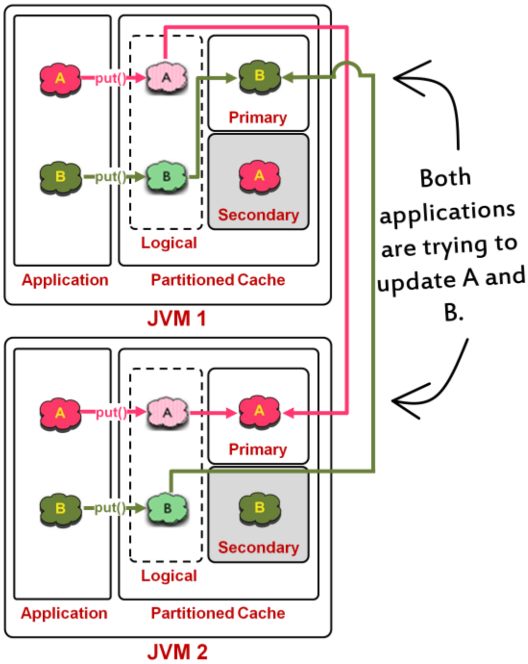
- Transactions: Unit of work that wholly succeeds or fails
- Maintains ACID properties
- Ensures data consistency
- Persisted to disk
- Can be coordinated with other data sources using XA with a resource manager

8 - 3

A database transaction is a unit of work that is performed against a database management system or a similar system that is treated in a coherent and reliable way independent of other transactions. A database transaction must be Atomic, Consistent, Isolated, and Durable (ACID). Transactions provide an “all-or-nothing” proposition, stating that work units performed in a database must be completed in their entirety, or take no effect whatsoever. Further, transactions must be isolated from other transactions, results must conform to existing constraints in the database, and transactions that complete successfully must be committed to durable storage.

Databases, and other data stores in which the integrity of data is important, often have the ability to handle transactions to ensure that the integrity of data is maintained. A single transaction is composed of one or more independent units of work, each reading and/or writing information to a database or other data store. In such a situation, it is important to ensure that the database or data store is in a consistent state.

## Managing Data Consistency and Transactions: Coherence




Coherence Data Consistency:

- `ConcurrentMap`
- `TransactionScheme` and Transaction Framework API
- JCA Adapter for joining managed XA transactions
- `EntryProcessor`

8 - 4

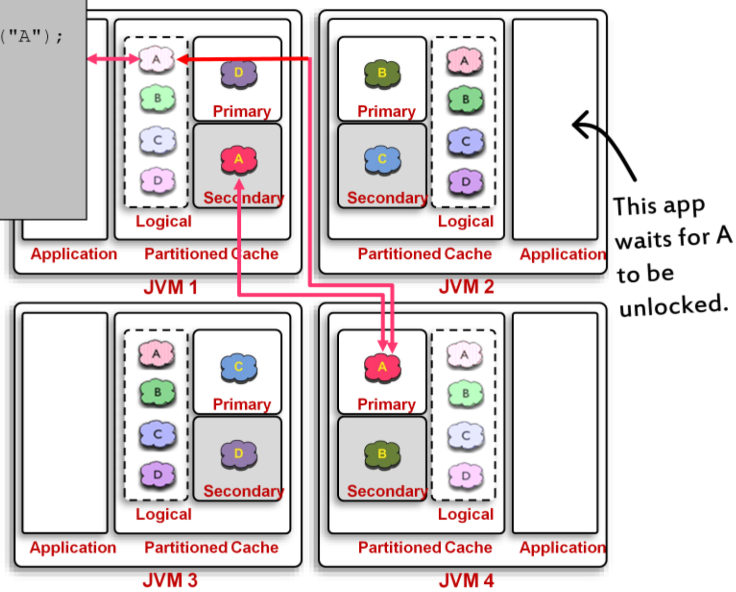
Just as it is important to ensure that a database is in a consistent state, it is also very important to ensure that data managed by Coherence is in a consistent state. Because there are multiple clients that can potentially have different copies of the cached data, it is important that the value of the data from the read until it is updated is accurate, and that some other process that has a copy of the data does not update it in the middle.

Coherence provides several options for managing concurrent access to data. This includes:

- **Explicit locking:** The `ConcurrentMap` interface (part of the `NamedCache` interface) supports explicit locking operations. Locks only block calls to `lock()`, much like Java `synchronized`. A lock can only be released when the locking client departs or by the same thread or another thread in the cluster node calling `unlock()`. This is determined by setting the `<lease-granularity>` element on the cache configuration to `thread` or member.

# ConcurrentMap

```
public void updateData() {
  myCache.lock("A", -1);
  try {
    Object a = myCache.get("A");
    //update a
    myCache.put("A", a);
  }
  finally {
    myCache.unlock("A");
  }
}
```




This code results in several network roundtrips:

1. Lock the entry on JVM4.
2. Get the entry from JVM4.
3. Put the updated entry on JVM4.
4. Update backup entry on JVM1.
5. Unlock the entry on JVM4.

8 - 6

Explicit locking can be very expensive. The simplest of use cases involves several network hops for a single update. And while the entry is locked (see the code example), applications that also use locking are blocked until object A is unlocked.

## ConcurrentMap

- `ConcurrentMap`:
  - Provides `lock()`/`unlock()` methods for keys (similar to Java synchronization)
  - Threads must coordinate reads/writes through locking
  - Locks are unaffected by server failure
  - Locks are released when client disconnects
  - `get()` and `put()` operations are allowed while key is locked

8 - 7

The `NamedCache` interface extends the `ConcurrentMap` interface which provides methods for locking and unlocking entries related to specific keys within a cache. Coherence lock functionality is similar to the Java `synchronized` keyword: locks only block locks. The difference is that `synchronized` is not distributed, whereas locks are distributed, as seen in the previous diagram. Threads must all use locking to coordinate access to data. If a key is locked, another thread can read the data without locking. If a server fails, the lock persists by failing over to the backup server. Conversely, if a client crashes and holds a lock to a key, the key will automatically be released immediately. Threads and clients have to coordinate locking/unlocking. `ConcurrentMap` locks are distributed globally, meaning they affect all cache clients, not just on one JVM. `ConcurrentMap` does not enforce that a lock is held prior to allowing a `get` operation. It would be up to the application to enforce that.

# `ConcurrentMap`: Example

```
NamedCache cache = CacheFactory.getCache("Airports");

Object key = "SFO";
cache.lock(key, -1);
try {
   Object value = cache.get(key);
   // application logic
  cache.put(key, value);
} finally {
  cache.unlock(key);
}
```

Lock the SFO key.

Get SFO data from cache.

Do something…

Update SFO data in cache.00

Always unlock in a finally block to ensure that uncaught exceptions do not leave data locked.

Application.java

8 - 8

Here is an example of the `ConcurrentMap` API used to lock and unlock data. The code accesses the `Airports` cache, and uses the `lock` method to lock the `SFO` key. The second argument passed is a `-1` that specifies that the application is willing to wait indefinitely for the lock. When the lock is obtained, the application gets the `SFO` data from the cache, does some processing, then puts the data into the cache to update the cache. The `unlock` method is then called as part of the `try/catch/finally` block to release the lock so other applications can potentially work with the `SFO` object.

## Issues with Locking

- Locking can cause scalability problems:
  - Applications must wait for locks to be released before accessing data.
  - Locking increases latency via the number of network round trips.
- `EntryProcessors` provide scalable, lock-free updates of data.

8 - 9

Both the `ConcurrentMap` and Transaction Framework APIs provide locking mechanisms to control concurrent access to data, which is quite similar to how transactions are done in relational databases. The transaction API also provides for commits and rollbacks. You can update one or more objects and commit the transaction as one atomic transaction. If something goes wrong, you can roll back the transaction and undo the updates that occurred.

When you lock an object in an application, the lock prevents other applications and other Coherence users from updating that object or accessing that object. As a result, locking can cause scalability problems. If you lock an object, other requests from other users to access the object are affected. Typically, the application waits for that object to be unlocked (perhaps indefinitely). This can cause performance and scalability issues. While that transaction occurs, other users, who are waiting, see the hourglass icon on their screens.

## Updating Data in Coherence

- So far, you performed only single inserts or updates by key for updating data in Coherence:

```
myCache.put(key, object);
```

- What if you want to update data without the key? What if the key is unknown?

  In SQL, this would be something like:

```
UPDATE orders SET priority = 1
 WHERE order_amount > 1000
```

- What if you want to update a lot of data simultaneously in Coherence?

8 - 11

So far you performed single inserts and updates by key. You used the `put()` method of `NamedCache`. However, there can be situations where you might be required to update the data without knowing the value of the key. For example, you might need to execute statements that are similar to SQL, such as, "`UPDATE orders SET priority=1 WHERE order_amount > 1000`."

Because Coherence does not support SQL, you must perform these operations in Coherence without using SQL. However, the CohQL query feature, covered in lesson titled “Querying and Aggregating Data in the Cache,” can be used to set a query filter that is SQL-like. In Coherence, you have the capability of performing mass updates such as the one discussed in the slide. Not only can you perform a mass update, updating a lot of data simultaneously, you can do it without locking and without causing scalability issues.

# What Is an EntryProcessor?

- An EntryProcessor is an agent that performs processing against entries where they are managed:
  - Requests are sent directly to owners to do the work.
  - Requests are queued, so locks are not necessary.
- This is equivalent to “agents” executing services in parallel on the data in the cluster.
- Entry processing:
  - May mutate cache entries, including creating, updating, or removing them
  - May perform simple calculations or any other type of processing

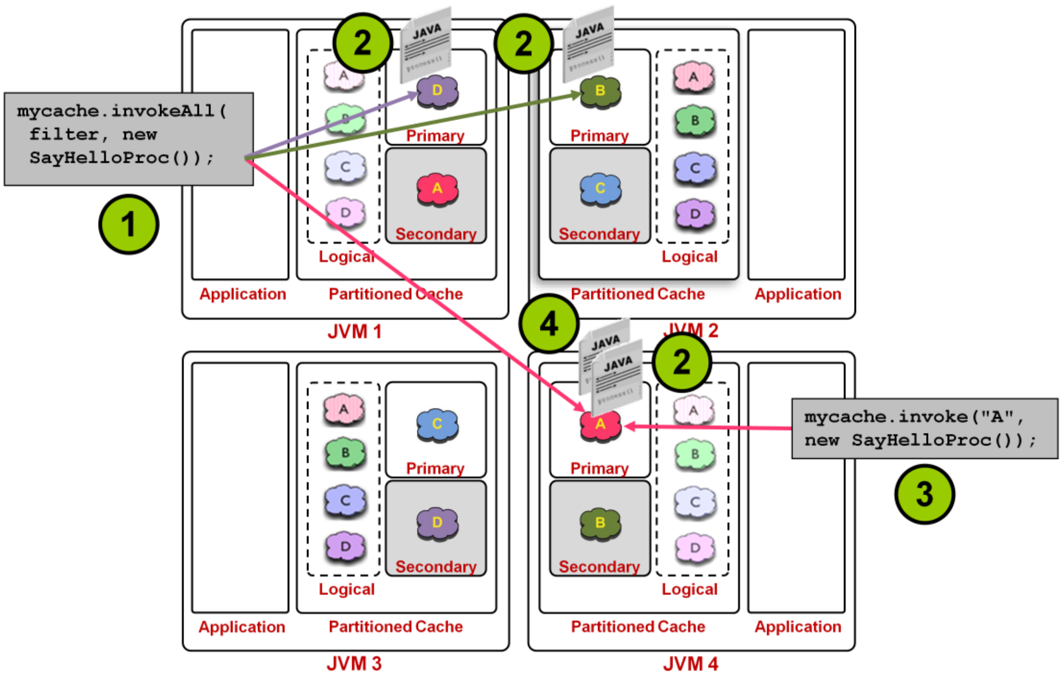
8 - 12

Coherence supports a lock-free programming model through the EntryProcessor API. This minimizes contention and latency, and improves system throughput, without compromising the fault tolerance of data operations.

Every NamedCache implements the InvocableMap interface. The EntryProcessor interface (contained within the InvocableMap interface) is the agent that performs processing against the entries directly where they are managed. So, on the client, method of an EntryProcessor can be invoked remotely This method is sent directly to the storage JVM where the data is located, and directly to the owners who actually do the work.

The requests are queued, so locks are not necessary. If there are multiple requests to update the same object, Coherence automatically queues them and performs the updates one after the other. This results in the ability to perform updates on the system without locks. EntryProcessors are equivalent to agents executing services in parallel on the data in the cluster. For example, if a request to update many objects in the cluster is executed, Coherence automatically performs the update in parallel on all the storage JVMs that own that data. This is done automatically, and the client does not have to launch the agents or manage any agents.

# Entry Processor Execution Diagram




8 - 14

When an `EntryProcessor` is invoked, the processor class that is passed as the argument to the call is serialized and sent to the JVM where each entry that matches the keyset exists. The processor is deserialized at each location across the cluster, the entry in question is passed as an argument to the processor, and the entries are processed in parallel.

This diagram shows two kinds of Entry Processor method invocations, one that uses a filter, and one that uses a single key directly. Here is what happens when these methods are invoked:

1. A filter has been constructed either using the Coherence filtering mechanism or the CohQL feature, and for the sake of argument, only objects A, B, and D meet the filter specifications when the query is executed. The application calls `invokeAll(filter, new SayHelloProc())`, where the filter is the filter described previously, and `SayHelloProc` is a serializable Java class that implements a method called process.

## Out-of-the-Box `EntryProcessors`

- There are a number of provided `EntryProcessors`:

  `AbstractProcessor`, `CompositeProcessor`, `ConditionalProcessor`, `ConditionalPut`, `ConditionalPutAll`, `ConditionalRemove`, `ExtractorProcessor`, `NumberIncrementor`, `NumberMultiplier`, `PreloadRequest`, `PropertyProcessor`, `UpdaterProcessor`, `VersionedPut`, and `VersionedPutAll`

See docs for more details!

```
//put BZE into the cache if it's not already there
String key = "BZE";
Airport airport = new Airport(key, "Goldson International");
airportCache.invoke(key, new ConditionalPut(
  new NotFilter(PresentFilter.INSTANCE), airport));
```

Updates only if not already present in cache.

Application.java

8 - 16

There are a number of default `EntryProcessors` that are provided by Coherence. Typically, you can also write your own, but there are some `EntryProcessors`, such as `ConditionalProcessor` or `ConditionalPut`, that enable you to update the data only if it meets a certain value or if the value exists.

- `NumberIncrementor`: An `EntryProcessor` that is similar to a sequence in Oracle Database where you can automatically increment a number
- `CompositeProcessor`: `CompositeProcessor` represents a collection of `EntryProcessor` objects that are invoked sequentially against the same entry.

An agent implements the `EntryProcessor` interface, typically by extending the `AbstractProcessor` class. A number of agents are included with Coherence, including:

- `AbstractProcessor`: An abstract base class for building an `EntryProcessor`
- `ExtractorProcessor`: Extracts and returns a specific value (such as a property value) from an object that is stored in an `InvocableMap`
- `ConditionalProcessor`: Conditionally invokes an `EntryProcessor` if a filter against the entry-to-process evaluates to `TRUE`

# `EntryProcessor` **Requirements**

To implement and execute an entry process:

- Create a class which implements the `EntryProcessor` or extends `AbstractProcessor`

```
. . .
class ExampleEntryProxcessor extends AbstractProcessor {
     . . .
     Object process(Entry entry) { . . . }
}
```

- Call one of the `InvocableMap` methods to invoke the `EntryProcessor` on certain entries

```
someNamedCache.invokeAll(AlwaysFilter.INSTANCE,
                      new ExampleEntryProxcessor ());
```



The steps to develop an `EntryProcessor` include:

- Implementing the `EntryProcessor` interface with code that interacts with cache entries.
- The entries can be cast into their original class type, and the class' methods can be invoked directly. The `Entry` is passed in to the `EntryProcessor` method as an `InvocableMap.Entry` type. This interface provides methods to interact with the entry.
- Determine which `InvocableMap` method to use, and pass the `EntryProcessor` object to use as the second parameter. When the method is called, the `EntryProcessor` is passed to the JVMs of all matching entries for execution against each entry.

The next few slides cover each of these interfaces in more detail.

# InvocableMap Interface

```
package com.tangosol.util;

public interface InvocableMap {

  public Object invoke(Object oKey,
                       InvocableMap.EntryProcessor processor);
```

Invokes the passed EntryProcessor against the entry specified by the passed key, returning the result of the invocation

```
  public Map invokeAll(Collection keys,
                       InvocableMap.EntryProcessor processor);
```

Invokes the passed EntryProcessor against the entries specified by the passed keys, returning the result of the invocation for each entry

```
  public Map invokeAll(Filter filter,
                       InvocableMap.EntryProcessor processor);
```

Invokes the passed EntryProcessor against the set of entries that are selected by the given filter, returning the result of the invocation for each entry

```
    . . .
}
```

InvocableMap.java

An informative subset of InvocableMap methods. See docs for complete list!

8 - 19

An `InvocableMap` is a map against which both entry-targeted processing and aggregating operations can be invoked. Though a traditional model for working with a map is to have an operation access and mutate the map directly through its API, `InvocableMap` allows that model of operation to be inverted such that the operations against the map contents are executed by (and thus within the localized context of) a map. This is particularly useful in a distributed environment, because it enables the processing to be moved to the location at which the entries-to-be-processed are managed, thus providing efficiency by localization of processing. The methods of the `InvocableMap` interface include:

- `invoke()`: Invokes the passed `EntryProcessor` against the entry specified by the passed key, returning the result of the invocation. The `InvocableMap` interface on the client can call invoke. A particular key is passed to update a single object, and the `EntryProcessor` object to execute is passed. The `EntryProcessor` is passed to the JVM where the specified entry exists, and passes that `Entry` to the `EntryProcessor`.
- `invokeAll(`*`keys`*`)`: Invokes the passed `EntryProcessor` against the entries specified by the passed keys, returning the result of the invocation for each entry. When `invokeAll` is executed, a collection of keys is passed if the keys for the entries to update are known.

# `InvocableMap.EntryProcessor` **Interface**

```
package com.tangosol.util;

public static interface InvocableMap.EntryProcessor
  extends Serializable {

  public Object process(InvocableMap.Entry entry);
```

Processes a `Map.Entry` object

```
  public Map processAll(Set setEntries);
```

Processes a set of `InvocableMap.Entry` objects (implementation typically provided by a super-class)

```
}
```

If processAll() is not implemented, Coherence executes process() repeatedly.

`InvocableMap.EntryProcessor.java`

8 - 21

Aside from the out-of-the-box `EntryProcessors`, Coherence allows implementing custom `EntryProcessors` to execute on the server. The `EntryProcessor` interface methods include:

- `process()`: The `process` method processes a single `InvocableMap.Entry` object at a time. As seen in the earlier example, an entry is what is passed into the `process` method. This is done automatically by Coherence. The server passes each entry that matches the associated filter or keys.
- `processAll()`: The `processAll` method processes a set of `InvocableMap.Entry` objects. It is optional to implement the `processAll` method (if extending the `AbstractProcessor`), which provides bulk processing for a large number of entries. The `processAll` method is not required for handling multiple entries, as the `EntryProcessor's` `process` method is called repeatedly when the `processAll` method is not implemented.

# `InvocableMap.Entry` Interface

InvocableMap.Entry.java

```
package com.tangosol.util;

public static interface InvocableMap.Entry extends QueryMap.Entry {

  public Object getKey();
```
Returns the key corresponding to this entry
```
  public Object getValue();
```
Returns the value corresponding to this entry
```
  public boolean isPresent();
```
Determines whether the entry exists in the map
```
  public void remove(boolean isSynthetic);
```
Removes this entry from the map if present
```
  public Object setValue(Object value);
```
Stores the value corresponding to this entry
```
  public void setValue(Object value, boolean isSynthetic);
```
Stores the value corresponding to this entry
```
}
```

8 - 22

An `InvocableMap.Entry` contains additional information and exposes additional operations that the basic `Map.Entry` does not. It allows nonexistent entries to be represented, thus allowing their optional creation. It allows the existing entries to be removed from the map. It supports several optimizations that can ultimately be mapped through to indexes and other data structures of the underlying map. Some of the methods that are available in `EntryProcessor` are:

- `getKey()`: Returns the key corresponding to this entry. This gets the key of the entry that is passed into the `EntryProcessor`.
- `isPresent()`: Determines whether the entry exists in the map.
- `remove()`: Removes an entry from the cache. A Boolean value specifying if this entry is synthetic is passed as the parameter to this method. This parameter tells the system whether it is a synthetic entry or not.

In Coherence, there are *synthetic* and *nonsynthetic* updates. A synthetic update is something that Coherence does. For example, when the server JVM is taken off the cluster, or it crashes, Coherence automatically redistributes data as discussed earlier. This is an event or an update to the cache. Such an event is referred to as synthetic because it is Coherence that initiates it.

## `EntryProcessor` Semantics

- `EntryProcessor`s invoked against the same key are locally queued. This means responsibility-free lock (high throughput) processing.
- `EntryProcessor`s can return any “serializable” value. This includes null if a result is not required.
- An `EntryProcessor` can be invoked against entries that do not yet exist. The `Entry.isPresent()` method determines if an entry exists.

8 - 24

If multiple clients are executing an `EntryProcessor` against the same entry, each local storage JVM will automatically queue those `EntryProcessors`. This is done without locking or invoking any code on the client. Each entry executes one after the other, in the order they are queued. `EntryProcessors` can return values, and that value can be any serializable object. If a return value is not required, the `EntryProcessor` returns `null`. An `EntryProcessor` can perform aggregations or actions that return results to the client. An `EntryProcessor` can be invoked against entries that do not currently exist. Within the `EntryProcessor`, there is the `isPresent()` method to determine whether the entry exists or not.

## `EntryProcessor` Behavior

- `EntryProcessors` are synchronous. The client waits until the `invoke()` or `invokeAll()` executes.
- `invoke()` is an atomic operation. It either succeeds entirely or fails entirely.
- `invokeAll()` is *not* an atomic operation.
- `EntryProcessor`s must be idempotent. If the client fails:
  - An `invoke()` that returns successfully is executed
  - An `invokeAll()` may be partially executed. Some operations may succeed and others may not be executed.
- If the server (storage JVM) fails, the `EntryProcessor` is guaranteed to be executed on the surviving storage JVMs.

8 - 25

It is important to understand the behavior of `EntryProcessors`, and that it is quite different from transactions in a database.

`EntryProcessors` are synchronous. The client waits until `invoke()` or `invokeAll()` executes. For example, when `invokeAll()` is performed, the client blocks until all the storage JVMs have executed `invokeAll`() before continuing to the next line of code. `Invoke()` is an atomic operation. It either succeeds entirely or fails entirely. If it fails, Coherence performs a rollback. However, `invokeAll()` is *not* an atomic operation. If the client executes an `invokeAll()`, and it fails, any `EntryProcessor` that returns successfully is executed. An `invokeAll()` could be partially executed, that is, some operations may succeed whereas others may not be executed.

If the server JVM fails in the middle of an `EntryProcessor`, the `EntryProcessor` is guaranteed to be executed among the surviving storage JVMs. The queues are backed up as data. If the primary JVM goes down, Coherence guarantees that the `EntryProcessor` is executed. So, when the JVM goes down, there is a backup of that queue on another JVM.

## Contract for Writing EntryProcessors

- Exceptions thrown within EntryProcessors are wrapped and rethrown to the application-calling thread.
- Failure to call entry.setValue(...) or entry.remove() on a value means that no cache entry mutation will occur.
- If a filter is used, the EntryProcessor should not affect the filter evaluation.
  Example:

```
invokeAll(New GreaterEqualsFilter("getAmount", 100), UpdateOrderAmount());
```

- EntryProcessors essentially have READ COMMITTED isolation. They read the most recently updated copy of the entry.

8 - 27

Exceptions thrown within EntryProcessors are wrapped and rethrown to the calling thread.

EntryProcessors do not automatically update entries. Updates only occur when setValues() or remove() is called. If a filter is used, the EntryProcessor should not affect the filter evaluation. An example of an EntryProcessor's filter affecting filter evaluation is when invoke() is called with a greater than or equals filter of getamount=100, and the EntryProcessor updates the order amount of each entry. This creates a circular dependency. In this example, the EntryProcessor is executed on entries where the amount is greater than 100, but also updates the order amount in the same EntryProcessor. This can have unpredictable results.

EntryProcessors, in database terminology, essentially have read-committed isolation. That is, EntryProcessors execute the most recently updated entries. If there are transactions occurring in Coherence, any uncommitted data is not going to be available to EntryProcessors. There are no dirty read isolations. That is, with EntryProcessors, you can be assured that they will execute only on entries that are actually committed into the Coherence grid.

## Example: Creating a Custom EntryProcessor

- Example ChangeAirportCountryProcessor is as follows:

```
class ChangeAirportCountryProcessor extends AbstractProcessor {
   ...
   Object process(Entry entry) {
    Airport airport = (Airport)entry.getValue();
    airport.setCountry(this.country);
    entry.setValue(airport);
    return null;
   }
}
```

ChangeAirportCountryProcessor.java

- Now run this EntryProcessor on all entries:

```
// now run the EntryProcessor within the client application
airportCache.invokeAll(AlwaysFilter.INSTANCE,
                        new ChangeAirportCountryProcessor("USA"));
```

Application.java

8 - 28

This example demonstrates the following:

- An implementation of EntryProcessor called ChangeAirportCountryProcessor that does a bulk update on the system to change the country of all airports to USA.
- The EntryProcessor class extends AbstractProcessor, which is in the com.tangosol.util.processors package.
- The EntryProcessor class implements the process() method, which takes an Entry as the parameter. Within the process() method, the code gets the passed in Entry object, invokes its getValue() method to obtain the wrapped Airport object, and casts it into an actual Airport object.
- The code then invokes the Airport object's setCountry() method, using the value that was set in the constructor (not shown) when new was called on the ChangeAirportCountryProcessor object.
- After the value is changed in the Airport object, the new Airport object is updated within the Entry by calling the entry's setValue() method with the updated Airport object.

## Concurrent Processing and Entry Processors

- Entry processor methods are executed concurrently.
- The server isolates the operation from all other operations on the same key.

```
public Object process(Entry entry) {

    // get the value of the entry
    Integer iValue = (Integer)entry.getValue();

    // Update the value
    iValue++;

    // Update the value, holding the lock on this entry
    entry.setValue(iValue);

    return iValue;
}
```

Entry processors implicitly hold a lock on the key for which they are executing.

As a result, any code executing within the entry processors process method is atomic.

8 - 30

Because an entry processor's process method is held against a specific key, the method can be considered to hold a lock against that key. As a result, the developer does not need to be concerned about concurrency issues. All processing within the process method can be considered atomic.

## What Is an Invocation Service?

An Invocation Service:

- Allows you to execute an agent on cluster members
- Does not execute against data entries in the cache
- Is suitable for cluster-wide management tasks
- Can be invoked synchronously or asynchronously
- Provides an “at most once” QOS:
  - Idempotent tasks
  - Non-critical tasks

8 - 31

An invocation service is similar to an `EntryProcessor`, except that instead of executing against data entries in the cache, it executes against one or more members of the cluster. The concept is the same in that the invocation agent, which is conceptually like the `EntryProcessor` agent, is serialized and sent over to the Coherence servers for execution. Because they do not execute against data entries in the cache, they are suited more for management tasks in the cluster. Unlike `EntryProcessors` that can only be invoked synchronously, invocable agents can be executed either synchronously or asynchronously. Also unlike `EntryProcessors` that ensure that each entry is processed once and only once, the invocation service only guarantees that it will attempt to run the code once. Any failures are the responsibility of the developer to capture events and retry execution of the agent. There is no way to tell for sure if the code ran successfully or not, so invocation services are suitable only for non-critical or idempotent tasks.

## Implementing the `Invocable` Interface

`Invocable` select methods:

```
package com.tangosol.net;

public interface Invocable extends Runnable, Serializable {

  public void init(InvocationService service);
  public void run();
  public Object getResult();
}
```

Allows for initializing the agent

The method to execute on the cluster members

Returns the result of the agent execution after the run method completes

Invocable.java

8 - 32

The `Invocable` interface extends the `Runnable` interface by adding the `init()` and `getResult()` methods. Note that the class must be serializable in order to be passed to the cluster members for execution.

The interface methods work as follows:

- The `init`() method provides a hook for initializing the agent.
- The `run`() method is the method that contains the code to execute on the cluster members.
- The `getResult`() method is the method that is called to get the results after the execution of the `run` method.

## Implementing the `AbstractInvocable` Interface

`AbstractInvocable` select methods:

```
package com.tangosol.net;

public abstract class AbstractInvocable implements
                                              Invocable, Serializable {

  public void init(InvocationService service) { m_service = service; }
```
Already implemented.
```
  public void run(); // Does not implement run!
```
Intentionally left unimplemented. Use setResult() to set result for calls to getResult().
```
  public Object getResult() { return m_oResult; }
```
Already implemented.
```
  protected void setResult(Object oResult) { m_oResult = oResult; }
```
Already implemented.
```
}
```
AbstractInvocable.java

8 - 33

The `AbstractInvocable` interface is really an abstract class that extends the `Invocable` interface and provides a default implementation for the `init()`, `getResult()`, and `protected` `setResult()` methods. Writing a class that extends `AbstractInvocable` allows the developer the ease of only implementing the `run()` method. When using this approach, the `run` method should call the `setResult()` method prior to returning.

## `AbstractInvocable` Implementation: Example

Example class that extends `AbstractInvocable`:

```
public class CheckNumberOfCPUsAgent extends AbstractInvocable {
  public void run() {
    setResult(Runtime.getRuntime().availableProcessors());
  }
}
```

CheckNumberOfCPUsAgent.java

Returns the number of CPUs available on the machine where the code executes

8 - 34

Here is an example of using the `AbstractInvocable` class to create an invocable agent. This is a class called `CheckNumberOfCPUsAgent` that simply implements the `run` method. The execution of this agent on a target cluster member obtains the number of available processors on the machine where the member exists, and sets the value using `setResult()`. A subsequent call to the agent's `getResult()` method would return the value to the calling client.

# Examining the `InvocationService` Interface

`InvocationService` select methods:

```
package com.tangosol.net;

public interface InvocationService extends Service {

  public abstract Map query(Invocable invocable, Set set);

  public abstract void execute(Invocable invocable, Set set,
     InvocationObserver invocationobserver);
}
```

Synchronous

Synchronous execution method. Return value is a Map of results keyed by the member.

Asynchronous

Asynchronous execution method. Results are returned as notifications to the InvocationObserver object.

InvocationService.java

8 - 35

After an `Invocable` agent is implemented, and an `InvocationService` is configured, the next step is to write the code that causes the agent to execute against the cluster. This is done by invoking one of the methods of the `InvocationService` interface:

- The `query()` method is for invoking the agent synchronously. The `Invocable` agent class is passed as the first parameter, and the set of target cluster members is passed as the second parameter. If `null` is passed for the member set, the agent is executed on all cluster members. This method returns a Map of results that are keyed by members.
- The `execute()` method is for invoking the agent asynchronously. It takes the same parameters as the `query()` method, with the addition of an implementation of the `InvocationObserver` interface as a third parameter, which provides the callbacks for results and other notifications related to the execution of the agent. The `InvocationObserver` interface is shown on a later slide.

# Executing an Invocation Agent Synchronously

Synchronous Execution Example:

Get invocation service.

Invoke synchronously.

```
InvocationService service = (InvocationService)
    CacheFactory.getService("InvocationService");

Map<Member, Integer> numCPUMap =
    service.query(new CheckNumberOfCPUsAgent(), null);

for (Map.Entry<Member, Integer> numCPU : numCPUMap.entrySet()) {
  System.out.println("Member: " + numCPU.getKey() +
                     " Number of CPUs: " + numCPU.getValue());
}
```

Application.java

Iterate through returned results.

8 - 36

This is an example of application code that executes the `CheckNumberOfCPUsAgent` invocable agent synchronously using the `InvocationService` `query()` method. The code performs the following steps:

1. Obtains a reference to the `InvocationService` named "`InvocationService`" using the `CacheFactory.getService()` method. Note that this is the name specified in the example configuration shown previously.
2. Uses the returned service reference to call the `query()` method, passing in a new `CheckNumberOfCPUsAgent` class as the first parameter, and `null` as the second parameter. This causes the `CheckNumberOfCPUsAgent` class to get executed on all members of the cluster synchronously. Results of the execution are returned and captured in the `numCPUMap` variable.
3. Iterates through the `numCPUMap` result set, and prints out the results

# Executing an Invocation Agent Asynchronously

Invocation Observer Class Example:

```
private static class CPUInvocationObserver implements
    InvocationObserver {

  public void memberCompleted(Member member, Object result) {
    System.out.println("Member: " + member + " Num CPUs: " + result);
  }
```

Called when each member completes asynchronous agent execution

```
  public void memberFailed(Member member, Throwable throwable) {}
```

Called when a member fails during asynchronous agent execution

```
  public void memberLeft(Member member) {}
```

Called when a member leaves the cluster during asynchronous agent execution

```
  public void invocationCompleted() {}
}
```

Called when all members have completed asynchronous agent execution

CPUInvocationObserver.java

8 - 37

In order to invoke an agent asynchronously, a class that implements the `InvocationObserver` interface must be written. This class provides the callback hooks when notifications related to the execution of the agent occur, including the return of results from each cluster member as well as when all cluster members have completed execution. The next slide shows how this class is used to invoke the agent asynchronously. Keep in mind that this class must be implemented as a thread safe class. Additionally, a member listener can be used to keep the application informed of cluster member state to make the use of the Invocation Service as efficient as possible.

# Executing an Invocation Agent Asynchronously

Asynchronous Execution:

```
InvocationService service = (InvocationService)
    CacheFactory.getService("InvocationService");

service.execute(new CheckNumberOfCPUsAgent(), null,
    new CPUInvocationObserver());
}
```

Application.java

Get invocation service.

execute() is called with the agent and invocation observer.



This is an example of application code that executes the `CheckNumberOfCPUsAgent` invocable agent asynchronously using the `InvocationService` `execute()` method. The code performs the following steps:

1. The code obtains a reference to the `InvocationService` named "`InvocationService`" using the `CacheFactory.getService()` method.
2. Uses the returned service reference to call the `execute()` method, passing in a new `CheckNumberOfCPUsAgent` class as the first parameter, `null` as the second parameter, and a new `CPUInvocationObserver` object shown on the previous slide for receiving notifications. This causes the `CheckNumberOfCPUsAgent` class to get executed on all members of the cluster asynchronously. Results of the execution are returned by each cluster member by calling the `CPUInvocationObserver's` `memberCompleted()` method.
3. Each member calls the `memberCompleted()` method that prints out the results.

## Registering an Invocation Service

Invocation Services are registered:

- Declaratively within a cache configuration
- Within an `<invocation-scheme>` element, nested in a `<caching-schemes>` element

```
<caching-schemes>
  . . .
  <invocation-scheme>
    <scheme-name>MyInvocationServiceScheme</scheme-name>
    <service-name>InvocationService</service-name>
    <thread-count>5</thread-count>
    <autostart>true</autostart>
  </invocation-scheme>
  . . .
</caching-schemes>
```

coherence-cache-config.xml

8 - 39

To register an invocation service in Coherence, edit the `coherence-cache-config.xml` file and add the `<invocation-scheme>` element and related child elements as shown in the example. Specifying a thread count provides a thread pool for running multiple invocation agents in parallel; otherwise, the main invocation service thread performs the execution.

## Obtaining Member Sets

The Invocation Service can execute on all members or select members by providing a list of members.

To obtain a set of members:

```
import com.tangosol.net.InvocationService;
import com.tangosol.net.Member;
import java.util.Set;

InvocationService service = (InvocationService)
    CacheFactory.getService("InvocationService");    1

Member thisMember = service.getCluster().getLocalMember();    2

Set allMembers = service.getCluister.getServiceMembers();    3
```

8 - 40

Invocation services can be executed on all members or a specific set of members. To narrow the set of invocations to only specific members:

1. Obtain an instance of the Invocation service using code similar to:
   `InvocationService service = (InvocationService) CacheFactory.getService("InvocationService");`
2. Optionally obtain the individual member:
   `Member thisMember = service.getCluster().getLocalMember();`
3. Obtain the list of all members
   `Set allMembers = service.getCluister.getServiceMembers();`
4. (not shown) remove the current member.
   `allMembers.remove(thisMember);`

## Summary

In this lesson, you should have learned how to:

- Manage concurrent access to data
- Update data in Coherence without locking
- Invoke methods of the `InvocableMap` interface
- Implement a custom `EntryProcessor`
- Implement a custom `Invocable` Agent

8 - 41