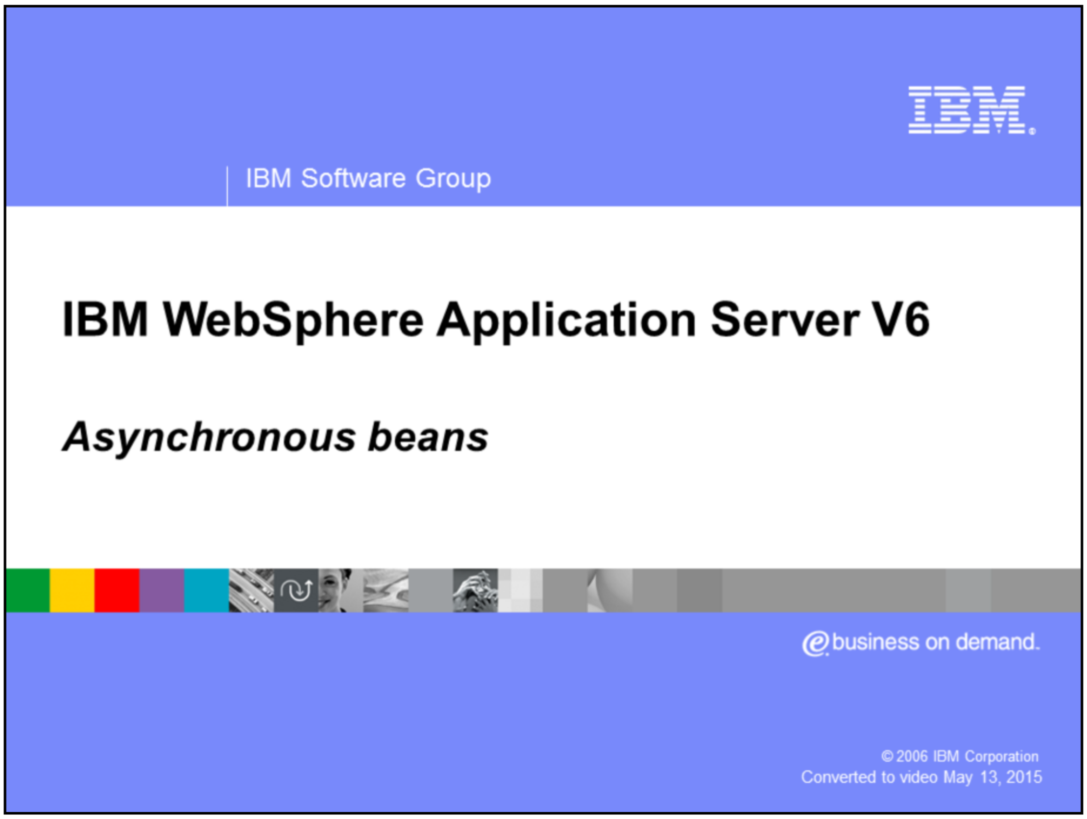


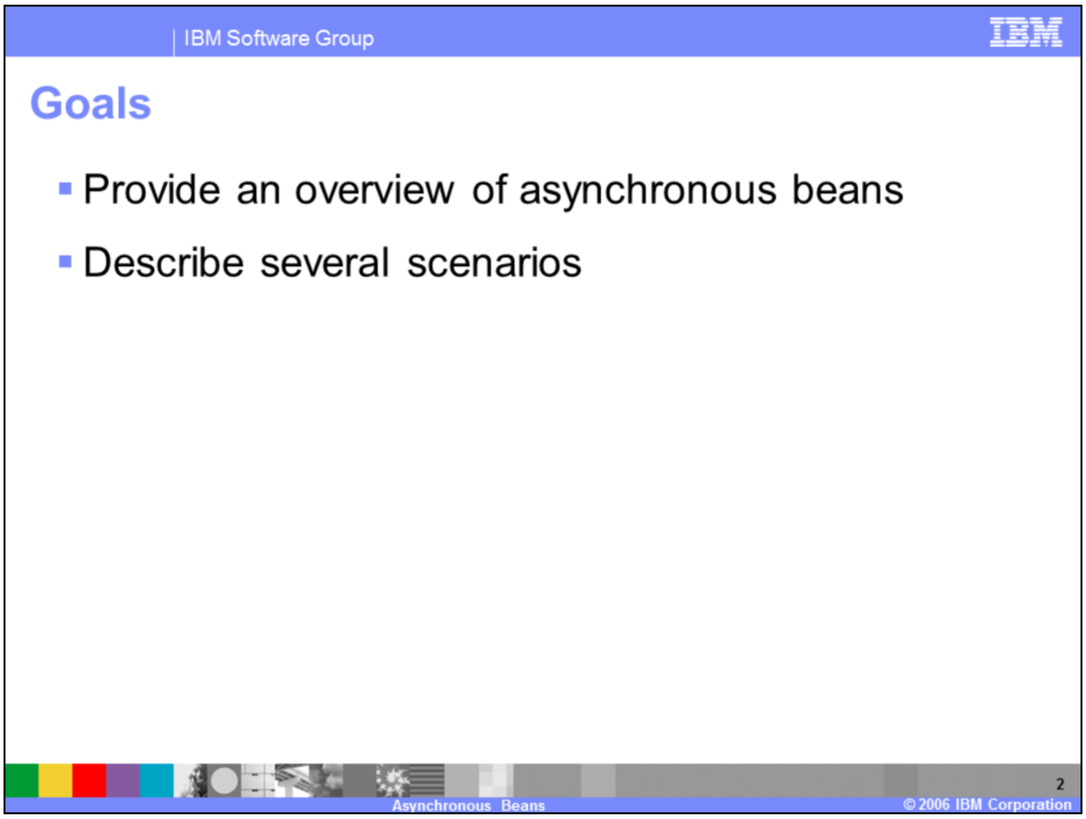
IBM Software Group

IBM WebSphere Application Server V6

Asynchronous beans

business on demand.

This presentation will discuss the Asynchronous Beans programming model extension in WebSphere Application Server V6.



IBM Software Group

Goals

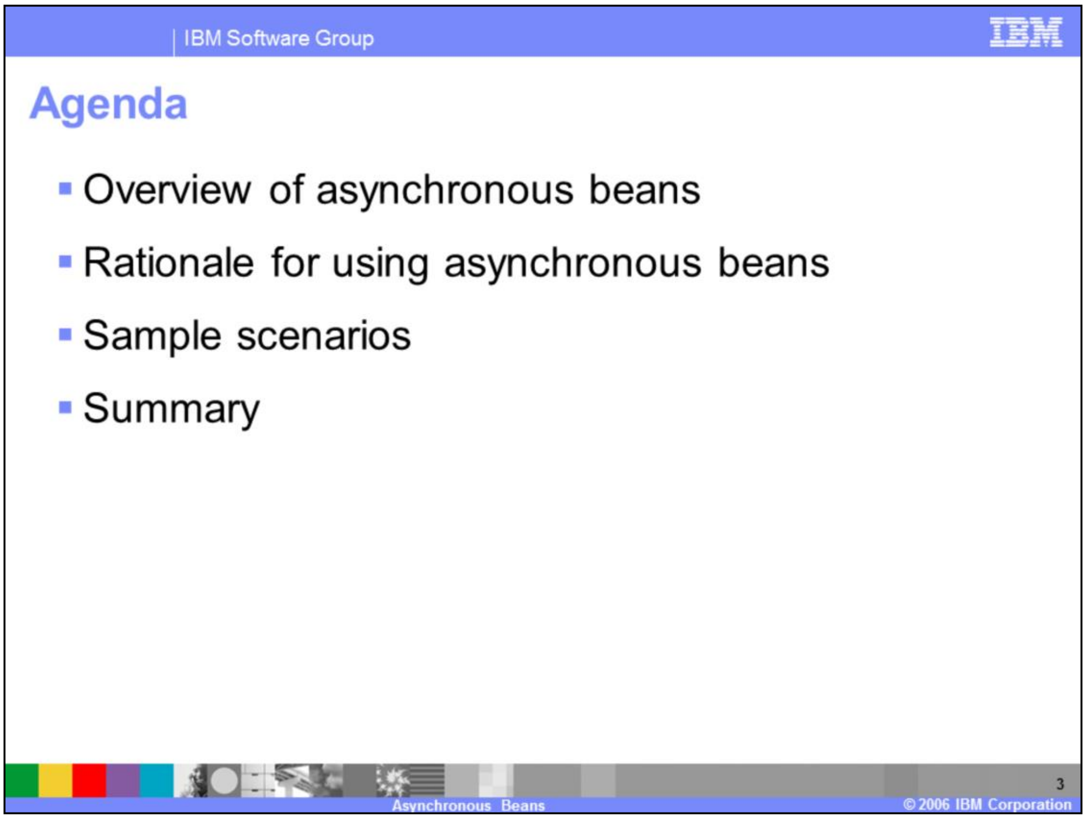
- Provide an overview of asynchronous beans
- Describe several scenarios

2

Asynchronous Beans

© 2006 IBM Corporation

The goal of this presentation is to describe the asynchronous programming capabilities and functionality provided by the Asynchronous Beans extension and to review several scenarios where this is useful.


IBM Software Group

IBM

Agenda

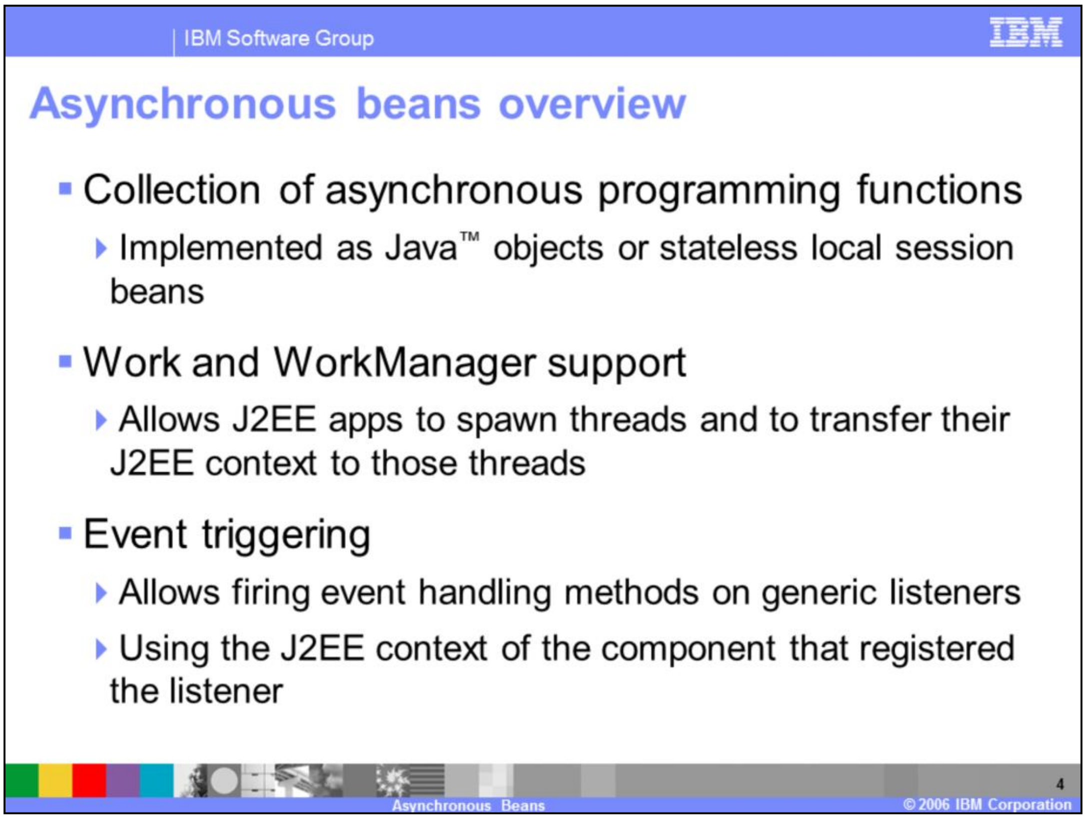
- Overview of asynchronous beans
- Rationale for using asynchronous beans
- Sample scenarios
- Summary

3

Asynchronous Beans

© 2006 IBM Corporation

The agenda for this presentation includes an overview of the Asynchronous Beans functionality in WebSphere Application Server V6 and why you would want to take advantage of this feature. You will also be shown several scenarios to help you understand where you might be able to take advantage of this very powerful set of asynchronous programming APIs.


IBM Software Group

Asynchronous beans overview

- Collection of asynchronous programming functions
  - Implemented as Java™ objects or stateless local session beans
- Work and WorkManager support
  - Allows J2EE apps to spawn threads and to transfer their J2EE context to those threads
- Event triggering
  - Allows firing event handling methods on generic listeners
  - Using the J2EE context of the component that registered the listener

4

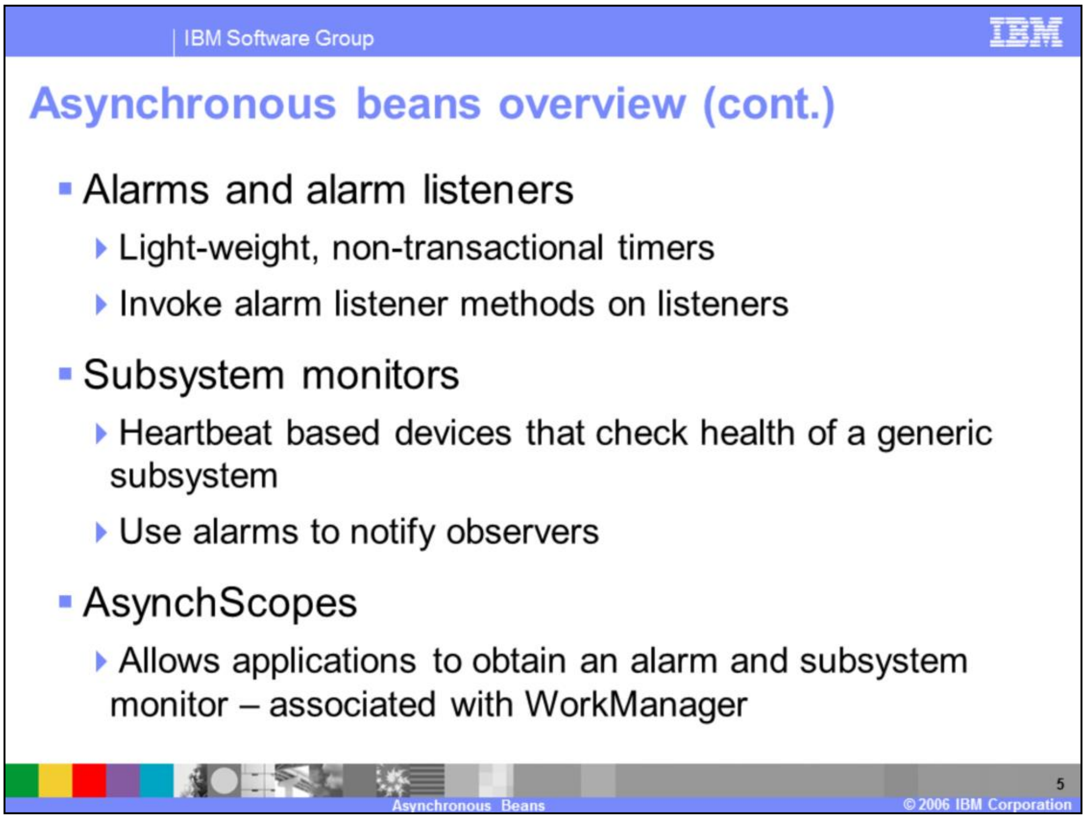
Asynchronous Beans

© 2006 IBM Corporation

The Asynchronous Beans extension covers a variety of functions to address asynchronous programming requirements. One of the main features is the capability to spawn threads and propagate the J2EE context to those threads. This is by submitting work to the WorkManager.

Work Manager for Application Server JSR 237 http://www.jcp.org/en/jsr/detail?id=237

Timer for Application Servers JSR 236 http://www.jcp.org/en/jsr/detail?id=236


IBM Software Group

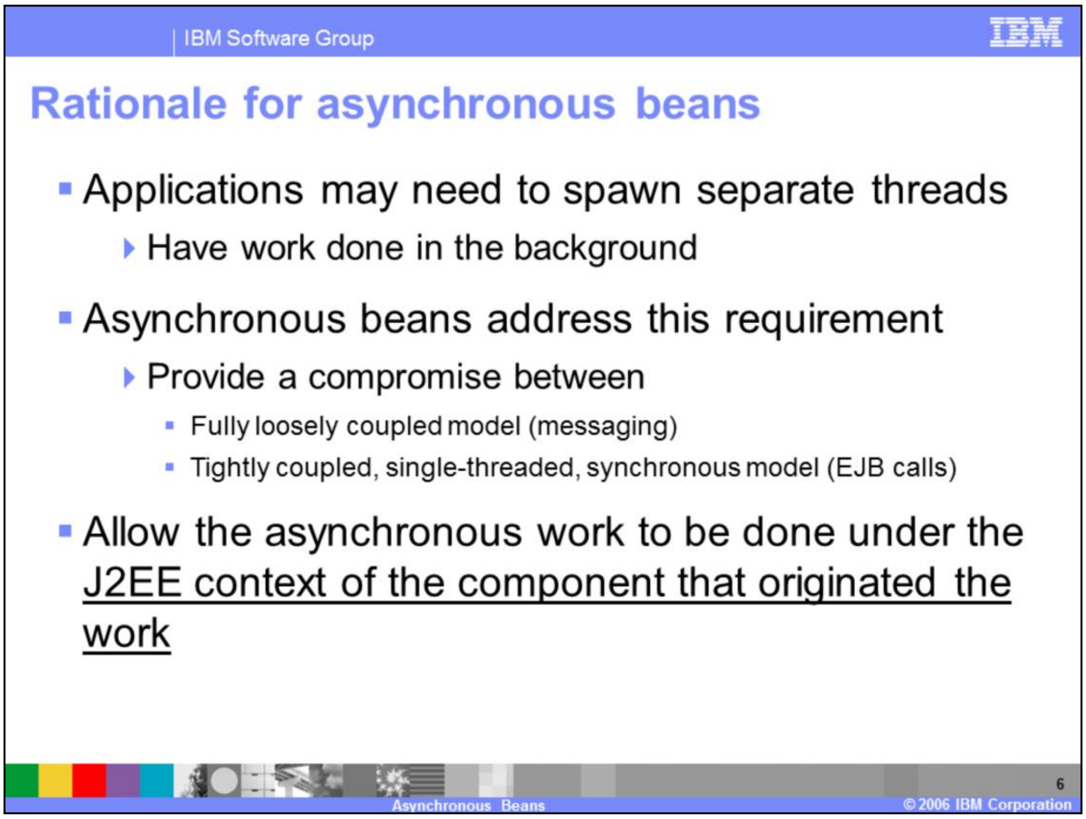
Asynchronous beans overview (cont.)

- Alarms and alarm listeners
  - Light-weight, non-transactional timers
  - Invoke alarm listener methods on listeners
- Subsystem monitors
  - Heartbeat based devices that check health of a generic subsystem
  - Use alarms to notify observers
- AsynchScopes
  - Allows applications to obtain an alarm and subsystem monitor – associated with WorkManager

5

Asynchronous Beans © 2006 IBM Corporation

You also see the other robust and high performance concurrency utilities, such as event triggering, alarms and alarm listeners, subsystem monitors, and AsynchScopes. You will learn more about each of these in the following charts.



IBM Software Group

Rationale for asynchronous beans

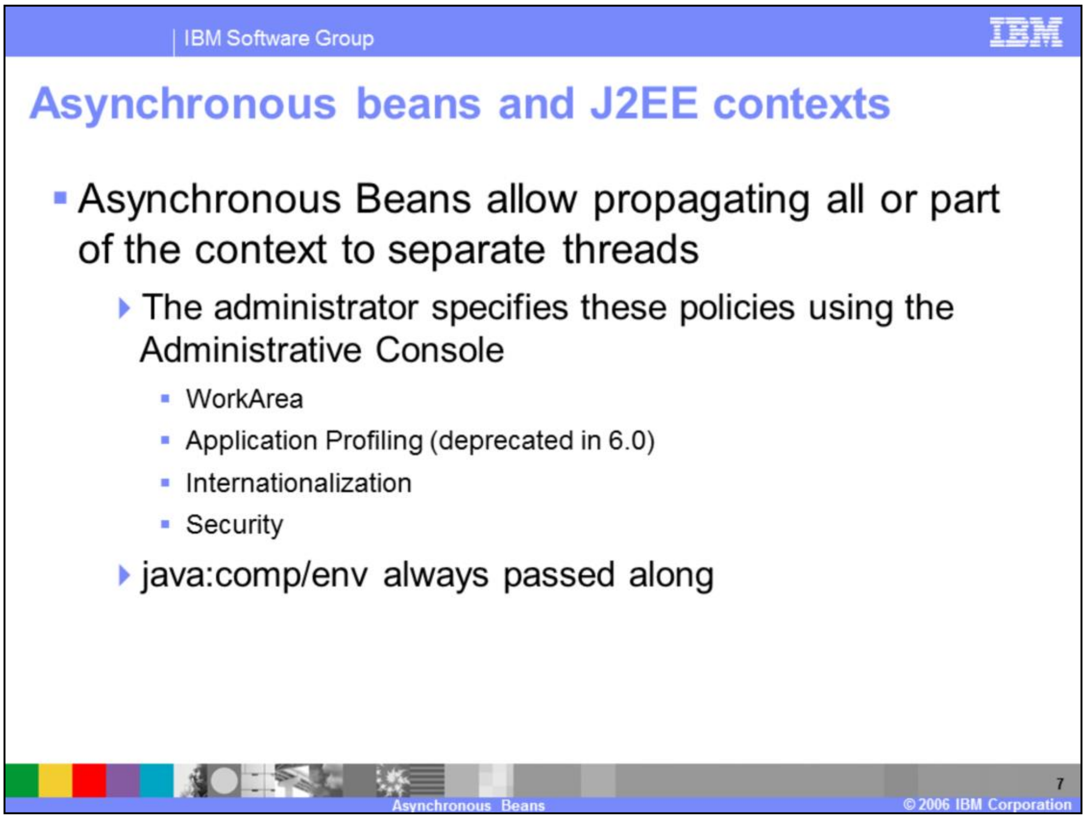
- Applications may need to spawn separate threads
  - Have work done in the background
- Asynchronous beans address this requirement
  - Provide a compromise between
    - Fully loosely coupled model (messaging)
    - Tightly coupled, single-threaded, synchronous model (EJB calls)
- Allow the asynchronous work to be done under the J2EE context of the component that originated the work

6

Asynchronous Beans © 2006 IBM Corporation

The primary motivation behind Asynchronous Beans is to enable Java™ 2 Enterprise Edition (J2EE) applications with full multi-threading support. By using asynchronous beans, your J2EE components will be able to submit code to be run on a separate thread in an asynchronous manner. The component that submitted the asynchronous bean will not wait for the bean to run, although typically there are mechanisms for the submitter to interact with the asynchronous bean. The asynchronous bean runs under the same J2EE context as the submitter - rather than with a J2EE context of its own. The asynchronous bean will therefore look up EJBs, access resources, and fall under the same authorization scheme as the component that created it and submitted it to run asynchronously.

This model represents a very interesting compromise between the loosely coupled approach used with messaging, where there is no propagation of context - and the tightly coupled approach of traditional J2EE programming, that runs in a single thread.


IBM Software Group

Asynchronous beans and J2EE contexts

- Asynchronous Beans allow propagating all or part of the context to separate threads
  - The administrator specifies these policies using the Administrative Console
    - WorkArea
    - Application Profiling (deprecated in 6.0)
    - Internationalization
    - Security
  - java:comp/env always passed along

Asynchronous Beans

© 2006 IBM Corporation

7

Normally, J2EE context can flow from a J2EE component to another component as long as both are called on the same thread. The Asynchronous Beans framework makes it possible to transfer all or some of the context to separate threads.

The administrator can also configure which parts of the context are going to be transferred, when a WorkManager is created. Any combination of the following four options is supported:

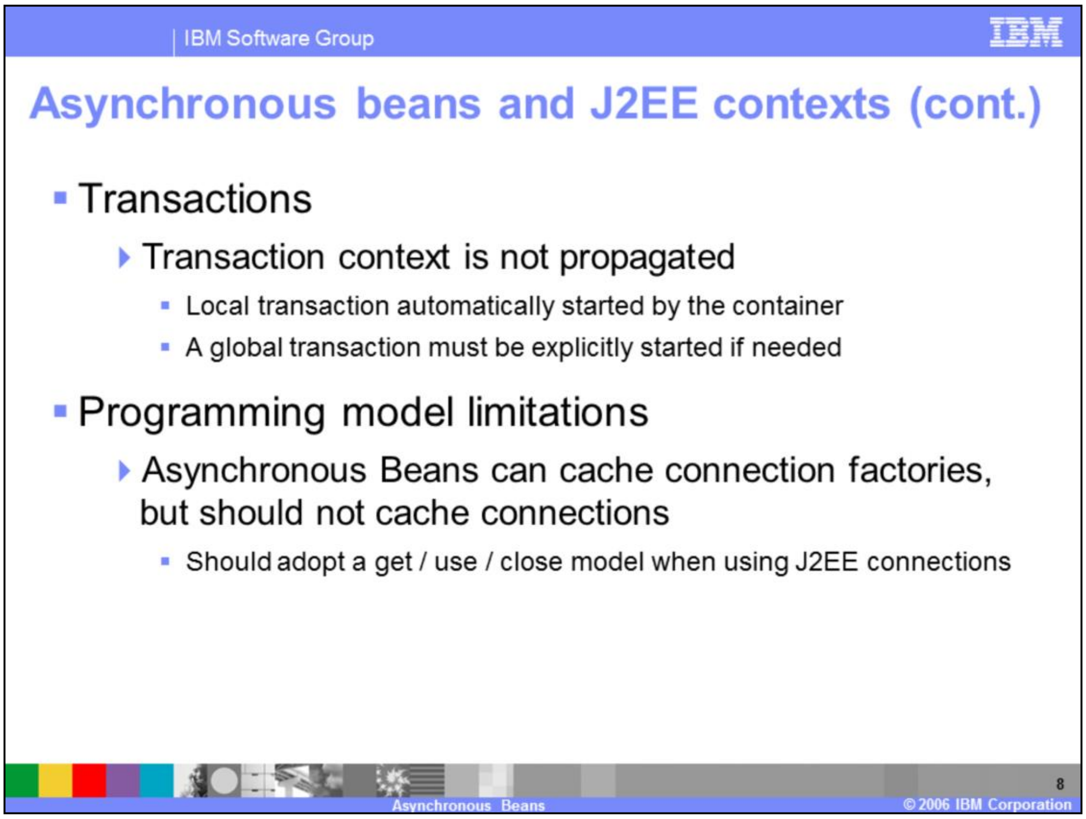
WorkArea

Internationalization context

Application profiling definitions (Transferring Application Profiling context is deprecated in V6.0).

Security

The java:comp/env local namespace is always passed along with Asynchronous Beans.

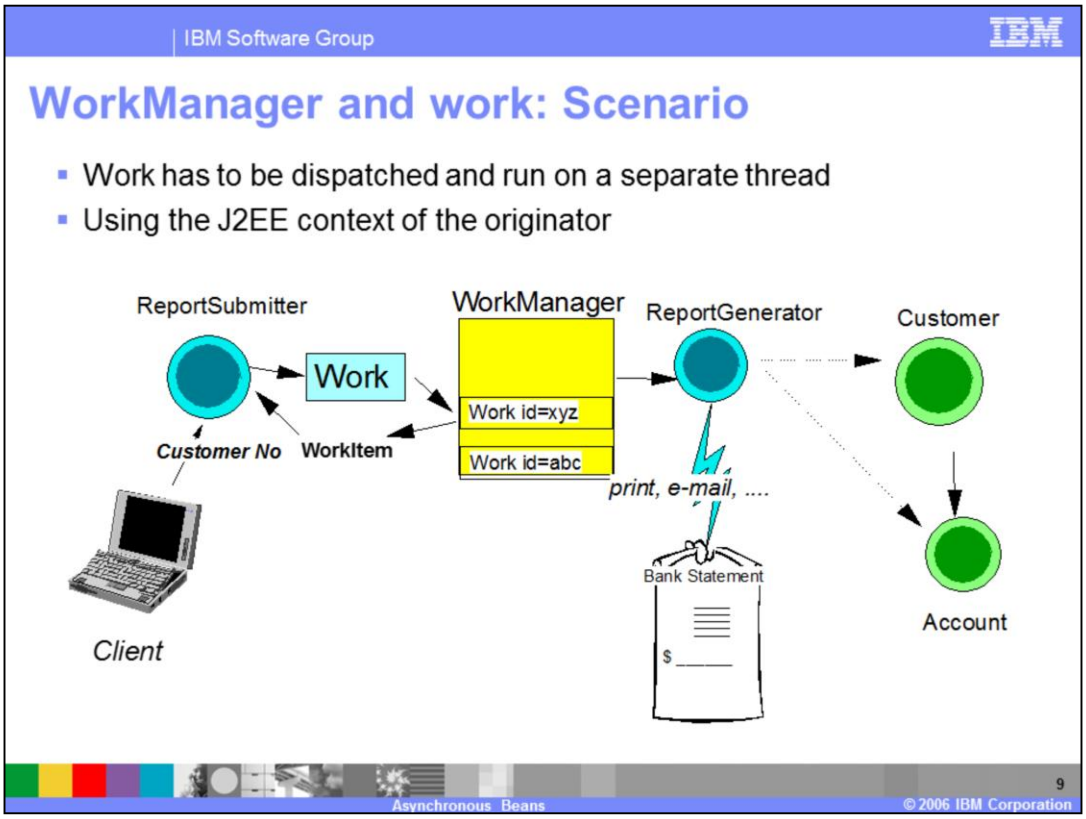


Asynchronous beans and J2EE contexts (cont.)

- Transactions
  - Transaction context is not propagated
    - Local transaction automatically started by the container
    - A global transaction must be explicitly started if needed
- Programming model limitations
  - Asynchronous Beans can cache connection factories, but should not cache connections
    - Should adopt a get / use / close model when using J2EE connections

Due to the asynchronous nature of this programming model, transactional contexts are not passed to the asynchronous beans. The asynchronous beans get their own local transaction, or they can start global transactions if XA resources are involved.

Avoid caching connections to avoid running out of connections to the backend system. Also, note that WebSphere does not support caching connections across multiple threads. You can cache connection factories, use them to get a connection, use the connection and then release it when the asynchronous process finishes running. That way, connection pooling is used at its best: the connection is not held by an asynchronous bean that is waiting to be run, rather it is used only when the bean actually runs.


IBM Software Group

WorkManager and work: Scenario

- Work has to be dispatched and run on a separate thread
- Using the J2EE context of the originator

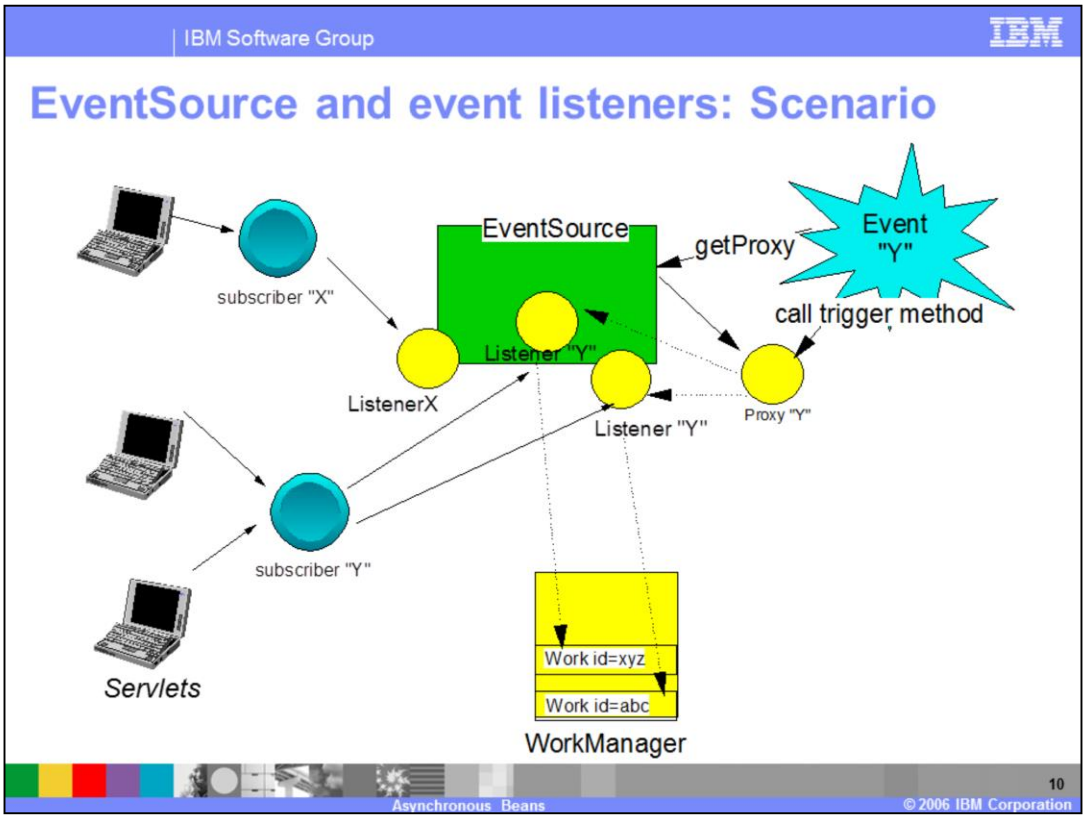
Asynchronous Beans

9

© 2006 IBM Corporation

This chart illustrates a simple example where an application creates an instance of a Work implementation that uses a ReportGenerator EJB to print a statement of account for a certain customer. The Work can be submitted to run in parallel on a separate thread.

Notice that the Work instance can still look up EJBs as if it were called directly by the ReportSubmitter bean.



This is a scenario where three servlet applications have registered a listener each with an EventSource. The three listeners implement different interfaces (X and Y), meaning they monitor different events.

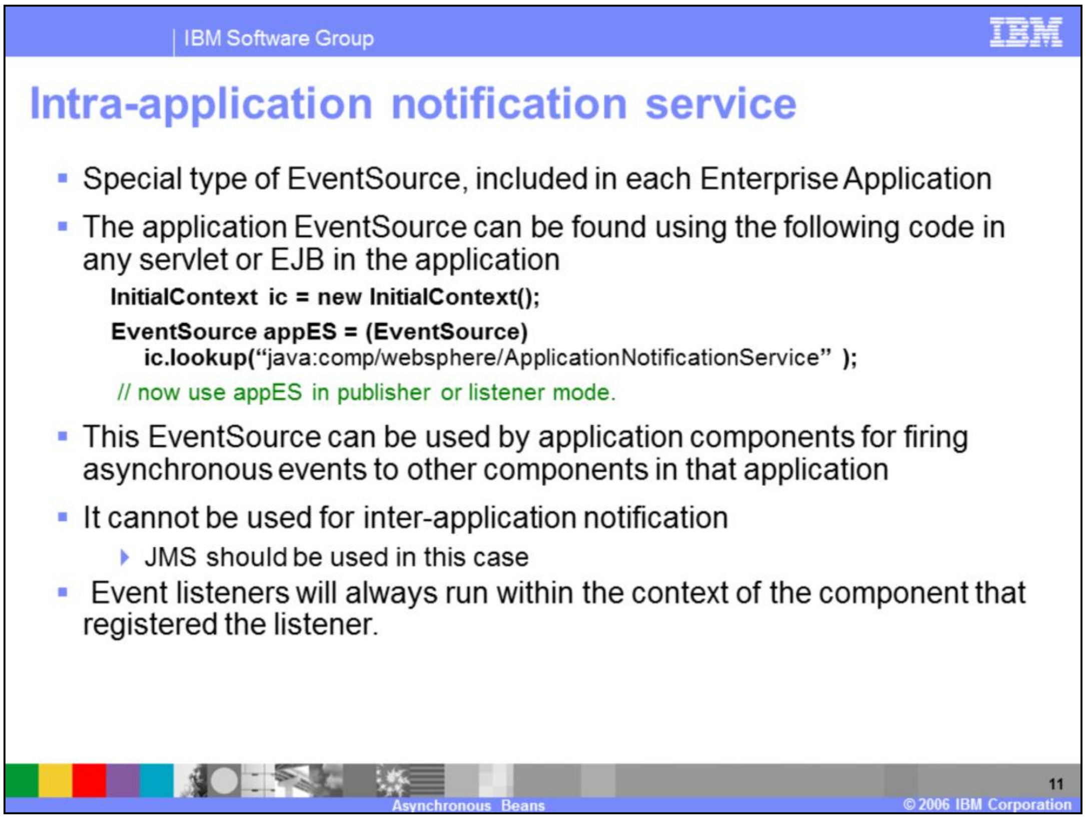
When a certain event occurs, the application that has generated it needs to acquire the event source, and ask it for a proxy to the listeners of a certain type. If the event Y occurs: the originator acquires a proxy to all the "Listener Y" objects that are currently registered.

Next, the event originator calls the firing method on the proxy (also called the event trigger), and that will cause the same method to be invoked on all the "Listener Y" objects.

Listeners have their methods invoked synchronously on the same thread as the publisher. Therefore, care needs to be taken with synchronization blocks. If the publisher has acquired a monitor on an object and if listeners are called on the same thread then they will also be able to acquire a monitor on that lock because it is the same thread. If the application wants the listeners to be called on a different thread then the application should start a Work instance that fires the event.

The order listeners are fired can be ordered by specifying an listener sequence number when calling addListener. If the sequence number is not specified then it defaults to 5. Valid sequence numbers are integers from 0..9. Listeners are invoked in ascending order, 0 first and 9 last.

In any case, the listeners use the same J2EE context as the servlet application that registered them.



Intra-application notification service

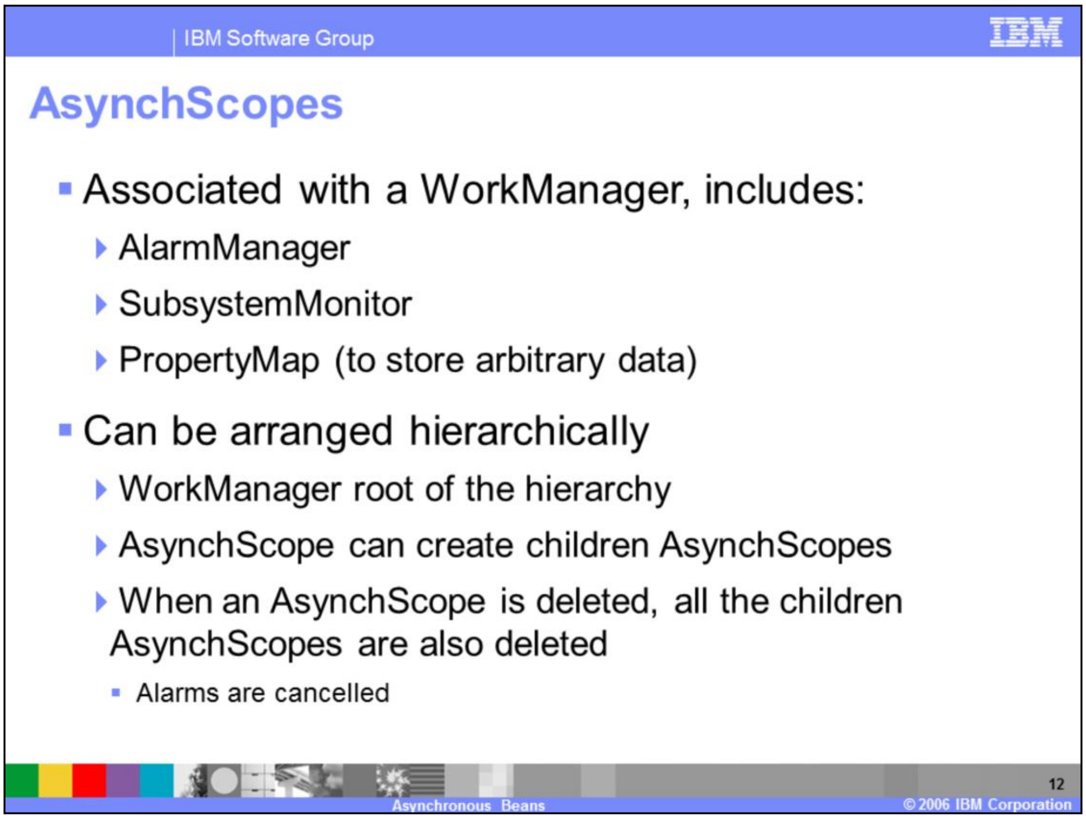
- Special type of EventSource, included in each Enterprise Application
- The application EventSource can be found using the following code in any servlet or EJB in the application

```
InitialContext ic = new InitialContext();
EventSource appES = (EventSource)
   ic.lookup("java:comp/websphere/ApplicationNotificationService" );
// now use appES in publisher or listener mode.
```

- This EventSource can be used by application components for firing asynchronous events to other components in that application
- It cannot be used for inter-application notification
  - JMS should be used in this case
- Event listeners will always run within the context of the component that registered the listener.

During the application lifetime, individual J2EE components (servlets or enterprise beans) within a single enterprise application (EAR file) might need to signal each other. There is an event source in the java:comp namespace that is bound into all components within an EAR file. This intra-application event notification service makes it possible for components that belong to the same application to communicate with each other through event notification (an EJB could subscribe a listener, another EJB could fire an event).

Keep in mind that this mechanism is not valid for communications that cross the boundaries of a single EAR - where JMS should rather be utilized.
Event listeners will always run within the context of the component that registered the listener.


IBM Software Group

IBM

AsynchScopes

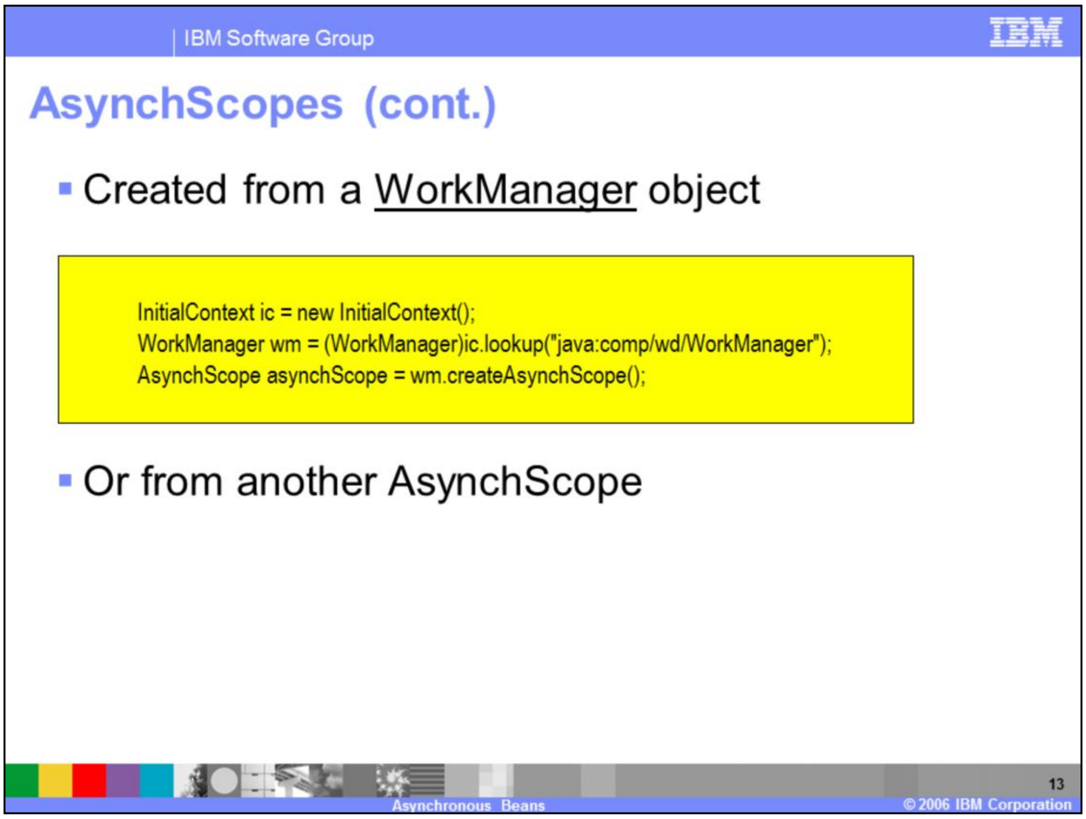
- Associated with a WorkManager, includes:
  - AlarmManager
  - SubsystemMonitor
  - PropertyMap (to store arbitrary data)
- Can be arranged hierarchically
  - WorkManager root of the hierarchy
  - AsynchScope can create children AsynchScopes
  - When an AsynchScope is deleted, all the children AsynchScopes are also deleted
    - Alarms are cancelled

12

Asynchronous Beans

© 2006 IBM Corporation

The AsynchScopes are key objects in the Asynchronous Beans framework that allow applications to obtain an Alarm Manager, a Subsystem Monitor, or a PropertyMap. They are associated with a WorkManager, which is the anchor point of the entire Asynchronous Beans architecture. This hierarchical structure is useful when you need to monitor complex subsystems that may have a hierarchical structure themselves.



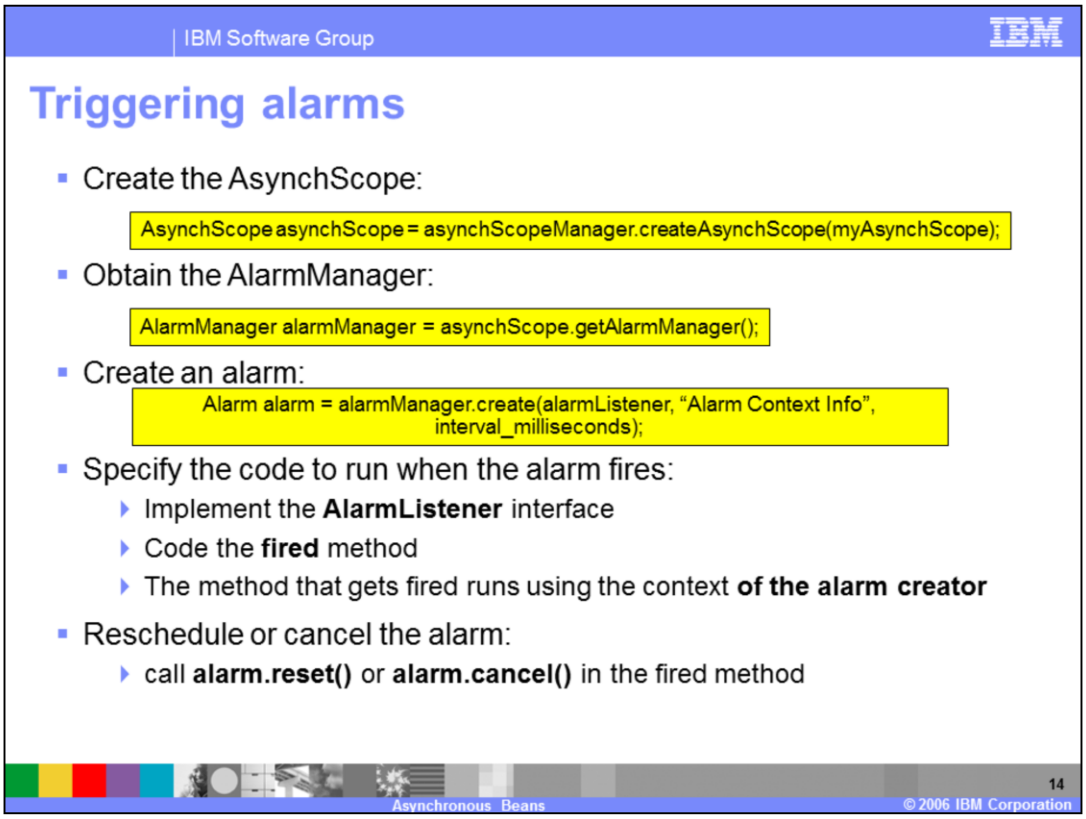
AsynchScopes (cont.)

- Created from a WorkManager object

```
InitialContext ic = new InitialContext();
WorkManager wm = (WorkManager)ic.lookup("java:comp/wd/WorkManager");
AsynchScope asynchScope = wm.createAsynchScope();
```

- Or from another AsynchScope

You can use the WorkManager to create instances of AsynchScopes. In turn, AsynchScopes themselves can be used to create other AsynchScopes. Those second, third, ... generation asynchronous scopes will be hierarchically connected to their "parents" - when a parent is destroyed, all the children (and implicitly all the alarms scheduled on the children) will go away as well.


IBM Software Group

Triggering alarms

- Create the AsynchScope:

```
AsynchScope asynchScope = asynchScopeManager.createAsynchScope(myAsynchScope);
```

- Obtain the AlarmManager:

```
AlarmManager alarmManager = asynchScope.getAlarmManager();
```

- Create an alarm:

```
Alarm alarm = alarmManager.create(alarmListener, "Alarm Context Info",
interval_milliseconds);
```

- Specify the code to run when the alarm fires:
  - Implement the **AlarmListener** interface
  - Code the **fired** method
  - The method that gets fired runs using the context **of the alarm creator**
- Reschedule or cancel the alarm:
  - call **alarm.reset()** or **alarm.cancel()** in the fired method

14

Asynchronous Beans

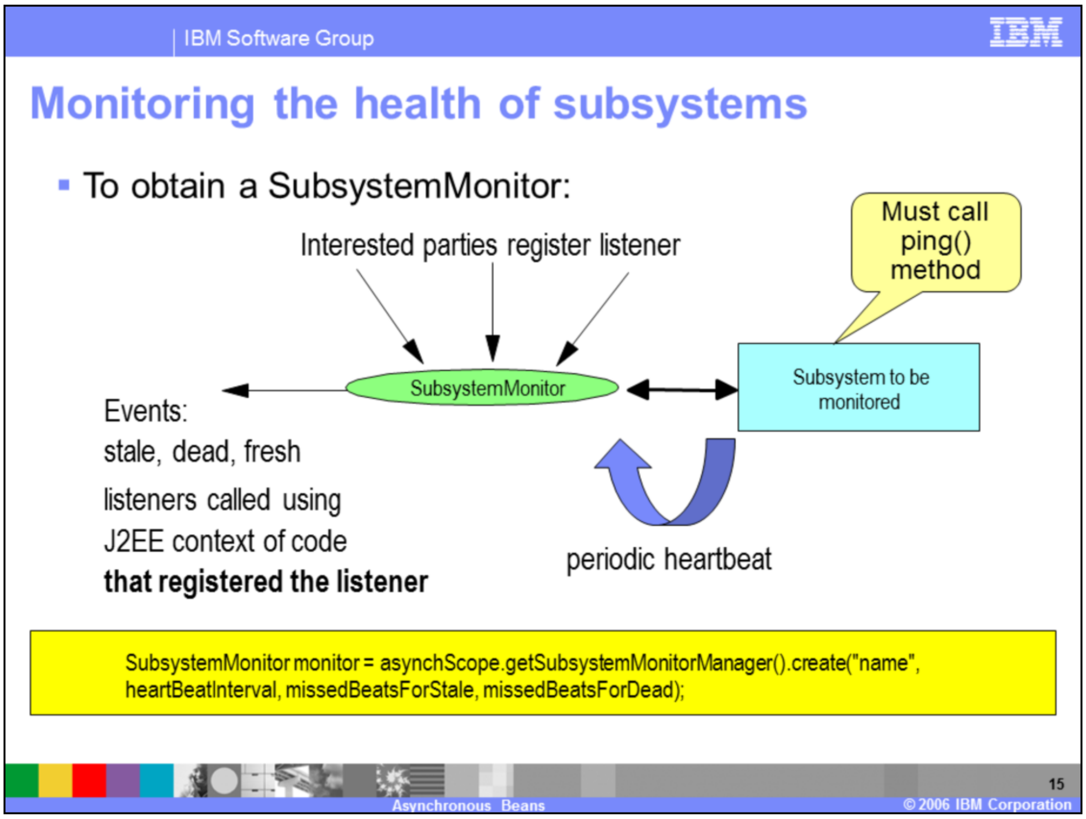
© 2006 IBM Corporation

An Alarm is a transient, high-performance timer similar to java.util.Timer. Applications can get an AlarmManager instance from an AsynchScope. The AlarmManager can then be used to create new alarms. Each alarm will run on an available thread at the configured time using a thread from the Alarm thread pool in the associated WorkManager.

When the alarm goes off, it will call the fired() method on the listener, using the J2EE context of the alarm creator. The creator can interact with the alarm manager and reset or cancel the alarm if necessary.

For example, consider the scenario where you need to implement a safety mechanism where a certain event has to occur every 5 seconds, or security has to be alerted. You can create an alarm and register a listener that will be called if the alarm goes off, after 5 seconds. If the event is received before the time limit, the alarm can be reset and the counter can be restarted.

These alarms are high-performance and transient. Applications that use them, need to recreate them after shutdown and restart of the application server..



Monitoring the health of subsystems

- To obtain a SubsystemMonitor:

```
SubsystemMonitor monitor = asynchScope.getSubsystemMonitorManager().create("name",
heartBeatInterval, missedBeatsForStale, missedBeatsForDead);
```

The objective of the SubsystemMonitor is to ensure that a certain application, or daemon, is alive and well, by receiving periodic heartbeats. SubsystemMonitors are very useful when time-dependent information and data currency are critical. For example, in a stock trading application, the requirement is to make sure that the stock quotes are fresh, and therefore there is a need to monitor the "subsystem" that publishes the quotes, in order to make sure that it is up and running.

The SubsystemMonitor can be obtained by an asynchronous bean and configured to fire two types of listeners:

One for the "stale" state of the subsystem

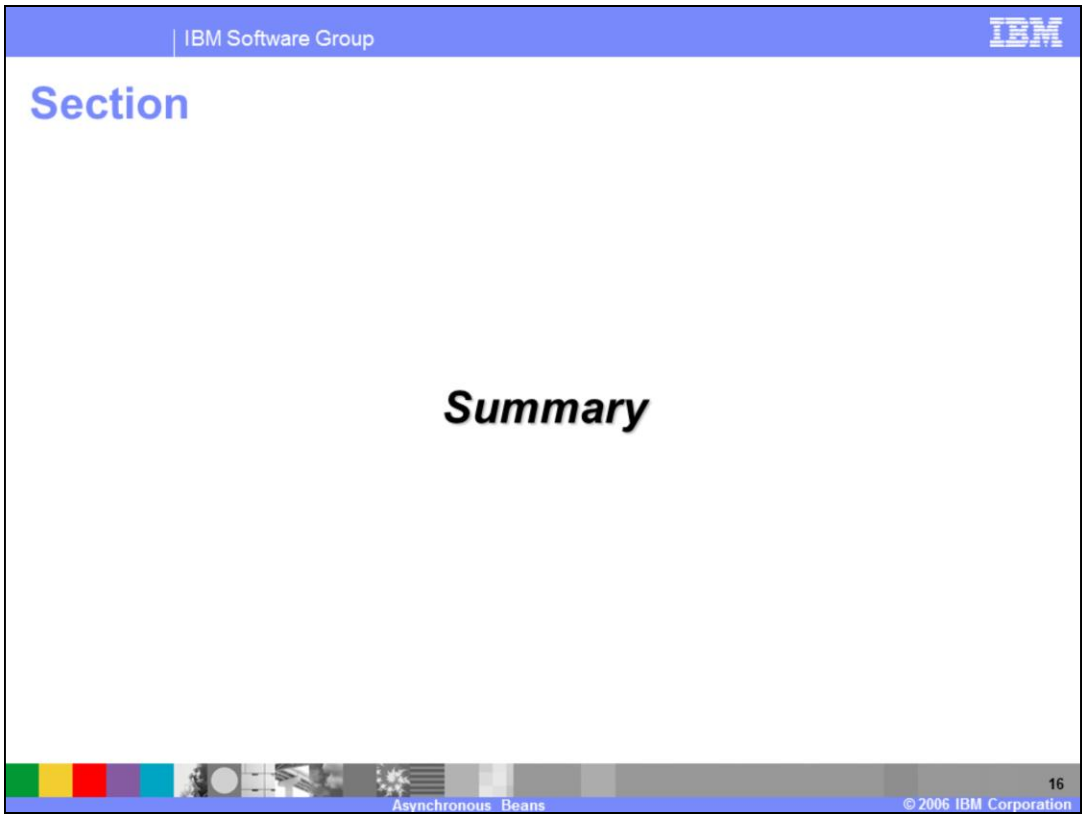
One for the "dead" state of the subsystem

You will be able to configure how often the monitored subsystem has to call the ping() method on the SubsystemMonitor, or the frequency of the heartbeat. You will also be able to configure how many missed heartbeats make a subsystem "stale", and how many missed heartbeats make it a "dead" subsystem.

How the subsystem sends heartbeats, and how they are detected in order that the ping method is called, is entirely application dependent and must be explicitly coded

Applications can register listeners with the SubsystemMonitor. The fact that the notification listeners are called using the J2EE context of the code that registered the listener is a fundamentally important aspect of the asynchronous functionality provided by the

WorkManager.



IBM Software Group

IBM

Section

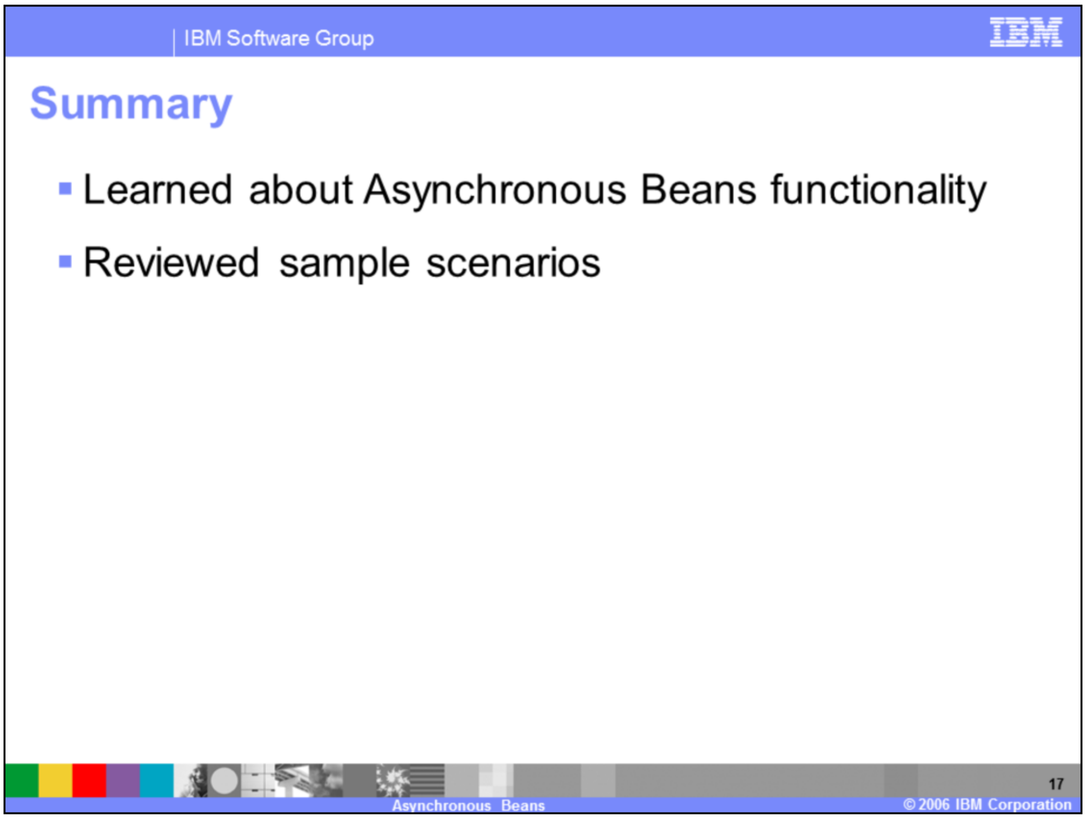
Summary

16

Asynchronous Beans

© 2006 IBM Corporation

And, in summary…


IBM Software Group

IBM

Summary

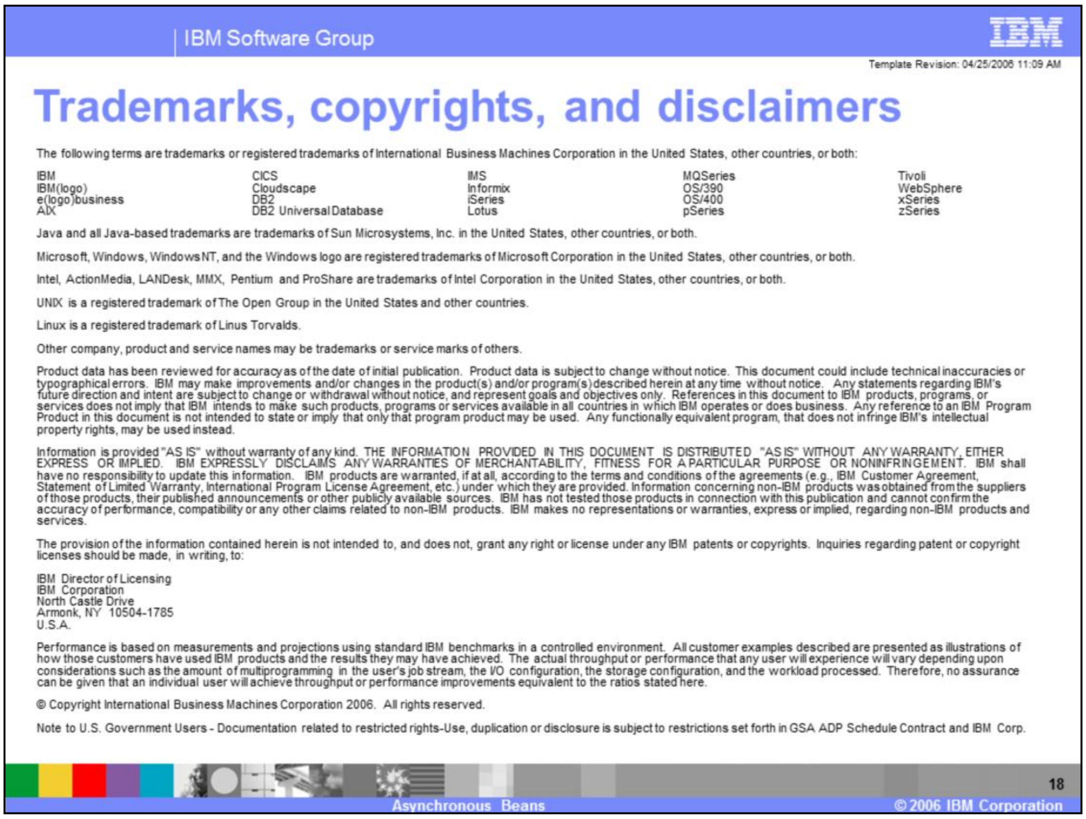
- Learned about Asynchronous Beans functionality
- Reviewed sample scenarios

17

Asynchronous Beans

This presentation covered the Asynchronous Beans programming model extension that covers a variety of functions to address asynchronous programming requirements. It also included several scenarios to help reinforce the understanding of where you might be able to take advantage of this functionality.

There are two Java Specification Requests (JSRs) related to this extension. There is Work Manager for Application Server JSR 237 http://www.jcp.org/en/jsr/detail?id=237, and Timer for Application Servers JSR 236 http://www.jcp.org/en/jsr/detail?id=236.



Template Revision: 04/25/2006 11:09 AM

Trademarks, copyrights, and disclaimers