



Enterprise Java Beans (EJB)

Aplicações Distribuídas
Cliente/Servidor
Corporativas



Introdução

- Desenvolver e distribuir Servlets e aplicativos EJB.
- Desenvolver e distribuir aplicativos Enterprise JavaBeans (EJB).



Introdução

- Simples aplicativos Web :
Servlets e JSP,
Evitar EJB.
- **Robustez e Escalonamento :**
considerar o desenvolvimento de um
aplicativo EJB.



Introdução

- Existem muitos benefícios que tornam EJB atraentes.

- EJB depende de outras tecnologias Java:

RMI – como protocolo de comunicação entre dois enterprise beans e um enterprise bean e seu cliente



Introdução

EJB com **RMI-IIOP** -

versão mais portátil de RMI, usada em comunicações entre um cliente e um enterprise bean.



Introdução

EJB usa **JNDI** (Java Naming and Directory Interface) - como serviço de nomeação que liga um nome com um enterprise bean.

- Nota: para entender EJB é obrigatório entender dessas tecnologias de suporte.



Introdução

- Introduzir EJB :
Definindo e mostrando os benefícios que a maioria dos quais não estão disponíveis em Servlets e JSP.
- Arquitetura e as funções no aplicativo EJB.
- O ciclo de vida de distribuição.



Introdução

- Um aplicativo de exemplo.
- Revisão do pacote `javax.ejb` .
- Dois aplicativos-clientes para testar o aplicativo de exemplo.
- Enterprise JavaBean = Enterprise Bean.
- Bean significa Enterprise Bean.



O que é um Enterprise JavaBean

- É um componente do lado do servidor.
- Componente que encapsula a lógica da aplicação comercial.
- Precisam estar de acordo com as especificações EJB.



O que é um Enterprise JavaBean

- Só são distribuídos e podem executar apenas em um container EJB, assim como um servlet executa dentro de um container servlet.



O que é um Enterprise JavaBean

- Um container servlet oferece serviços para servlets: gerenciamento de sessão e segurança.
- Um container EJB oferece **serviços no nível de sistema** para aplicativos EJB.
- O container é a essência do EJB.



Container EJB

- Plataformas de middleware orientadas a objeto, como o ORB do CORBA ou RMI, livram o desenvolvedor de aplicações distribuídas, dos aspectos de rede, provendo mecanismos para **localização**, **marshaling/ unmarshaling** dos dados de mensagens, entre outras coisas.



Container EJB

- O conceito de um **container** segue esta idéia, por simplificar outros aspectos não triviais de uma aplicação distribuída, tais como, **segurança**, **coordenação de transações** e **persistência de dados**.



O que é um Container EJB

- Protocolo de Comunicação
- Serviço de Localização de Objetos
- Serviço de Transações
- Serviço de Segurança
- Ativação, Desativação e Persistência de Objetos



Container EJB através de CORBA

- ORB sobre IIOP
- JNDI sobre CORBA Naming Service (+ Trade Service)
- OTS
- CORBA Security
- POA



Benefícios do EJB

- Por que as organizações querem investir em EJB?



Benefícios do EJB

- Resposta:

Depois que se conhece os detalhes práticos de EJB, escrever um aplicativo é uma tarefa mais fácil e se pode usufruir de alguns benefícios oferecidos pelo container EJB.



Benefícios do EJB

- O desenvolvedor de aplicativo EJB pode se concentrar na lógica. Ao mesmo tempo ele usa os serviços fornecidos pelo container, que são produtos bem mais caros, quando adquiridos isoladamente.
- EJBs são componentes.



Benefícios do EJB

- A especificação EJB assegura que os beans desenvolvidos por outros podem ser usados em seu aplicativo.



Benefícios do EJB

- Há uma clara divisão de trabalho no desenvolvimento, disponibilização e administração de um aplicativo EJB.
- Isto torna o processo de desenvolvimento e disponibilização ainda mais rápido.



Benefícios do EJB

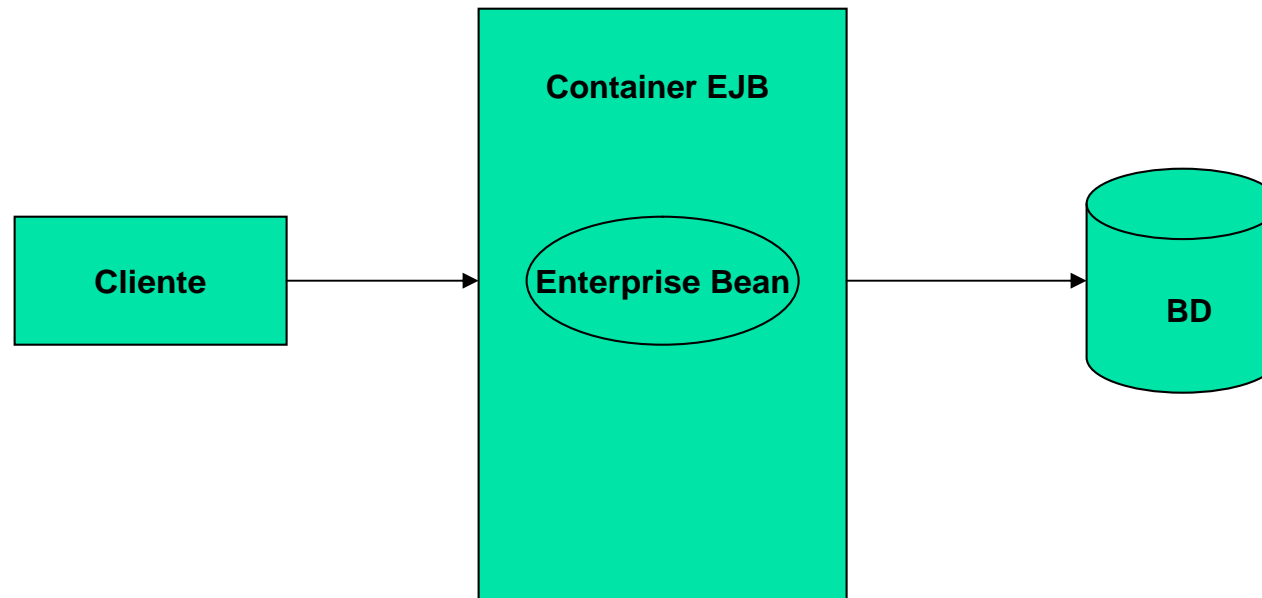
- O container EJB gerencia determinadas APIs de nível baixo, sem que o desenvolvedor precise entendê-las.



Benefícios do EJB

- A arquitetura EJB é compatível com outras APIs Java.

Arquitetura de Aplicativo EJB



A arquitetura de aplicativo EJB.



Arquitetura de Aplicativo EJB

- Os clientes de um enterprise bean podem ser um aplicativo tradicional Java, um applet, uma página JSP ou um servlet, um outro EJB Bean ou outros.



Arquitetura de Aplicativo EJB

- Um cliente nunca chama diretamente os métodos de um bean.
- A comunicação entre clientes e beans é feita através do container EJB.



Arquitetura de Aplicativo EJB

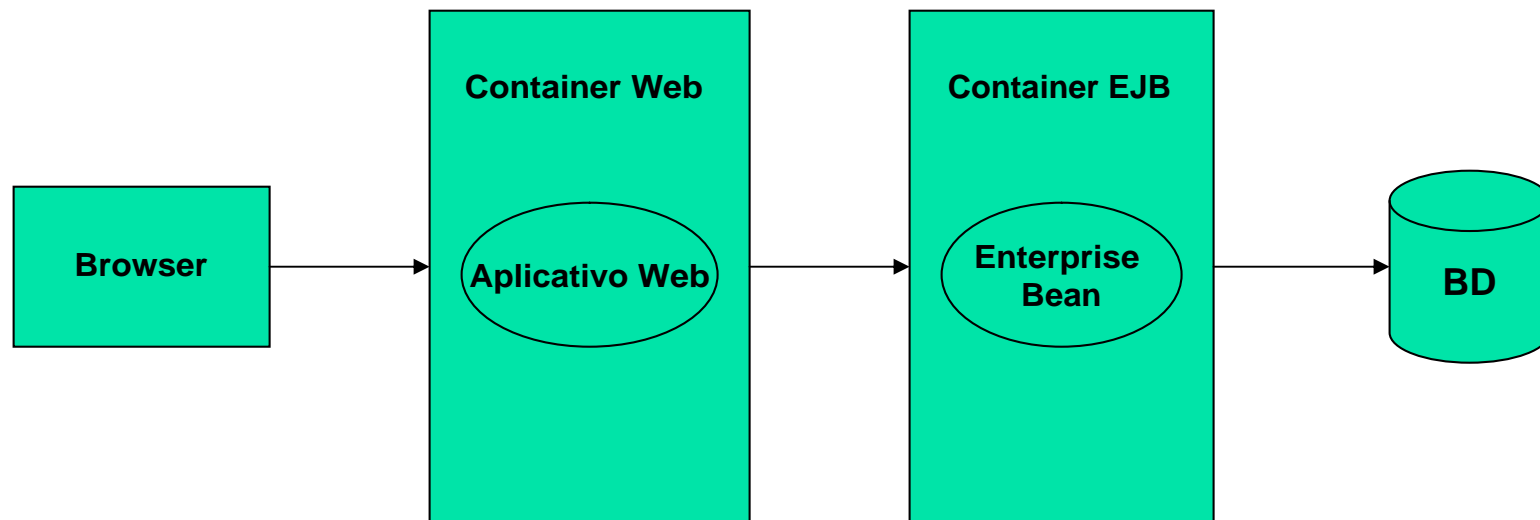
- Compare isto com um aplicativo Web, onde um cliente browser web precisa usar o container de web para usar um servlet ou uma página JSP.



Arquitetura de Aplicativo EJB

- Quando um cliente é um servlet ou uma página JSP, a estrutura de um aplicativo EJB se parece com a arquitetura a seguir:

Arquitetura de Aplicativo EJB



Um aplicativo EJB com um servlet como cliente.



As seis funções EJB

- Funções no desenvolvimento de aplicativo EJB e ciclo de vida de distribuição:
 - Desenvolvedor Bean,
 - Montador do Aplicativo EJB,
 - Disponibilizador (Deployer),
 - Administrador de Sistema,
 - Provedor de Serviço EJB,
 - Provedor de Container EJB.



Desenvolvedor Bean

- É o programador que desenvolve os enterprise bean.
- Precisa entender a lógica comercial do aplicativo.



Montador de Aplicativo

- Tipicamente, um aplicativo EJB consiste de mais de um enterprise bean.
- Em aplicativos maiores, vários desenvolvedores bean podem ser contratados para construir os bens.



Montador de Aplicativo

- O montador é a pessoa que agrupa todos os beans escritos pelos desenvolvedores.
- O montador também escreve o disponibilizador do aplicativo EJB.



Distribuidor (Deployer)

- É o encarregado de disponibilizar o aplicativo EJB em um container, ou em containers, se houver mais do que um container sendo usado.



Disponibilizador (Deployer)

- Essa pessoa pega os enterprise beans desenvolvidos e o disponibilizador escrito pelo montador.
- Essa pessoa precisa ser especialista no container EJB.



Distribuidor

- Você será apresentado a essa função durante o desenvolvimento e a disponibilização do aplicativo EJB, que será apresentado.



Administrador de Sistema

- É função do administrador garantir que o aplicativo execute 24 horas por dia, sem interrupção.
- É responsável por gerenciar a segurança.



Provedor de Container

- É um fabricante que tem os recursos para construir um container EJB e garantir que o software esteja de acordo com a especificação EJB.



Provedor de Servidor EJB

- O provedor de servidor oferece um servidor EJB, que por sua vez hospeda um container EJB.
- A maioria dos containers EJB vem empacotados em um servidor EJB.



Tipos de Enterprise Bean

- Session Bean
(Bean de Sessão)
- Entity Bean
(Bean de Entidade)
- Message-Driven Bean
(Bean Direcionado por Mensagem)



Session Bean

- É um componente que executa determinada tarefa para o cliente.
- Contém a lógica de negócio da aplicação.



Entity Bean

- Representa uma entidade no banco de dados ou outra armazenagem persistente.



Message-Driven

- Extensão ao EJB 2.0
- Serve como um listener para a API Java Message Service, que possibilita o processamento de mensagem assíncrona.



Como escrever o seu Primeiro Enterprise Bean

- Um simples aplicativo EJB é apresentado (Session Bean).
- O exemplo executa em um servidor de aplicação JBOSS.
- Depois de criar o aplicativo EJB, é preciso criar o aplicativo-cliente.



Etapas *desenvolver e distribuir*

- Desenvolver e disponibilizar um aplicativo EJB, requer as seguintes etapas:
 1. Escrever o bean e compilar com sucesso.
 2. Escrever o descritor de disponibilização.
 3. Criar um arquivo de disponibilização.
 4. Disponibilizar o bean.
 5. Escrever o aplicativo-cliente para testar o bean.



Um Aplicativo EJB

- Apenas um enterprise bean, que pode fazer a adição de dois inteiros a e b.
- Compilar o bean com sucesso.
- Escrever o descritor de distribuição e distribuir o bean.
- Criar um servlet, no caso, o cliente do aplicativo EJB.



Iniciando o desenvolvimento

- Considere o pacote:

`com.brainysoftware.ejb`

- Criar a estrutura de diretório apropriada para um aplicativo EJB. Veja as pastas:

classes (aberta)

- com

- brainysoftware

+.ejb

+ META-INF



Arquivos do aplicativo EJB

- Três arquivos Java para escrever.
- Todos fazendo parte do pacote `com.branysoftware.ejb` .
- Os três arquivos são:
 - AdderHome.java (interface home)
 - Adder.java (interface remota)
 - AdderBean.java (classe session bean)



Interface Home

- A interface Home controla o ciclo de vida de um enterprise bean. Ela especifica operações para criar (create), encontrar (find) e remover (remove) enterprise beans (objetos EJB).



Interface Home

- Auxilia a **criar, localizar e remover** instâncias de um enterprise bean.



A interface Home

- A interface exerce um papel similar ao objeto factory que pode ser construído ou usado como design pattern no RMI e no CORBA.



A interface AdderHome

```
package    com.brainysoftware.ejb;
import    java.rmi.RemoteException;
import    javax.ejb.CreateException;
import    javax.ejb.EJBHome;

public    interface    AdderHome    extends    EJBHome
{
        Adder    create()    throws    RemoteException,
                                CreateException;
}
```



A interface Remote

- Especifica os métodos de negócio (os métodos da lógica da aplicação), que são providos pelo enterprise bean.
- Base de todas as interfaces remotas.



A classe *implementation*

- Implementa a funcionalidade definida nas interfaces Home e Remote.



A classe AddBean

```
package    com.brainysoftware.ejb;
import    java.rmi.RemoteException;
import    javax.ejb.SessionBean;
import    javax.ejb.SessionContext;

public class  AdderBean implements SessionBean
{
    public int add(int a, int b) {
        System.out.println ("from AdderBean");
        return (a + b);
    }
    ...
    ... (**)
    ...
}
```



(**)

```
public void ejbCreate() {  
}  
public void ejbRemove() {  
}  
public void ejbActivate() {  
}  
public void ejbPassivate() {  
}  
public void setSessionContext(SessionContext sc) {  
}
```




ejbCreate()

- O ciclo de vida de um session bean inicia quando um cliente invoca um método **create()** sobre a interface Home do session bean.



ejbCreate()

- A implementação da interface Home é provida pelo container, baseada na classe de implementação do session bean.



ejbCreate()

- O container cria uma nova instância do session bean, inicializa ela e retorna uma referência de objeto de volta ao cliente.



setSessionContext()

- Durante este processo, primeiro o método `setSessionContext()`, e então o método `ejbCreate()` são invocados na implementação do session bean (a qual implementa a interface do session bean).



`ejbCreate()`, `setSessionContext()`

- O provedor do enterprise bean pode usar estes métodos para inicializar um session bean.



setSessionContext()

- Estabelece um contexto de sessão.
- O contexto determina as políticas para executar o bean.
- Por exemplo, o comportamento transacional ou o controle e acesso de um enterprise bean.



setSessionContext()

- O contexto de sessão provê métodos para acessar propriedades de runtime (tempo de execução) no qual uma sessão roda.

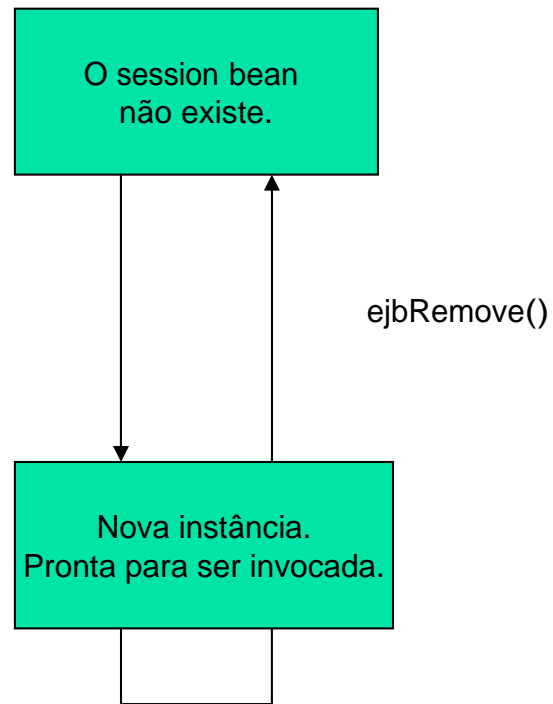


ejbRemove()

- Notifica um session bean que ele é para ser removido.
- O container remove o session bean, quando o cliente invoca uma operação **Remove()** sobre a **interface Home ou Remote**. Ele chama a operação **ejbremove()** sobre a implementação do bean.

Ciclo de Vida de um Stateless Session Bean governado pelas políticas do Container

1. newInstance()
2. setSessionContext()
3. ejbCreate()



Método de Negócio



ejbActivate()

- Notifica o session bean para que ele seja ativado.



ejbPassivate()

- Notifica o session bean que ele é para ser desativado.



Ativação e Desativação

- Tais notificações permitem implementações avançadas de beans, no sentido de gerenciar recursos, que os beans podem controlar.



Descritor de disponibilização (Deployment)

- Um aplicativo EJB precisa ter um **descritor de disponibilização** que descreva cada enterprise bean naquele aplicativo.
- O arquivo descritor de disponibilização é chamado **ejb-jar.xml** .



O descritor de disponibilização

```
<?xml version="1.0" encoding="UTF-8"?>

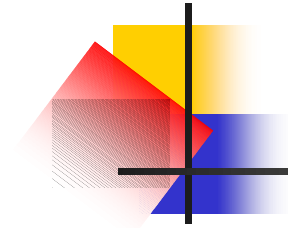
<!DOCTYPE ejb-jar PUBLIC
"-//Sun Microsystems, Inc.//DTD Enterprise JavaBeans 2.0//EN"
"http://java.sun.com/dtd/ejb-jar_2_0.dtd">

<ejb-jar>
  <description>Your first EJB application</description>
  <display-name>Adder Application</display-name>
  <enterprise-beans>
    <session>
      <ejb-name>Adder</ejb-name>
      <home>com.branysoftware.ejb.AdderHome</home>
      <remote>com.branysoftware.ejb.Adder</remote>
      <ejb-class>com.branysoftware.ejb.AdderBean</ejb-class>
      <session-type>Stateless</session-type>
      <transaction-type>Bean</transaction-type>
    </session>
  </enterprise-beans>
</ejb-jar>
```



Criando um arquivo de disponibilização

- Depois de desenvolver seu enterprise bean, precisamos empacotar todos os arquivos de classe (* .class) em um arquivo .jar .



- Use a estrutura de diretório mostrada anteriormente.
- O diretório **ejb** conterá três arquivos:
AdderHome.class,
Adder.class,
AdderBean.class
- O diretório META-INF conterá o arquivo **ejb-jar.xml**



Siga estas etapas para criar o arquivo de distribuição:

- 1. Vá para os diretórios **com** e **META-INF**
- 2. Supondo que **jar.exe** já existe no caminho, digite o seguinte:

```
jar -cfv adder.jar  
    com/brainysoftware/ejb/*  
    META-INF/ejb-jar.xml
```

- Isto cria uma arquivo **.jar** chamado **adder.jar**



Disponibilização

- Copie o arquivo adder.jar na pasta de distribuição do JBOSS.
- Reinicie JBOSS.
- JBOSS deve chuviscar algumas mensagens ...



JBoss ... executando ...

```
...
[Auto  deploy]    Starting
[Auto  deploye]   Auto  deploy of
                  file:/C:/jboss/deploy/adder.jar
...
[J2EE  Deployer]  Create  application  adder.jar
[J2EE  Deployer]  Installing EJB  package:  adder.jar
[J2EE  Deployer]  Starting  module  adder.jar
...
[J2EE  Deployer[  J2EE  application;
                  file:/C:/jboss/deploy/adder.jar
is deployed.
```

O aplicativo EJB foi distribuído com sucesso.



JBoss Application Server

- <http://www.jboss.org/products/jbossas>



Como escrever aplicativos-clientes

- Um enterprise bean é um componente do lado do servidor, aguardando por chamadas de clientes.
- Entender como um cliente obtém acesso.
- Entender como um cliente chama o seu bean do aplicativo EJB.

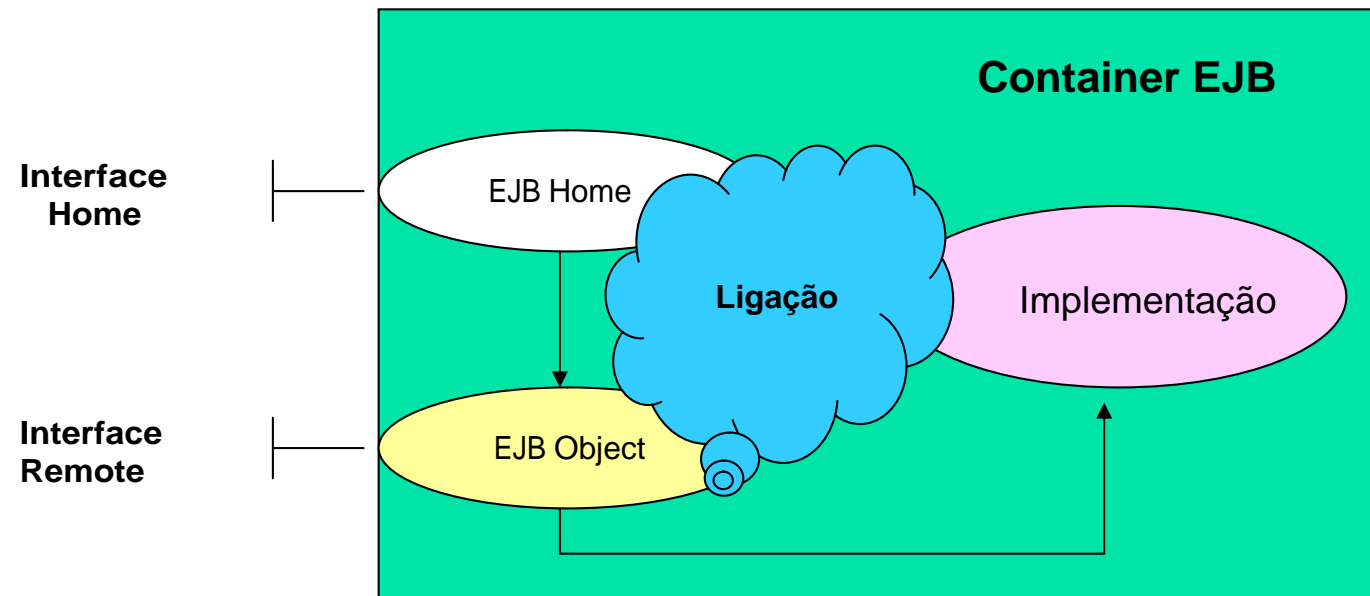


Como escrever aplicativos-clientes

- Um cliente EJB não chama métodos, diretamente, na classe do enterprise bean.
- Um cliente só pode exercer as interfaces **Home** e **Remote** do bean.

Componentes de um EJB Bean

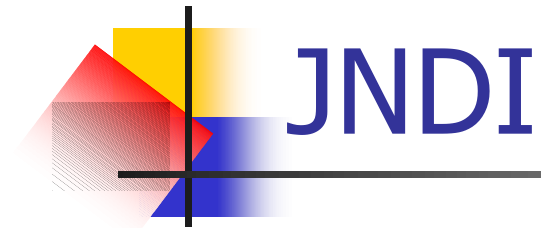
- EJB Home, EJB Remote e a Ligação são gerados pelo Container.



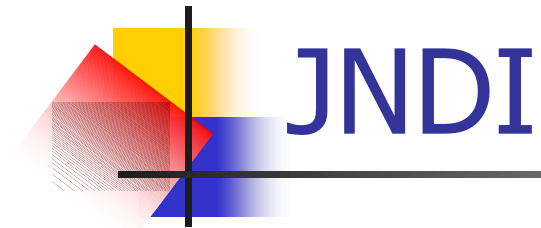


Como escrever aplicativos-clientes

- Os aplicativos-clientes acessam os beans através da API JNDI (Java Naming and Directory Interface).
- Exemplos de Clientes:
 - (a) Aplicativo Java de apenas 1 classe.
 - (b) Uma página JSP que mostra como usar um bean a partir de um servlet.



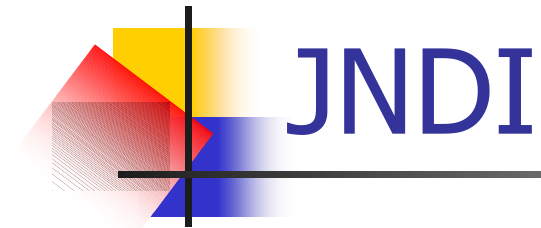
- JNDI oferece dois serviços:
 1. **Serviço de Nomeação,**
 2. Serviço de Diretório,
(não usado em EJB).



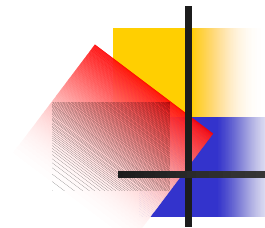
- **Serviço de Nomeação:**

Encontra um objeto associado a um determinado nome.

- Em EJB, um enterprise bean é encontrado, se for fornecido o nome dado ao bean.

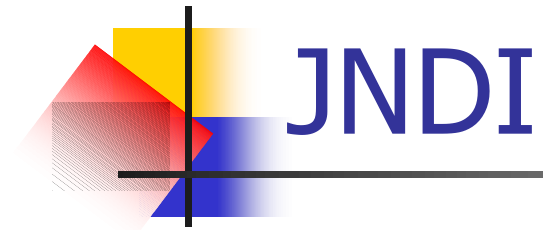


- **Serviço de Diretório:**
É uma extensão para o Serviços de Nomeação.
- Um **Serviço de Diretório** associa nomes com objetos, mas também permite aos objetos terem propriedades que descrevem os objetos.



JNDI

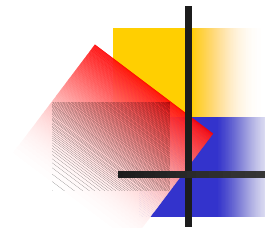
- Serviço de Diretório
Esse serviço permite que se encontre objetos sem conhecer o seu nome.
- Neste caso, os objetos são encontrados através das suas propriedades.



- **javax.naming**
- javax.naming.directory
- javax.naming.event
- javax.naming.ldap
- javax.naming.spi

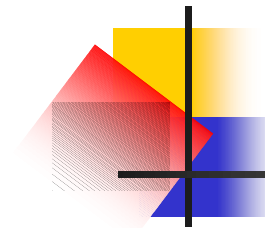


- Ao escrever um **aplicativo-cliente** que use o serviço de um enterprise java bean, só precisamos entender dois membros do pacote **java.naming**: a **interface Context** e a **classe InitialContext**.



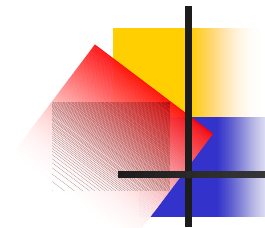
JNDI

- A classe **InitialContext** implementa a interface **Context**.
- A interface `java.naming.Context` representa um **contexto de nomeação**.



JNDI

- Um **contexto de nomeação** é um conjunto de **associações nome-a-objeto**.
- Uma associação em termos de JNDI é chamada de **binding**.



JNDI

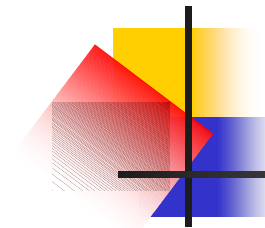
- A **interface Context** contém métodos para examinar e modificar as associações nome-a-objeto.
- O método usado com mais frequência é o método **lookup()**.



JNDI

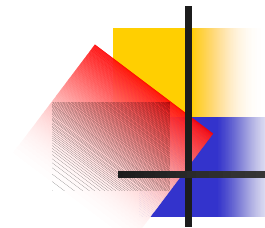
- **lookup()** retorna, dado um nome de objeto, uma referência a um objeto.

- `public Object`
 `lookup(javax.naming.Name, name)`
 throws
 `javax.naming.NamingException`



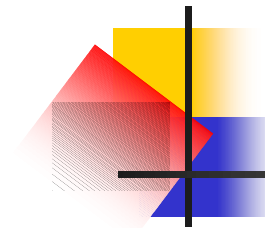
JNDI

- `public Object
lookup(String name)
throws
javax.naming.NamingException`
- `lookup()` é usado para se obter uma **referência a um objeto Home** do enterprise java bean.



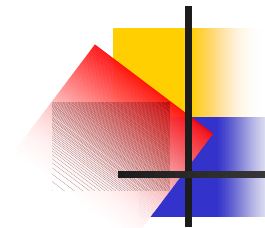
JNDI

- O método **lookup()** lança uma exceção, representada pelo objeto `javax.naming.NamingException`, caso a resolução de nomes venha a falhar.



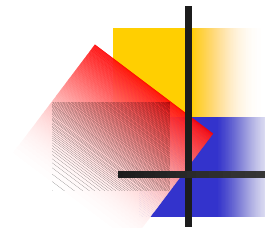
JNDI

- As operações de nomeação são relativas a um **contexto**.
- A **classe InitialContext** implementa a **interface Context** e fornece, assim, o **contexto de início** para o processo de resolução de nomes.



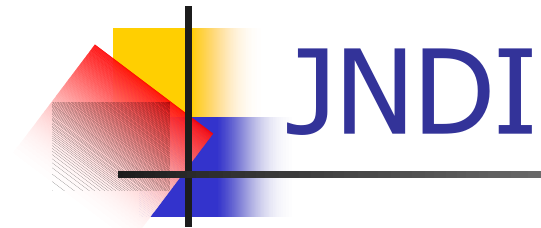
JNDI

- Para um **contexto**, é preciso definir um conjunto de **propriedades** para o **ambiente do contexto**.
- Como exemplo: uma resolução de nomes pode ser restrita a um grupo de usuários autorizados.



JNDI

- Neste caso, as propriedades representam as credenciais de um usuário, e essas precisam ser fornecidas.
- Usa-se então o objeto `java.util.Properties`, com seu método `put()` para fornecer pares `(chave, valor)` representando as credenciais de um usuário.



- Ao acessar um enterprise java bean, precisamos fornecer duas propriedades:

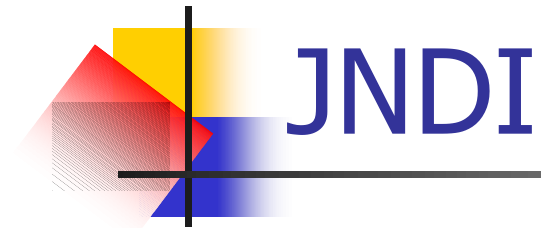
(a) propriedade de ambiente,

`java.naming.factory.initial`

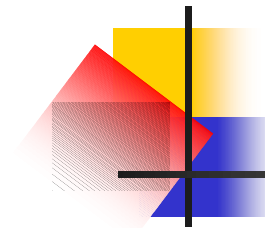
cujo valor é o nome qualificado da

classe que será usada para criar o

contexto inicial.



- A segunda **propriedade** é `java.naming.provider.url` que será uma **propriedade de ambiente**, cujo valor fornece informações de **configuração** para uso do **serviço do provedor EJB**.



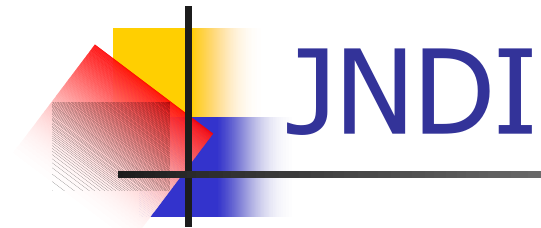
JNDI

- Estas duas propriedades estão presentes na Interface Context:
 - Context.INITIAL_CONTEXT_FACTORY
 - Context.PROVIDER_URL



JNDI

- Para acessar um enterprise bean, é necessário adicionar estas duas propriedades ao pacote `java.util.Properties`, usando o método `put()`, como mostrado no seguinte código:



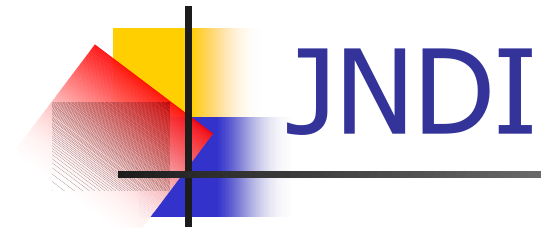
```
import java.util.Properties;

//Create a java.util.Properties object
    Properties properties = new Properties();

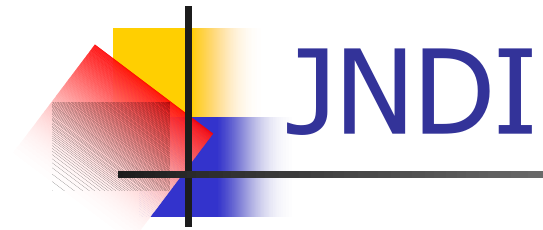
//Add two properties: "java.naming.factory.initial"
//e "java.naming.provider.url"

    properties.put(Context.INITIAL_CONTEXT_FACTORY,
        "org.jnp.interfaces.NamingContextFactory");

    properties.put(Context.PROVIDER_URL,
        "localhost:1099");
```

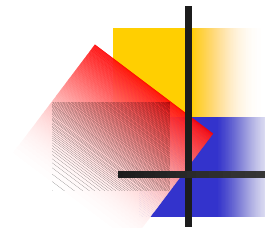


- Porque estas propriedades são necessárias ao criar um contexto inicial, o objeto `Properties` é passado ao construtor da classe `InitialContext`, como no código seguinte:



```
// Get an initial context
```

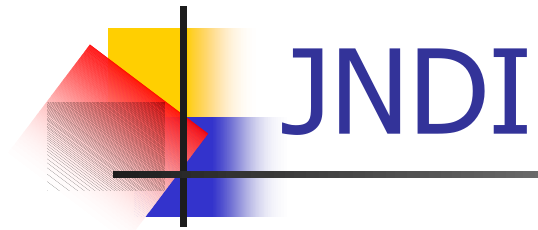
```
InitialContext jndiContext = new  
    InitialContext(properties);
```



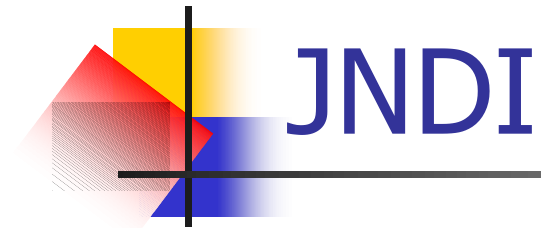
JNDI

- Usar o método lookup() da interface javax.naming.Context para obter uma referência ao objeto Home do bean, passando o nome do bean:

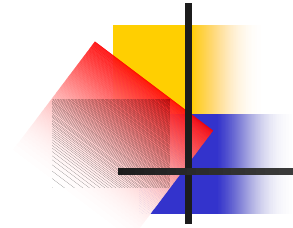
```
// Get a reference to the Bean
Object ref =
    jndiContext.lookup( "Adder" );
```



- Usando JNDI , consegue-se obter uma referência ao objeto Home do bean.



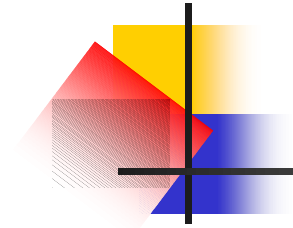
- Mas, a referência obtida do objeto Home é um objeto RMI do tipo `java.lang.Object` e para chamar o método `create()` do objeto Home, primeiro é preciso fazer o “cast” como segue:



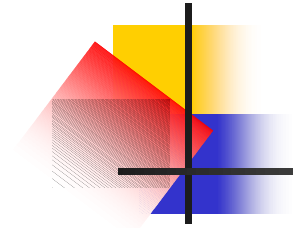
```
// Get a reference from this to the  
    Bean's Home interface
```

```
AdderHome home =(AdderHome) ref
```

onde, `ref` é a referência obtida da busca do nome do contexto inicial de JNDI.

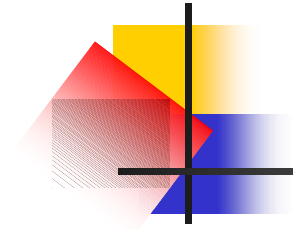


- Para estar em conformidade com o RMI-IIOP, é preciso usar um outro método para o “cast” explicado antes.
- Assim, usa-se o método estático `narrow()` da classe `java.rmi.PortableRemoteObject`.

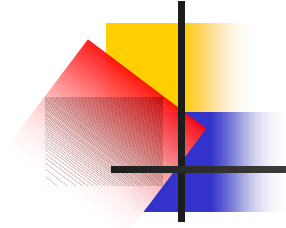


- Esse método é usado para garantir que um objeto de um tipo de interface remota ou abstrata possa ser classificado para um tipo desejado.
- A assinatura deste método é:

```
public static Object  
    narrow(Object narrowFrom,  
          Class narrowTo) throws  
          ClassCastException
```

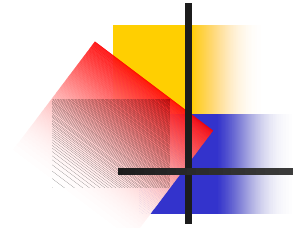


- Onde `narrowFrom` é o objeto a ser verificado e `narrowTo` é o objeto desejado.



- Se `narrowFrom` não puder ser convertido para `narrowTo`, então o método retornará um objeto do tipo `Object`, que poderá ser convertido para o tipo desejado.

```
AdderHome home =(AdderHome)  
    PortableRemoteObject.narrow (ref,  
                                AdderHome.class);
```

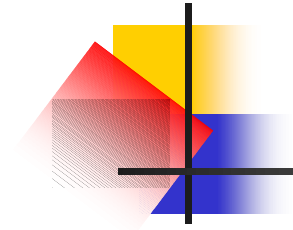


- A etapa seguinte é construir uma cópia do bean no servidor, usando-se o método `create()` da interface `Home` do bean.

```
// Create an Adder object from the Home  
interface
```

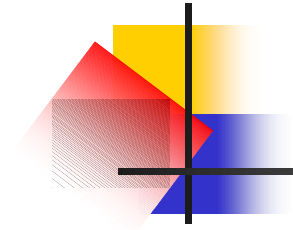
```
Adder adder = home.create();
```

```
System.out.println ("2 + 5 = " +  
    adder.add(2, 5));
```



- Agora pode-se chamar quaisquer métodos do bean.
- Para o exemplo,

```
adder.add(2, 5);
```

- No que segue é mostrada a implementação de uma classe chamada `BeanClient` que implementa o aplicativo-cliente para testar o `AdderBean` do exemplo.



Um Aplicativo-Cliente

```
import javax.naming.*;  
import javax.rmi.PortableRemoteObject;  
import java.util.Properties;  
import com.brainysoftware.ejb.Adder;  
Import com.brainysoftware.ejb.AdderHome;
```



Um Aplicativo-Cliente

```
public class BeanClient {  
  
    public static void main(String[] args) {  
  
        // preparing properties for constructing  
        an InitialContext object  
        Properties properties = new Properties();  
  
        properties.put(Context.INITIAL_CONTEXT_FACTORY,  
            "org.jnp.interfaces.NamingContextFactory");  
  
        properties.put(Context.PROVIDER_URL,  
            "localhost:1099");  
    }  
}
```



Um Aplicativo-Cliente

```
try {  
  
    // Get an initial context  
    InitialContext jndiContext = new  
        InitialContext(properties);  
  
    System.out.println("Got context");  
  
    // Get a reference to the Bean  
    Object ref = jndiContext.lookup("Adder");  
  
    System.out.println("Got reference");  
}
```



Um Aplicativo-Cliente

```
// Get a reference from this to the  
Bean's Home interface
```

```
AdderHome home =(AdderHome)  
    PortableRemoteObject.narrow (ref,  
                                AdderHome.class);
```



Um Aplicativo-Cliente

```
// Create an Adder object from the Home
    interface
Adder adder = home.create();
        System.out.println ("2 + 5 = " +
            adder.add(2, 5));
    }
    catch(Exception e) {
        System.out.println(e.toString());
    }
}
}
```