
MONASH University
MONASH
INFORMATION
TECHNOLOGY
FIT5192 Lecture 9:
Advanced Applications of Java Enterprise Beans
GROUP
OF EIGHT
AUSTRALIA

MONASH University

Last Lecture

- Provide an understanding of the role Enterprise JavaBeans has in developing enterprise web applications with the Java EE platform.
- Review the role of Session Beans in implementing business logic code.

GROUP OF EIGHT AUSTRALIA

MONASH University

This Lecture

Understand some of the finer aspects of Enterprise JavaBeans implementations in the Java EE platform.

- Session Bean Life Cycle
- Callback Annotation
- Asynchronous Calls
- Timer Service

GROUP OF EIGHT AUSTRALIA

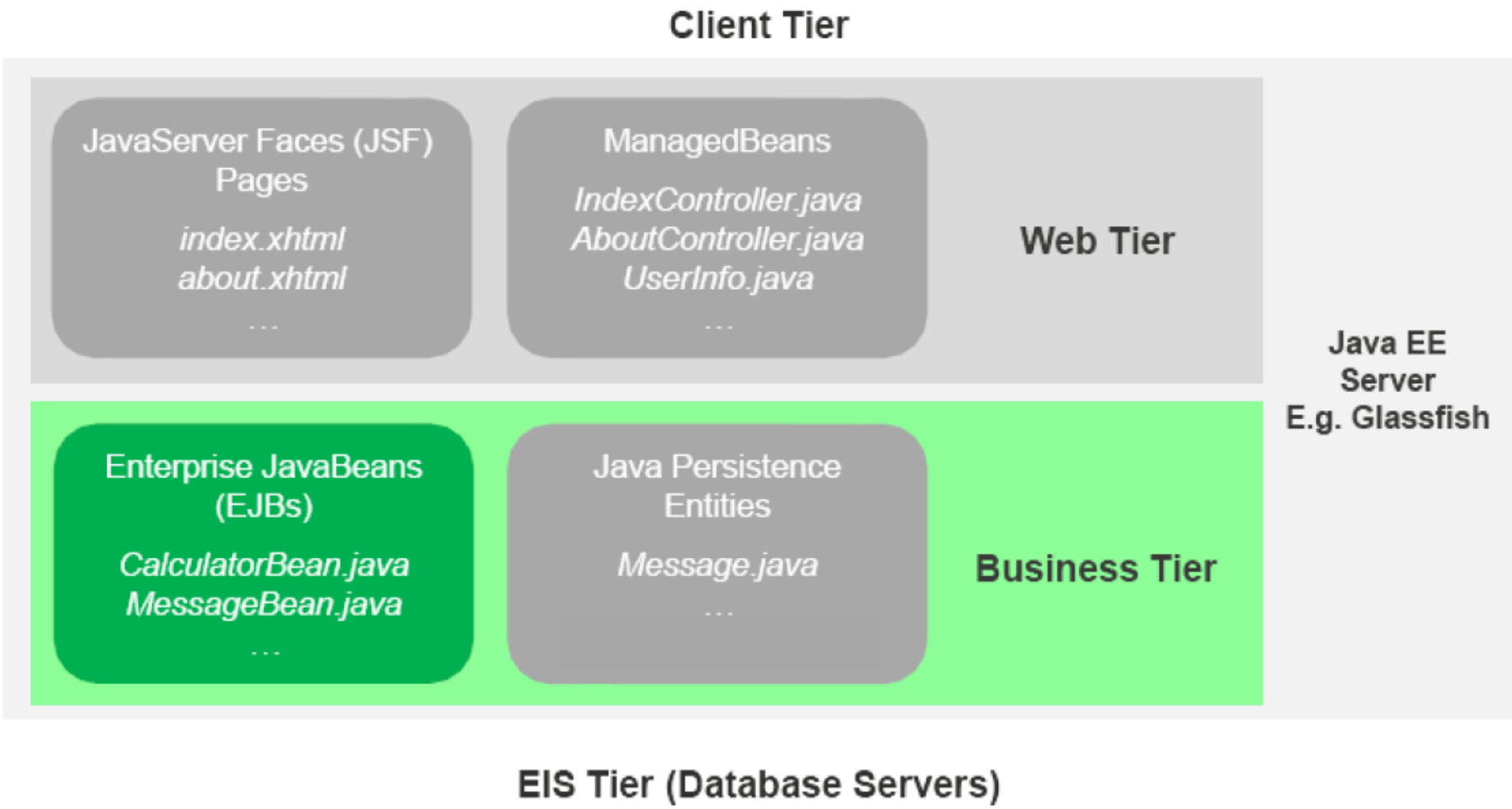
MONASH
University

Recap Enterprise JavaBeans

Recap on Enterprise JavaBeans

- **Specifically deals with the business logic programming for enterprise applications.** This is a major component for the "Business Tier" of the Java EE platform.
- **Enables ManagedBeans to handle communication between the user interface (JSF) and business data operations (EJBs).** It is considered a good practice to decouple client logic with business logic.
- Similar to how we don't want our CSS rules to be embedded inside the HTML document markup!
- **Using this approach makes it easier to develop and maintain complex enterprise applications.**

Overview of EJBs in Java EE Environment



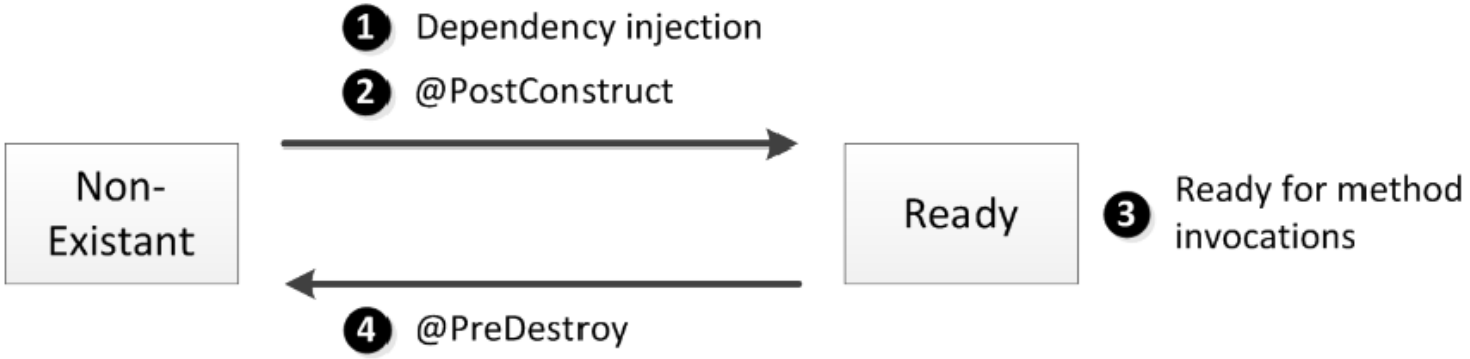
Session Bean Life Cycle

Session Beans Life Cycle

- **Clients do not need to manually create an instance of an EJB since we are getting a reference using dependency injection (DI) or through a JNDI lookup.**
  - The EJB container has the responsibility of monitoring the life cycle of each Session Bean.
- **Very similar to the life cycle of our Managed Beans.**
- While Stateless and Singleton Session Beans have the same life cycle, Stateful Session Beans require a slightly different cycle due to its potential persistent state.

Life Cycle: Stateless and Singleton

- **Don't maintain conversational state with a client.**
- Stateless is accessed on per-instance basis.
- Singleton provides concurrent access to a single instance.



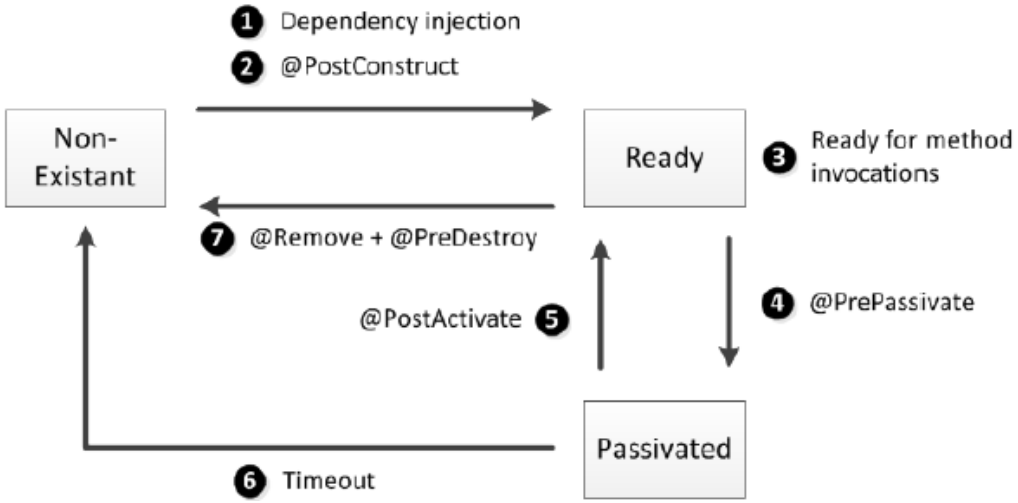
1. **Created via Dependency Injection (E.g. @EJB, @Inject)**
2. @PostConstruct methods are called upon creation.
3. **Bean has been created, ready for method invocations.**
4. @PreDestroy methods called before instance removal.

Life Cycle: Stateful (1)

- **Maintains a conversational state with the client.**
  - The EJB container will generate an instance and assign it to **one client.**
  - Each following request from that client is then sent to that instance.
- Following this approach allows us to gain a one-to-one relationship between a client and Stateful Session Bean.
  - Example. 100 users = 100 Stateful Session Beans Ideal for scenarios like a Shopping Cart

- **EJB container uses *Passivation* which serialises the bean instance to a storage medium / database when not required. *Activation* is used to bring back an instance into memory.**

Life Cycle: Stateful (2)



1. Created via Dependency Injection (E.g. @EJB, @Inject)
2. @PostConstuct methods are called upon creation.
3. Bean has been created, ready for method invocations.
4. Bean not in use, @PrePassivate methods called before serialisation
5. @PostActivate methods called for re-initialisation.
6. Timeout has occurred for the bean, remove it entirely.
7. @Remove and @PreDestroy methods called before removal.

Life-Cycle Callback Annotations

Annotation	Description
@PostConstruct	Marks a method to be invoked immediately after you create a bean instance and the container does dependency injection. This annotation is often used to perform any initialization.
@PreDestroy	Marks a method to be invoked immediately before the container destroys the bean instance. The method annotated with @PreDestroy is often used to release resources that had been previously initialized. With stateful beans, this happens after timeout or when a method annotated with @Remove has been completed.
@PrePassivate	Stateful beans only. Marks a method to be invoked before the container passivates the instance. It usually gives the bean the time to prepare for serialization and to release resources that cannot be serialized (e.g., it closes connections to a database, a message broker, a network socket, etc.).
@PostActivate	Stateful beans only. Marks a method to be invoked immediately after the container reactivates the instance. Gives the bean a chance to reinitialize resources that had been closed during passivation.

Example: Life-Cycle Callback

```
@Stateless
public class GreetingsBean {
   @PostConstruct
   private void init() {
      System.out.println("EJB has been
   initialised.");
   }
...
}
```

Asynchronous Calls

Asynchronous Method Calls

- **By default, all method invocations via EJBs are synchronous.**
  - Client invokes a method and it remains blocked until processing has been completed.
  - Problematic in scenarios such as multi-threading or operations that take considerable time (such as network requests).
- **Asynchronous processing is a very common requirement for enterprise applications since many clients may need many requests to be processed at once.**
  - We design asynchronous methods when we want to process actions of an unknown or long duration.

Asynchronous Methods in EJBs

- **EJB 3.1+ enable us to support asynchronous processing for methods without relying on external technologies such as Java Messaging Service.**
  - Prevents a huge amount of backend functionality to be added to an application to support this type of processing.
- **We declare asynchronous methods in our EJB via the @Asynchronous annotation.**

Example:

```
@Asynchronous
public void sendNetworkRequest(String data) {
// Long processing goes here
}
```

Getting Data from Asynchronous Calls (1)

- **Calling asynchronous methods is done like any other method call however we run into issues in returning data.**
  - How can we get data from a method when we don't know when it will be completed?
- **We can return data by implementing the Future<V> interface where V represents the result type.**
  - Future objects in the Java Concurrent API enable us to return values from methods executed in another thread.
  - Client can use the Future API to receive the result or if needed, cancel the call.

Getting Data from Asynchronous Calls (2)

- **Checking status of Asynchronous Method Call**
  - Future<V>.isDone()
    - Returns *true* if processing completed normally, was cancelled or if an exception is raised.
- **Retrieving result from Asynchronous Method Call**
  - Future<V>.get()
    - Returns the result of the type specified in the Future object instance (client will wait for result!).
    - *CancellationException* is thrown if the call was cancelled or *ExecutionException* if processing failed by the EJB.
  - Also possible to assign a timeout value for get(…)

Getting Data from Asynchronous Calls (3)

- **Cancelling result for Asynchronous Method Call**
  - Future<V>.cancel(boolean mayInterruptIfRunning)
    - Returns *true* if processing completed normally, was cancelled or if an exception is raised.
    - Boolean flag parameter if *true* notifies the EJB session bean instance that the client has attempted to cancel the method call.
  - Future<V>.isCancelled()
    - Returns *true* if the method call was cancelled.
    - Helps client determine how method call ended.

MONASH
University

Timer Service

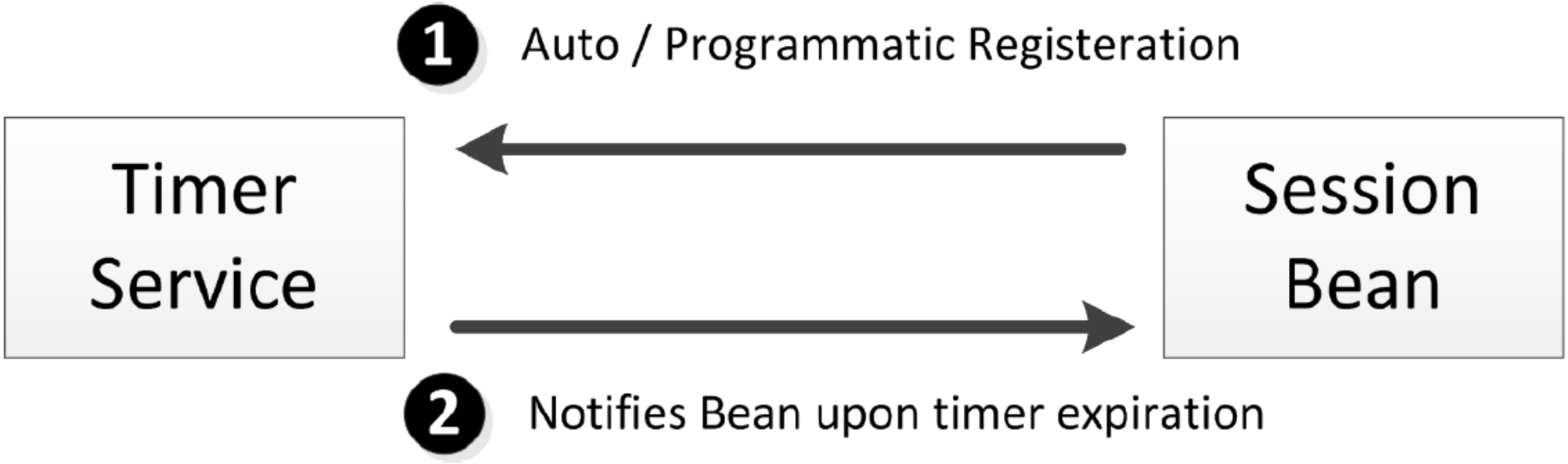
Scheduling Tasks

- **Many Enterprise web applications need to routinely schedule various types of operations:**
  - Backup operations
  - Cache operations
  - Database queries
  - Email notifications
  - Report production
  - And so on!
- **These type of operations can be executed automatically without a client specifically invoking a method.**
  - We can run these type of operations in Java EE using the *Timer Service.*

Timer Service

- **The EJB container provides us with the *Timer Service* which enables us to schedule timed notification for all EJB types *except* Stateful Session Beans.**
  - Stateful beans are conversation scoped and should *never* use scheduling functions of such a service.
- Timers can be established automatically by the container. Once deployed should the appropriate EJBs have methods annotated with the @Schedule annotation.
- **Additionally, Timers can also be created programmatically but must allocate one callback method for Timeout operations with the @Timeout annotation.**

Timer Service Interaction



Calendar-Based Expression (CE)

- **A Calendar-based syntax is used for establishing timers within an EJB:**
  - *ScheduleExpression* class is used for programmatic timer creation.
  - @Schedule annotation is used for automatic timer creation.
- **This is also possible to define via the deployment descriptor.**
- **Let's take a look at the various components of Calendar-based Expression:**
  - Will be referring to it as 'CE' for the following slides.

Common CE Attributes

Attribute	Description	Possible Values	Default Value	Examples
second	One or more seconds within a minute	0 to 59	0	second="30"
minute	One or more minutes within an hour	0 to 59	0	minute="15"
hour	One or more hours within a day	0 to 23	0	hour="13"
dayOfWeek	One or more days within a week	0 to 7 (0,7 = Sun) Sun, Mon, …, Sat	*	dayOfWeek="3" dayOfWeek="Mon"
dayOfMonth	One or more days within a month	1 to 31 -x (from last day of month) 1st, 2nd, 3rd, …, 31st Sun, Mon, …, Sat Last (Last day of month)	*	dayOfMonth="15" dayOfMonth="-3" dayOfMonth="Last" dayOfMonth="2nd Fri"

- Source: *Beginning Java EE 7, Antonio Goncalves - pg. 270*

Specifying multiple values in CE

Form	Description	Example
Single Value	The attribute has only one possible value.	year = "2010“
Wildcard	This form represents all possible values for a given attribute	second = "*“ dayOfWeek = "*"
List	The attribute has two or more values separated by a comma.	year = "2006, 2008, 2010"
Range	The attribute has a range of values separated by a dash.	second = "1-10“ dayOfWeek = "Mon-Fri"
Increments	The attribute has a starting point and an interval separated by a forward slash.	minute = "*/15“ second = “30/10"

Programmatic Timers

- **Creating timers programmatically requires access to the javax.ejb.TimerService interface using DI or JNDI lookup approach.**

Annotation	Description
`createTimer`	Creates a timer based on dates, intervals, or durations. These methods do not use calendar-based expressions.
`createSingleActionTimer`	Creates a single-action timer that expires at a given point in time or after a specified duration. The container removes the timer after the timeout callback method has been successfully invoked.
`createIntervalTimer`	Creates an interval timer whose first expiration occurs at a given point in time and whose subsequent expirations occur after specified intervals.
`createCalendarTimer`	Creates a timer using a calendar-based expression with the ScheduleExpression helper class.
`getAllTimers`	Returns the list of available timers.

Programmatic Timers

- **ScheduleExpression helper class allows us to create CEs programmatically.**

- **Example:**

```
new ScheduleExpression().dayOfMonth("Mon").month("Jan");
new ScheduleExpression().second("10,30,50").hour("10-14");
new ScheduleExpression().dayOfWeek("1").timezone("Europe/Lisbon");
```

- **Once the timer has expired, EJB container calls the associated @Timeout method of the bean, passing the *Timer* object.**
  - A bean can only have ***one*** @Timeout method.

Example: Programmatic Timer

```
@Stateless
public class NewsletterBean {
    @Resource
    TimerService timerService;
    // ...
    public void createNewsletter(Newsletter ns) {
        // ...
        ScheduleExpression sendDate = new
        ScheduleExpression().dayOfWeek("Sun");
        timerService.createCalendarTimer(sendDate, new
        TimerConfig(ns, true));
    }
    @Timeout
    public void sendNewsletter(Timer t) {
        Newsletter ns = (Newsletter) timer.getInfo();
    // ...
}}
```

Automatic Timers

- **We will be more concerned with using Automatic Timers since they are easier to deploy.**
  - This approach has the added benefit of supporting multiple automatic timeout methods.

- **These timers are created by the EJB container when an Enterprise Bean contains methods that are annotated with either the @Schedule or @Schedules annotation.**
  - @Schedules within an EJB allows multiple automatic timeout methods to be specified.

Example: Automatic Timer

```
@Stateless
public class ReportBean {
    @Schedule(dayOfMonth = "1", hour = "20")
    public Report generateReport() {
        ...
    }
    @Schedules({
        @Schedule(hour = "2"),
        @Schedule(dayOfWeek = "Sun")
    })
    public void sendEmailNewsletter() {
        …
    }
}
```

Summary

- Understand some of the finer aspects of Enterprise JavaBeans implementations in the Java EE platform.
  - Session Bean Life Cycle
  - Callback Annotation
  - Asynchronous Calls
  - Timer Service

Next Lecture

- **Web Services**

MONASH University

See you in the Studio !

Readings

MONASH University

- **Chapter 8 in *Beginning Java EE 7 by Antonio Goncalves***

- ***Chapter 28 Running the Enterprise Bean Examples* in The Java EE 7 Tutorial**
  - It would be a benefit reviewing the sample projects listed as they provide very clear examples of key EJB implementations