

CC68J APLICACIONES EMPRESARIALES CON JEE

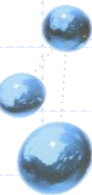
# ESTÁNDAR JEE

En qué consiste y para qué sirve

Profesor: Andrés Farías

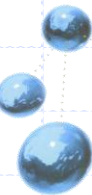
# Agenda

- Introducción
  - ✓ Algo de historia.
  - ✓ Patrones arquitecturales.
- Objetivos del estándar
- Arquitectura J2EE
- (Repaso) Java 1.5

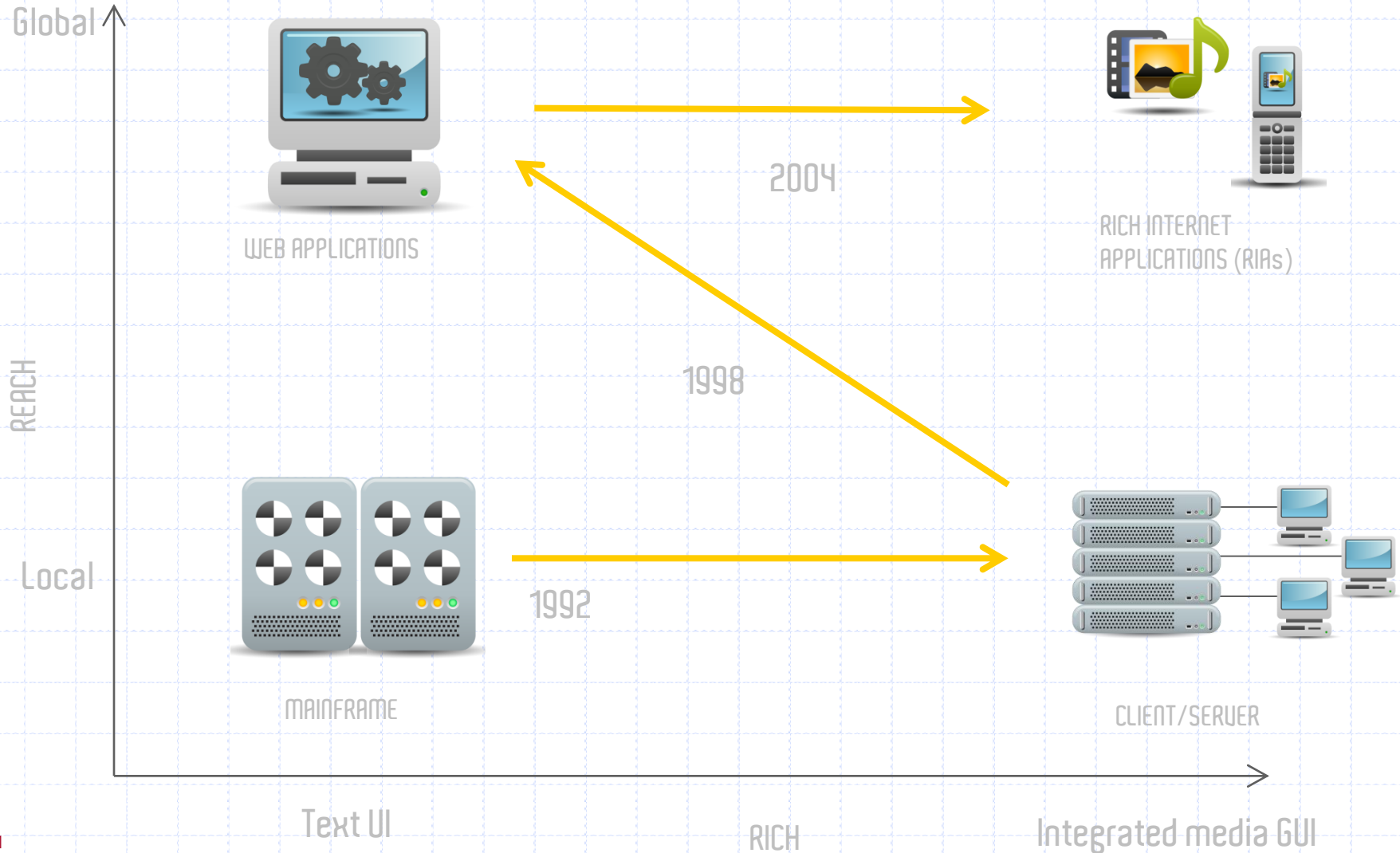


# Introducción

1. Evolución
2. Patrones Arquitecturales



# Evolución de las aplicaciones



## Mainframes

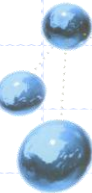
- Un gran servidor que acepta conexiones desde terminales tontos.
- Toda la lógica de negocio y, muy acoplados a estos en general, los datos.
- Se considera el Mainframe como una BDD.



cliente

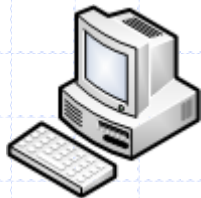


Servidor de BD



## Cliente-servidor

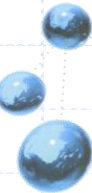
- Utilizado normalmente para consultar, modificar y administrar los datos de una organización
- El programa cliente era la interfaz inteligente entre el usuario y la base de datos.
- Los programas podían contener algo de lógica de negocio, pero no necesariamente datos persistentes.



cliente

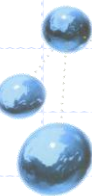


Servidor de BD



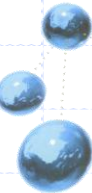
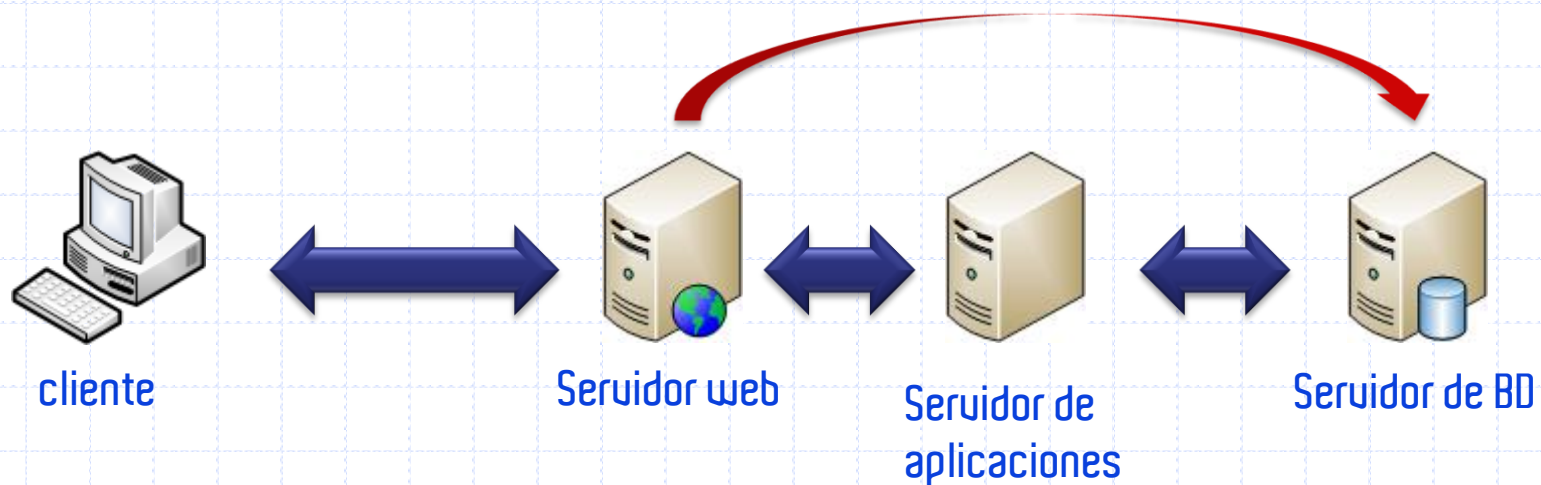
## Cliente-servidor: Inconvenientes

- Aplicaciones monolíticas difíciles de mantener
  - ✓ Toda la inteligencia está en el cliente
  - ✓ Los servidores son sólo servidores de datos
- Distribución del código que cambia
- Poca reusabilidad del código
  - ✓ No está orientado al desarrollo de componentes
- No se puede usar de cualquier lado
- Poca seguridad
- Mucho tráfico de datos



## N-Capas

- El cliente es el browser
- La aplicación misma está en el servidor web
- Puede existir un servidor de aplicaciones





## N-Capas: Ventajas

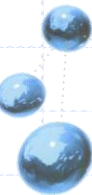
### ■ Arquitectura:

- ✓ Separación de las funciones de control de la interfaz y presentación de datos con la lógica de la aplicación.
- ✓ Independencia de la interfaz del cliente y la arquitectura de datos.
- ✓ Mejores posibilidades de balancear la carga.

### ■ Reusabilidad de componentes.

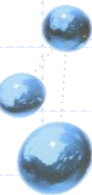
### ■ Mantenibilidad y Transferabilidad:

- ✓ Uso de protocolos abiertos.



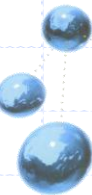
# ARQUITECTURA J2EE

1. Introducción
2. El estándar J2EE
3. Las API J2EE
4. Contenedores



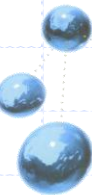
## La Web y J2EE

- En los 90's inicia la Word Wide Web con la distribución de información basado en hipertexto.
- Las aplicaciones web están basadas en clientes (navegadores) y servidores (HTTP).



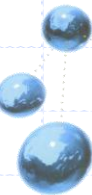
# LA ARQUITECTURA J2EE

## 2. El estándar J2EE



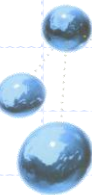
## Aparece un estándar

- La plataforma de Java, Enterprise Edition (J2EE) define el estándar para desarrollar componentes empresariales basados en multicapa u orientadas a servicios.
- Simplifica la construcción de aplicaciones empresariales estables, escalables y que se integran fácilmente datos y aplicaciones heredadas.



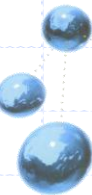
## Objetivos del estándar

- Independencia de la plataforma
  - ✓ Especialmente del lado del cliente
  - ✓ De la arquitectura física
- Reusabilidad
- Modularidad
- Escalabilidad
- Facilidad de administración y mantenimiento
- La idea es la de proveer un estándar simple y unificado para aplicaciones distribuidas a través de modelos de aplicación basado en componentes.



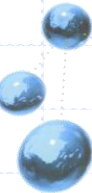
## La plataforma J2EE

- Es en esencia un ambiente de servicios de aplicaciones distribuidas, basado en contenedores.
- Se compone de:
  - ✓ Una infraestructura de ejecución para el hospedaje de aplicaciones (runtime hosting applications).
  - ✓ Una familia de APIs para construir aplicaciones.



# LA ARQUITECTURA J2EE

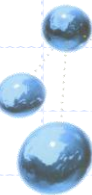
## 3. Las API J2EE





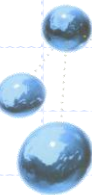
## APIs J2EE

1. Java Database Connectivity (JDBC)
2. Remote Method Invocation Inter-ORB protocol (RMI-IIOP)
3. Enterprise Java Beans (EJB)
4. Java Servlets 2.2
5. Java Server Pages 1.1 (JSP)
6. Java Message Service (JMS)
7. Java Naming and Directory Interface (JNDI)
8. Java Transaction API
9. Java Mail
10. Standard Java API



## Java IDL (Interface Definition Language)

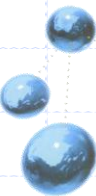
- Se utiliza cuando se desea tener interoperabilidad y conectividad basada en estándares **CORBA**.
- Permite que Aplicaciones Web Java distribuidas invoquen operaciones de manera transparente en una red de servicios remotos utilizando **CORBA IDL** y **IIOP** (*Internet Inter-ORB Protocol*)
- Componentes de ejecución incluyen un **ORB** Java para computación distribuida usando protocolos **IIOP**.



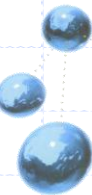
## RMI-IIOP

(1/2)

- Permite que programadores **RMI** usen el protocolo de comunicación **IIOP** de **Corba** para comunicarse con clientes de cualquier tipo.
  - ✓ Los clientes pueden ser hechos en Java o componentes escritos en cualquier otro lenguaje compatible con **Corba** (Corba-Compliant).
- Permite la programación de servidores **Corba** y aplicaciones vía la API **RMI**.
  - ✓ Escribe todo el código en **Java**,
  - ✓ Utiliza el compilador **rmic** para generar el código para conectar aplicaciones vía **IIOP** a otras aplicaciones escritas en cualquier lenguaje **CORBA-compliant**.
- Combina lo mejor de Java **RMI** con lo mejor de **CORBA**.

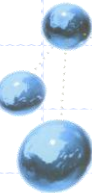


- Permite escribir aplicaciones CORBA para la plataforma Java sin tener que aprender el IDL CORBA.
- Permite pasar cualquier objeto serializable Java (Objects By Value) entre componentes de una aplicación.
- Incluye la funcionalidad completa de un ORB CORBA (Object Request Broker)



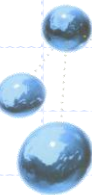
## Java IDL vs RMI-IIOP

- El IDL de Java sirve a programadores que quieren programar en Java basados en las interfaces definidas en el CORBA IDL.
- RMI-IIOP es para los programadores que quieren programar interfaces RMI, pero usan IIOP como el protocolo de transporte subyacente.
  - ✓ Provee interoperabilidad con otros objetos CORBA implementados en varios lenguajes – pero sólo las interfaces remotas están originalmente definidas como interfaces RMI Java.
- Utilizado en Enterprise JavaBeans (EJB) 1.1, desde que el modelo de objetos remotos para EJB's se basó en RMI.



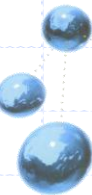
## JNDI

- Provee las funcionalidades de directorio y nombres para aplicaciones Java independientes de cualquier implementación específica.
- La Arquitectura consiste en una API y un SPI (*Service Provider Interface*)
- Estos son los SPIs provistos:
  - ✓ LDAP
  - ✓ COS Naming (*Common Object Services*)
  - ✓ RMI registry



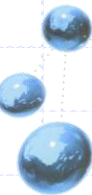
## Mensajería y JMS

- Método de comunicación entre componentes de software y aplicaciones.
- JMS es un sistema de mensajería:
  - ✓ Con facilidades peer-to-peer.
  - ✓ Clientes de mensajería pueden enviar mensajes hacia y recibir mensajes desde, cualquier otro cliente.
- Agentes de mensajería proveen facilidades para:
  - ✓ Crear mensajes
  - ✓ Enviar mensajes
  - ✓ Recibir mensajes



## JMS

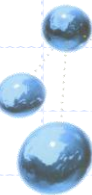
- Cada cliente se conecta al agente de mensajería.
- Comunicación de mensajes
  - ✓ Bajo acoplamiento
  - ✓ Asíncrona
  - ✓ Confiable.





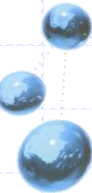
## ¿Cuándo usar mensajería?

- Cuando los componentes deben ser independientes de las interfaces del otro, de manera que éstos puedan ser fácilmente reemplazados.
- Cuando el sistema debe funcionar de manera independiente a si todos los componentes están corriendo de manera simultánea.
- El modelo del negocio obliga a un componente a enviar información a otro componente y continuar operando sin recibir una respuesta inmediata.
- Ejemplos
  - ✓ Una plata de ensamblaje hace una orden de inventario con una fábrica cuando los suministros están bajos.
  - ✓ Sistemas transaccionales que deben funcionar incluso estando fuera de línea.



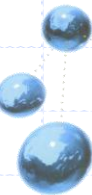
# LA ARQUITECTURA J2EE

## 4. Contenedores



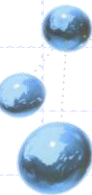
## Administración de componentes basados en contenedores

- Que es un Contenedor? Un ambiente de ejecución que provee servicios.
- Algunos de estos servicios son:
  - ✓ Administración del ciclo de vida
  - ✓ Seguridad
  - ✓ Despliegue
  - ✓ Threading



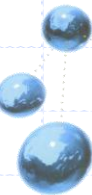
## Los Contenedores J2EE

- La arquitectura se compone de cuatro contenedores
  - ✓ Un contenedor de WEB. Para hospedar Java servlets y páginas JSP.
  - ✓ Un contenedor de EJB. Para hospedar componentes JavaBeans
  - ✓ Un contenedor de aplicaciones clientes. Para ejecutar aplicaciones clientes de consola.
- Un contenedor es un ambiente de ejecución (runtime) Java 1.2 Standar Edition para componentes de aplicaciones. Ej:
  - ✓ Sun JSE
  - ✓ BEA JRockit



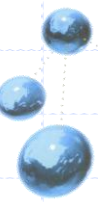
## Arquitectura de un contenedor

- Componentes de aplicaciones
  - ✓ Incluye servlets, jsp, ejb, etc.
  - ✓ Pueden ser empaquetados en archivos (jar, war, ear).
- Especificaciones de despliegue (deployment descriptors).
  - ✓ Un archivo en XML que describe los componentes de la aplicación.
  - ✓ Incluye también información adicional para la administración efectiva de componentes de la aplicación.

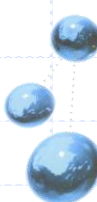
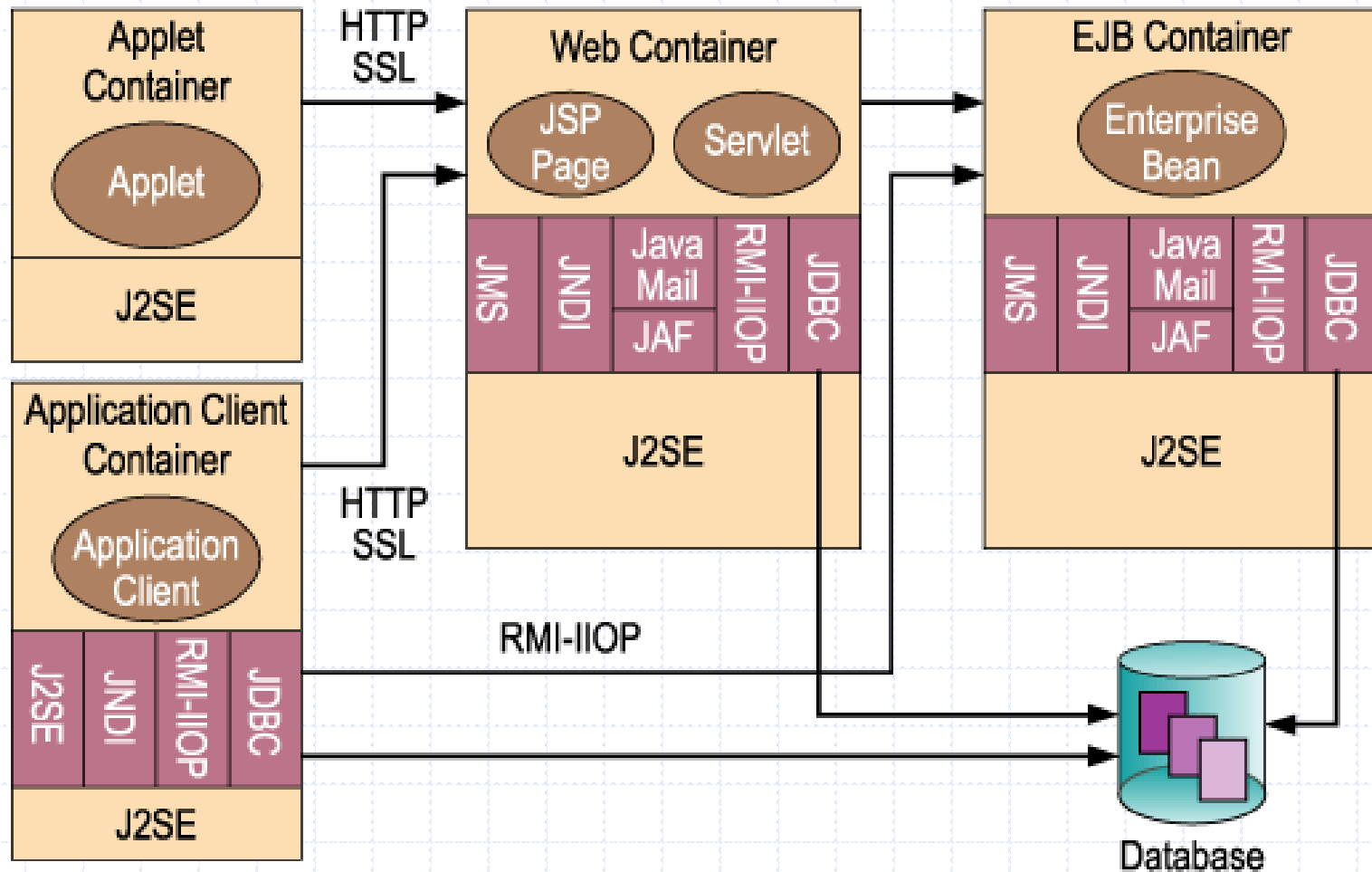


### Sus partes

- Contratos de componentes
  - ✓ Definidos por la JVM
  - ✓ Para el contenedor WEB el contrato es para indicar las APIs de Servlets y JSP utilizadas.
- APIs de servicios del contenedor
  - ✓ Un contenedor provee una vista global a varios APIs corporativos especificados en el estándar J2EE.
  - ✓ Accesibles vía JNDI
- Servicios declarativos
  - ✓ Son servicios o acciones que se llevan a cabo por el contenedor en el que se encuentra una aplicación vía una invocación específica en los descriptores de despliegue.
- Otros servicios del contenedor
  - ✓ Recolección de basura
  - ✓ Colecciones de recursos (resource pooling)

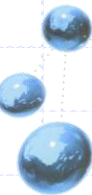


# Servicios y Contenedores J2EE



## Contenedor Web

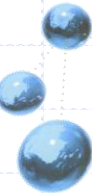
- Ambiente de ejecución para una aplicación web.
- Services adicionales:
  - ✓ Contexto de nombres
  - ✓ Administración del ciclo de vida



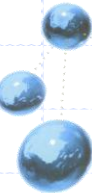
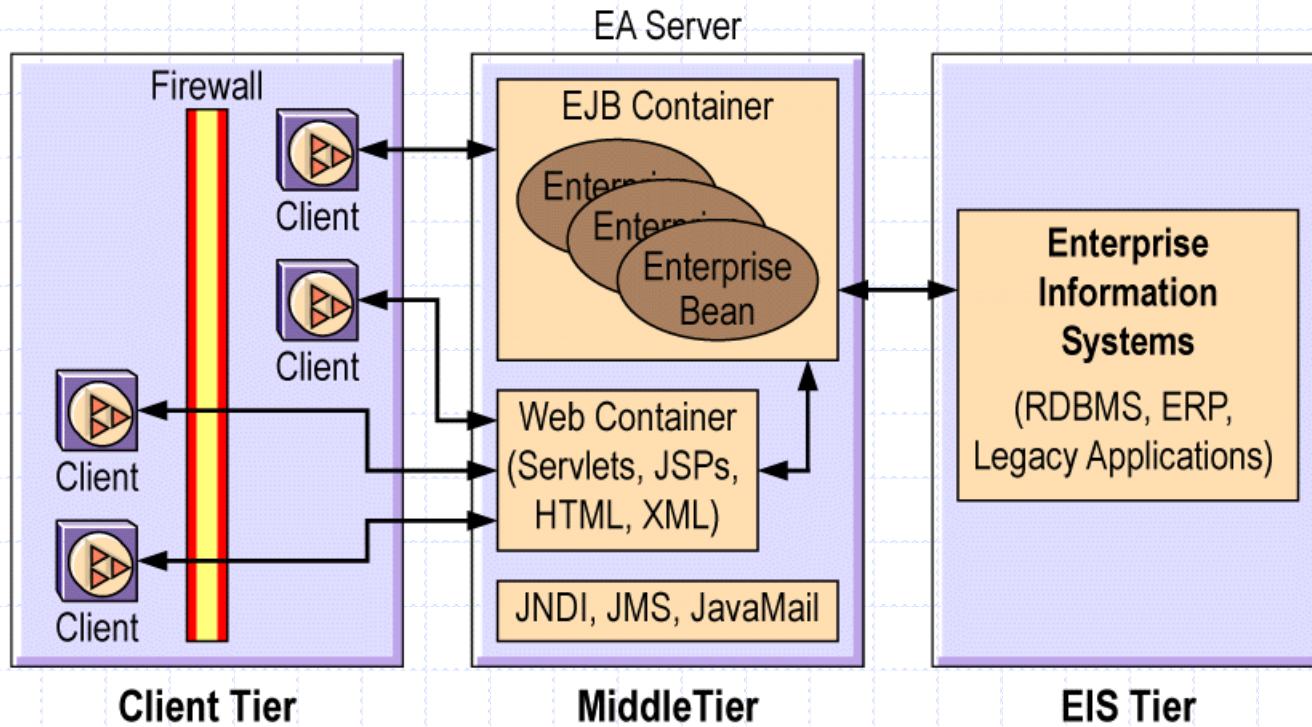


## Contenedor EJB

- Ambiente de ejecución para EJBs
- Services adicionales:
  - ✓ Transaccional
  - ✓ Persistencia

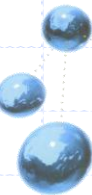


# EAServer y J2EE



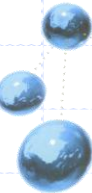
## EAServer J2EE y mucho más

- Opciones de mensajería avanzada
  - ✓ Agente de eventos empresariales (Enterprise Event Broker)
  - ✓ Agente de mensajes (Message Broker)
- Alto desempeño y disponibilidad
  - ✓ Balanceador de carga
  - ✓ Failover Stateful y stateless
- Seguridad
  - ✓ SSL
  - ✓ Encryption



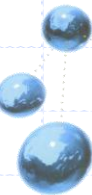
# LA ARQUITECTURA J2EE

## Resumen

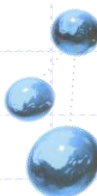
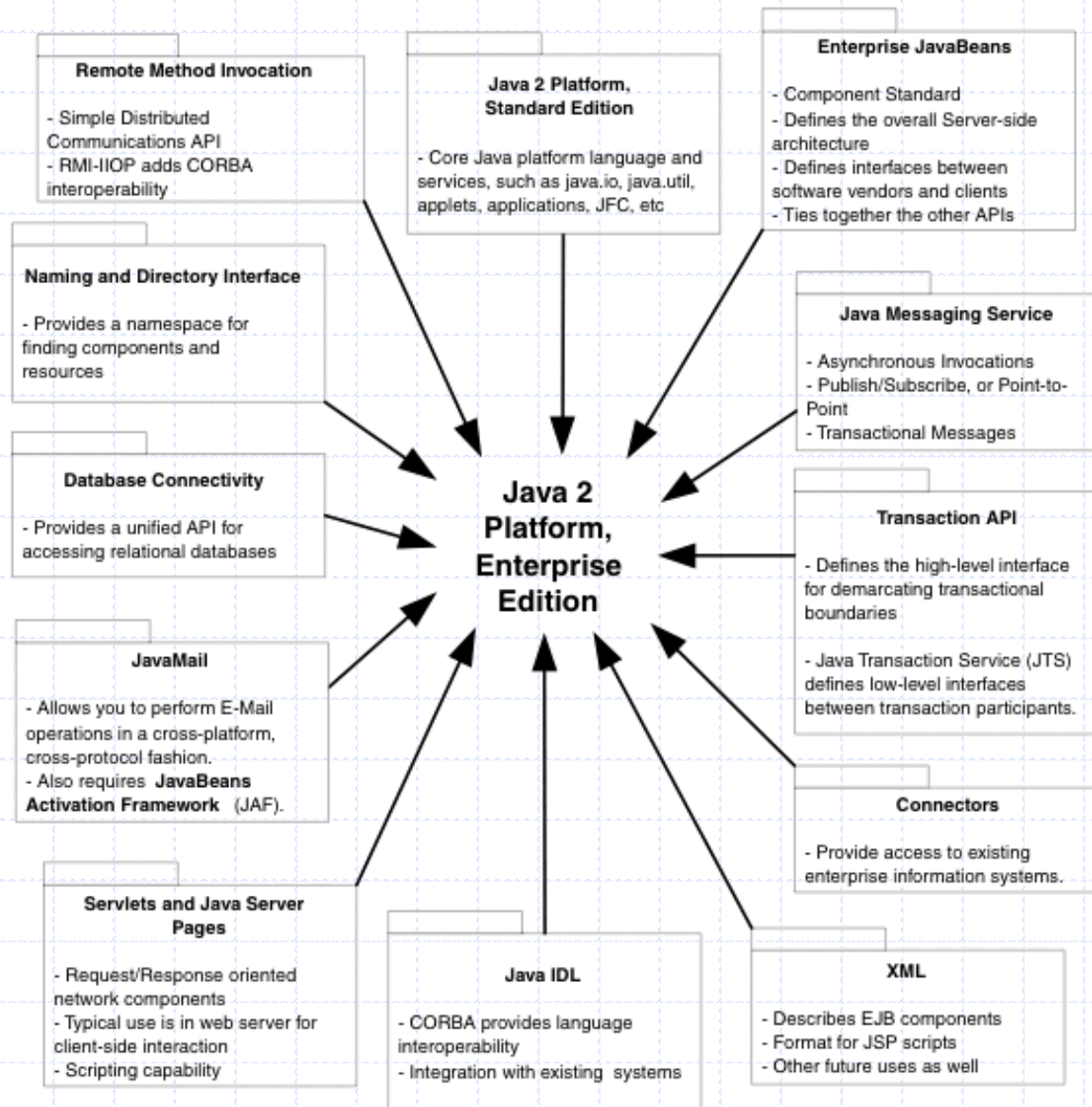


# Tecnologías en J2EE

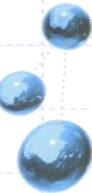
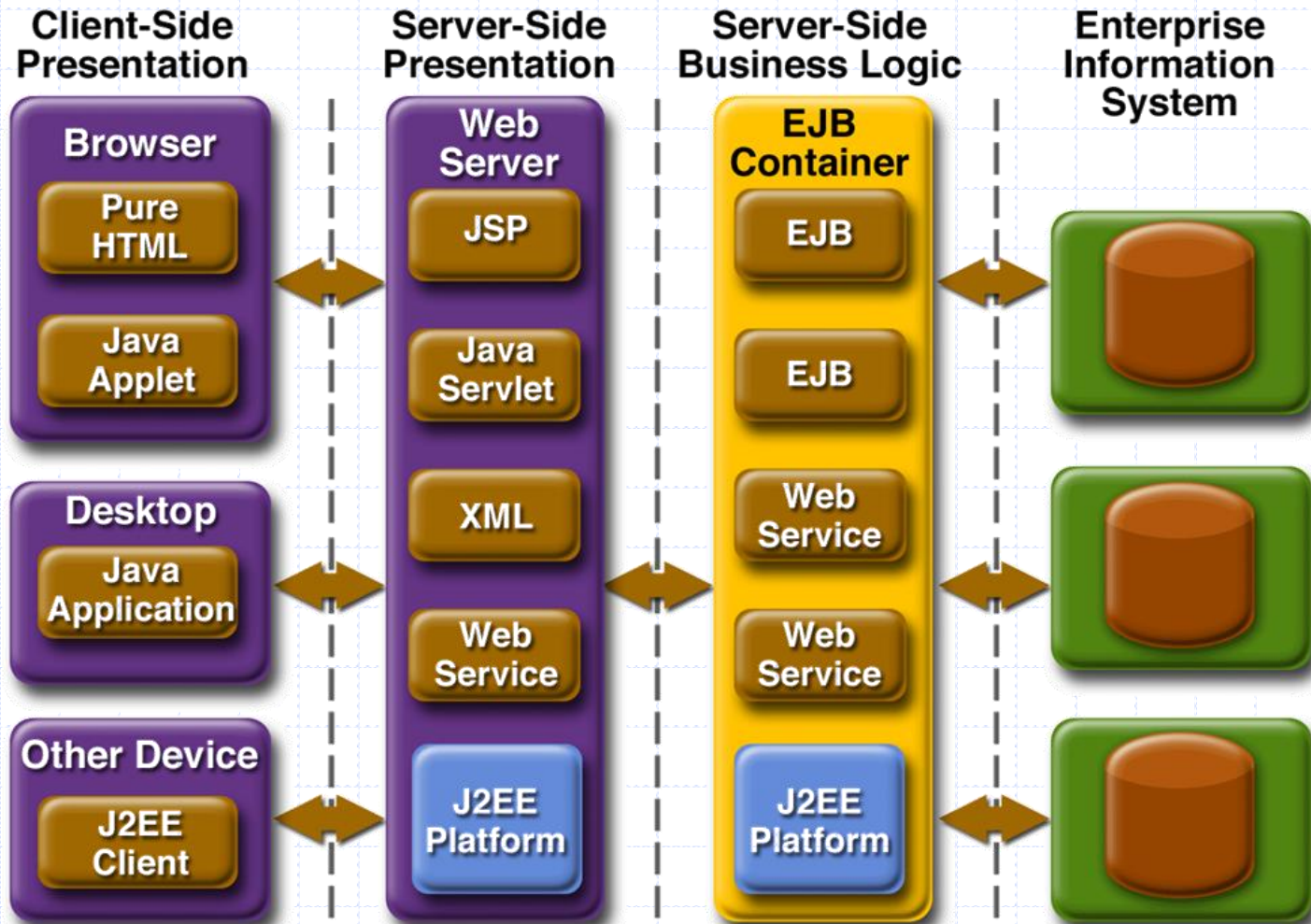
- Tecnologías de componentes
  - ✓ Contienen la lógica de la aplicación
    - ◆ Servlets, JSP, EJB
- Tecnologías de servicio
  - ✓ Servicios de soporte para los componentes de aplicaciones
    - ◆ JDBC, JNDI, JTA
- Tecnologías de comunicación
  - ✓ Proveen los mecanismos de comunicación entre las diferentes partes de la aplicación, ya sean locales o remotas
    - ◆ HTTP, TCP/IP, SSL, RMI, RMI-IIOP, JMS, JavaMail



## Arquitectura J2EE



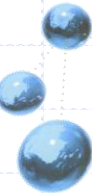
# Arquitectura J2EE





## Arquitectura J2EE

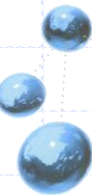
- J2EE es una plataforma para el desarrollo de aplicaciones empresariales, multi-capas independientes de la plataforma basadas en componentes.
- Diversos estándares constituyen la plataforma J2EE
  - ✓ Servlet
  - ✓ JSPs
  - ✓ EJBs
- J2EE define la administración de componentes basada en contenedores.
- EAServer provee 2 de los 4 contenedores definidos por J2EE así como otros servicios.



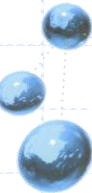


# Especificaciones J2EE de Interés

- Java Server Pages 2.0 [Specification](#),
- Server API 2.4 [Specification](#),
- Sun BluePrints<sup>tm</sup> [Design Guidelines for J2EE](#),
- J2EE 1.4 [Specification](#),

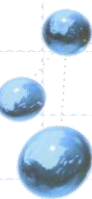


# WHAT'S NEW ON JAVA 1.5



## Introducción

- J2SE 5 representa la mayor innovación en la tecnología Java desde su creación
- Añade las siguientes características:
  - ✓ Mecanismos de autoboxing y unboxing de tipos primitivos
  - ✓ Mejora a la sentencia de control for
  - ✓ Métodos con número de argumentos variable (varargs)
  - ✓ Enumeraciones
  - ✓ Generics (templates de C++) proporcionan seguridad de tipos en tiempo de compilación
  - ✓ Imports estáticos
  - ✓ Nuevo método (printf) para generar salida por consola

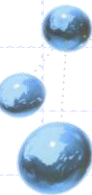


## Autoboxing y Unboxing

- El siguiente código es incorrecto en J2SE 1.4:  

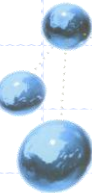
```
List numbers = new ArrayList();  
numbers.add(89);
```
- Lo correcto hubiera sido:  

```
numbers.add(new Integer(89));
```
- Sin embargo, en Java 5, la sentencia `numbers.add(89)` sería correcta gracias al mecanismo de Boxing:
  - ✓ El compilador automáticamente añade el código para convertir un tipo primitivo en su clase correspondiente
- Mecanismo de unboxing
  - ✓ El proceso opuesto de convertir un objeto (tipo `Integer`) en un valor (un entero)
  - ✓ `System.out.println(numbers.get(0));`



# Autoboxing y Unboxing

Tipo Primitivo	Clase Referenciada
boolean	Boolean
byte	Byte
double	Double
short	Short
int	Integer
long	Long
float	Float



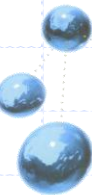
## Bucle For Mejorado

- Hasta Java 1.4 para iterar por una colección o array hacíamos lo siguiente:

```
// numbers es una lista de números
for (Iterator it = numbers.iterator(); it.hasNext(); )
{
    Integer number = (Integer) it.next();
    // Hacer algo con el número...
}
```

- Lo mismo en Java 5 puede hacerse del siguiente modo:

```
for(Integer number: numbers)
{
    // Do something with the number...
}
```



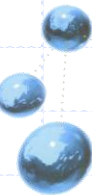
## Número Variable de Argumentos

- Una manera tradicional de emular esto sería:

```
int sum(Integer[] numbers) {  
    int mysum = 0;  
    for(int i: numbers)  
        mysum += i;  
    return mysum;  
}  
sum(new Integer[] {12,13,20});
```

- En Java 5.0, sin embargo se realizaría del siguiente modo:

```
int sum(Integer... numbers) {  
    int mysum = 0;  
    for(int i: numbers)  
        mysum += i;  
    return mysum;  
}  
sum(12,13,20);
```



## Número Variable de Argumentos

- Una clara aplicación del paso de parámetros variables a un método es la implementación de `System.out.printf`:

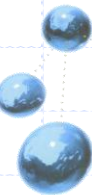
```
// Usando System.out.println y System.out.printf
int x = 5;
int y = 6;
int sum = x + y;
```

```
// Antes de Java 5.0 haríamos
System.out.println(x + " + " + y + " = " + sum);
```

```
// Pero ahora podemos hacer
System.out.printf("%d + %d = %d\n", x, y, sum);
```

- Otro ejemplo más complejo, ¿qué hace?:

```
System.out.printf("%02d + %02d = %02d\n", x, y,
    sum);
```

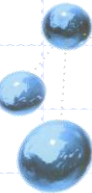




## Enumeraciones

- Antes para declarar una enumeración en Java hacíamos lo siguiente:

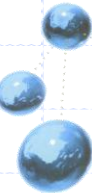
```
public Class Color {  
    public static int Red = 1;  
    public static int White = 2;  
    public static int Blue = 3;  
}  
  
int myColor = Color.Red;  
// incorrecto semánticamente  
int myColor = 999;
```



# Enumeraciones

- Una manera de implementar esto con validación sería:

```
public class Color {  
    // Color value;  
    int _color;  
  
    // Constructor  
    protected Color (int color) {  
        _color = color;  
    }  
  
    private static final int _Red = 1;  
    private static final int _White = 2;  
    private static final int _Blue = 3;  
  
    public static final Color Red = new Color(_Red);  
    public static final Color White = new Color(_White);  
    public static final Color Blue = new Color(_Blue);  
}  
// Invocar con: Color myColor = Color.Red;
```



## Enumeraciones

- La solución en Java 5 es mucho más sencilla:

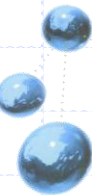
```
public enum Color
{
    Red,
    White,
    Blue
}
```

```
// Color myColor = Color.Red;
```

- Podríamos enumerar su contenido como:

```
for (Color c : Color.values())
    System.out.println(c);
```

- Otros métodos disponibles en una enum son: `name()`, `toString()`, `ordinal()` o `compareTo()`.



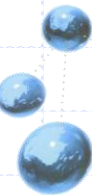
## Imports Estáticos

- Extiende el modo en que **import** funciona en Java.
- Por ejemplo para usar la función **ceil()** habría que importar primero `java.lang.Math` y luego escribir:

```
double y = Math.ceil(3.2); // = 4,0
```

- Sin embargo, ahora con Java 5 se podría hacer: **double y = ceil(x)**, ya que se pueden hacer imports del siguiente modo:

```
✓ import static java.lang.Math.ceil;  
✓ import static java.lang.Math.*;
```



# Generics

- Su principal cometido es implementar seguridad de tipos en tiempo de compilación.
- El siguiente código compilaría pero lanzaría un error en tiempo de ejecución:

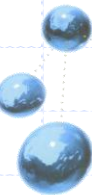
```
// Declare Class A  
class A {}
```

```
// Declare Class B  
class B {}
```

```
// Somewhere in the program create a Vector  
Vector v = new Vector();
```

```
// Add an object of type A  
v.add(new A());
```

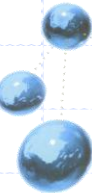
```
// And sometime later get the object back  
B b = (B) v.get(0);
```



## Generics

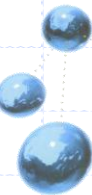
- Lo mismo con Java 5 usando Generics no compilaría:

```
Vector<A> v = new Vector<A>();  
// Añadir un objeto de tipo A  
v.add(new A());  
// Y luego intentar recuperarlo como B  
B b = (B) v.get(0);
```



# Desarrollando aplicaciones J2EE

- Desarrollo del componente de la aplicación
  - ✓ Se modela la lógica del sistema en la forma de componentes de aplicaciones
- Composición de los componentes de la aplicación en módulos
  - ✓ Los componentes de la aplicación se empaquetan en módulos. Se proveen deployment descriptors para cada módulo
- Composición de módulos en la aplicación
  - ✓ Integración de múltiples módulos en aplicaciones J2EE proveyendo sus deployment descriptors
- Despliegue de la aplicación
  - ✓ Se despliega e instala la aplicación empaquetada en un servidor J2EE



# PREGUNTAS?

