Developing Business Logic With Session Beans

Objectives

After completing this lesson, you should be able to do the following:

- Describe session beans
- Create stateless and stateful session beans by using annotations
- Describe the passivation and activation of stateful session beans
- Use interceptor methods and classes
- Timer

EJBs and EJB Container

- Enterprise JavaBean
  - Server-side component that encapsulates the business logic of an application
- EJB container
  - An environment in which EJBs are executed
  - Provides system-level services, such as transactions and security, to the enterprise beans
- Types of EJB containers:
  - Full implementation
  - Embeddable

```
import javax.ejb.embeddable.*;
...
EJBContainer container = EJBContainer.createEJBContainer();
Context ctx = container.getContext();
SomeBean foo = (SomeBean)ctx.lookup("java:global/SomeBean");
...
```

Enterprise JavaBean Types

Enterprise JavaBean classification:

Message-Driven
Act as a message consumer for a Queue or a Topic

Session		
Stateful	Stateless	Singleton
Perform tasks that require to hold client specific information across method invocations	Perform tasks that do not require to retain any client specific context across method calls	Perform tasks that use information shared across the entire application
	May be exposed as a WebService	May be exposed as a WebService

What Is a Session Bean?

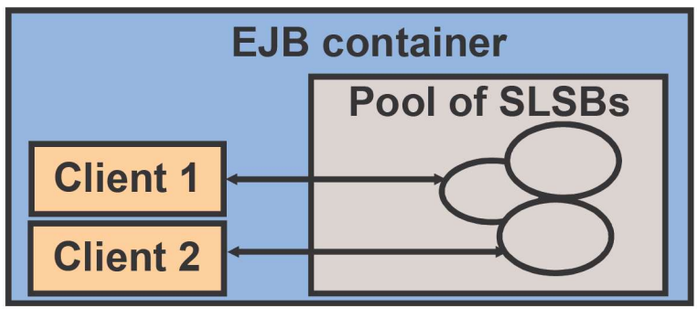
A session bean is a type of Enterprise JavaBean (EJB) that:

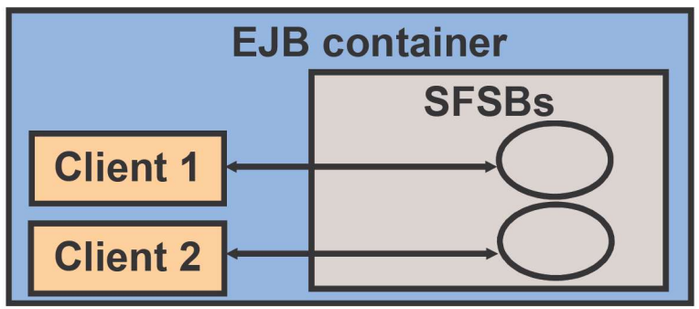
- Implements a business process
- Represents a client/server interaction
- Has a short lifespan
- Lives in memory rather than in persistent storage
- Is used to create a Session Facade

Stateless Versus Stateful Session Beans

There are two types of session beans:

- Stateless session bean (SLSB):
  - Conversation is contained in a single method call.
  - Business process does not maintain client state.
- Stateful session bean (SFSB):
  - Conversation may invoke many methods.
  - Business processes can span multiple method requests, which may require maintaining a client state.

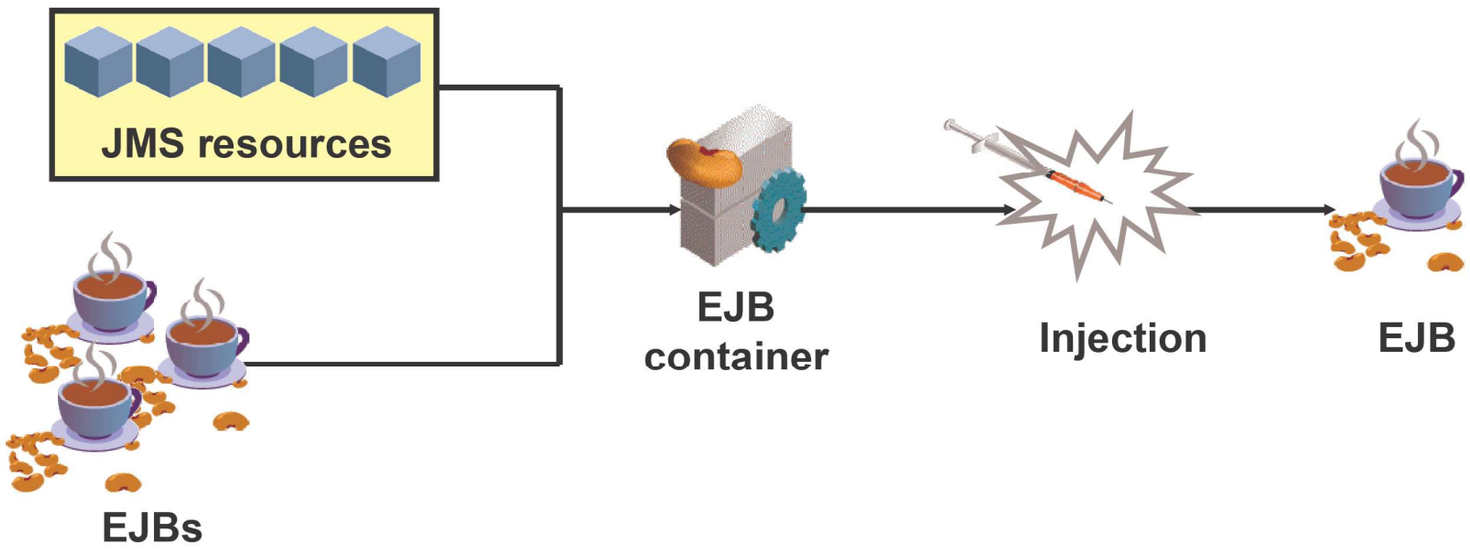




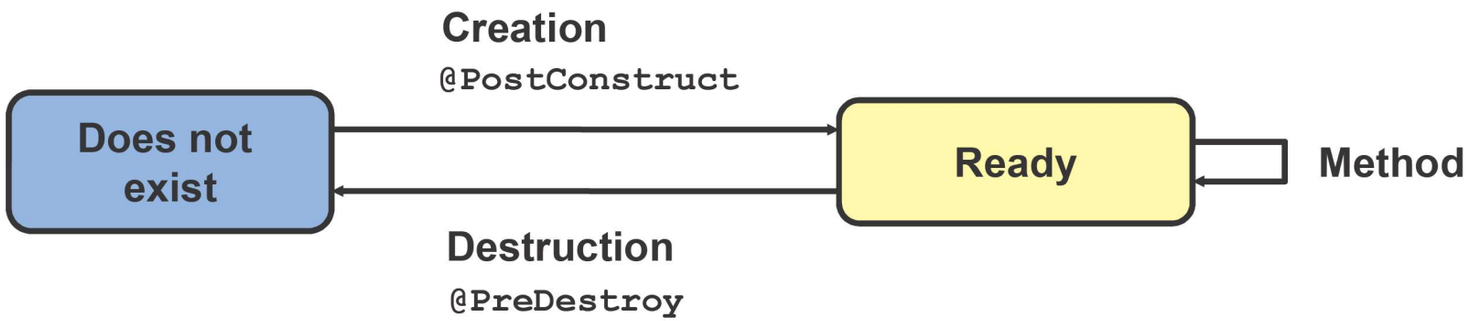
Dependency Injection in EJB

The concept of dependency injection implies that it is:

- A programming design pattern that enables you to inject resources into a session bean at run time
- Based on the principle of Inversion of Control (IOC)
- An alternative to the JNDI implementation



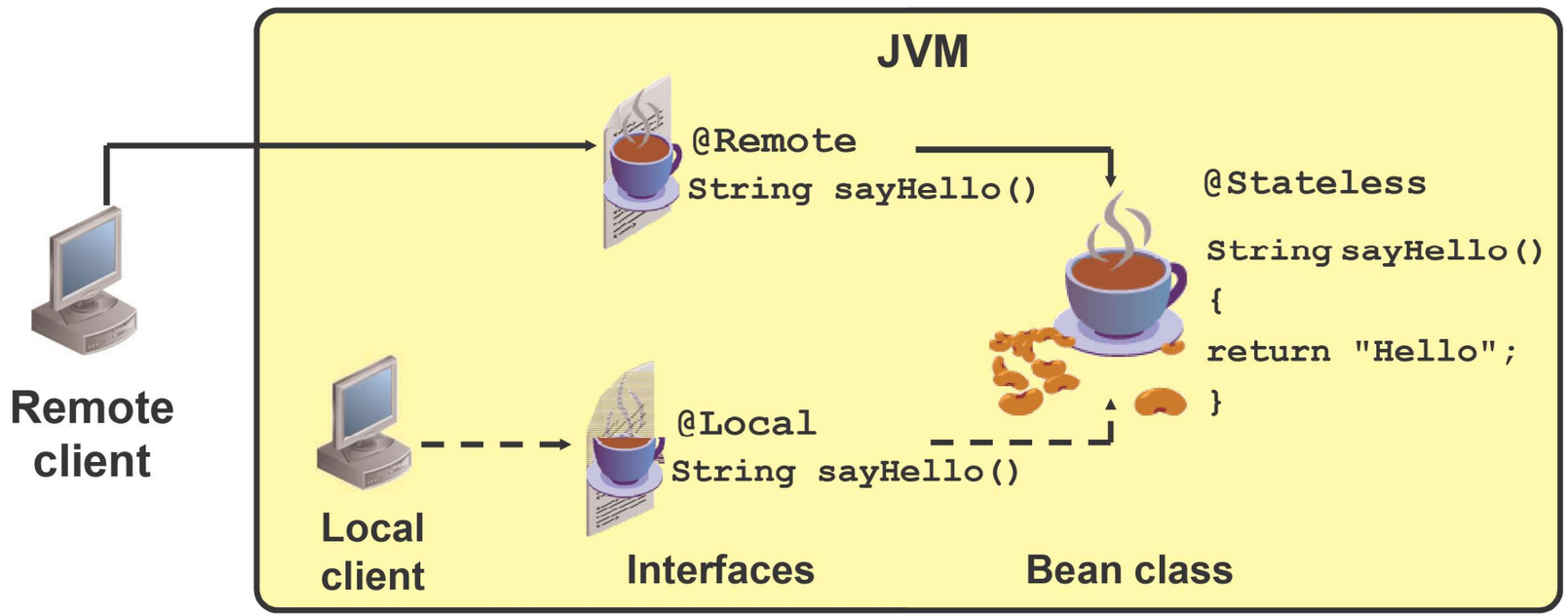
Life Cycle of a Stateless Session Bean



Elements of a Stateless Session Bean

A stateless session bean contains:

- Business interfaces, which contain the business method definition
- A bean class, which implements the business method



Defining the Stateless Session Bean

```
// HelloWorldBean.java
package helloworld.ejb;
import javax.ejb.Stateless;

@Stateless(name="HelloWorld",
                   mappedName="HelloWorldSessionEJB")

public class HelloWorldBean implements HelloWorld
{
  public String sayHello()
  {
    return "Hello World!";
  }
}
```

1 2 3 4 5

Analyzing the Remote and Local Interfaces

```
// HelloWorld.java
package helloworld.ejb;                    1
import javax.ejb.Remote;

@Remote                                    2
                                           3
public interface HelloWorld {
    public String sayHello();              4
}
```

```
// HelloWorldLocal.java
package helloworld.ejb                     1
import javax.ejb.Local;

@Local                                     2
                                           3
public interface HelloWorldLocal {
    public String sayHello();              4
}
```

Creating a Test Client for the SLSB

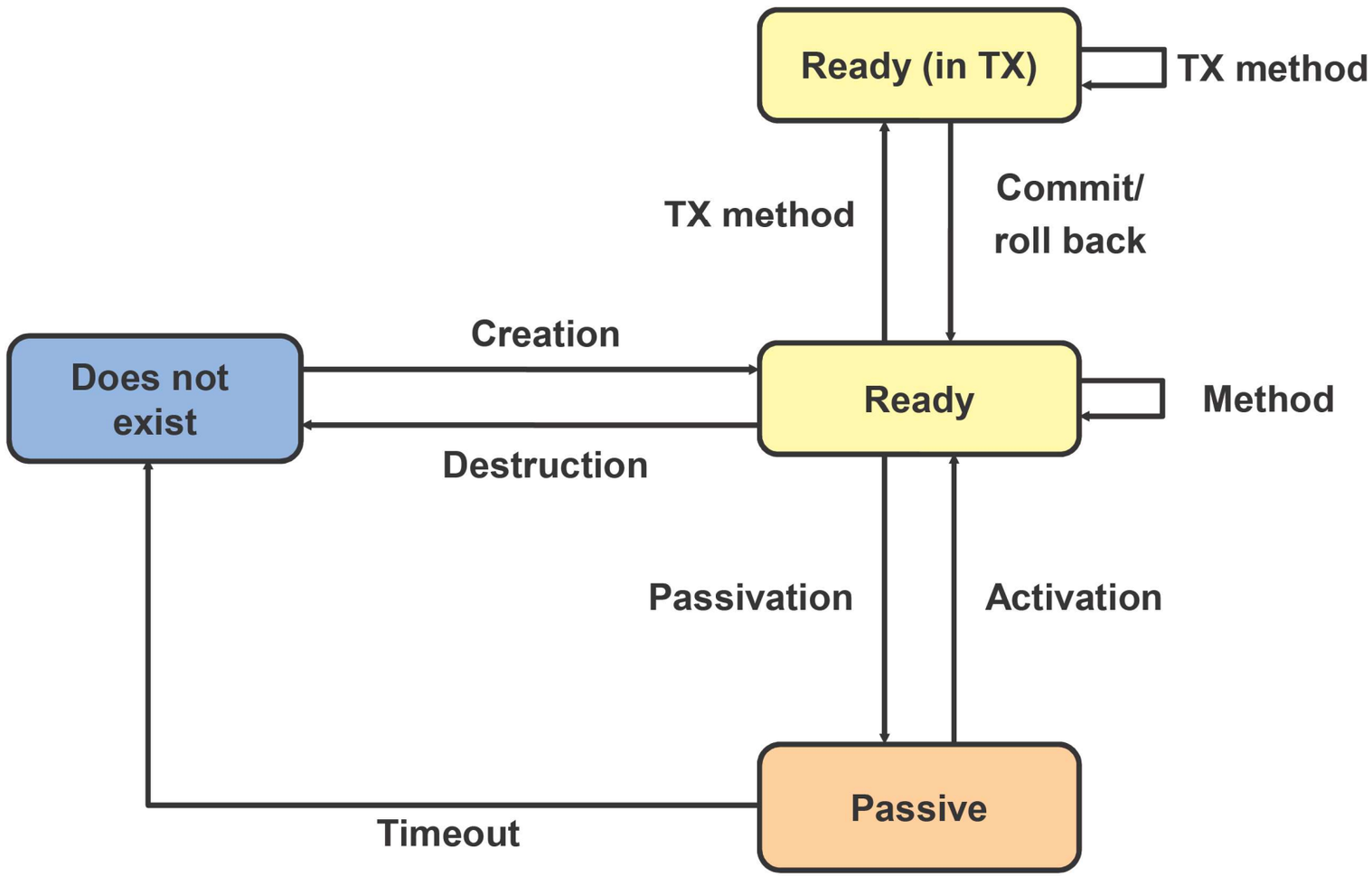
```
// HelloWorldClient.java
import helloworld.ejb;
import javax.naming.Context;
import javax.naming.InitialContext;
import javax.naming.NamingException;

public class HelloWorldClient {
  public static void main(String [] args)
    throws NamingException {
    try {
      final Context context = new InitialContext();
      HelloWorld helloWorld =
        (HelloWorld)context.lookup(
          "HelloWorldSessionEJB#helloworld.ejb.HelloWorld");
      System.out.println(helloWorld.sayHello());
    } catch (Exception ex) { ex.printStackTrace(); }
  };
}
```

1

2

3

Life Cycle of a Stateful Session Bean



Passivation and Activation Concepts

Passivation and activation are stages in a session bean's life cycle controlled by the EJB container.

- Passivation:
  - Serializes the bean state to secondary storage
  - Removes the instance from memory
- Activation:
  - Restores the serialized bean's state from secondary storage
  - Creates a new bean instance or uses a bean from the pool (initialized with the restored state)

Creating a Stateful Session Bean

To create a stateful session bean, perform the following steps:

1. Define the `@Stateful` class-level annotation in any standard Java class.
2. Implement the business methods defined in the business interfaces.

Defining the Stateful Session Bean

```
// CartBean.java
package cart.ejb;
import javax.ejb.Stateful;
…
@Stateful(name="Cart", mappedName="CartSessionEJB")
public class CartBean implements Cart {
  private ArrayList items;

  @PostConstruct
  public void initialize() { items = new ArrayList(); }
  public void addItem(String item) { items.add(item); }
  public void removeItem(String item) {
    items.remove(item); }
  public Collection getItems() { return items; }
  @PreDestroy
  public void cleanup() {System.out.println("CLEAR!");}
  @Remove
  public void dumpCart() {System.out.println("BYE!");}
}
```

1
2
3
4
5

Analyzing the Remote and Local Interfaces

```
// Cart.java
package cart.ejb;
import javax.ejb.Remote;                    1
…
@Remote                                     2
public interface Cart {
    public void addItem(String item);       3
    public void removeItem(String item);
    public Collection getItems();
}
```

```
// CartLocal.java
package cart.ejb
import javax.ejb.Local;                     1
…
@Local                                      2
public interface CartLocal { …
```

Creating a Test Client for the SFSB

```
// CartClient.java
import …
public class CartClient {
  public static void main(String[] args) throws
   Exception {
    Context context = new InitialContext();
    Cart cart = (Cart)
        context.lookup("CartSessionEJB#cart.ejb.Cart");
    cart.addItem("Item1");
    cart.addItem("Item2");
    Collection items = cart.getItems();
    for (Iterator i = items.iterator(); i.hasNext();) {
      String item = (String) i.next();
      System.out.println("  " + item);
    }
  }
}
```

Calling a Stateless Bean from a Stateful Bean by Implementing DI

```
…

@Stateful(name="Cart"", mappedName="CartSessionEJB")
public class CartBean implements Cart {

@EJB private HelloWorld obj;
…
// A demo method to call the ‘sayHello()’ method defined
// in ‘HelloWorld’ stateless session bean

  public String callEJB() {
    return obj.sayHello();
  }
…

}
```

Interceptor Methods and Classes

EJB 3.0 introduces the ability to create custom interceptor methods and classes that are called before invoking the methods they intercept. Interceptors:

- Are available only for session beans (stateless and stateful) and message-driven beans
- Provide more granular control of a bean's method invocation flow
- Can be used to implement custom transaction or security processes instead of having those services provided by the EJB container

Interceptor Method

```
import javax.ejb.Stateless;
import javax.interceptor.AroundInvoke;
import javax.interceptor.InvocationContext;

@Stateless
public class HelloWorldBean implements HelloWorld
{
  @AroundInvoke
  public Object myInterceptor(InvocationContext ctx)
    throws Exception {
     System.out.println("Entering method: " +
                                ctx.getMethod().getName());
     return ctx.proceed();
  }

  public void sayHello()
  { System.out.println("Hello World!"); }
}
```

1 2 3 4

Interceptor Classes

External interceptor classes can be created to abstract the behavior of interceptors and to define multiple interceptors for a bean.

```
// Bean Class
@Stateless(name = ... , mappedName = ... )
@Interceptors(CheckUserInterceptor.class)
@Interceptors(LogActivity.class)
public class HelloWorldBean implements HelloWorld
{… }
```

Attaching the interceptor class in the bean class

```
// Interceptor Class
public class CheckUserInterceptor {
  @AroundInvoke
  public Object checkId(InvocationContext ctx) {…}
}
```

Defining the interceptor class and specifying the interceptor method

Timers

Purpose of timers is to execute code at some point in time or periodically.

- Timers can be applied to Stateless, Singleton, Message-Driven, and 2.1 Entity Beans.
- By default timers are stateful, so they are redelivered after server restart.
- Automatic timers can be configured with annotations or deployment descriptors.

Timer types:

- Programmatic
  - Created using TimerService object
  - When such timer expires (goes single method annotated @Ti implementation class.
- Automatic
  - Created with multiple @Schedule or @Schedules

Timeout methods:

- Must not return values (be declared with void return type)
- Optionally take Timer object as the only parameter

```
@Resource
private TimerService timerService;
public void someMethod() {
timerService.createTimer(...);
}
@Timeout
```

```
@Schedule(dayOfWeek="Sun", hour="0")
public void doThings() {...}

@Schedule(minute="*/15", hour="9-17")
public void doMoreThings() {...}
```

Calendar-Based Timer Expressions

Timer expressions can be used within:

- @Schedule annotation
- ScheduleExpression Object
- The `ejb-jar.xml` file

Attribute	Allowed Values	Default	Examples
second	0 to 59	0	second="30"
minute	0 to 59	0	minute="15"
hour	0 to 23	0	hour="13"
dayOfWeek	0 to 7 (both 0 and 7 refer to Sunday) Sun, Mon, Tue, Wed, Thu, Fri, Sat	*	dayOfWeek="3" dayOfWeek="Mon"
dayOfMonth	1 to 31 –7 to –1 (a negative number means the nth day or days before the end of the month) Last [1st, 2nd, 3rd, 4th, 5th, Last] [Sun, Mon, Tue, Wed, Thu, Fri, Sat]	*	dayOfMonth="15" dayOfMonth="–3" dayOfMonth="Last" dayOfMonth="2nd Fri"
month	1 to 12 Jan, Feb, Mar, Apr, May, Jun, Jul, Aug, Sep, Oct, Nov, Dec	*	month="7" month="July"
year	A four–digit calendar year	*	year="2010"

Define Programmatic Timers

Programmatic Timers allow developers to create Timer Objects dynamically.

- TimerService object controls creation of timers, with various methods, such as:
  - `createCalendarTimer`
  - `createIntervalTimer`
  - `createSingleActionTimer`
  - `createTimer`
- Time may be triggered as a one-off
- or periodic event, or both.
- Timers may use Schedule Expressions.
- Timeout annotation designated the method
- that timer should trigger.
- Timer persistency can be switch off:
  - TimerConfig.setPersistent(false)

```
@Stateless
public class SomeBean {
   @Resource
   private TimerService timerService;

   @PostConstruct
   public void init() {
      ScheduleExpression e = new ScheduleExpression();
      e.second("30").minute("*/10").hour("*");
      timerService.createCalendarTimer(e);
   }
   public void createSimpleTimer(Date date){
      timerService.createSingleActionTimer(date,
                                            new TimerConfig());
   }
   @Timeout
   public void someMethod(Timer timer) {...}
}
```

Define Automatic Timers

Automatic Timers allow developers to create a number of Schedules to trigger different operations upon timer expiration.

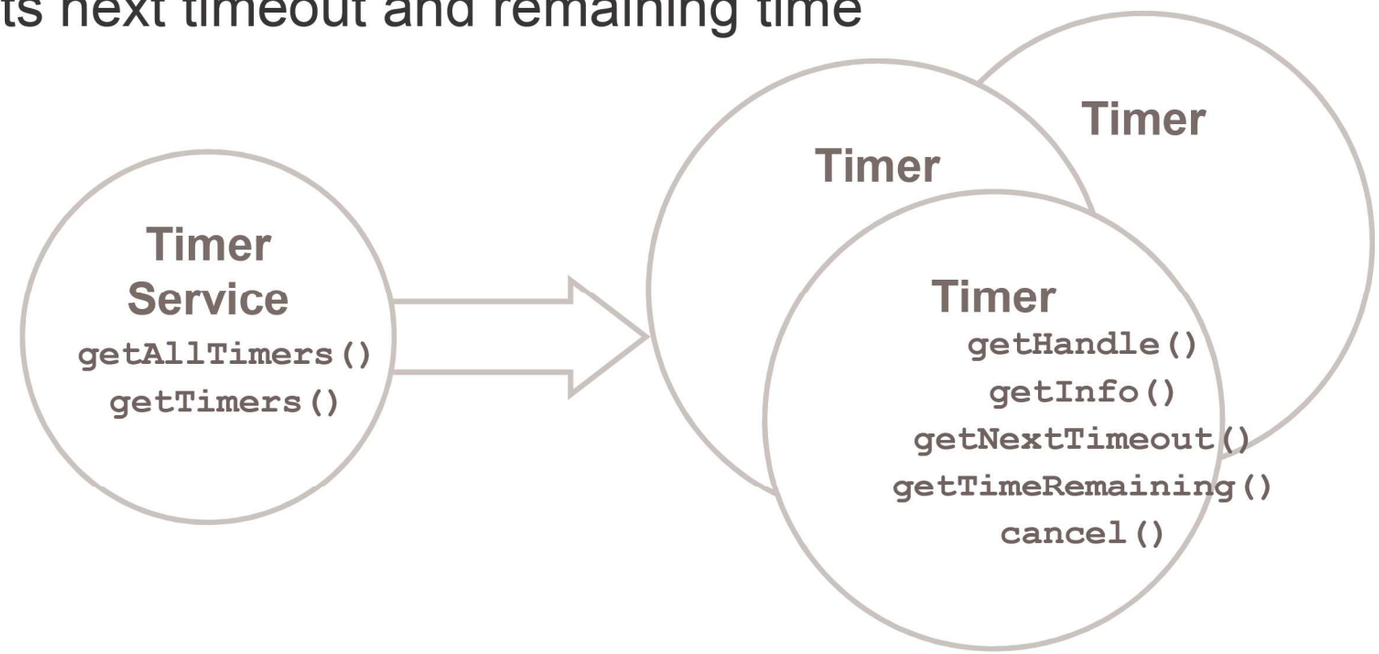
- EJB Container creates automatic timers upon successful deployment of an enterprise bean that contains one or more method that is annotated with the java.ejb.Schedule or java.ejb.Schedules annotations, or describes automatic timers in the `ejb-jar.xml` file.
- An info element can contain any serializable information that you want to associate with this timer.
- Timer persistency can be switched off using the `persistent=false` attribute.

```
@Singleton
public class SomeBean {
   @Schedules({
      @Schedule(hour="*", minute="*", second="*/10", persistent=false),
      @Schedule(hour="*", minute="*", second="*/45")
   })
   public void doThings() {...}

   @Schedule(minute="*/15", hour="9-17", info="Something")
   public void doMoreThings() {...}
}
```

Manage Timers

TimerService object can be used to retrieve active Timer objects. Each Timer object can be used to:

- Obtain its handler
- Obtain an info object associated with this timer
- Obtain information about its next timeout and remaining time
- Cancel Timer



Practice: Overview

- These practices covers the following topics:
  - Creating a stateless session bean and implementing a business method
  - Creating a test client to invoke the stateless session bean
  - Creating a stateful session bean that calls the stateless session bean by implementing dependency injection
  - Creating a test client to invoke the stateful session bean
  - Creating an interceptor class that contains a method to calculate the time to execute a bean's method

Summary

In this lesson, you should have learned how to:

- Describe session beans
- Create stateless and stateful session beans by using annotations
- Describe the passivation and activation of stateful session beans
- Use interceptor methods and classes
- Timer