Enterprise Java

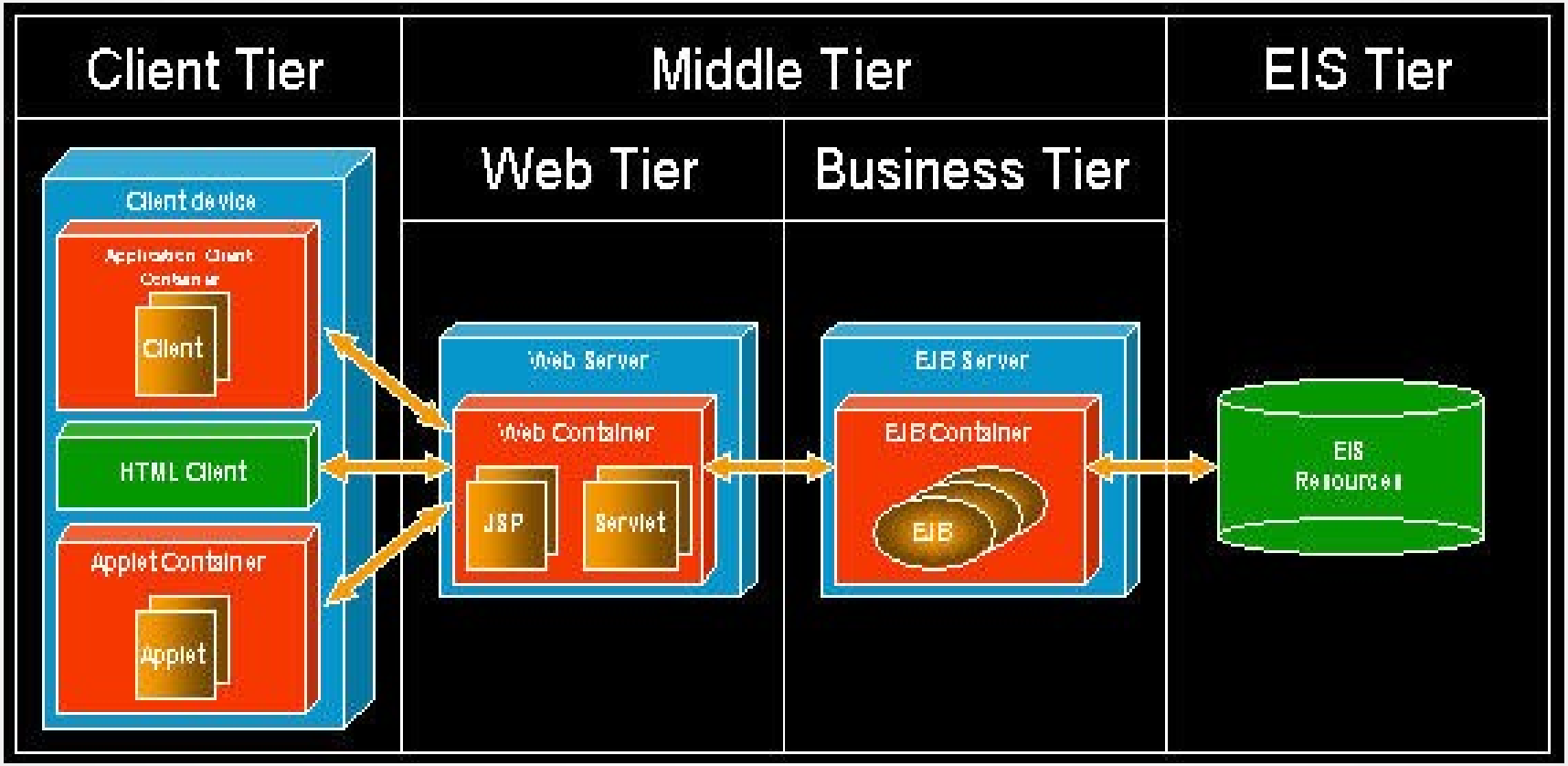
Java EE™

JAKARTA EE

Dr. Kamlendu Pandey
Dept. Of ICT

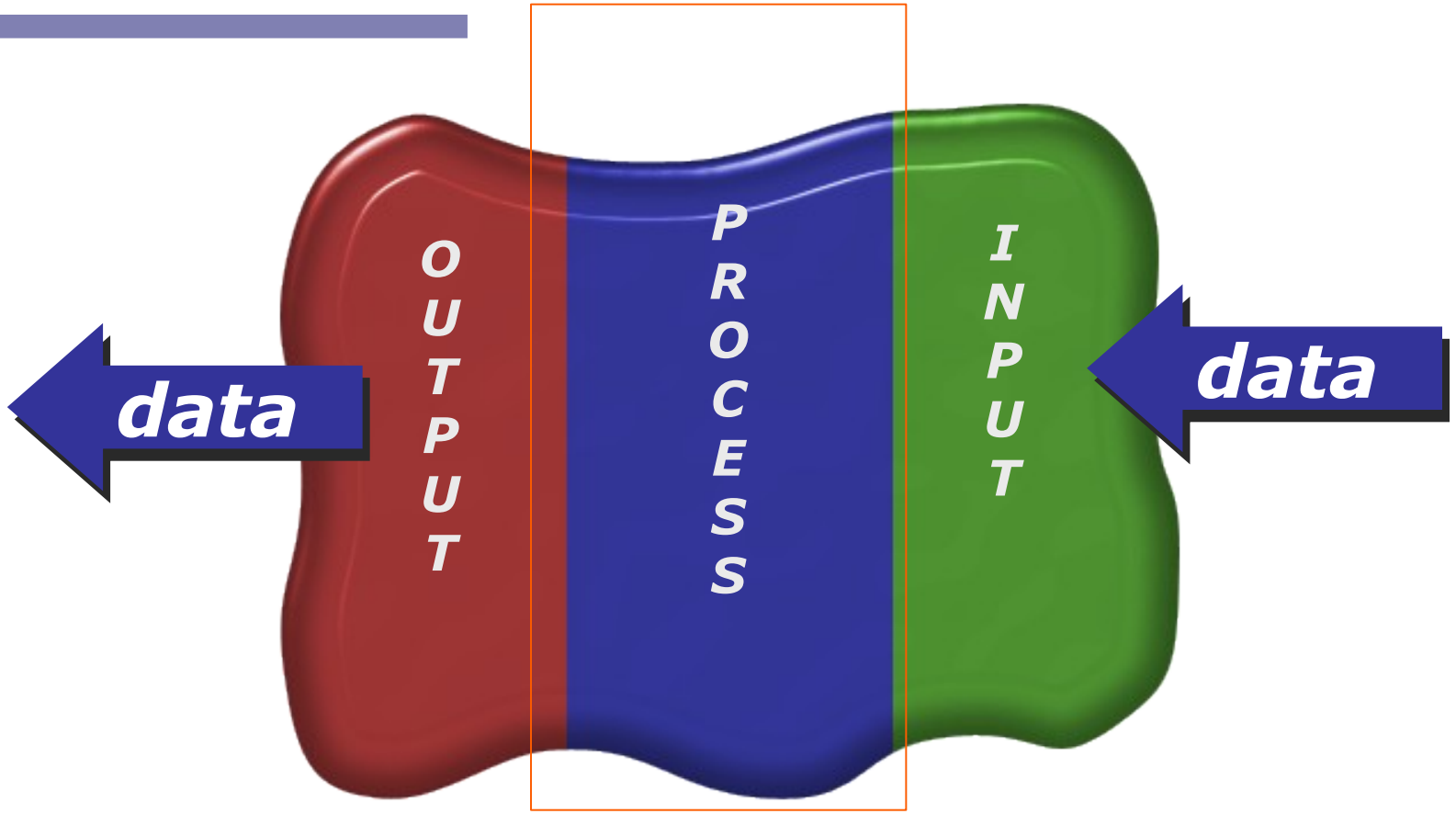
VNSGU

Java EE Layered Architecture



Three Tier Application

Business Logic



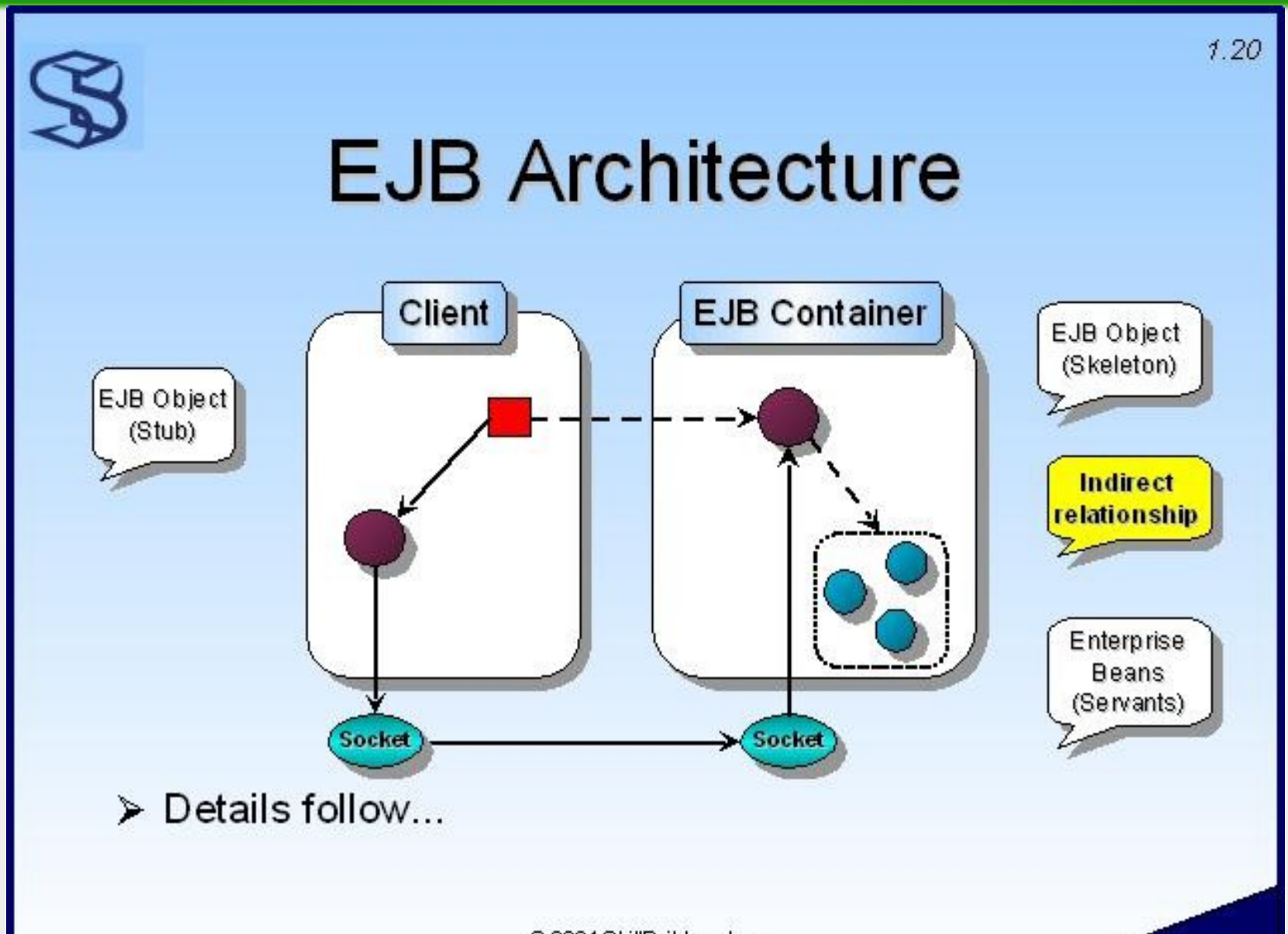
Enterprise Java Beans

What is EJB?

- An EJB is a specialized, non-visual JavaBean that runs on a server.

- EJB technology supports application development based on a multitier, distributed object architecture in which most of application's logic is moved from the client to the server.

EJB Architecture



What EJB Accomplishes

You can take any Java class and with little effort make it a distributed, secure, transactional class

You can take any data source and make the data source appear to be a collection of Java objects

-> Eliminates distinction between data from a database and any other source

-> All information is accessed through Java objects

->All SQL is cleanly encapsulated in Java objects

->true object-oriented programming

->high reusability

What EJB Means to Us

- Developers can focus on writing business logic rather than writing low-level infrastructure like data access, concurrency, transactions, threading, etc.
  - Reduces development time
  - Reduces complexity
  - Increases quality and reliability
- The knowledge about EJB is portable among many different products because EJB products are based on a common standard
- Greater reuse because code is located in shareable, server objects

So....let's take a look at Enterprise JavaBeans

- A specification from JavaSoft
- *Enterprise JavaBeans defines a server component model for the development and deployment of Java applications based on a multi-tier, distributed object architecture*
- The Enterprise JavaBeans specification defines:
  - A container model
  - A definition of the services the container needs to provide to an Enterprise JavaBean, and vice versa
  - How a container should manage Enterprise JavaBeans

Java Application Servers

- A Java application server provides an optimized execution environment for server-side Java application components.
- A Java application server delivers a high-performance, highly scalable, robust execution environment specifically suited to support Internet enabled application systems.

Application Server

Provides a Runtime Environment

- The EJB Server provides system services and manages resources
  - Process and thread management
  - System resources management
  - Database connection pooling and caching
  - Management API

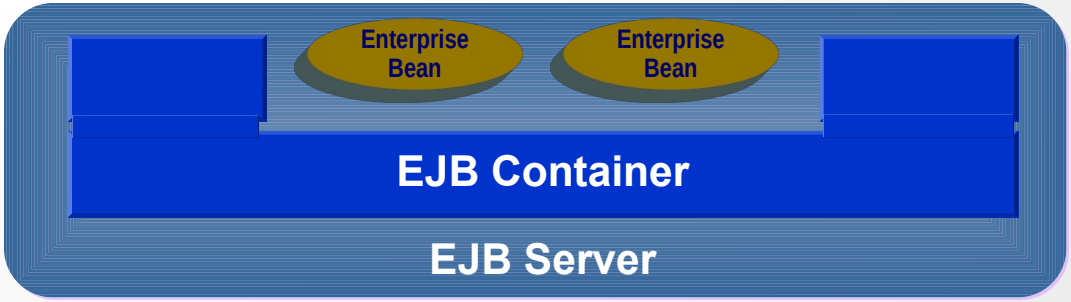
EJB Server and Container

Provides a Run-time Environment for an Enterprise Bean

- Hosts the Enterprise JavaBeans
- Provides services to Enterprise JavaBeans
  - Naming
  - Life cycle management
  - Persistence (state management)
  - Transaction Management
  - Security
- Likely provided by server vendor

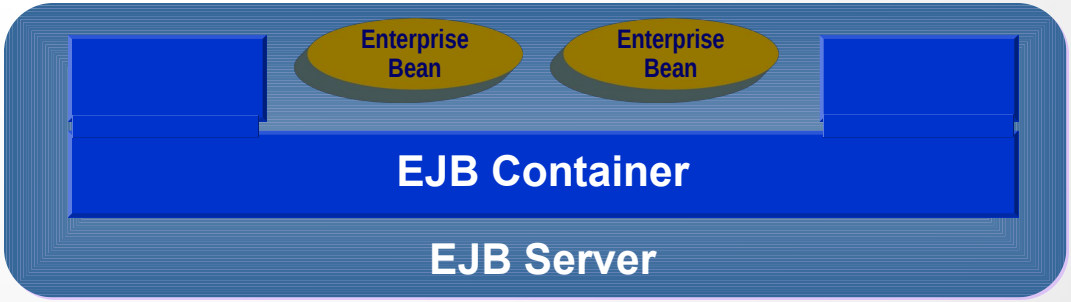
Enterprise JavaBeans

- A specialized Java class where the real business logic lives
  - May be developer-written or tool-generated
- Distributed over a network
- Transactional
- Secure
- Server vendors provide tools that automatically generate distribution, transaction and security behavior



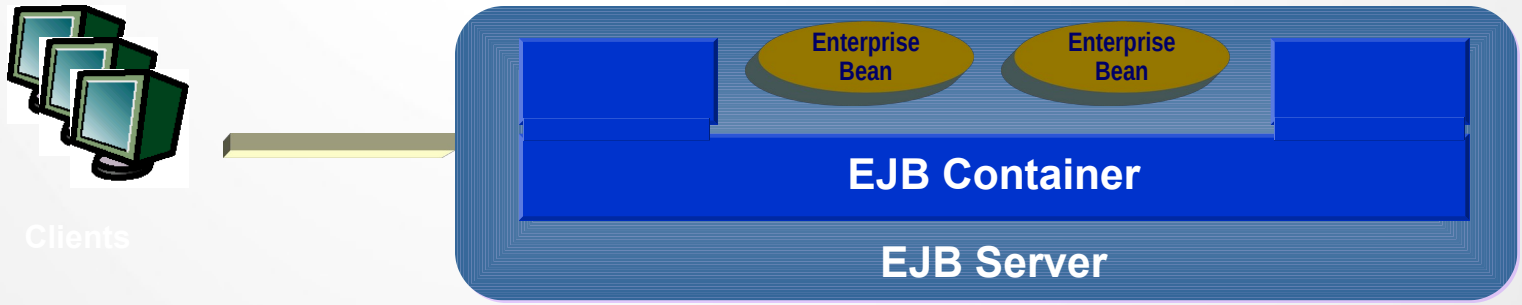
Enterprise JavaBeans

- A specialized Java class where the real business logic lives
  - May be developer-written or tool-generated
- Distributed over a network
- Transactional
- Secure
- Server vendors provide tools that automatically generate distribution, transaction and security behavior



EJB Clients

- Client access is controlled by the container in which the enterprise Bean is deployed
- Clients locates an Enterprise JavaBean through Java Naming and Directory Interface (JNDI)
- RMI is the standard method for accessing a bean over a network



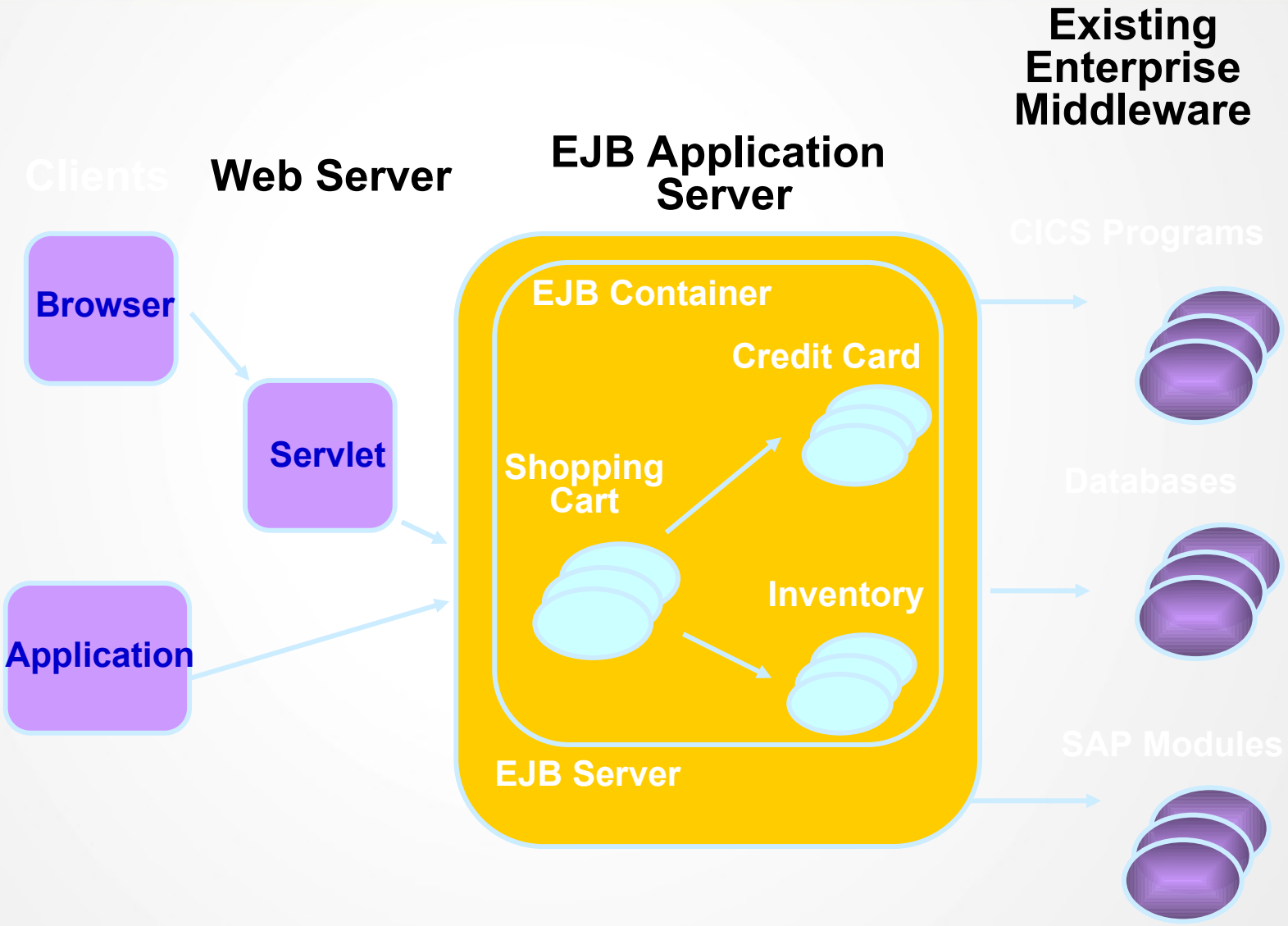
What's Unique About EJB

Declarative Programming Model

- Mandates a container model where common services are declared, not programmed
  - At development and/or deployment time, attributes defining the bean's transaction and security characteristics are specified
  - At deployment time, the container introspects the Enterprise JavaBean attributes for the runtime services it requires and wraps the bean with the required functionality
  - At runtime, the container intercepts all calls to the objectProvides transactional, threading and security behavior required before the method invocationInvokes the method on the object
  - Cleans up after the call

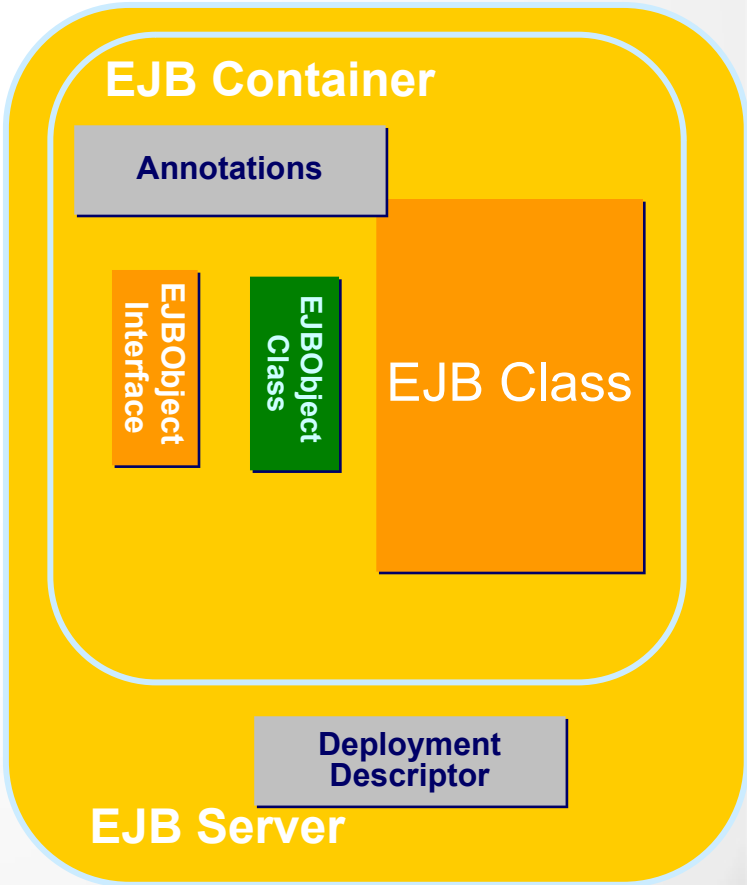
Understanding EJB Components

Enterprise EJB Scenario



An inside look at the various components of EJB

- Bean Class is written by the developer
- EJBObject interfaces or/and classes control access to the Bean class
- Annotations describe security and transactional characteristics of the Bean



Written by Developer

Generated at Development

Generated at Deployment

TYPES OF EJB

Session Beans

Singleton Beans

Messaging Beans

Asynchronous Beans

Session Beans

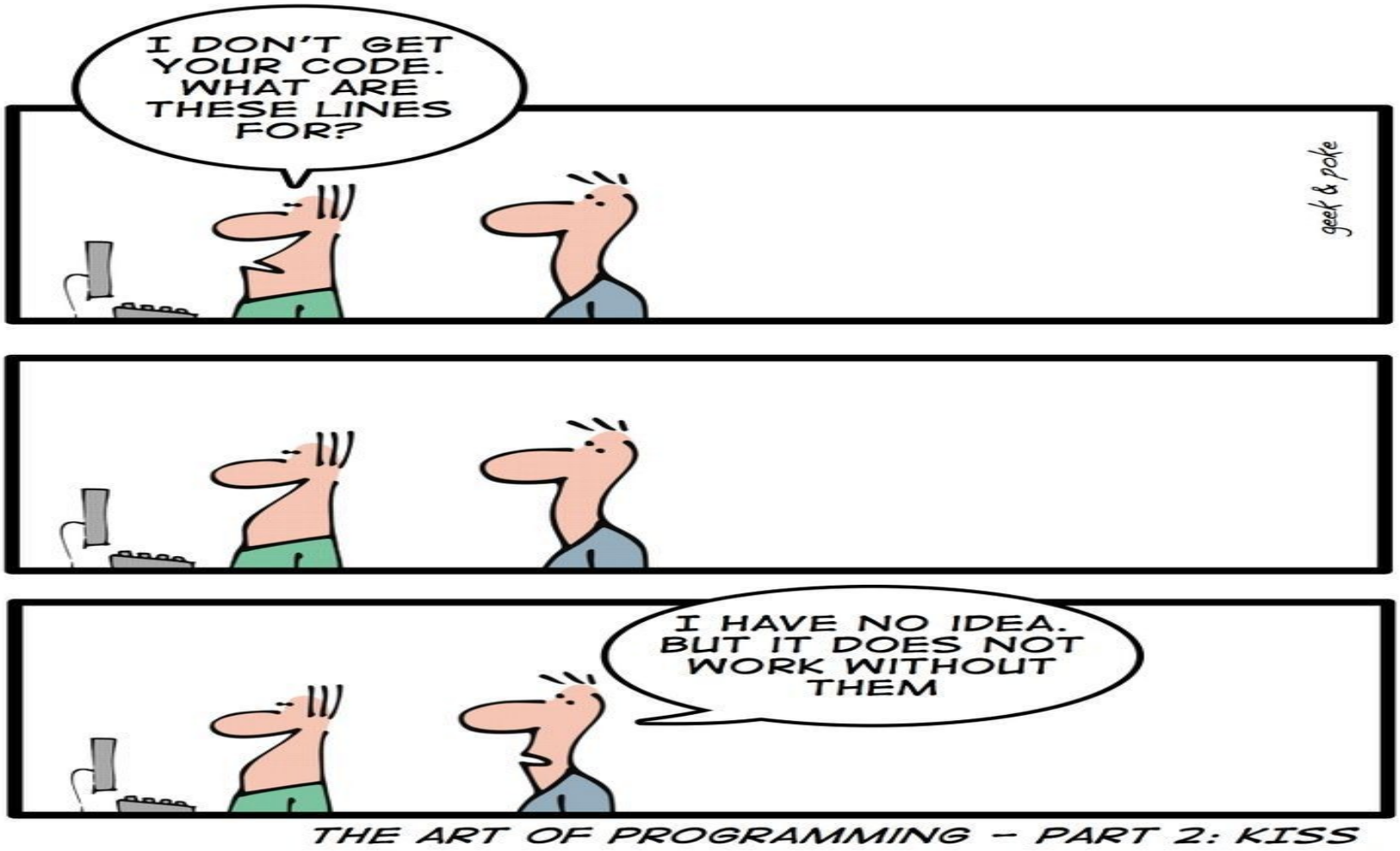
Represents *Process*

- A transient agent for an individual client that executes on a server (e.g. inventory, ShoppingCart, loginmodule)
- Session beans are often a client of multiple java client programs
- Implements `javax.ejb.SessionBean` interface
- State management types for session EJBs
  - stateful - session bean may maintain state information across method calls
  - stateless - session bean may be used to service multiple clients
  - a stateless session bean does not maintain state

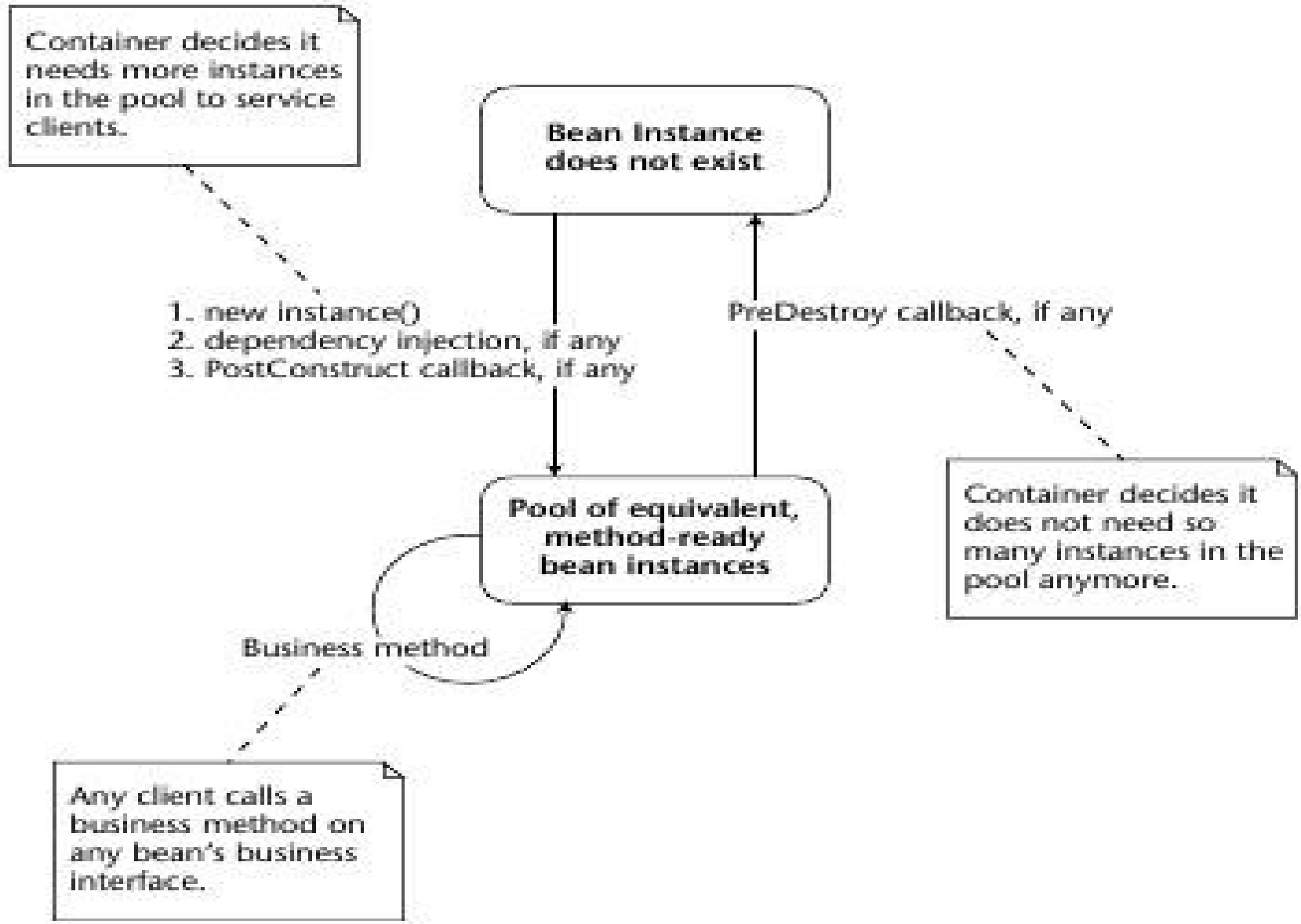
Session Beans - Stateful or Stateless

- A Stateful Session Bean maintains a one-to-one relationship with a Client. It maintains a user “session”. Most common example is an e-commerce application with a “shopping cart” unique for each user.
  - Container will automatically “swap out” the Session bean if it is inactive. Here the container calls the ejbPassivate() method to save any private data to some physical storage.
  - When container receives new request, the Container will call the ejbActivate() method to restore the Session Bean.
- A Stateless Session Bean can be accessed by multiple incoming clients and keeps no private data. It does not maintain a unique session with a client.
  - Keeps no persistent data. If it crashes, container simply starts another one and the client transparently connects.
  - All access to the Bean is serialized.

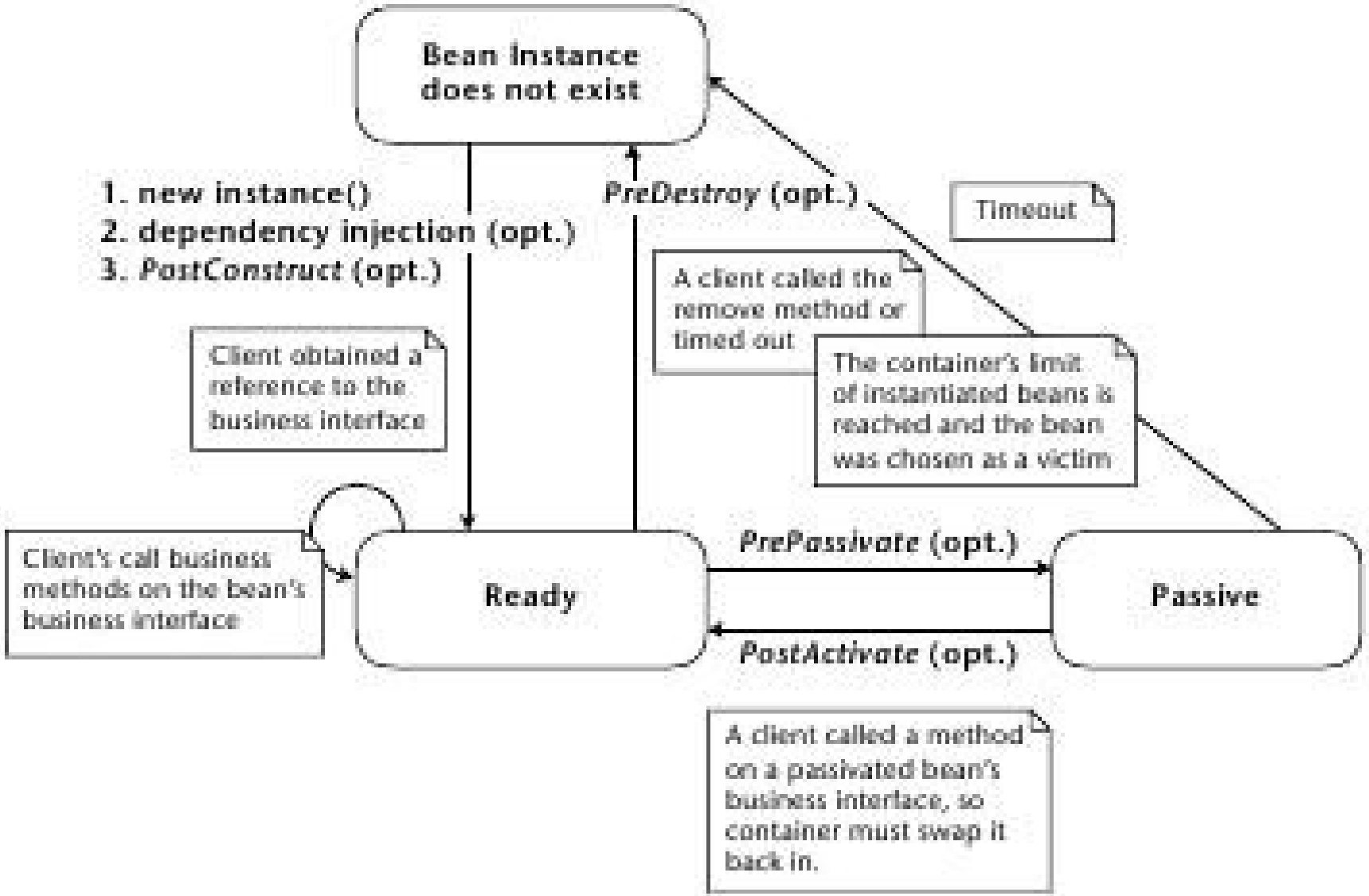
Laughter is the best medicine



Lifecycle of Stateless Session Bean



Life Cycle of Stateful Bean



EJB Callback Methods

- These annotations are used to tag any methods to the bean class
- @PostConstruct
- @PreDestroy
- @PrePassivate
- @PostActivate
- @Init
- @Remove

EJB 3.1 – The new standard

- EJB 3.1 focuses on faster productivity and manageability of the server side components and client side applications
- Much better resource management, transaction management, security checking, access control, memory management, connection pooling etc

Some useful terms in EJB 3.0

POJO

POJO stands for Plain Old Java Objects, which are defined by the user and can be tested even outside the container. All Session beans and Entity beans are also POJOs

POJI

POJI stands for plain old Java Interfaces used to develop an API against interfaces instead of implementations. POJI is an interface that user defines

- Annotations
  - An annotation is a piece of text with which you can adorn your code. Annotation let you express the information about the code in code. Annotations are compiled and type-checked. They can not perform any computation They start with @ sign eg.

CONTEXT and DEPENDENCY INJECTIONS (CDI)

Dependency injection is a mechanism followed by the container to inject the requested environmental entry and make it available to the bean instance before any business methods are invoked on that particular instance. The container injects these entries into the bean's variables or methods. The bean provider has to convey to the container where these dependencies should be injected at the runtime, such as in which variables or methods. The provider can use a deployment descriptor and annotations to specify this. Bean methods that are targets of dependency injection should be defined using the JavaBeans naming convention for properties in that they should follow the setXXX() convention.

Accebility of Session Beans

Local Beans

Remote Beans

Accebility of Session Beans

Local Accessibility of EJB

These Beans use a local bussiness interface which is annotated with @Local. Such beans can be accessed locally by Enterprise application by being a part of the EJB Client application. A Local Bean may or may not implement bussiness interface (New in EJB 3.1)

Accebility of Session Beans

Remote Accebility of EJB

These Beans use a remote bussiness interface which is annotated with @Remote. Such beans can be accessed remotely by Enterprise application without being a part of the EJB Client application. A Remote Bean shall have to implement bussiness interface

Java Naming and Directory Interfaces (JNDI)

Java Naming and Directory Interface (JNDI) provides a standard interface for locating users, machines, networks, objects and services. Naming service allows you to store various types of objects and associate or bind names to these objects. It also provides facility to search or look up an object based on a name. The objects can be a file on your disk or a computer in the network or an EJB object.

How JNDI is used in EJB

Default JNDI names are given to different beans which can be used by the clients to access them

For local access in the same application (EAR file)

java:app[/module name]/enterprise bean name[/interface name]

For local access in the same module (JAR file)

java:module/[BeanClassName/InterfaceName]

For Remote Access (EAR file)

java:global/[modulename/BeanClassName/InterfaceNam]

Dependency Injections to Access EJBs

The dependency injections can be used to access EJBs

The Local Bean can be accessed as

@EJB(mappedName=<JNDI_NAME>MyBeanLocal local;

The remote interface can be accessed as

@EJB(mappedName=corbaname:iiop:<ip_address>:<port_no>#<JNDI_NAME>)

MyBeanRemote remote;

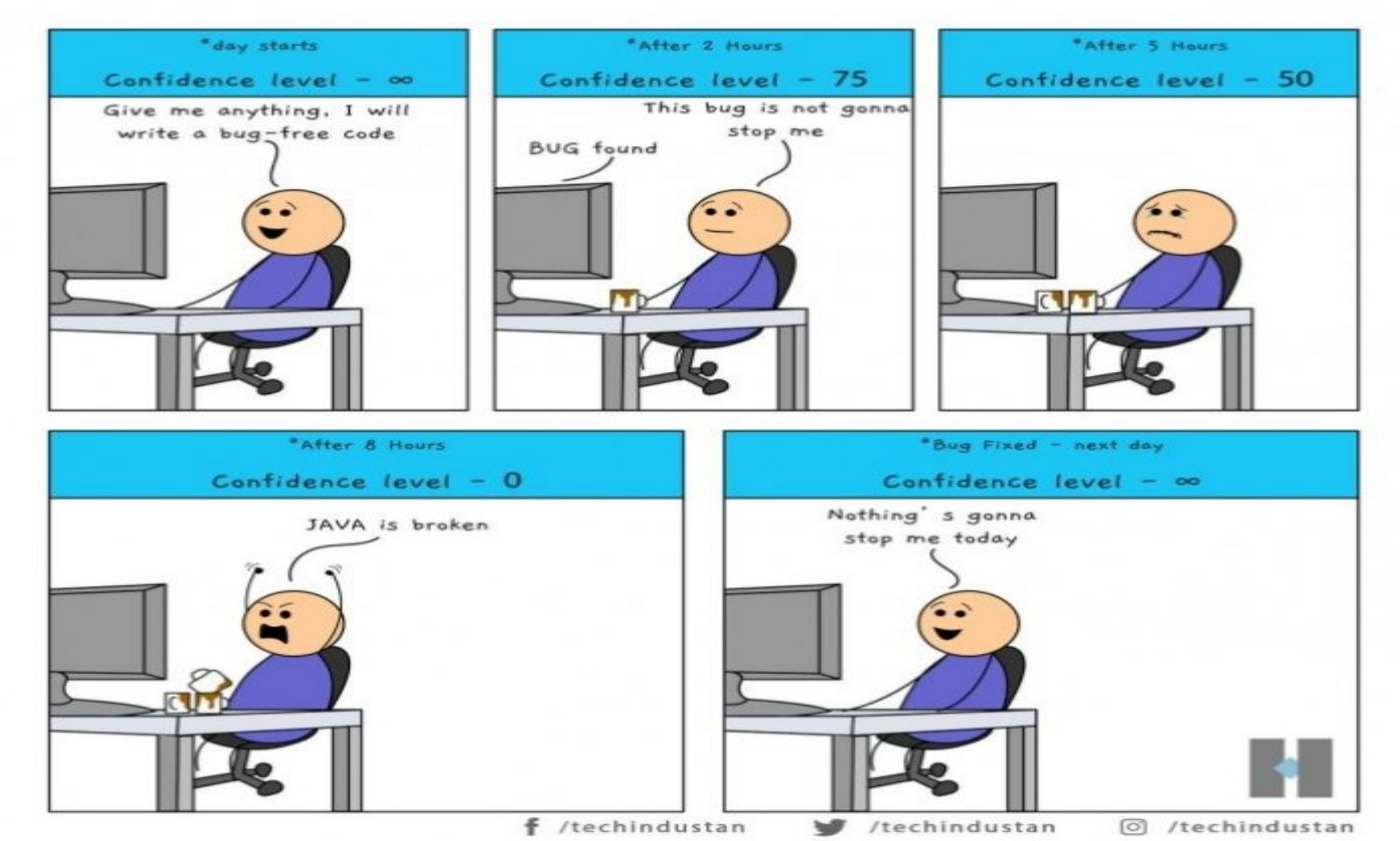
Example Code to Access EJB

Accessing EJB by Injections

```
Class MyServlet extends HttpServlet {
 @EJB(mappedName=”java:module/MyBean”)
MyBeanLocal local;
@EJB(mappedName=”java:global/MyModule/RemoteBean”)
MyBeanRemote remote;

Public void service(request....., response.....) throws IOException
{
PrintWriter out = response.getWriter();
out.println(local.count());
out.println(remote.getEmployeeName(23));
}
```

Java Joke of the Day



Message Driven Beans

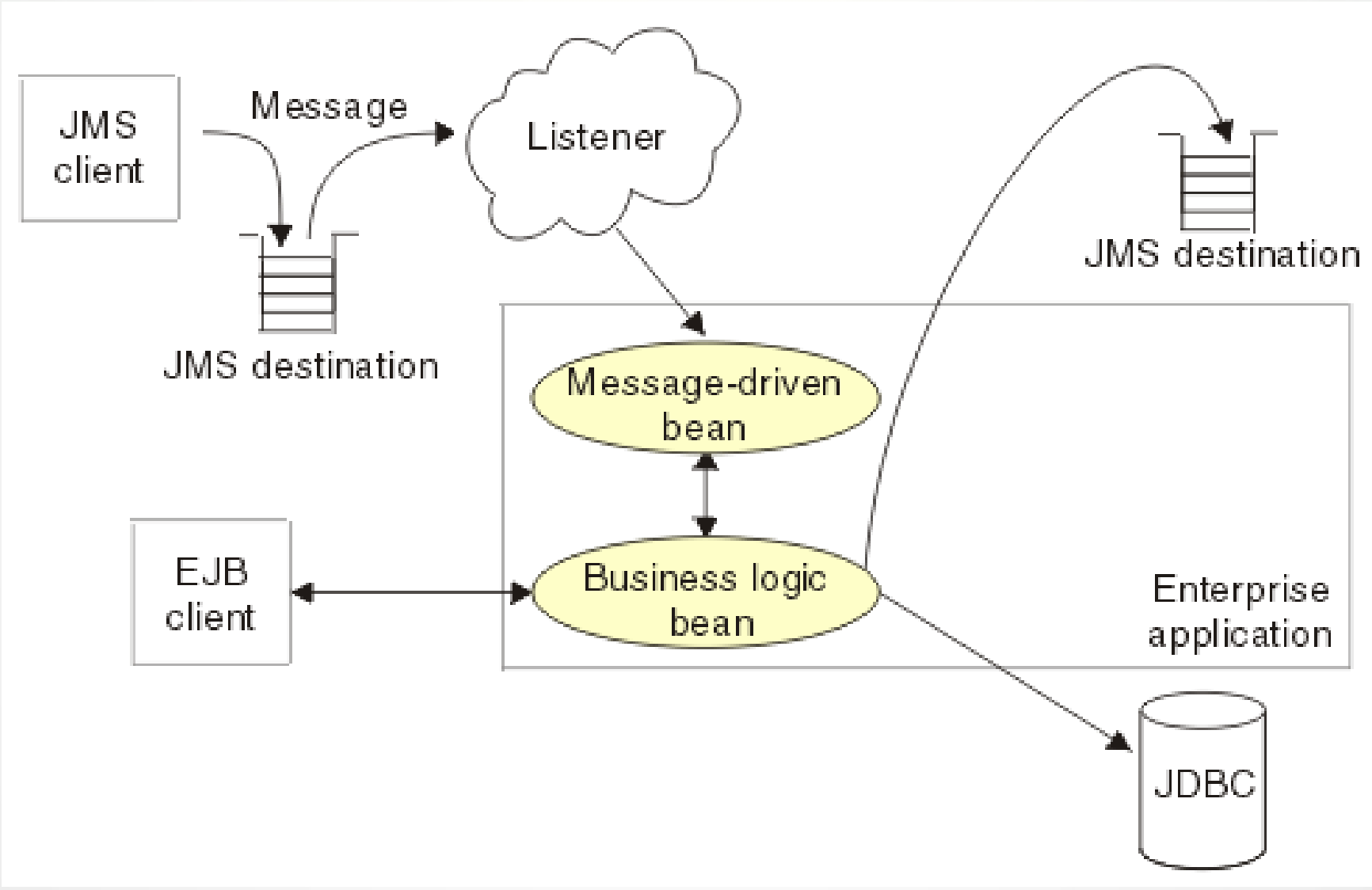
Message Driven Beans

This is a special bean which acts as a message consumer in the JMS messaging system. They receive messages from JMS Queue or Topic and perform bussiness logic based on the message contents. MDB allows Java EE applications to receive the JMS messages asynchronously. Senders and Receivers are independent of each other

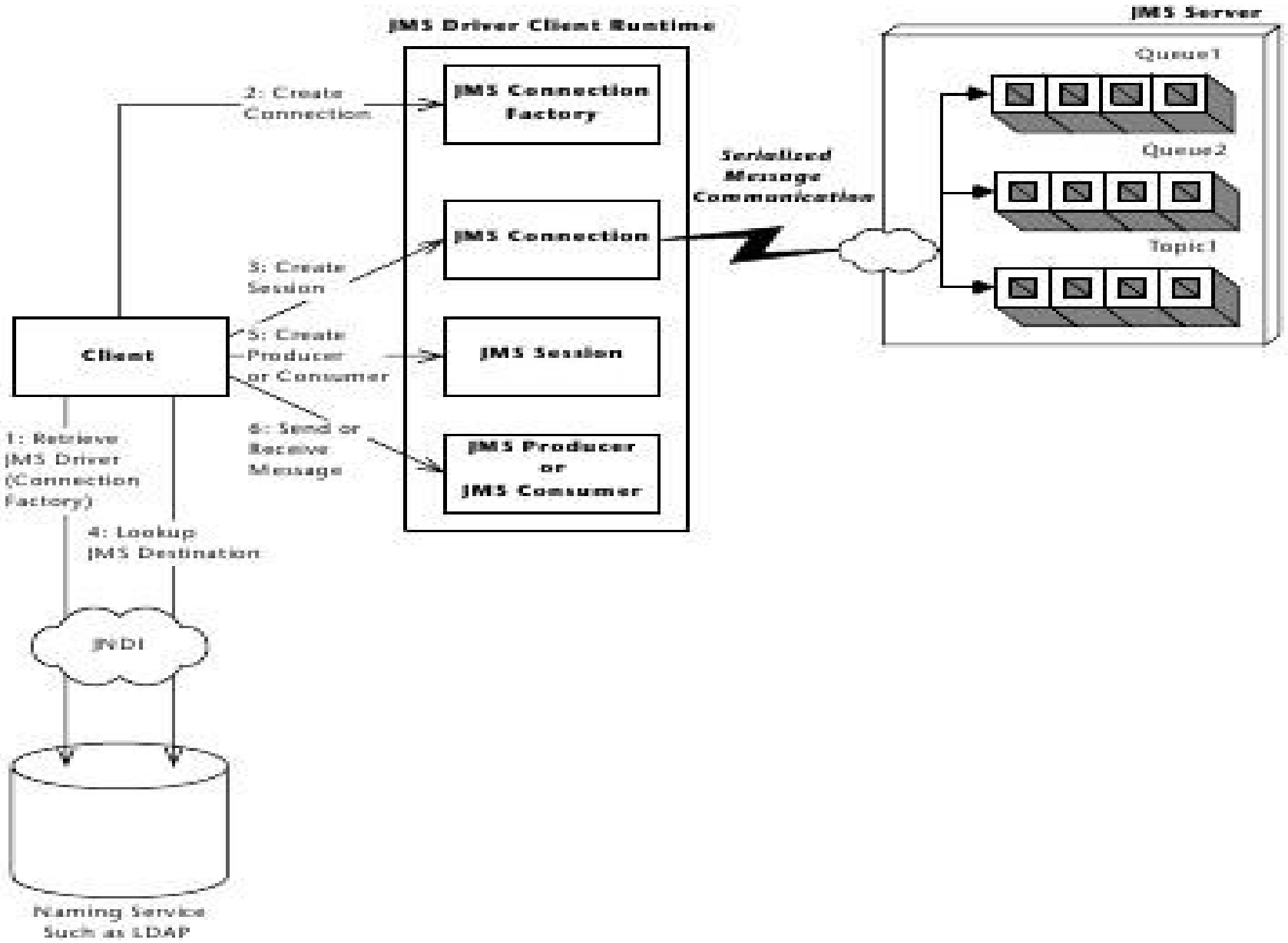
Messaging Beans

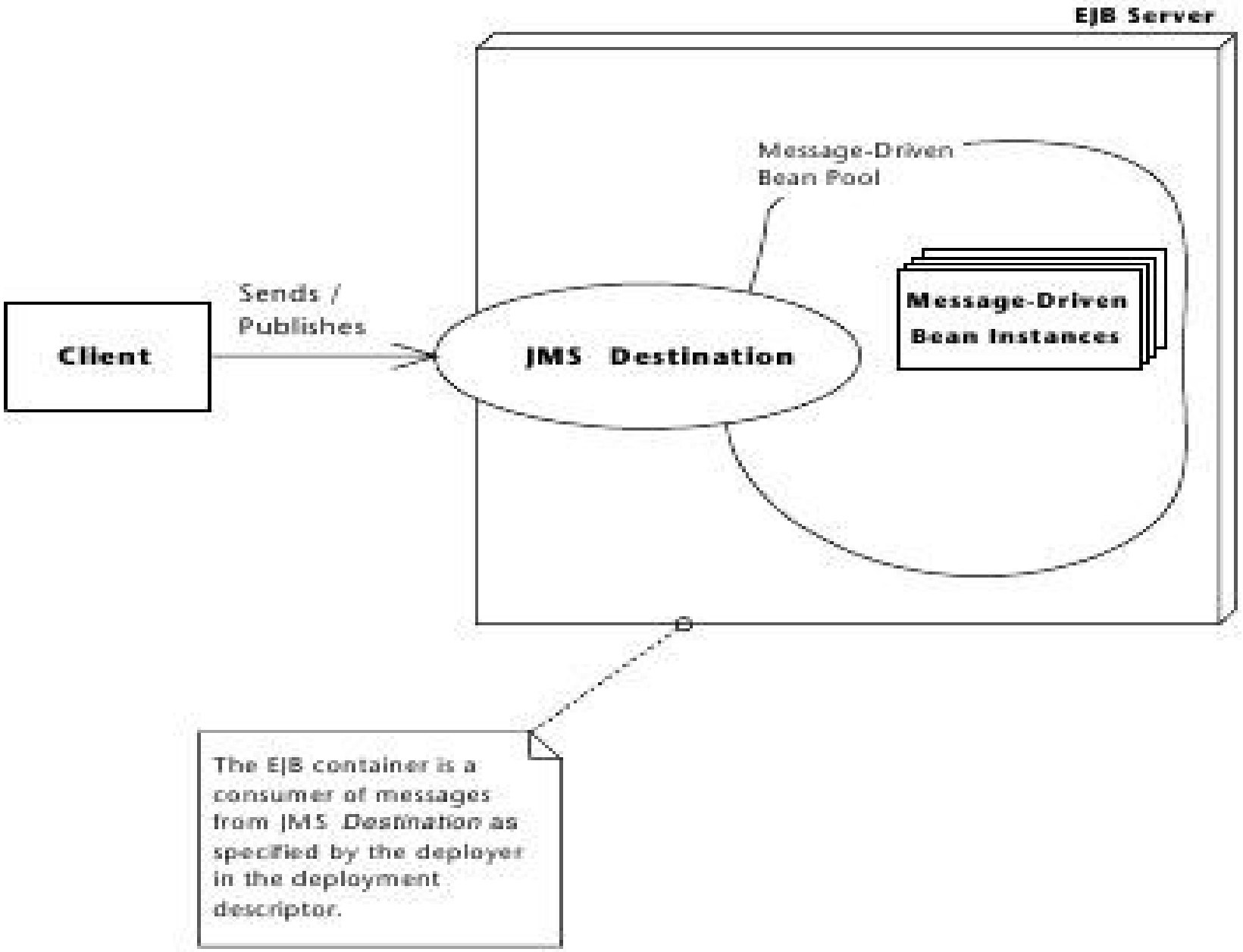
A message driven bean is a stateless, server-side, transaction-aware component that is driven by a Java message (javax.jms.message). It is invoked by the EJB Container when a message is received from a JMS Queue or Topic. It acts as a simple message listener

Message Sending Process

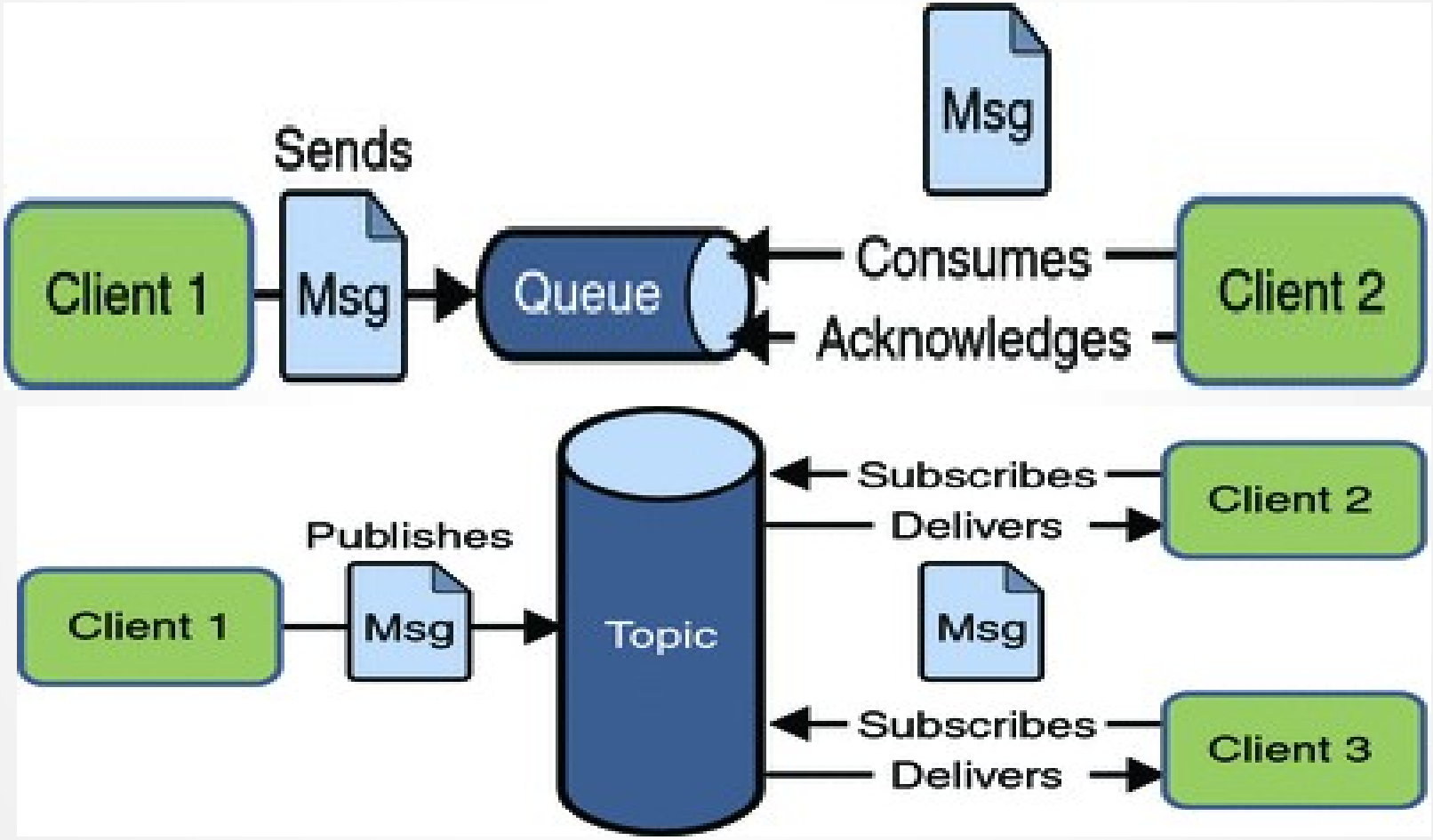


Client View of JMS

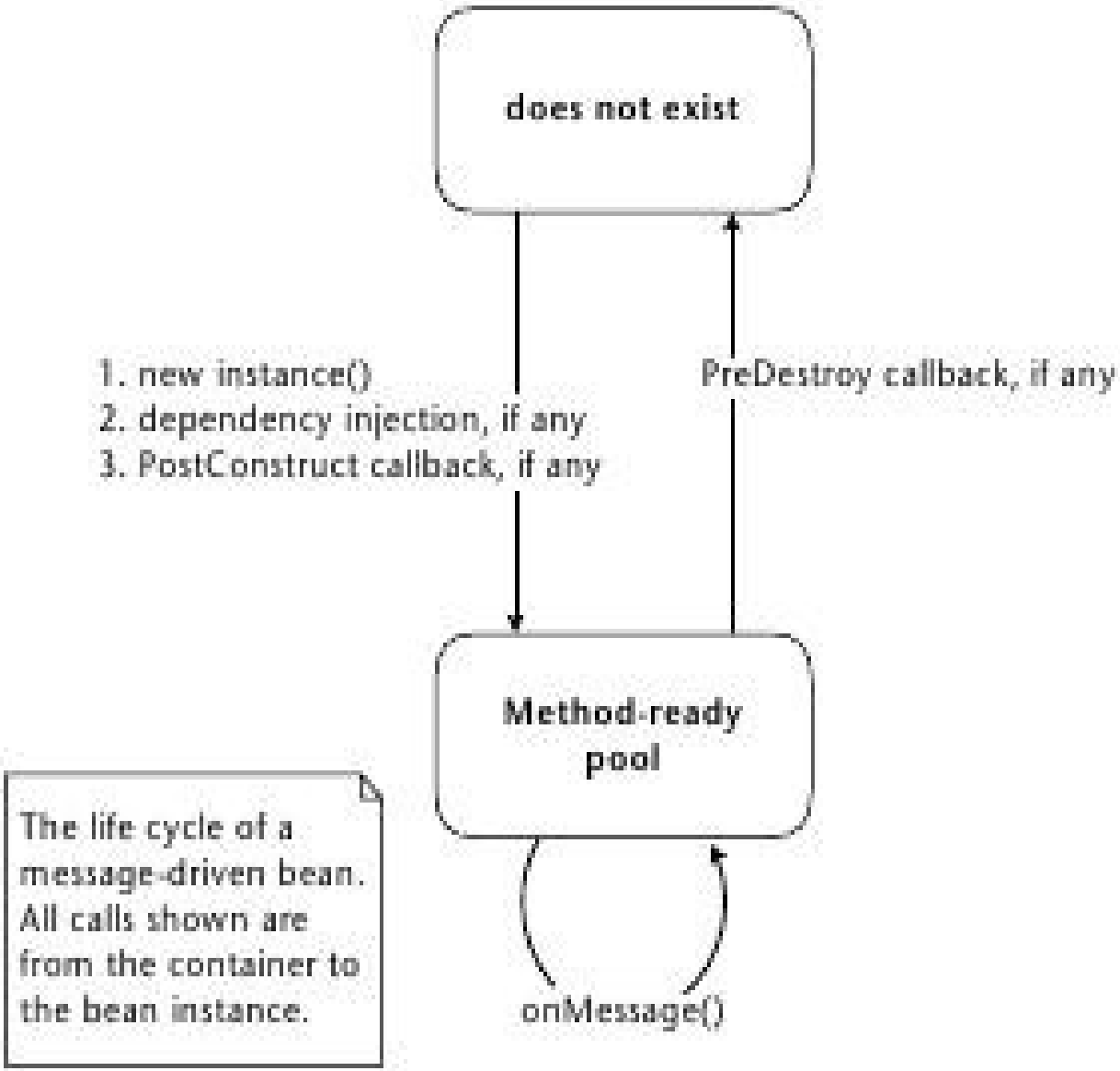



EJB Server
Message-Driven
Bean Pool
Message-Driven
Bean Instances
Sends /
Publishes
Client
JMS Destination
The EJB container is a
consumer of messages
from JMS Destination as
specified by the deployer
in the deployment
descriptor.

Queue and Topic

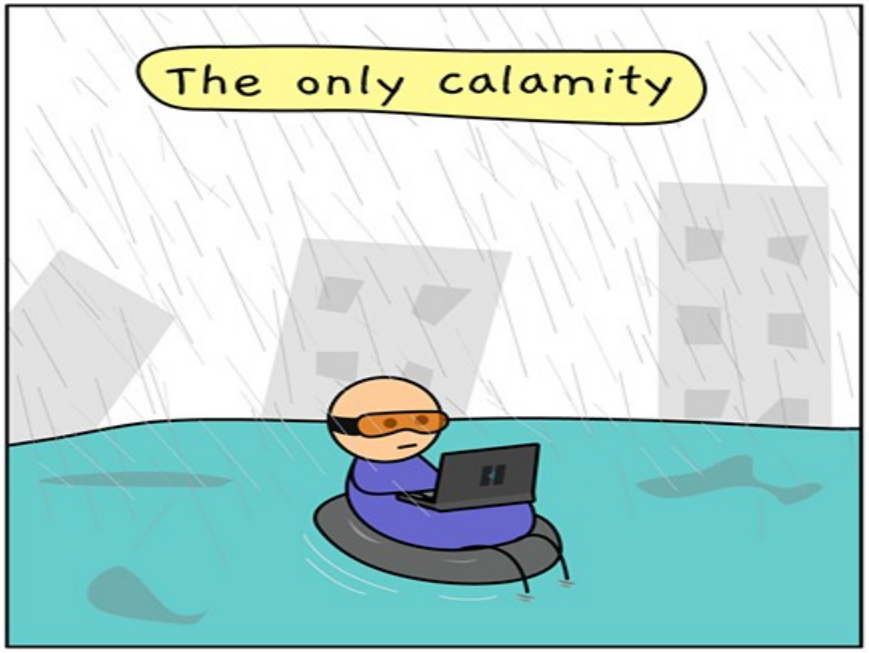


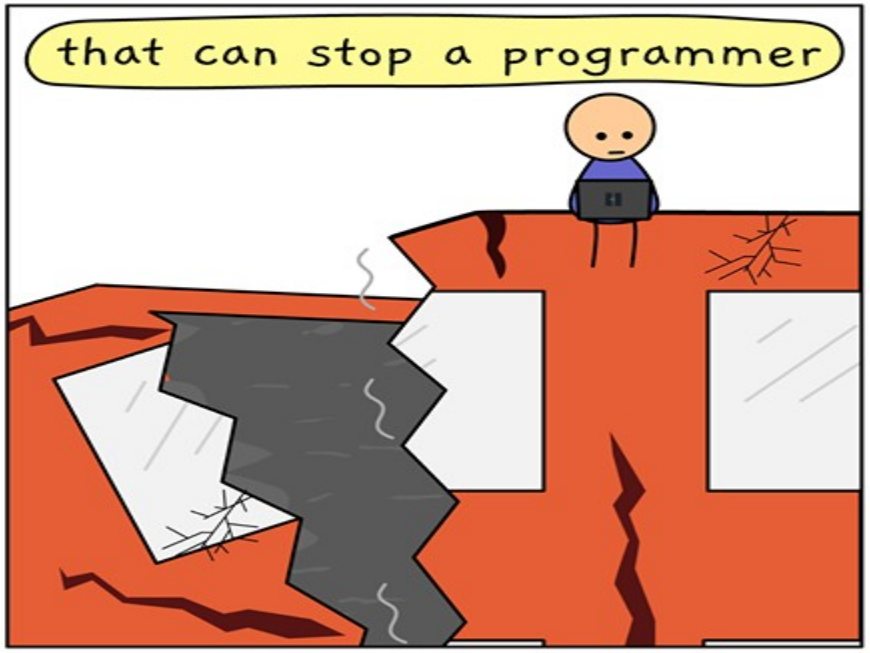
LifeCycle of Messaging Beans

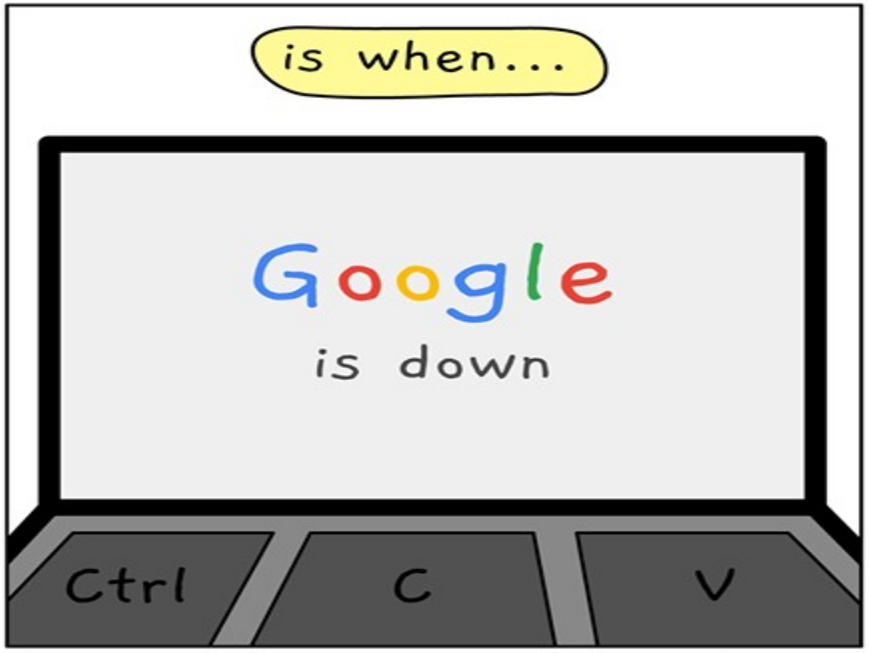


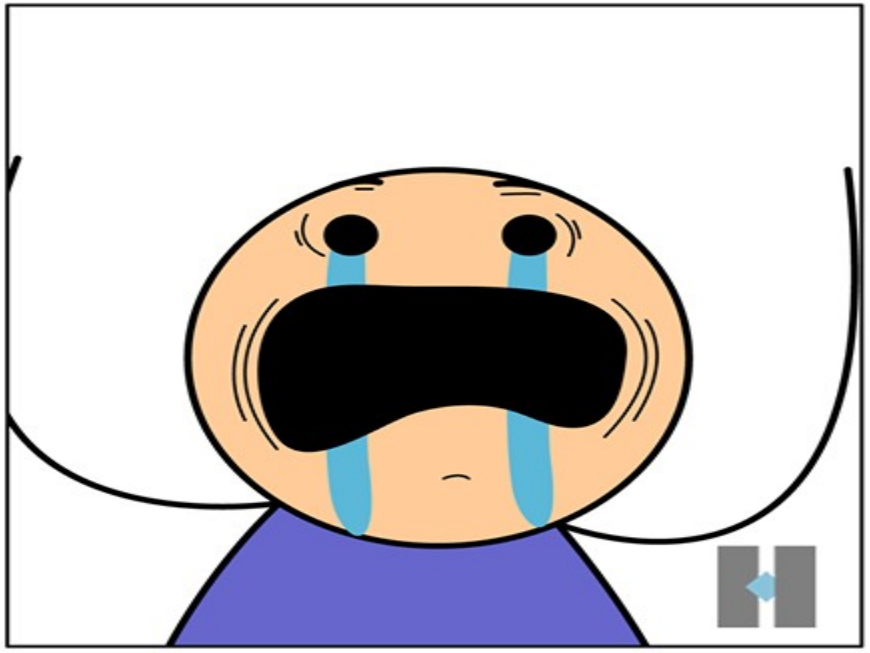
MDB in EJB 3.0

- Bean class in annotated with @MessageDriven and @ActivationConfigProperty
- Dependency Injection is used to aquire MessageDrivenContext
- @Resource javax.ejb.MessageDrivenContext mdc
- Jndi reference to the queue is given by
- @Resource (mappedname=”mdb/simpleQueue”)
- An MDB implements MessageListener Interface and overrides onMessage() method which will be invoked if some message is send by the client to corresponding JMS destination









Singleton Beans

THE EJBS which has only one instance in the project life time is called Singleton Bean,

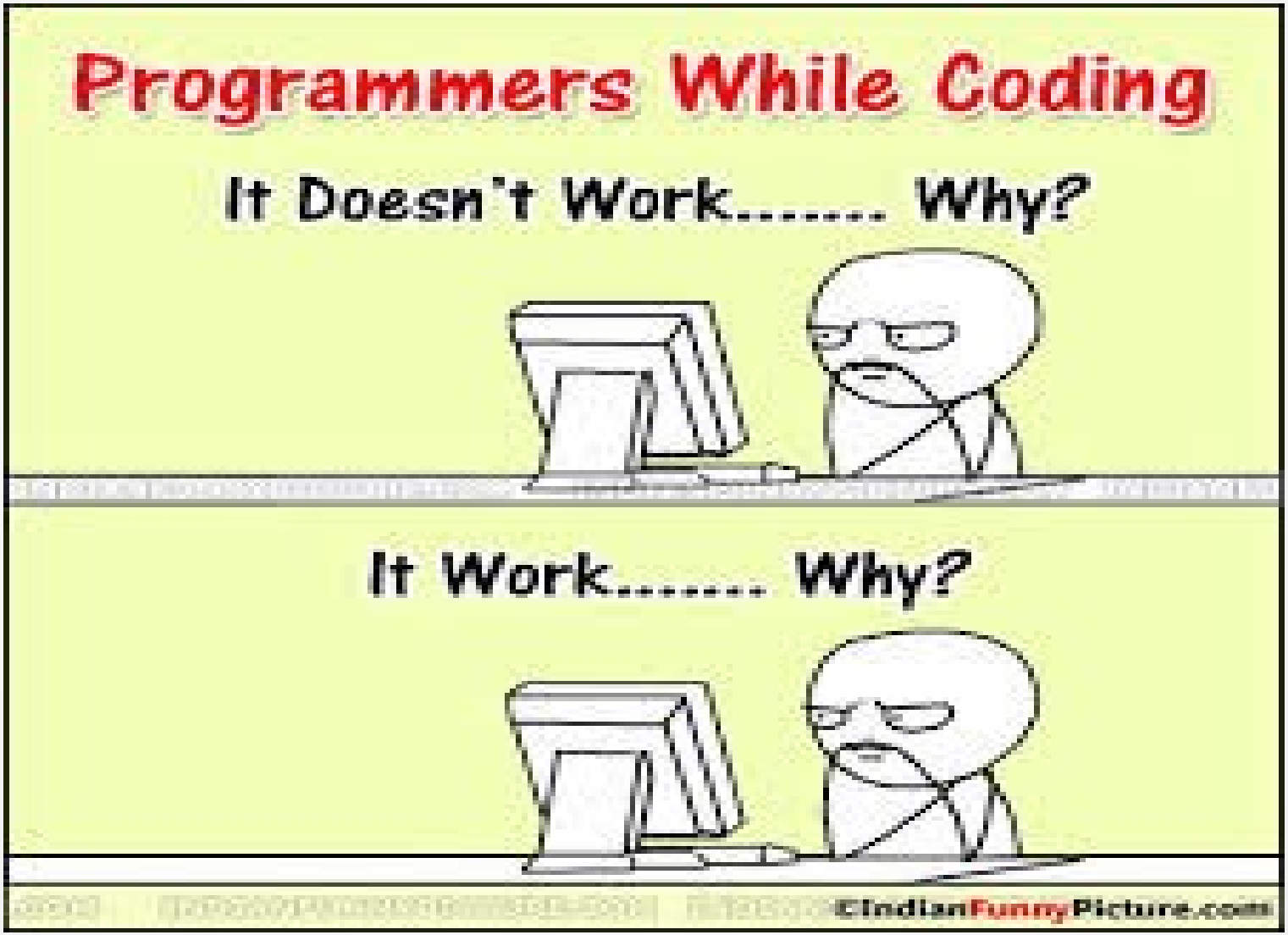
Singleton beans can be controlled programatically for concurrent access ad dependencies

Asynchronous Beans

The Enterprise java Beans which are capable of processing multiple tasks parallely are called Asynchronous Beans

EJB TIMERS

Lets Laugh a Little



EJB Timers

Applications that model business work flows often rely on timed notifications. The timer service of the enterprise bean container enables you to schedule timed notifications for all typesof enterprise beans except for stateful session beans. You can schedule a timed notification to occur according to a calendar schedule, at a specific time, after a duration of time, or at timed intervals. For example, you could set timers to go off at 10:30 AM on May 23, in 30 days, or every 12 hours.

EJB Timers Types

- PROGRAMMATIC TIMERS

Programmatic timers are set by explicitly calling one of the timer creation methods of the TimerService interface.

AUTOMATIC TIMERS

Automatic timers are created upon the successful deployment of an enterprise bean that contains a method annotated with the java.ejb.Schedule or java.ejb.Schedules annotations.

Creating Prog. Timers

// Single Instance Expiry Timer

- long duration = 6000;
- Timer timer = timerService.createSingleActionTimer(duration, new TimerConfig());

//Single Instance- Fixing Date and Time when Timer will expire

SimpleDateFormatter formatter = new SimpleDateFormatter("MM/dd/yyyy 'at' HH:mm");

- Date date = formatter.parse("05/01/2010 at 12:05");
- Timer timer = timerService.createSingleActionTimer(date, new TimerConfig());

EJB Timers Types

- PROGRAMMATIC TIMERS

Programmatic timers are set by explicitly calling one of the timer creation methods of the TimerService interface.

AUTOMATIC TIMERS

Automatic timers are created upon the successful deployment of an enterprise bean that contains a method annotated with the java.ejb.Schedule or java.ejb.Schedules annotations.

EJB Timers

Different Types of Timers:

EJB basically supports two forms of Timer objects:

* Single Action Timer
* Interval Timer

Creating Prog. Timers

// Interval Timer – will timeout after every given time interval again and again

- long interval = 6000;
- Timer timer = timerService.createTimer(interval, “Timer Created”);

//Creating Schedule Timer – will timeout on every Monday, 12 to 5.00 PM evey hour and at 11.00 PM

ScheduleExpression schedule = new ScheduleExpression();

- schedule.dayOfWeek("Mon");
- schedule.hour("12-17, 23");
- Timer timer = timerService.createCalendarTimer(schedule);

Single Action Timers

A single action timer (or a single interval timer) is a one which will expire only once (as opposed to interval timer, where multiple expirations are possible).

Accordingly EJB supports two different ways for constructing a single interval timer.

i) One is to create the timer in such a way that it will expire at a particular point of time which is specified as a Date

ii) The other possible way is make the timer to expire after certain period of time (say after 10 hours or 1 day) which is usually specified in milliseconds. After the timer expires, the enterprise bean will receive a kind of notification, i.e the container will call the ejbTimeout() method (or the method that is annotated with @Timeout annotation).

Interval Timers

Interval timer (or multiple action timer), as its name suggests will recur (or happen) at multiple intervals of time. That is, these kinds of timer will have multiple expirations at regular intervals of time.

Two different ways of constructing the Timers are available:

i) The first approach is to create a timer to have an initial expiration at some point of time (which is usually specified as a Date) and to have the subsequent expirations happen at a specified interval.

ii) The second approach is to construct the timer whose initial expiration will happen after an elapsed duration of time (in milliseconds) and to have the subsequent expirations happen after a specific interval. For every expirations of the Timer object, the container continues to call the ejbTimeout() method (or the method that is annotated with @Timeout annotation) until the bean explicitly calls the cancel() method.

Interval Timers

Interval timer (or multiple action timer), as its name suggests will recur (or happen) at multiple intervals of time. That is, these kinds of timer will have multiple expirations at regular intervals of time.

Two different ways of constructing the Timers are available:

i) The first approach is to create a timer to have an initial expiration at some point of time (which is usually specified as a Date) and to have the subsequent expirations happen at a specified interval.

ii) The second approach is to construct the timer whose initial expiration will happen after an elapsed duration of time (in milliseconds) and to have the subsequent expirations happen after a specific interval. For every expirations of the Timer object, the container continues to call the ejbTimeout() method (or the method that is annotated with @Timeout annotation) until the bean explicitly calls the cancel() method.

Creating Timers

Timer objects are usually created by one or more enterprise beans. Only stateless session beans and message-driven beans can create timer objects.In the EJB or MDB class do the following steps to use timer service

1) A dependency Injection

```
@Resource
Private SessionContext sessionCtx;  // OR
private MessageDrivenContext mdc;
```

2)Getting a Reference to TimerService:

```
TimerService timerService = sessionCtx.getTimerService();
```

3) Create Singleton Action Timer Object which will expire 10 seconds

```
Calendar now = Calendar.getInstance();
Timer timer = timerService.createTimer(now.getTimeInMillis() + (10 * 1000), null);
```

Creating Timers

ii) This Singleton Action Timer object will expire on the 1st of March 2007.

```
Calendar firstMarch2007 = new GregorianCalendar(2007, Calendar.MARCH, 1);
Timer timer = timerService.createTimer(firstMarch2007, null);
```

iii) Creating Interval Timer which will expire after one week and subsequent expirations after every 2 weeks for ever

```
long oneWeek = (7 * 24 * 60 * 60 * 1000);
Timer timer = timerService.createTimer(oneWeek, (oneWeek * 2), null);
```

How to handle Timeouts

Timed out method Call back

The method which will handle expiry of Timers every time should be annotated by @Timeout Annotation in EJB/MDB Class

```
@Timeout
public void reportExpiry(Timer timer)
{
System.out.println("The timer expired at "+ new Date());

// This method can have any logic like sending periodic emails or updating stock prices etc.
}
```

Automatic Timers

The following timeout method uses @Schedule to set a timer that will expire every Sunday atmidnight:

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


//Setting multiple expiry timers
@Schedules ({
@Schedule(dayOfMonth="Last"),
@Schedule(dayOfWeek="Fri", hour="23")
})
public void doPeriodicCleanup() { ... }
@Schedule(minute="*/3", hour="*")
public void automaticTimeout() {
```

Creating Calendar-Based Timer Expressions

TABLE 16–1 Calendar-Based Timer Attributes

Attribute	Description	Allowable 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`[1] `Sun`, `Mon`, `Tue`, `Wed`, `Thu`, `Fri`, `Sat`	`*`	`dayOfWeek="3"` `dayOfWeek="Mon"`

[1] Both `0` and `7` refer to Sunday.

Creating Calendar-Based Timer Expressions

TABLE 16–1 Calendar-Based Timer Attributes *(Continued)*

Attribute	Description	Allowable Values	Default Value	Examples
`dayOfMonth`	One or more days within a month.	1 to 31	*	`dayOfMonth="15"`
		-7 to –1[2]		`dayOfMonth="-3"`
		`Last`		`dayOfMonth="Las`
		`[1st, 2nd, 3rd, 4th, 5th, Last] [Sun, Mon, Tue, Wed, Thu, Fri, Sat]`		`dayOfMonth="2nd Fri"`
`month`	One or more months within a year.	1 to 12	*	`month="7"`
		`Jan, Feb, Mar, Apr, May, Jun, Jul, Aug, Sep, Oct, Nov, Dec`		`month="July"`
`year`	A particular calendar year.	A four-digit calendar year.	*	`year="2010"`

[2] A negative number means the xth day or days before the end of the month