VŠB TECHNICKÁ UNIVERZITA OSTRAVA

www.vsb.cz

All Java platforms consist of a Java Virtual Machine (VM) and an application programming interface (API).
The Java Virtual Machine is a program, for a particular hardware and software platform, that runs Java technology applications.
An API is a collection of software components that you can use to create other software components or applications.

Each Java platform provides a virtual machine and an API, and this allows applications written for that platform to run on any compatible system with all the advantages of the Java programming language: platform-independence, power, stability, ease-of-development, and security.

Java standard edition include java language and Java SE API as fundamental parts. Main future besides running standard desktop is running java program in web page like flash application. This functionality is realized by technology java applet or java web start.
Java SE themselves is fundamental part of all other java platforms.

Java FX platform is designed for creating rich internet application like Adobe Flash. First version of that platform contains JavaFX script language for defining rich graphic interface and behavior bound to that interface. But from version 2.0 Oracle scrapped scripting language and JavaFX is more like java library for creating rich graphics application that is usable almost with all java platforms.

Java EE is designed for developing and running large-scale, multi-tiered application. In most case is used for web based enterprise application. That platform contains, as part of specification, frameworks and technology for developing web application, like servlets, JavaServerPages, JavaServerFaces developed by Sun (Oracle) and many others like Apache Struts 2, Apache Wicket, IceFaces, JBoss Seam, Oracle ADF, Spring, Vaadin developed by others companies.

Java ME is designed for running application on small devices like mobile phones or smart card or any other devices, where was implemented java virtual machine for Java ME. In these days Java ME is used in older mobile phones but on smart phones with Windows OS or iOS is not too popular. Some kind of resurrection of Java for mobile phones come with Android OS, where application is developed in Java language with API based on Java ME API, on the other hand result application is not in java bytecode but in Google Android proprietary format.

When most people think of the Java programming language, they think of the Java SE API. Java SE's API provides the core functionality of the Java programming language. It defines everything from the basic types and objects of the Java programming language to high-level classes that are used for networking, security, database access, graphical user interface (GUI) development, and XML parsing.

In fact Java language is one of part Java SE platform. Language is defined by keywords and syntax and is compiled to Java bytecode with javac compiler.

Other parts of Java SE platform are:

- Java SE API (Application Programming Interface) contains libraries of classes that provide various functionalities. From basic types and object to high level classes for networking, security, database access, GUI, XML and others.
- Virtual Machine that provide unified virtual environment to run Java programs on different hardware and software platforms.
- Set of development tools and utilities to perform monitoring, testing, optimization and others important tasks.
- Deployment Technology contains ways how to run developed software on client computer. In Java SE platform we can run desktop application directly using java runtime environment or using web browser as java applet in secured environment with no access to client computer. Last way of deployment is Java Web Start technology that combines two previous ways and java program can be run from computer desktop in secured way and program is downloaded and cached from server.

As mentioned in previous said Java FX platform is designed for creating rich internet application. Newest version don’t use script language but use Java programing language and JavaFX API to define application GUI and behavior of GUI. GUI is created as tree of component called scene graph. Any part of tree component can be modified (move, rotate, …). Graphical style (color, font, shadow) of component can be defined directly in Java code or by using CSS styles similar as HTML pages.

The Java EE platform is built on top of the Java SE platform. The Java EE platform provides:

- Huge API with support for internet communication, HTTP connectivity, Web development, data persistency, …
- Runtime environment often called JavaEE container that provide server side runtime environment often included as part of web server.

We discuss this platform later in that course.

Java ME platform is designed to run java application on devices with limited resources (memory, small display) like mobile phones, game consoles, smart TV.

Basically Java ME platform mainly API is subset of Java SE API. Most noticeable different is in graphics user interfaces, because embedded device often has very small display like pager.

Because variety of devices (memory size, processor speed, display, input devices) there exist configurations and profiles. That profiles and configuration defines different capabilities of java ME platform on different devices.

http://en.wikipedia.org/wiki/Application_server

http://en.wikipedia.org/wiki/ActiveX

http://www.cs.vsb.cz/benes/vyuka/pte/prednasky/07-javabeans.pdf

1.1 Component granularity

There are a range of different kinds of JavaBeans components:

1. Some JavaBean components will be used as building blocks in composing applications. So a user may be using some kind of builder tool to connect together and customize a set of JavaBean components s to act as an application. Thus for example, an AWT button would be a Bean.
2. Some JavaBean components will be more like regular applications, which may then be composed together into compound documents. So a spreadsheet Bean might be embedded inside a Web page.

Portability

One of the main goals of the JavaBeans architecture is to provide a platform neutral component architecture. When a Bean is nested inside another Bean then we will provide a full functionality implementation on all platforms. However, at the top level when the root Bean is embedded in some platform specific container (such as Word or Visual Basic or ClarisWorks or Netscape Navigator) then the JavaBeans APIs should be integrated into the platform's local component architecture.

Beans v. Class Libraries

Not all useful software modules should necessarily turn into beans. Beans are appropriate for software components that can be visually manipulated and customized to achieve some effect. Class libraries are an appropriate way of providing functionality that is useful to programmers, but which doesn't benefit from visual manipulation. So for example it makes sense to provide the JDBC database access API as a class library rather than as a bean, because JDBC is essentially a programmatic API and not something that can be directly presented for visual manipulation. However it does make sense to write database access beans on top of JDBC. So for example you might write a "select" bean that at customization time helped a user to compose a select statement, and then when the application is run uses JDBC to run the select statement and display the results

Design time v. run-time

Each Java Bean component has to be capable of running in a range of different environments. There are really a continuum of different possibilities, but two points are particularly worth noting. First a bean must be capable of running inside a builder tool. This is often referred to as the *design environment. Within this design environment it is very important that the bean should* provide design information to the application builder and allow the end-user to *customize the* appearance and behaviour of the bean. Second, each bean must be usable at run-time within the generated application. In this environment there is much less need for design information or customization. The design time information and the design time customization code for a component may potentially be quite large. For example, if a component writer provides a "wizard" style customizer that guides a user through a series of choices, tthe run-time code for the bean. We therefore wanted to make sure that we have a clear split between the design-time aspects of a bean and the run-time aspects, so that it should be possible to deploy a bean at run-time without needing to download all its design time code. So, for example, we allow the design time interfaces (described in chapters 8 and 9) to be supported in a separate class from the run-time interfaces (described in the other chapters).hen the customization code may easily dwarf

Security Issues

Java Beans are subject to the standard Java security model. We have neither extended nor relaxed the standard Java security model for Java Beans. Specifically, when a Java Bean runs as part of an untrusted applet then it will be subject to the standard applet security restrictions and won't be allowed to read or write arbitrary files, or to connect to arbitrary network hosts. However when a Java Bean runs as part of a stand-alone Java application, or as part of a trusted (signed) applet, then it will be treated as a normal Java application and allowed normal access to files and network hosts. In general we advise Java Bean developers to design their beans so that they can be run as part of untrusted applets. The main areas where this shows up in the beans APIs are:

• *Introspection. Bean developers should assume that they have unlimited access to the* high level Introspection APIs (Section 8) and the low-level reflection APIs in the design-time environment, but more limited access in the run-time environment. For

example, the standard JDK security manager will allow trusted applications access to even private field and methods, but will allow untrusted applets access to only public fields and methods. (This shouldn't be too constraining - the high-level Introspection

APIs only expose "public" information anyway.) • *Persistence. Beans should expect to be serialized or deserialized (See Section 5) in both*

the design-time and the run-time environments. However in the run-time environment, the bean should expect the serialization stream to be created and controlled by their parent application and should not assume that they can control where serialized data is read from or written to. Thus a browser might use serialization to read in the initial state for an untrusted applet, but the applet should not assume that it can access random files.

• *GUI Merging. In general untrusted applets will not be permitted to perform any kind of* GUI merging with their parent application. So for example, menubar merging might occur between nested beans inside an untrusted applet, but the top level menubar for the untrusted applet will be kept separate from the browser's menubar.

None of these restrictions apply to beans running as parts of full-fledged Java applications, where the beans will have full unrestricted access to the entire Java platform API.

What should be saved

When a bean is made persistent it should store away appropriate parts of its internal state so that it can be resurrected later with a similar appearance and similar behaviour. Normally a bean will store away persistent state for all its exposed properties. It may also store away additional internal state that is not directly accessible via properties. This might include (for example) additional design choices that were made while running a bean Customizer (see Section 5) or internal state that was created by the bean developer.

A bean may contain other beans, in which case it should store away these beans as part of its internal state.

However a bean should not normally store away pointers to external beans (either peers or a parent container) but should rather expect these connections to be rebuilt by higher-level software. So normally it should use the "transient" keyword to mark pointers to other beans or to event listeners. In general it is a container's responsibility to keep track of any inter-bean wiring it creates and to store and resurrect it as needed.

For the same reasons, normally event adaptors should mark their internal fields as "transient".

on a**Invisible beans**

Many Java Beans will have a GUI representation. When composing beans with a GUI application builder it may often be this GUI representation that is the most obvious and compelling part of the beans architecture. However it is also possible to implement invisible beans that have no GUI representation. These beans are still able to call methods, fire events, save persistent state, etc. They will also be editable in a GUI builder using either standard property sheets or customizers (see Chapter 9). They simply happen to have no screen appearance of their own. Such invisible beans can be used either as shared resources within GUI applications, or as components in building server applications that have no GUI appearance at all. These invisible beans may still be represented visually in an application builder tool, and they may have a GUI customizer that configures the bean. Some beans may be able to run either with or without a GUI appearance depending where they are instantiated. So if a given bean is run in a server it may be invisible, but if it is run user’s desktop it may have a GUI appearance.

If illustrate all in class diagram from language UML the situation is as follows. Class FooEvent represents individual events and must be a subclass of class java.util.EventObject. An important feature of this class is that it have to contains pointer to object, that is source of events. So it must be pass as parameter in constructor and can not be NULL, otherwise a run-time exception occurs.

Furthermore, there must be FooListener interface, whose methods determine what events listeners will receive. These methods must have a first parameter of class FooEvent. This interface must derive from interface java.util.EventListener. Specific listeners (MyListener, MyOtherListener) must then implement this interface.

EventSource the class must contain methods addFooListener, removeFooListener or getFoolisteners. A good habit is also create a method fireSomethingHappend to distribute a specific event to all listeners. In this class we must not forget the collection of all registered listeners (collection of object which implements interface FooListener) to which an individual listeners are added or removed using the methods addFooListener and removeFooListener.

Animation description:

At the beginning we have an object that is the source of some events and any listeners. The first listener, object of class MyListener, calls the method addFooListener to the source and pass pointer to itself (this) as method parameter. Source stores this pointer in the list of listeners. The same steps will be followed for the second object of class MyOtherListener.

If an event occurs, the object of class Source calls himself method fireSomethingHappend. This method creates a object of class FooEvent that sends as a parameter of method somethingHappend to all listeners whose pointers are stored in the list of listeners.

FooListener class code may look as follows. Most important is the constructor, the first parameter must have an object that represents a source of events. This object have to by passed to constructor of superclass at first line of method.

If you need a class can contain additional information about the event as in our case, textual information (read property MyInfo).

Description FooListener interface is straightforward, we must not only forget about parameter of type FooEvent in each method.

In our case, the interface has three methods that represent three different events that can occur and the listener will be informed about them.

The largest source code is for class EventSource, which must contain a definition of collection for storing a listeners. Furthermore, methods addFoolistener and removeFoolistener, which is used for adding and removing objects to and from this collection.

It is also a good practice to create a method fireSomethingHappned that iterate over the collection of listeners and call method somethingHappend to all object in this collection.

So if we have component counter, which generate two type of events: valueChanged and counterReset, we have to created CounterListener interface with methods counterReset and valueChanged. Adapter for this interface is very simple, it is a class that implements the interface CounterListener and implementation of both methods are empty.

Bound properties

Sometimes when a bean property changes then either the bean's container or some other bean
may wish to be notified of the change.
A component can choose to provide a change notification service for some or all of its properties.
Such properties are commonly known as *bound properties, as they allow other components*
to bind special behaviour to property changes.
The *PropertyChangeListener event listener interface is used to report updates to simple bound*
properties. If a bean supports bound properties then it should support a normal pair of multicast
event listener registration methods for PropertyChangeListeners:
public void addPropertyChangeListener(PropertyChangeListener x);
public void removePropertyChangeListener(PropertyChangeListener x);
When a property change occurs on a bound property the bean should call the PropertyChange-
Listener.propertyChange method on any registered listeners, passing a *PropertyChangeEvent*

object that encapsulates the locale-independent name of the property and its old and new values.
The event source should fire the event *after updating its internal state.*
For programming convenience, we provide a utility class PropertyChangeSupport that can be
used to keep track of PropertyChangeListeners and to fire PropertyChange events.

Bound properties

Sometimes when a bean property changes then either the bean's container or some other bean
may wish to be notified of the change.
A component can choose to provide a change notification service for some or all of its properties.
Such properties are commonly known as *bound properties, as they allow other components*
to bind special behaviour to property changes.
The *PropertyChangeListener event listener interface is used to report updates to simple bound*
properties. If a bean supports bound properties then it should support a normal pair of multicast
event listener registration methods for PropertyChangeListeners:
public void addPropertyChangeListener(PropertyChangeListener x);
public void removePropertyChangeListener(PropertyChangeListener x);
When a property change occurs on a bound property the bean should call the PropertyChange-
Listener.propertyChange method on any registered listeners, passing a *PropertyChangeEvent*

object that encapsulates the locale-independent name of the property and its old and new values.
The event source should fire the event *after updating its internal state.*
For programming convenience, we provide a utility class PropertyChangeSupport that can be
used to keep track of PropertyChangeListeners and to fire PropertyChange events.

Bound properties

Sometimes when a bean property changes then either the bean's container or some other bean may wish to be notified of the change.

A component can choose to provide a change notification service for some or all of its properties.

Such properties are commonly known as *bound properties, as they allow other components* to bind special behaviour to property changes.

The *PropertyChangeListener event listener interface is used to report updates to simple bound* properties. If a bean supports bound properties then it should support a normal pair of multicast event listener registration methods for PropertyChangeListeners:

public void addPropertyChangeListener(PropertyChangeListener x);

public void removePropertyChangeListener(PropertyChangeListener x);

When a property change occurs on a bound property the bean should call the PropertyChange-Listener.propertyChange method on any registered listeners, passing a *PropertyChangeEvent*

object that encapsulates the locale-independent name of the property and its old and new values.

The event source should fire the event *after updating its internal state.*

For programming convenience, we provide a utility class PropertyChangeSupport that can be used to keep track of PropertyChangeListeners and to fire PropertyChange events.

```
private PropertyChangeSupport changes = new PropertyChangeSupport(this);
private VetoableChangeSupport vetos = new VetoableChangeSupport(this);
```

AWT Abstract Window Toolkit (import java.awt.*)

- Included in JDK from first version of Java language, basic build blocks for creating complex user interface. Design of AWT use many design patters (most significant design pattern - Model-View-Controller).

- Dynamic layout management of visual components – When Java was introduced to the market, was one of its main advantages to create a GUI that automatically magnified and shrunk the individual elements of the window depending on window size. This functionality was not easily accessible in the other languages and technologies.

Swing (import javax.swing.*)

Extends existing AWT toolkit, included in JDK from version 2, contains lot of new components, standard dialog windows, Look & Feel. Use AWT classes as parent classes and also use many design patterns.

Component are designed as lightweight, that means, that appearance and behavior is implemented directly in Java.

Dynamic layout management of visual components – 8 type of basic layout managers, but exist many others.

Part of JFC (Java Foundation Classes).

Support for data transfer (Cut/Copy/Paste, Drag & Drop).

Include Undo Framework (support for Undo a Redo operation).

Internationalization, Accessibility (disclosure of the

JavaFX

The newest technology for creating user interfaces.

Focused on multimedia and easy creation of rich user interfaces.

This technology began as a single platform, but today it is applicable in the form of library.

Allows use of cascading style sheet (CSS) known from creation of the web pages.

Another multiplatform alternative is SWT toolkit from IBM.

Neplést s architekturou MVC

2.17.9 Virtual Machine Exit

The Java virtual machine terminates all its activity and exits when one of two things happens:

All the threads that are not daemon threads (§2.19) terminate.

Some thread invokes the exit method of class Runtime or class System, and the exit operation is permitted by the security manager.

A program can specify that all finalizers that have not been automatically invoked are to be run before the virtual machine exits. This is done by invoking the method runFinalizersOnExit of the class System with the argument true.[4] By default finalizers are not run on exit. Once running finalizers on exit has been enabled it may be disabled by invoking runFinalizersOnExit with the argument false. An invocation of the runFinalizersOnExit method is permitted only if the caller is allowed to exit and is otherwise rejected by the security manager.

Here is a summary of the key SAX APIs:

SAXParserFactory A SAXParserFactory object creates an instance of the parser determined by the system property, javax.xml.parsers.SAXParserFactory.

SAXParser The SAXParser interface defines several kinds of parse() methods. In general, you pass an XML data source and a DefaultHandler object to the parser, which processes the XML and invokes the appropriate methods in the handler object.

SAXReader The SAXParser wraps a SAXReader. Typically, you do not care about that, but every once in a while you need to get hold of it using SAXParser's getXMLReader() so that you can configure it. It is the SAXReader that carries on the conversation with the SAX event handlers you define.

DefaultHandler Not shown in the diagram, a DefaultHandler implements the ContentHandler, ErrorHandler, DTDHandler, and EntityResolver interfaces (with null methods), so you can override only the ones you are interested in.

ContentHandler Methods such as startDocument, endDocument, startElement, and endElement are invoked when an XML tag is recognized. This interface also defines the methods characters() and processingInstruction(), which are invoked when the parser encounters the text in an XML element or an inline processing instruction, respectively.

ErrorHandler Methods error(), fatalError(), and warning() are invoked in response to various parsing errors. The default error handler throws an exception for fatal errors and ignores other errors (including validation errors). This is one reason you need to know something about the SAX parser, even if you are using the DOM. Sometimes, the application may be able to recover from a validation error. Other times, it may need to generate an exception. To ensure the correct handling, you will need to supply your own error handler to the parser.

DTDHandler Defines methods you will generally never be called upon to use. Used when processing a DTD to recognize and act on declarations for an unparsed entity.

EntityResolver The resolveEntity method is invoked when the parser must identify data identified by a URI. In most cases, a URI is simply a URL, which specifies the location of a document, but in some cases the document may be identified by a URN - a public identifier, or name, that is unique in the web space. The public identifier may be specified in addition to the URL. The EntityResolver can then use the public identifier instead of the URL to find the document-for example, to access a local copy of the document if one exists.

A typical application implements most of the ContentHandler methods, at a minimum. Because the default implementations of the interfaces ignore all inputs except for fatal errors, a robust implementation may also want to implement the ErrorHandler methods.

Jak chápete slovo Adresář?

Directory service
From Wikipedia, the free encyclopedia
Jump to: navigation, search
In software engineering, a directory is similar to a dictionary; it enables the look up of a name and information associated with that name. As a word in a dictionary may have multiple definitions, in a directory, a name may be associated with multiple, different pieces of information. Likewise, as a word may have different parts of speech and different definitions, a name in a directory may have many different types of data. Based on this rudimentary explanation of a directory, a **directory service** is simply the software system that stores, organizes and provides access to information in a directory.
Directories may be very narrow in scope, supporting only a small set of node types and data types, or they may be very broad, supporting an arbitrary or extensible set of types. In a telephone directory, the nodes are names and the data items are telephone numbers. In the DNS the nodes are domain names and the data items are IP addresses (and alias, mail server names, etc.). In a directory used by a network operating system, the nodes represent resources that are managed by the OS, including users, computers, printers and other shared resources. Many different directory services have been used since the advent of the Internet but this article focuses mainly on those that have descended from the X.500 directory service.
Adresářová služba je v softwarovém inženýrství aplikace shromažďující a poskytující informace o pojmenovaných objektech, ke kterým bývá intenzivně přistupováno, ale mění se jen zřídka. Informace jsou uloženy ve formě atributů hierarchicky pojmenovaných záznamů (DIT), které jsou pro lepší integraci systémů standardizovány. Adresářová služba je často ústřední bezpečnostní komponenta a udržuje odpovídající záznamy pro řízení přístupu (jakým způsobem může někdo operovat s nějakým objektem).
V adresářové službě mohou být udržovány informace například o uživatelích (telefon, e-mail, pracovní zařazení), tiskárnách či počítačích, ke kterým přistupují ostatní systémy skrze síť.
Za předchůdce lze považovat X.500 protokol DAP (Directory Access Protocol) vytvořený v 70. letech 20. stol. v souvislosti s ISO/OSI modelem.
[editovat] Srovnání s databázemi
Adresářové služby neposkytují pokročilé databázové techniky, jako jsou transakce nebo udržování integrity. Datový model adresářů není normalizován, např. položka „pracovní telefon“ obsahuje dvě tel. čísla nebo pro zvýšení výkonu mohou být vybraná data uložena duplicitně.
[editovat] Soudobé adresářové služby
DNS – specializovaná jmenná služba pro překlad jmen domén a číselných IP adres, spíše předchůdce dnešních adresářových služeb
LDAP (Lightweight Directory Access Protocol) a jeho implementace OpenLDAP
Active Directory – služba ve Windows, od verze Windows 2000 Server
XNS eXtensible Name Service – pro webové služby
UDDI (Universal Description, Discovery, and Integration) – pro webové služby

LDAP (*Lightweight Directory Access Protocol*) je definovaný protokol pro ukládání a přístup k datům na adresářovém serveru. Podle tohoto protokolu jsou jednotlivé položky na serveru ukládány formou záznamů a uspořádány do stromové struktury (jako ve skutečné adresářové architektuře). Je vhodný pro udržování adresářů a práci s informacemi o uživatelích (např. pro vyhledávání adres konkrétních uživatelů v příslušných adresářích, resp. databázích). Protokol LDAP je založen na doporučení X.500, které bylo vyvinuto ve světě ISO/OSI, ale do praxe se ne zcela prosadilo, zejména pro svou „velikost“ a následnou „těžkopádnost“.

Protokol LDAP již ve svém názvu zdůrazňuje fakt, že je „odlehčenou“ (lightweight) verzí, odvozenou od X.500 (X.500 - Mezinárodní standard, vyvinutý spolkem International Consultative Commitee of Telephony and Telegraphy, pro formátování elektronických zpráv přenášených přes sítě nebo mezi počítačovými sítěmi).

Aplikace funguje na bázi klient-server. V komunikaci využívá jak synchronní tak asynchronní mód. Součástí LDAP je autentizace klienta. Při provádění požadavku lze nedokončený požadavek zrušit příkazem abandon.

Adresářová služba

Z Wikipedie, otevřené encyklopedie

Skočit na: Navigace, Hledání

Informační model

Úkolem informačního modelu LDAP je definovat datové typy a informace, které lze v adresářovém serveru ukládat. Data jsou uchovávána ve stromové struktuře pomocí záznamů. Pod pojmem **záznam** si můžeme představit souhrn atributů (dvojice jméno - hodnota). **Atributy** nesou informaci o stavu daného záznamu. Záznamy, uložené v adresáři, musí odpovídat přípustnému schématu. Pod pojmem **schéma** si představme soubor povolených **objektových tříd** a k nim náležících atributů. Z faktu, že každý záznam je instancí objektové třídy, vyplývá, že musí obsahovat všechny atributy vedené u dané objektové třídy jako povinné. Mimo to může obsahovat i atributy nepovinné, nicméně opět musí vybírat pouze z množiny příslušící dané objektové třídě. To je nejlépe vidět na příkladě konkrétní definice objektové třídy, např. třídy person ve schématu (např. v serveru OpenLDAP je tato třída součástí základního schématu core.schema).

Jmenný model

Úkolem jmenného modelu LDAP je definovat, jakým způsobem budou data v adresáři organizována a jak je možné se na ně odkazovat.

Každý záznam musí být jednoznačně identifikovatelný pomocí svého **rozlišovacího jména (DN = Distinguished Name)** v rámci celého stromu serveru. Musí být také jednoznačný pomocí **relativního rozlišovacího jména (RDN = Relative Distinguished Name)** v rámci jedné úrovně větve v adresáři. RDN se skládá ze jména a hodnoty identifikujícího atributu. Není vhodné za RDN považovat např. atribut pro křestní jméno s hodnotou Jana (givenName=Jana), protože nositelů tohoto jména může být na dané úrovni více. Vhodněji vybraným atributem může být např. emailová adresa (atribut mail) nebo uživatelské jméno pro vstup do nějakého systému (atribut uid).

K záznamům přistupujeme pomocí cesty. **Cesta** je synonymem pro výše zmíněné rozlišovací jméno DN. Rozlišovací jméno je závislé na zvoleném sufixu a na poloze záznamu v adresářovém stromu. **Sufix** je část rozlišovacího jména, která je společná všem záznamům, často bývá odvozena od lokality, nebo od internetové domény. To proto, aby zaručila danému adresáři jedinečnost (i v rámci celého světa). Např. sufix patřící firmě ABC by mohl mít následující podobu: dc=abc, dc=cz (dc je povinným atributem objektové třídy dcObject, zastupující komponenty internetové domény). Sufix je ale pouze jednou z částí, pomocí které identifikujeme záznam v rámci adresáře. A zatímco ten je pro každý záznam v adresáři shodný, další část rozlišovacího jména se musí pro každý záznam lišit. Při tvorbě rozlišovacího jména postupujeme **„zdola nahoru“**, na rozdíl od tvorby cest v klasickém adresářové struktuře, kde postupujeme od kořene „shora dolů“. Rozlišovací jméno poskládáme z relativních rozlišovacích jmen předchůdců daného záznamu.

Rozlišovací jméno zaměstnance z adresářového serveru firmy ABC může mít následující podobu: uid=jana.jiraskova, dc=abc, dc=cz. Zatímco pro stejného zaměstnance platí relativní rozlišovací jméno uid=jana.jiraskova.

[editovat]

"cn" for common name, "dc" for domain component, "mail" for e-mail address and "sn" for surname.

Naming and directory services play a vital role in intranets and the Internet by providing network-wide sharing of a variety of information about users, machines, networks, services, and applications.

JNDI is an API specified in Java technology that provides naming and directory functionality to applications written in the Java programming language. It is designed especially for the Java platform using Java's object model. Using JNDI, applications based on Java technology can store and retrieve named Java objects of any type. In addition, JNDI provides methods for performing standard directory operations, such as associating attributes with objects and searching for objects using their attributes.

JNDI is also defined independent of any specific naming or directory service implementation. It enables applications to access different, possibly multiple, naming and directory services using a common API. Different naming and directory service providers can be plugged in seamlessly behind this common API. This enables Java technology-based applications to take advantage of information in a variety of existing naming and directory services, such as LDAP, NDS, DNS, and NIS(YP), as well as enabling the applications to coexist with legacy software and systems.

Using JNDI as a tool, you can build new powerful and portable applications that not only take advantage of Java's object model but are also well-integrated with the environment in which they are deployed.

Directory Concepts

Many naming services are extended with a *directory service*. A directory service associates names with objects and also allows such objects to have *attributes*. Thus, you not only can look up an object by its name but also get the object's attributes or search for the object based on its attributes. An example is the telephone company's directory service. It maps a subscriber's name to his address and phone number. A computer's directory service is very much like a telephone company's directory service in that both can be used to store information such as telephone numbers and addresses. The computer's directory service is much more powerful, however, because it is available online and can be used to store a variety of information that can be utilized by users, programs, and even the computer itself and other computers.

A *directory object* represents an object in a computing environment. A directory object can be used, for example, to represent a printer, a person, a computer, or a network. A directory object contains *attributes* that describe the object that it represents.

Attributes

A directory object can have *attributes*. For example, a printer might be represented by a directory object that has as attributes its speed, resolution, and color. A user might be represented by a directory object that has as attributes the user's e-mail address, various telephone numbers, postal mail address, and computer account information.

An attribute has an *attribute identifier* and a set of *attribute values*. An attribute identifier is a token that identifies an attribute independent of its values. For example, two different computer accounts might have a "mail" attribute; "mail" is the attribute identifier. An attribute value is the contents of the attribute. The email address, for example, might have an attribute identifier of "mail" and the attribute value of "john.smith@somewhere.com".

Directories and Directory Services

A *directory* is a connected set of directory objects. A *directory service* is a service that provides operations for creating, adding, removing, and modifying the attributes associated with objects in a directory. The service is accessed through its own interface. Many examples of directory services are possible. The Novell Directory Service (NDS) is a directory service from Novell that provides information about many networking services, such as the file and print services. Network Information Service (NIS) is a directory service available on the Solaris operating system for storing system-related information, such as that relating to machines, networks, printers, and users. The SunONE Directory Server is a general-purpose directory service based on the Internet standard LDAP.

Searches and Search Filters

You can look up a directory object by supplying its name to the directory service. Alternatively, many directories, such as those based on the LDAP, support the notion of *searches*. When you search, you can supply not a name but a query consisting of a logical expression in which you specify the attributes that the object or objects must have. The query is called a *search filter*. This style of searching is sometimes called *reverse lookup* or *content-based searching*. The directory service searches for and returns the objects that satisfy the search filter. For example, you can query the directory service to find all users that have the attribute "age" greater than 40 years. Similarly, you can query it to find all machines whose IP address starts with "192.113.50".

Combining Naming and Directory Services

Directories often arrange their objects in a hierarchy. For example, the LDAP arranges all directory objects in a tree, called a *directory information tree (DIT)*. Within the DIT, an organization object, for example, might contain group objects that might in turn contain person objects. When directory objects are arranged in this way, they play the role of naming contexts in addition to that of containers of attributes.

JNDI Overview

The Java Naming and Directory Interface™ (JNDI) is an application programming interface (API) that provides naming and directory functionality to applications written using the Java™ programming language. It is defined to be independent of any specific directory service implementation. Thus a variety of directories--new, emerging, and already deployed--can be accessed in a common way.

Architecture

The JNDI architecture consists of an API and a service provider interface (SPI). Java applications use the JNDI API to access a variety of naming and directory services. The SPI enables a variety of naming and directory services to be plugged in transparently, thereby allowing the Java application using the JNDI API to access their services. See the following figure. **Packaging**

The JNDI is included in the Java 2 SDK, v1.3 and later releases. It is also available as a Java Standard Extension for use with the JDK 1.1 and the Java 2 SDK, v1.2. It extends the v1.1 and v1.2 platforms to provide naming and directory functionality. To use the JNDI, you must have the JNDI classes and one or more service providers. The Java 2 SDK, v1.3 includes three service providers for the following naming/directory services:

Lightweight Directory Access Protocol (LDAP)

Common Object Request Broker Architecture (CORBA) Common Object Services (COS) name service

Java Remote Method Invocation (RMI) Registry

Other service providers can be downloaded from the JNDI Web site or obtained from other vendors. When using the JNDI as a Standard Extension on the JDK 1.1 and Java 2 SDK, v1.2, you must first download the JNDI classes and one or more service providers. See the Preparations lesson for details on how to install the JNDI classes and service providers.

The JNDI is divided into five packages:

javax.naming
javax.naming.directory
javax.naming.event
javax.naming.ldap
javax.naming.spi

If we have context (initialization end without any exception) we can work with it. These example code shows look up of object bind with specified name. But the naming context provide functionality for creating new entries, bind them with object, delete entries, unbind objects, rename entry, list all names in current context and others.

Directory context provide extended functionality such as access to attributes stored with an entry. Obtaining directory context is very similar as obtaining standard context. After set of environment variables (same as in case of standard context) it is necessary to create object of class InitialDirContext.

If InitialDirContext is created (no exception occur), it can be used to obtain information same as In case of standard context and moreover it can obtain information about attributes and other directory specific information and functionality. Example code obtain all attributes of entry with distinguish name "cn = Ted Geisel, ou=People" and after that obtain value of attribute "sn“ (sure name – LDAP standard attribute).

There exist only two operation with context in JNDI API. Operation for creating new context and deleting existing context. For renaming context there is no special opperation but application could use method Context.rename() same as for object, because that method rename entry even it si name bound with object or whole context or subcontext.

JNDI directory context provide access method for attributes. These method return special object of type javax.naming.directory.Attributes, that holds collection of attributes and provide acces method like getAll() for obtaining enumerator of all attributes or getIDs() that return enumerator of all names of attributes (no values of attributes).

Because directory services often contains many entries with many attributes, JNDI API provide methods for modification of whole collection of entries or attributes. For example method modifyAttributes can replace, add or remove set of attributes in entry specified by parameter name.

Another way to change set of attributes for specified object use class ModificationItem, that define individual type of operation with individual attribute. With that class application can specific individual type of operation to individual attribute instead same operation to set of attributes as on previous slide.

JNDI Directory context has extended support for searching entries or their attributes. There exist two ways o searching:

- Specifying collection of attributes and their values and directory context search for all entries that have all specified attributes and their values same.
- Application can specify search filter formula, that can be much more complicated.

~= přesný význam se na jednotlivých serverech může lišit. Záleží také na typu atributu. Pro řetězce je například běžné že ~= znamená zní jako, problém pro různé jazyky

Neplést s compositeName

Nutno opravit na základě
http://download.oracle.com/javaee/6/tutorial/doc/gipjf.html

Portable JNDI Syntax
Three JNDI namespaces are used for portable JNDI lookups: java:global, java:module, and java:app.
■ The java:global JNDI namespace is the portable way of finding remote enterprise beans using JNDI lookups. JNDI addresses are of the following form:
java:global[/*application name*]/*module name*/*enterprise bean name*[/*interface name*]
Application name and module name default to the name of the application and module minus the file extension. Application names are required only if the application is packaged within an EAR. The interface name is required only if the enterprise bean implements more than one business interface.
■ The java:module namespace is used to look up local enterprise beans within the same module. JNDI addresses using the java:module namespace are of the following form:
java:module/*enterprise bean name*/[*interface name*]
The interface name is required only if the enterprise bean implements more than one business interface.
■ The java:app namespace is used to look up local enterprise beans packaged within the same
application. That is, the enterprise bean is packaged within an EAR file containing multiple Java EE modules. JNDI addresses using the java:app namespace are of the following form:
java:app[/*module name*]/*enterprise bean name*[/*interface name*]
The module name is optional. The interface name is required only if the enterprise bean implements more than one business interface.
For example, if an enterprise bean, MyBean, is packaged within the web application archive myApp.war, the module name is myApp. The portable JNDI name is java:module/MyBean An equivalent JNDI name using the java:global namespace is java:global/myApp/MyBean.

JDBC Architecture

The JDBC API contains two major sets of interfaces: the first is the JDBC API for application writers, and the second is the lower-level JDBC driver API for driver writers. JDBC technology drivers fit into one of four categories. Applications and applets can access databases via the JDBC API using pure Java JDBC technology-based drivers, as shown in this figure:

Left side, Type 4: Direct-to-Database Pure Java Driver

This style of driver converts JDBC calls into the network protocol used directly by DBMSs, allowing a direct call from the client machine to the DBMS server and providing a practical solution for intranet access.

Right side, Type 3: Pure Java Driver for Database Middleware

This style of driver translates JDBC calls into the middleware vendor's protocol, which is then translated to a DBMS protocol by a middleware server. The middleware provides connectivity to many different databases.

JDBC support four types of drivers:

- JDBC – ODBC bridge driver, that is not driver to real database but bridge to another unified access to database called ODBC which is supported in Windows systems.
- Driver that is partially written in Java and partially in native code of hosted operating system. Such as driver is not portable between different platform.
- Pure Java Driver – that communication directly with database server through the proprietary network protocol.
- Pure Java Driver – that communicate with database middleware server through standardized network protocol.

Cited from:

Types of Drivers

There are many possible implementations of JDBC drivers. These implementations are categorized as follows:

Type 1 - drivers that implement the JDBC API as a mapping to another data access API, such as ODBC. Drivers of this type are generally dependent on a native library, which limits their portability. The JDBC-ODBC Bridge driver is an example of a Type 1 driver.

Type 2 - drivers that are written partly in the Java programming language and partly in native code. These drivers use a native client library specific to the data source to which they connect. Again, because of the native code, their portability is limited.

Type 3 - drivers that use a pure Java client and communicate with a middleware server using a database-independent protocol. The middleware server then communicates the client?s requests to the data source.

Type 4 - drivers that are pure Java and implement the network protocol for a specific data source. The client connects directly to the data source.

Before application connect to database, it must initialize JDBC driver. It is very simple, application just ask for class specified by provider of the driver. These class have some initialization code in static constructor, that is performed when class is loaded to the memory. For some older version of Java, there was an error and new instance of object had to be created to invoke these initialization code.

Class DriverManager provide static method getConnection to establish connection with database. That method has string parameter called “url”. URL parameter contains connection string to database. That connection string consist from prefix “jdbc:” to identify JDBC protocol. And continue with driver identification like “mysql”, “oracle” or “derby”. Rest of connection string has syntax specified by driver provider.

Warning: Even almost all program languages use indexing of arrays form ZERO, indexes connected with JDBC start often with ONE, like indexing of columns in result set.

When result set is returned cursor point before first row and next() method return false if cursor move after last row, that allow easily iterate over all result set row by simple cycle while(resultSet.next()){} (see example code). Class result set provide more methods for moving cursor over result set. But all this method must respect type of result set. If the result set type is TYPE_FORWARD_ONLY the method can't move cursor backward anyway.

Application can read data only form row that is referenced by cursor. Because databases support many data types that not exactly match with java data types, class ResultSet provide methods for reading Java data types, that convert requested data automatically if it is possible. For Example method getString(int), read data from column with index passed as parameter and convert them to string (almost all database data types can be transformed to string).

Data in database can be updated in two ways:

1. Using SQL statement for updating data and method Statement.executeUpdate()
2. Using update support in class ResultSet. If application obtain data from database in result set, the data from individual columns can be modified using method like updateString() or updateInteger() and all changes have to be confirmed by method updateRow(). In same way rows can be deleted or inserted by method .

Some database tables generate unique keys for all newly inserted records.

If application need know these keys it must use following lines of code:

```
stmt.executeUpdate("INSERT INTO autoincSample (column1) VALUES ('Record
1')", Statement.RETURN_GENERATED_KEYS);
rs = stmt.getGeneratedKeys();
```

Application have to pass flag ***RETURN_GENERATED_KEYS*** to method ***executeUpdate***

Application can obtain the keys using method ***getGeneratedKeys()*** and returned result set contains all generated key from executed statement.

Could be used if application repetitively process same statement only with different data.

Usage of prepared statement preserve application against:

SQL injection attack

Bad data transformation to string (application side) and back to proper data type (database side)

SQL statement is send to DBMS and compiled, after that can be processed repetitively (time saving, better security)

Warning: `Index of parameters (paces in SQL statement marked by ?) start with number 1.`

Transaction are important mechanizm of database systems, therefore JDBC has support for transaction.

Each single SQL statement is treated as transaction.

Don’t exist command „BeginTransaction“ it is performed automatically.

If application need more then one statement in one transaction, it have to use method **`setAutoCommit()`** `with attribute false. It switch of committing transaction automatically after each SQL statement. At the end transaction can be committed with method commit() in class Connection.`

HTTP is text protocol for transfer data between web server and client (often web browser). Protocol HTTP use port 80 in most cases.

Client – Server: client sent request to the server and server sent requested data as a response.

HTTP is a protocol from Application Layer (ISO-OSI model) and work over TCP protocol implicitly on port 80.

Each request contains method of request, that can be one of following : **GET, PUT, POST**, HEAD, DELETE, OPTIONS, TRACE, CONNECT. In these days are commonly used only first three methods (get, put, post), other are not used only in some special cases like application of “REST full” services.

Actual version of HTTP protocol is 1.1. That version can use persistent connection (HTTP keep-alive),

Request of HTTP protocol consist from:

- Line that define method of request and requested document on server (path and name within published directories on server) and specification of HTTP protocol version.
- Lines of head that define some important information like user agent (identifier of application which send request), host (name of computer which send request) and other parameters.
- One empty line that separate head from body.
- And body, that is empty in most cases for simple requests. It is used for example in some forms or in case of upload file to web server.

Response of protocol HTTP consist from:

- Line that define protocol and version of protocol. Status code and status message that define what happened on server side(If request was successfully fulfilled or there was some errors or warnings – most know error number 404).
- Head lines contains information about server and about sent content, like mime type of document (text, audio, video, zip file, …) length of sent data and others.
- One empty line that separate head from body.
- Body that contains raw data of sent file. If requested file is HTML page it contains text data, if requested file is image it contains binary data of image.

Protocol contains only plain text data so it is very clear and simple for human user. But it is dangerous send data (password) through that protocol especially if application use method GET that contains sended information directly in requested URL, so it is shown in web browser window. With method POST are data send in request body so they cannot be seen directly in browser but it is no problem capture data on way to server.

In case of sending sensitive information application should use encrypted transfer like HTTPS.

Protocol is stateless: server don’t have permanent connection to client so they cannot be uniquely identified – complication for web application.

How to identify client in secure way, that already pass through authentication?

Bad Solutions:

Transferring identification data in URL and in hidden fields of HTTP forms.

⇒ Τρανσφερρινγ αλλ ιδεντιφιχατιον δατα ιν αλλ ρεθυεστ ισ δανγερουσ.

Χοοκιεσ – Μεχηανισμ φορ στορινγ δατα σεντ βψ σερϖερ ιν βροωσερ. Τηατ δατα αρε αυτοματιχαλλψ σενδ το σερϖερ ιν εαχη ρεθυεστ.

⇒ Στορινγ ανδ τρανσφερρινγ αλλ ιδεντιφιχατιον δατα ιν αλλ ρεθυεστ ισ δανγερουσ εϖεν ωιτη χοοκιεσ.

Described disadvantages led to the introduction of sessions:

An identifier (called session id) is assigned to each new client and on server is stored pair of information the session id and client identification.

Session id is transferred to server with each request using cookies, parameter in URL or hidden form filed.

Advantage: Only this session id is transferred, complete identification is stored on server.

The support of session is important for development of web applications.

The Java EE platform is built on top of the Java SE platform. The Java EE platform provides:

- Huge API with support for internet communication, HTTP connectivity, Web development, data persistency, …
- Runtime environment often called JavaEE container that provide server side runtime environment often included as part of web server.

We discuss this platform later in that course.

The Java EE platform is built on top of the Java SE platform. The Java EE platform provides:

- Huge API with support for internet communication, HTTP connectivity, Web development, data persistency, …
- Runtime environment often called JavaEE container that provide server side runtime environment often included as part of web server.

We discuss this platform later in that course.

The Java EE platform is built on top of the Java SE platform. The Java EE platform provides:

- Huge API with support for internet communication, HTTP connectivity, Web development, data persistency, …
- Runtime environment often called JavaEE container that provide server side runtime environment often included as part of web server.

We discuss this platform later in that course.

Java EE 5 APIs
Java EE 5 illustrates the availability of the Java EE 5 platform APIs in each Java EE container type. The following sections give a brief summary of the technologies required by the Java EE platform, and the APIs used in Java EE applications.
Enterprise JavaBeans
An Enterprise JavaBeans(TM (EJB) component, *or enterprise bean, is a body of code having fields* and methods to implement modules of business logic. You can think of an enterprise bean as a building block that can be used alone or with other enterprise beans to execute business logic on the Java EE server.
There are two types of enterprise beans: session beans and message-driven beans. A *session bean represents a transient conversation with a client. When the client finishes executing, the* session bean and its data are gone. *A message-driven bean combines features of a session bean* and a message listener, allowing a business component to receive messages asynchronously from a JMS queue or topic.
In Java EE 5, entity beans have been replaced by Java persistence API entities. An entity represents persistent data stored in one row of a database table. If the client terminates, or if the server shuts down, the persistence manager ensures that the entity data is saved.
Java ServletTechnology
Java servlet technology lets you define HTTP-specific servlet classes. A servlet class extends the capabilities of servers that host applications that are accessed by way of a request-response programming model. Although servlets can respond to any type of request, they are commonly used to extend the applications hosted by web servers.
JavaServer PagesTechnology
JavaServer PagesTM (JSP) technology lets you put snippets of servlet code directly into a text-based document. Static HTML or SGML markup defines the servlet's response, while static data (which can be expressed in any text-based format such as HTML, WML, and XML) and JSP elements, which determine how the page constructs dynamic content.
JavaServer Pages Standard Tag Library
The JavaServer Pages Standard Tag Library (JSTL) encapsulates core functionality common to many JSP applications. Instead of rroving tags from numerous vendors in your JSP applications, you employ a single, standard set of tags. This standardization allows you to deploy your applications on any JSP container that supports JSTL and makes it more likely that the implementation of your tags will be optimized by web servers.
JSTL has iterator and conditional tags for handling flow control, tags for manipulating XML documents, internationalization tags, tags for accessing databases using SQL, and commonly used functions.
JavaServer Faces
JavaServer Faces technology is a user interface framework for building web applications. The main components of JavaServer Faces technology are as follows:
A GUI component framework.
A flexible architecture for rendering components in different kinds of HTML or different markup languages and technologies. A Renderer object generates the markup to render the component and converts the data stored in a model object to types that can be represented in the UI.
A standard RenderKit for generating HTML/4.01 markup.
The following features support the GUI components:
Validation
Event handling
Data conversion between model objects and components
Managed model object creation
Page navigation configuration
The application is configured using standard Java APIs and XML-based configuration files.
Java Message Service API
The JavaMessage Service (JMS) API is a messaging standard that allows Java EE application components to create, send, receive, and read messages. It enables distributed communication that is loosely coupled, reliable, and asynchronous.
JavaTransaction API
The Java Transaction API (JTA) provides a standard interface for demarcating transactions. The Java EE architecture provides a default auto commit to handle transaction commits and rollbacks. Auto commit means that a transaction is committed or rolled back after each call to the updated data after each database read or write operation.However, if your application performs two separate database access operations that depend on each other, you will wantto use the JTA to create a single unit where the entire transaction, including both operations, begins, rolls back, and commits.
JavaMail API
Many applications use the JavaMailTM API to send email notifications. The JavaMail API has two parts: an application-level interface used by the application components to send mail, and a service provider interface. The Java EE platform includes JavaMail with a service provider that allows application components to send Internet mail.
JavaBeans Activation Framework
The JavaBeans Activation Framework (JAF) is included because JavaMail uses it.JAF provides standard services to determine the type of arbitrary data, to encapsulate access to it, to discover the operations available on it, and create the appropriate JavaBeans components to perform those operations.
Java API for XML Processing
The Java API for XML Processing (JAXP), part of the Java SE platform, supports the processing of XML documents usingDocument ObjectModel (DOM), Simple API for XML (SAX), and eXtensible Stylesheet Language Transformations (XSLT). JAXP lets you parse and validate and transform XML documents independent of a particular XML processing implementation. JAXP also provides namespace support, which lets you work with schemas that might otherwise have naming conflicts.Designed to be flexible, JAXP lets you use any XML-compliant parser or XSL processor from within your application and supports the W3C schema. You can find information about W3C XML Schema at http://www.w3.org/XML/Schema.
Java API for XMLWeb Services (JAX-WS)
The JAX-WS specification provides support for web services that use the JAXB API for binding XML data to Java objects. The JAX-WS specification also provides more sophisticated web services as well as techniques for implementing web service endpoints. The Web Services for J2EE specification describes the deployment of JAX-WS-based services and clients. The EJB and servlet specifications also describe aspects of such deployment. It must be possible to deploy JAX-WS-based applications using any of these deployment models.
JAX-WS 2.0 supports message handlers, which are application-specific components that can process message requests and responses. In general, these message handlers execute in the same container and with the same privileges and execution context as the JAX-WS client or endpoint component to which they are associated. Handlers can be associated with a port, a service, or a qualified java.comjmx namespace as their associated component. Custom serializers and deserializers, if supported, are treated in the same way as message handlers.
Java Architecture for XML Binding
The Java Architecture for XML Binding (JAXB) provides a convenient way to bind an XML schema to a representation in java language programs. JAXB can be used independently or in combination with JAX-WS, where it provides a standard data binding for web service messages. All Java EE application client containers, web containers, and EJB containers support the JAXB API.
SOAP with Attachments API for Java
The SOAP with Attachments API for Java (SAAJ) is a low-level API on which JAX-WS and JAXR depend. It enables developers to produce and consume messages that conform to the SOAP 1.1 specification and SOAP with Attachments note.Most developers do not use the SAAJ API, instead using the higher-level JAX-WS API.
Java API for XML Registries
The Java API for XML Registries (JAXR) lets you access business and general-purpose registries over the Java platform. JAXR provides a unifiedbusiness and Repository standards and the emerging UDDI specifications. By using JAXR, developers can learn a single API and gain access to both of these important registry technologies.
Business Registries provide a place for material to be shared and search for material that others have submitted. Standards groups have developed schemas for particular kinds of XML documents; two businesses might, for example, agree to use the schema for their industry's purchase order documents. Because the schema is stored in a standard business registry, both parties can use JAXR to access it.
J2EE Connector Architecture
The J2EE Connector architecture is used by tools vendors and system integrators to create resource adapters that support access to enterprise information systems that can be plugged-in to any Java EE platform. The connector architecture defines a standardized interface that application components to access and interact with the underlying resource manager of the EIS. Because a resource adapter is specific to its resource manager, typically there is a different resource adapter for each type of database or enterprise information system.
The J2EE Connector architecture also provides a performance-oriented, secure, scalable, and message-based transactional integration of Java EE-based web services with existing EISs that are not Java-based. Integration of existing EISs is important because many business solutions. The J2EE Connector architecture into the Java EE platform can be exposed as XML-based web services by using JAX-WS and Java EE component models. Thus JAX-WS and the J2EE Connector architecture are complementary technologies for enterprise application integration (EAI) and end-to-end business integration.
Java Database Connectivity API
The Java Database Connectivity (JDBC) API lets you invoke SQL commands from java programming language methods. You use the JDBC API in an enterprise bean when you have a session bean access the database. You can also use the JDBC API from a servlet or a JSP page to access a database and process the results in a different way.
The JDBC API has two parts: an application-level interface used by the application components to access a database, and a service provider interface to attach a JDBC driver to the Java EE platform.
Java Persistence API
The Java Persistence API is a Java standards-based solution for persistence. Persistence uses an object-relational mapping approach to bridge the gap between an object oriented model and a relational database. Java Persistence consists of three areas:
The Java persistence API
The query language
Object/relational mapping metadata
Java Naming and Directory Interface
The JavaNaming and Directory InterfaceTM (JNDI) provides naming and directory functionality, enabling applications to access multiple naming and directory services, including existing naming and directory services such as LDAP, NDS,DNS, and NIS. It provides applications with methods for performing standard directory operations, such as associating attributes with objects and searching for objects using those attributes. Using JNDI, a J2EE application can store and retrieve any type of named Java object, allowing Java EE applications to consist with many legacy applications and systems.
Because the J2EE platform manages components, such as enterprise beans, and web components with access to a JNDI naming environment, a naming environment allows a component to be customized without the need to access or change the component's source code. A container implements the component's environment and provides it to the component as a JNDI naming context.
A Java EE component can locate its environment naming context using JNDI interfaces. A component can create a javax.naming.InitialContext object and looks up the environment naming context in InitialContext under the name java.comjenv. A component's naming operations are relative directly in the environment naming context or in any of its direct or indirect subcontexts.
A Java EE component can access named system-provided and user-defined objects. The names of these objects are contained in the Java EE platform's global JNDI namespace, its environment naming context, java.comjenv. The Java EE platform allows a component to name user-defined objects, such as enterprise beans, environment entries, JDBC DataSource objects, and message connections. An object should be named within a subcontext of the naming environment according to the type of the object. For example, enterprise beans are named in the subcontext java.comjenvejbejb-complexnejb, and JDBC DataSource references in the subcontext java.comjenvjdbc.
Java Authentication and Authorization Service
The Java Authentication and Authorization Service (JAAS) provides a way for a Java EE application to authenticate and authorize a specific user or group of users to run it.
JAAS is a java programming language version of the standard Pluggable Authentication Module (PAM) framework, which extends the Java Platform security architecture to support user-based authorization.
Simplified Systems Integration
The Java EE platform is a platform-independent, full systems integration solution that creates an open marketplace in which every vendor can sell to every customer.Such a marketplace benefits customers because it lets them buy the best products and services for their needs instead by trying to outdo each other in providing products and services that benefit customers, such as better performance, better tools, or better customer support.
Java EE 5 APIs provide a standard component architecture through the following:
Unified application model across tiers with enterprise beans
Simplified request-and-response mechanism with JSP pages and servlets
Reliable security model with JAAS
XML-based data interchange integration with JAXP, SAAJ, and JAX-WS
Scalable, transactional, and secure EIS connector architecture
Easy database connectivity with the JDBC API
Enterprise application integration with message-driven beans and JMS, JTA, and JNDI

4-tire application model presented in previous slides is not suitable for all application. The model can be changed based on possible communication way of containers.

For example, model of small simple application can consist from web client (web browser), web container and database.

Java EE 5 APIs
Java EE 5 illustrates the availability of the Java EE 5 platform APIs in each Java EE container type. The following sections give a brief summary of the technologies required by the Java EE platform, and the APIs used in Java EE applications.
Enterprise JavaBeans
An Enterprise JavaBeans(TM (EJB) component, *or enterprise bean, is a body of code having fields* and methods to implement modules of business logic. You can think of an enterprise bean as a building block that can be used alone or with other enterprise beans to execute business logic on the Java EE server.
There are two types of enterprise beans: session beans and message-driven beans. A *session bean represents a transient conversation with a client. When the client finishes executing, the* session bean and its data are gone. *A message-driven bean combines features of a session bean* and a message listener, allowing a business component to receive messages asynchronously from a client. *Session beans can be stateful or stateless.*
In Java EE 5, entity beans have been replaced by Java persistence API entities. An entity represents persistent data stored in one row of a database table. If the client terminates, or if the server shuts down, the persistence manager ensures that the entity data is saved.
Java ServletTechnology
Java servlet technology lets you define HTTP-specific servlet classes. A servlet class extends the capabilities of servers that host applications that are accessed by way of a request-response programming model. Although servlets can respond to any type of request, they are commonly used to extend the applications hosted by web servers.
JavaServer PagesTechnology
JavaServer PagesTM (JSP) technology lets you put snippets of servlet code directly into a text-based document. Static HTML or SGML markup defines the servlet's response, while static data (which can be expressed in any text-based format such as HTML, WML, and XML) and JSP elements, which determine how the page constructs dynamic content.
JavaServer Pages Standard Tag Library
The JavaServer Pages Standard Tag Library (JSTL) encapsulates core functionality common to many JSP applications. Instead of mixing tags from numerous vendors in your JSP applications, you employ a single, standard set of tags. This standardization allows you to deploy your applications on any JSP container that supports JSTL and makes it more likely that the implementation of your JSP code will be optimized.
JSTL has iterator and conditional tags for handling flow control, tags for manipulating XML documents, internationalization tags, tags for accessing databases using SQL, and commonly used functions.
JavaServer Faces
JavaServer Faces technology is a user interface framework for building web applications. The main components of JavaServer Faces technology are as follows:
A GUI component framework.
A flexible architecture for rendering components in different kinds of HTML or different markup languages and technologies. A Renderer object generates the markup to render the component and converts the data stored in a model object to types that can be represented in the UI.
A standard RenderKit for generating HTML/4.01 markup.
The following features support the GUI components:
Validation
Event handling
Data conversion between model objects and components
Managed model object creation
Page navigation configuration
All of these features are configurable using standard Java APIs and XML-based configuration files.
Java Message Service API
The JavaMessage Service (JMS) API is a messaging standard that allows Java EE application components to create, send, receive, and read messages. It enables distributed communication that is loosely coupled, reliable, and asynchronous.
JavaTransaction API
The Java Transaction API (JTA) provides a standard interface for demarcating transactions. The Java EE architecture provides a default auto commit to handle transaction commits and rollbacks. Every SQL statement that affects a database is treated as a transaction and the updated data after each database read or write operation.However, if your application performs two separate database access operations that depend on each other, you will want to use the JTA to create a single unit where the entire transaction, including both operations, begins, rolls back, and commits.
JavaMail API
Java EE applications use the JavaMailTM API to send email notifications. The JavaMail API has two parts: an application-level interface used by the application components to send mail, and a service provider interface. The Java EE platform includes JavaMail with a service provider that allows application components to send Internet mail.
JavaBeans Activation Framework
The JavaBeans Activation Framework (JAF) is included because JavaMail uses it.JAF provides standard services to determine the type of arbitrary data, to encapsulate access to it, to discover the operations available on it, and create the appropriate JavaBeans components to perform those operations.
Java API for XML Processing
The Java API for XML Processing (JAXP), part of the Java SE platform, supports the processing of XML documents usingDocument ObjectModel (DOM), Simple API for XML (SAX), and extensible Stylesheet Language Transformations (XSLT). JAXP lets you parse and validate some and transform XML documents independent of a particular XML processing implementation. JAXP also provides namespace support, which lets you work with schemas that might otherwise have naming conflicts.Designed to be flexible, JAXP lets you use any XML-compliant parser or XSL processor from within your application and supports the W3C schema. You can find information about W3C XML Schema at http://www.w3.org/XML/Schema.
Java API for XMLWeb Services (JAX-WS)
The JAX-WS specification provides support for web services that use the JAXB API for binding Web Services Description Language (WSDL) files and supporting the mapping of Web Services services as well as techniques for implementing web service endpoints. The Web Services for J2EE specification describes the deployment of JAX-WS-based services and clients. The EJB and servlet specifications also describe aspects of such deployment. It must be possible to deploy JAX-WS-based applications using any of these deployment models.
JAX-WS 2.0 supports handlers, which are low-level, transport-independent callbacks that can process message requests and responses. In general, these message handlers execute in the same container and with the same privileges and execution context as the JAX-WS client or endpoint component to which they are associated. Handlers can be associated with a port, or with a port-qualified java:comp/env namespace as their associated component. Custom serializers and deserializers, if supported, are treated in the same way as message handlers.
Java Architecture for XML Binding
The Java Architecture for XML Binding (JAXB) provides a convenient way to bind an XML schema to a representation in java language programs. JAXB can be used independently or in combination with JAX-WS, where it provides a standard data binding for web service messages. All Java EE application client containers, web containers, and EJB containers support the JAXB API.
SOAP with Attachments API for Java
The SOAP with Attachments API for Java (SAAJ) is a low-level API on which JAX-WS and JAX-RPC runtime systems are layered, but it can also be used directly to produce and consume SOAP 1.1 specification and SOAP with Attachments note.Most developers do not use the SAAJ API, instead using the higher-level JAX-WS API.
Java API for XML Registries
The Java API for XML Registries (JAXR) lets you access business and general-purpose registries over the Internet. Business registries, ebXML Registry and Repository standards and the emerging UDDI specifications. By using JAXR, developers can learn a single API and gain access to both of these important registry technologies.
Business registries allow businesses material to be shared and search for material that others have submitted. Standards groups have developed schemas for particular kinds of XML documents; two businesses might, for example, agree to use the schema for their industry's purchase order documents. Because the schema is stored in a standard business registry, both parties can use JAXR to access it.
J2EE Connector Architecture
The J2EE Connector architecture is used by tools vendors and system integrators to create resource adapters that support access to enterprise information systems that can be plugged in to any Java EE platform. The J2EE Connector architecture defines a standard set of application components to access and interact with the underlying resource manager of the EIS. Because a resource adapter is specific to its resource manager, typically there is a different resource adapter for each type of database or enterprise information system.
The J2EE Connector architecture also provides a performance-oriented, secure, scalable, and message-based transactional integration of Java EE-based web services with existing EISs that are not Java technology based. Integration of the J2EE Connector architecture with JAX-WS allows the J2EE Connector architecture into the Java EE platform can be exposed as XML-based web services by using JAX-WS and Java EE component models. Thus JAX-WS and the J2EE Connector architecture are complementary technologies for enterprise application integration (EAI) and end-to-end business integration.
Java Database Connectivity API
The Java Database Connectivity (JDBC) API lets you invoke SQL commands from java programming language methods. You use the JDBC API in an enterprise bean when you have a session bean access the database. You can also use the JDBC API from a servlet or a JSP page to access a database and process the results in a different way than an EJB.
The JDBC API has two parts: an application-level interface used by the application components to access a database, and a service provider interface to attach a JDBC driver to the Java EE platform.
Java Persistence API
The Java persistence API is a Java standards-based solution for persistence. Persistence uses an object-relational mapping approach to bridge the gap between an object oriented model and a relational database. Java Persistence consists of three areas:
The persistence API
The query language
Object/relational mapping metadata
Java Naming and Directory Interface
The JavaNaming and Directory Interface™ (JNDI) provides naming and directory functionality, enabling applications to access multiple naming and directory services, including existing naming and directory services such as LDAP, NIS,DNS, and NIS. It provides applications with methods for performing standard directory operations, such as associating attributes with objects and searching for objects using those attributes. Using JNDI, a application can store and retrieve any type of named Java object, allowing Java EE applications to consist with many legacy applications and systems.
Java EE components, such as enterprise beans, enterprise beans, and web components with access to a JNDI naming environment. A naming environment allows a component to be customized without the need to access or change the component's source code. A container implements the component's environment and provides it to the component as a JNDI naming context.
A Java EE component can locate its environment naming context using JNDI interfaces. A component can create a javax.naming.InitialContext object and looks up the environment naming context in initialContext under the name java:comp/env. A component's naming environment is stored directly in the environment naming context or in any of its direct or indirect subcontexts.
A Java EE component can access named system-provided and user-defined objects. The names of these objects are contained in the Java EE platform's naming context, which is an environment naming context, java:comp/env. The Java EE platform allows a component to name user-defined objects, such as enterprise beans, environment entries, JDBC DataSource objects, and message connections. An object should be named within a subcontext of the naming environment according to the type of the object. For example, enterprise beans are named in the subcontext java:comp/ejb, environment entries in the subcontext java:comp/env, JDBC DataSource references in the subcontext java:comp/jdbc.
Java Authentication and Authorization Service
The Java Authentication and Authorization Service (JAAS) provides a way for a Java EE application to authenticate and authorize a specific user or group of users to run it. JAAS is a java programming language version of the standard Pluggable Authentication Module (PAM) framework, which extends the Java Platform security architecture to support user-based authorization.
Simplified Systems Integration
The Java EE platform is a platform-independent, full systems integration solution that creates an open marketplace in which every vendor can sell to every customer.Such a marketplace benefits customers because it lets them buy the best products and services for their needs instead by trying to outdo each other in providing products and services that benefit customers, such as better performance, better tools, or better customer support.
Java EE 5 APIs provide simplified systems integration through the following:
Unified application model across tiers with enterprise beans
Simplified request-and-response mechanism with JSP pages and servlets
Reliable security model with JAAS
XML-based data interchange integration with JAXP, SAAJ, and JAX-WS
Reliable transaction model with the J2EE Connector architecture
Easy database connectivity with the JDBC API
Enterprise application integration with message-driven beans and JMS, JTA, and JNDI

To deploy a servlet to the java application server it is important include servlet class in a java web module. The web module is a ZIP file with extension “war” and predefined file structure.

Static content of module is included directly in root of module. Classes and other dynamic content are in folder “WEB-INF”. One of the most important file is “web.xml” from folder “WEB-INF” because contains configuration of whole web module.

Java EE Application Assembly and Deployment

A Java EE application is packaged into one or more standard units for deployment to any Java EE platform-compliant system. Each unit contains:

A functional component or components (such as an enterprise bean, JSP page, servlet, or applet)

An optional deployment descriptor that describes its content

Once a Java EE unit has been produced, it is ready to be deployed.Deployment typically involves using a platform's deployment tool to specify location-specific information, such as a list of local users that can access it and the name of the local database. Once deployed on a local platform, the application is ready to run.

Packaging Applications

A Java EE application is delivered in an Enterprise Archive (EAR) file, a standard Java Archive (JAR) file with an .ear extension. Using EAR files and modules makes it possible to assemble a number of different Java EE applications using some of the same components.No extra coding is needed; it is only a matter of assembling (or packaging) various Java EE modules into Java EE EAR files.

An EAR file (see Figure 1–6) contains Java EE modules and deployment descriptors. A *deployment descriptor is an XML document with an .xml extension that describes the* deployment settings of an application, a module, or a component. Because deployment descriptor information is declarative, it can be changed without the need to modify the source code. At runtime, the Java EE server reads the deployment descriptor and acts upon the application, module, or component accordingly.

There are two types of deployment descriptors: Java EE and runtime. A *Java EE deployment descriptor is defined by a Java EE specification and can be used to configure deployment settings* on any Java EE-compliant implementation. A *runtime deployment descriptor is used to* configure Java EE implementation-specific parameters.

For example, the Sun Java System Application Server Platform Edition 9 runtime deployment descriptor contains information such as the context root of a web application, the mapping of portable names of an application's resources to the server's resources, and Application Server implementation-specific parameters, such as caching directives. The Application Server runtime deployment descriptors are named sun-*moduleType.xml and are located in the same META-INF directory as the Java EE* deployment descriptor.

A *Java EE module consists of one or more Java EE components for the same container type and* one component deployment descriptor of that type. An enterprise bean module deployment descriptor, for example, declares transaction attributes and security authorizations for an enterprise bean. A Java EE module without an application deployment descriptor can be deployed as a *stand-alone module.*

The four types of Java EE modules are as follows:

EJB modules, which contain class files for enterprise beans and an EJB deployment descriptor. EJB modules are packaged as JAR files with a .jar extension.

Web modules, which contain servlet class files, JSP files, supporting class files, GIF and HTML files, and a web application deployment descriptor. Web modules are packaged as JAR files with a .war (Web ARchive) extension.

Application client modules, which contain class files and an application client deployment descriptor. Application client modules are packaged as JAR files with a .jar extension.

Resource adapter modules, which contain all Java interfaces, classes, native libraries, and other documentation, along with the resource adapter deployment descriptor. Together, these implement the Connector architecture (see "J2EE Connector Architecture" on page 61) for a particular EIS. Resource adapter modules are packaged as JAR files with an .rar (resource adapter archive) extension.

Web Applications

In the Java 2 platform, *web components provide the dynamic extension capabilities for a web* server. Web components are either Java servlets, JSP pages, or web service endpoints. The interaction between a web client and a web application is illustrated in Figure 3–1. The client sends an HTTP request to the web server. A web server that implements Java Servlet and JavaServer Pages technology converts the request into an HTTPServletRequest object. This object is delivered to a web component, which can interact with JavaBeans components or a database to generate dynamic content. The web component can then generate an HTTPServletResponse or it can pass the request to another web component. Eventually a web component generates a HTTPServletResponse object. The web server converts this object to an HTTP response and returns it to the client.

Servlets are Java programming language classes that dynamically process requests and construct responses. *JSP pages are text-based documents that execute as servlets but allow a more natural* approach to creating static content. Although servlets and JSP pages can be used interchangeably, each has its own strengths. Servlets are best suited for service-oriented applications (web service endpoints are implemented as servlets) and the control functions of a presentation-oriented application, such as dispatching requests and handling nontextual data. JSP pages are more appropriate for generating text-based markup such as HTML, Scalable Vector Graphics (SVG), WirelessMarkup Language (WML), and XML.

Since the introduction of Java Servlet and JSP technology, additional Java technologies and frameworks for building interactive web applications have been developed. Figure 3–2 illustrates these technologies and their relationships.

Notice that Java Servlet technology is the foundation of all the web application technologies, so you should familiarize yourself with the material in Chapter 4, “Java Servlet Technology,” even if you do not intend to write servlets. Each technology adds a level of abstraction that makes web application prototyping and development faster and the web applications themselves more maintainable, scalable, and robust.

Web components are supported by the services of a runtime platform called a *web container. A* web container provides services such as request dispatching, security, concurrency, and life-cycle management. It also gives web components access to APIs such as naming, transactions, and email.

Certain aspects of web application behavior can be configured when the application is installed, or *deployed, to the web container. The configuration information is maintained in a text file in* XML format called a *web application deployment descriptor (DD). ADDmust conform to the* schema described in the Java Servlet Specification.

This chapter gives a brief overview of the activities involved in developing web applications. First it summarizes the web application life cycle. Then it describes how to package and deploy very simple web applications on the Application Server. It moves on to configuring web applications and discusses how to specify the most commonly used configuration parameters. It then introduces an example, Duke’s Bookstore, which illustrates all the Java EE web-tier technologies, and describes how to set up the shared components of this example. Finally it discusses how to access databases from web applications and set up the database resources needed to run Duke’s Bookstore.

As soon as the web began to be used for delivering services, service providers recognized the need for dynamic content. Applets, one of the earliest attempts toward this goal, focused on using the client platform to deliver dynamic user experiences. At the same time, developers also investigated using the server platform for this purpose. Initially, Common Gateway Interface (CGI) scripts were the main technology used to generate dynamic content. Although widely used, CGI scripting technology has a number of shortcomings, including platform dependence and lack of scalability. To address these limitations, Java Servlet technology was created as a portable way to provide dynamic, user-oriented content.

What Is a Servlet?

A *servlet is a Java programming language class that is used to extend the capabilities of servers* that host applications accessed by means of a request-response programming model. Although servlets can respond to any type of request, they are commonly used to extend the applications hosted by web servers. For such applications, Java Servlet technology defines HTTP-specific servlet classes.

The javax.servlet and javax.servlet.http packages provide interfaces and classes for writing servlets. All servlets must implement the Servlet interface, which defines life-cycle methods. When implementing a generic service, you can use or extend the GenericServlet class provided with the Java Servlet API. TheHttpServlet class provides methods, such as doGet and doPost, for handling HTTP-specific services.

This chapter focuses on writing servlets that generate responses to HTTP requests.

This example of source code for servlet generate web page with simple text “Hello world!”.

Servlet responses only on HTTP requests with HTTP methods GET and POST, because only methods **doPost()** and **doGet()** are overridden. Because we don’t need different response on method POST and GET the method **doGet()** simply call method **doPost()**.

Method **doPost()** just generates HTML code with simple text “Hello world!”

File “web.xml” contains configuration of whole web module like name and default file names. Also it contains information for each servlet contained in web module, like name of servlet, description, main class of servlet and list of mapping URLs.

To deploy a servlet to the java application server it is important include servlet class in a java web module. The web module is a ZIP file with extension “war” and predefined file structure.

Static content of module is included directly in root of module. Classes and other dynamic content are in folder “WEB-INF”. One of the most important file is “web.xml” from folder “WEB-INF” because contains configuration of whole web module.

HTTP Servlet has access to information included in HTTP request. Mostly used information is value of parameters included in request. The pairs “parameter name” and “parameter value” are included in [query-string] or in body of request. HTTP request contains other information like host name, port number and request path. Request path can be divided to three parts:

Context path: Part of request path from root to application deploy directory.

Servlet Path: Part of request path matched by URL pattern defined in file “web.xml”.

Path info: Rest of request path after servlet path.

There are four scope objects in servlets which enables sharing information between web components.

Servlet context can store information for one servlet. All clients share information stored in servlet context.

Session can store information for one user session. All information is available for particular user until session destruction.

Request can store information only during request is processing.

Page context is used in JSP technology and hold information about one JSP page.

Servlet can set several request properties like content type, encoding, … . Many of those properties can be set only before first byte of response is written.

In most cases response contains only text (HTNL, XML, …) but servlet can use binary output stream to return binary data like a images or other multimedia.

Request filtering is another useful mechanism in web development. Java EE provides possibility of filter definition and mapping to URL pattern.

When client send request, web container build filter chain (ordered set of filters) according to requested URL. Request have to pass through all filters in the filter chain then is processed by servlet and have to go back through filter chain in reverse order.

Source code implements simple filter example. Shown filter just convert all text from response to upper case.

Implemented class contains annotation that can substitute configuration from “web.xml” file.

Method doFilter() just create response wrapper, call method FilterChain.doFilter() to pass control to next filter in the chain. When control is returned from method FilterChain.doFilter(), all other filters and servlet already process request and our filter can change text to upper case.

If filter want process data for client from servlet or other filter it need response wrapper.

The animation describes filtering process if filter doesn’t create response wrapper.

When request is passed to filtering process a response object is already created and contains output stream. Response output stream is connected directly to client and data passed to the output stream are immediately sent to client (web browser).

Servlet generate response data and pass the data to output stream.

Our filter “MyFilter” cannot convert already sent data to upper case.

This animation describes filtering process if our filter create a response wrapper.

A response wrapper implements interface HttpServletResponse and the default implementation of wrapper (class HttpServletResponseWrapper) just forward all methods call to the original response object.

Implementation of the response wrapper in our example just creates a data buffer and redirect output stream to the data buffer. Servlet generates data and pass them to the output stream. The output stream sent data to the buffer and our filter “MyFilter” can read data from the buffer and change all character to upper case.

Implementation of response wrapper from our example inherits from default response wrapper HttpServletResponseWrapper.

Our class add private field “buffer” of type CharArrayWriter, initialize the filed in constructor and override two methods getWriter() and toString().

Method getWriter() return output stream connected to buffer.

Method toString() return content of buffer as string.

Source code implements simple filter example. Shown filter just convert all text from response to upper case.

Implemented class contains annotation that can substitute configuration from “web.xml” file.

Method doFilter() just create response wrapper, call method FilterChain.doFilter() to pass control to next filter in the chain. When control is returned from method FilterChain.doFilter(), all other filters and servlet already process request and our filter can change text to upper case.

Filters are connected to the filter chain based on filter mapping. The filter mapping is defined in configuration file “web.xml” or can be specified by annotations in filter class.

The filter mapping contains URL pattern. If the URL pattern match with requested URL, the filter is added to the filter chain.

All URL pattern strings have to match excatly with requested URL except these:

Pattern contains characters “/*” at end of the pattern string. Requested URL match even if contains suffix string.

Pattern contains characters “*.” at the beginning of the pattern string. Requested URL match if ends with specified extension.

Servlets can include other servlets to generate common part of response like header, footer, menu bar and so on. Included servlet just add content data to existing response.

JSF is designed to create UI from predefined components (similar as AWT or SWING) and minimalize impact of HTML and http protocol onto application design. In fact JSF is one servlet that process HTTP requests and prepare environment for JSF components.

Configuration lines define a servlet from JSF java libraries and map usage of the servlet to defined URL patter. In this case to all URLs starts with prefix /guess/.

JSF main component is f:view that represent one view/dialog/page of application. View can contain other components. Components that need inputs from user have to be contained in component h:form.

Component structure is defined by HTML with special namespace and each JSF component is defined by xml tag from that namespace. It is possible use standart HTML tags but programmer should use only JSF tags.

JSF define three special namespaces.

Core: define tags for validator, converters, views and others mostly non visual elements of user interface.

HTML: Define visual components that are similar for all other UI frameworks. Many of components can be clearly map to standard HTML tags like h:form, h:inputTest.

Facelets: Define components used for templating mechanism.

Application (@ApplicationScoped): Application scope persists across all users' interactions with a web application.
Session (@SessionScoped): Session scope persists across multiple HTTP requests in a web application
View (@ViewScoped): View scope persists during a user's interaction with a single page (view) of a web application.
Request (@RequestScoped): Request scope persists during a single HTTP request in a web application.
None (@NoneScoped):
Indicates a scope is not defined for the application.
Custom (@CustomScoped): A user-defined, nonstandard scope. Its value must be configured as a map. Custom scopes are used infrequently.

Immediate expressions are executed immediately when text processed and the expression read values only.
Deferred expressions can be executed many times and the expressions read and write value of property.

Expressions navigate through objects and their properties.

h:outputText - This component render value of attribute “value” just as simple text. In this case value is defined by deferred expression.

t h:graphicImage – This component load and render image from specified URL.

h:inputText – This component render standard text field. Value of text field is bind with value of property “userNumber”. When page is rendered value is read form property and set to text field. When page is submitted value form text field is set to property.

h:panelGroup – This component define group of other components. PanelGroup is rendered to HTML as tag div. Value of attribute style is passed to tag div as CSS style.

h:panelGrid – This component allow layout components to table. Component define only number of columns. Number of rows is calculated automatically based on components contained in panel grid. This component is rendered to HTML page as tag table.

JSF treated both components h:commandButton and h:commandLink in same way. Only difference is in visual appearance for user. CommandButton is rendered as button and commandLiink is rendered as link (standard link in HTML page). Attribute outcome is very important for navigation to the next page.

h:dataTable – This component define table filed with data from a collection. Attribute “value” refer the collection, attribute “var” define name of variable used to store one element of collection.

Method expresion

Facelets is a templating technology that allows definition of overall visual structure and style of web application, like top header with logo and menu, left side tree navigation and so on.

Template is standard JSF page with at least one component ui:insert. The ui:insert component define named place for insert dynamic content.

Any page can use template if use component ui:composition. The page should contains components ui:define with name that corresponded with name of components ui:insert form used template.

Converter is an interface describing a Java class that can perform Object-to-String and String-to-Object conversions between model data objects and a String representation of those objects that is suitable for rendering.

Converter implementations must have a zero-arguments public constructor. In addition, if the Converter class wishes to have configuration property values saved and restored with the component tree, the implementation must also implement StateHolder.

Starting with version 1.2 of the specification, an exception to the above zero-arguments constructor requirement has been introduced. If a converter has a single argument constructor that takes a Class instance and the Class of the data to be converted is known at converter instantiation time, this constructor must be used to instantiate the converter instead of the zero-argument version. This enables the per-class conversion of Java enumerated types.

If any Converter implementation requires a java.util.Locale to perform its job, it must obtain that Locale from the UIViewRoot of the current FacesContext, unless the Converter maintains its own Locale as part of its state.

Method Summary java.lang.Object **getAsObject**(FacesContext context, UIComponent component, java.lang.String value)

Convert the specified string value, which is associated with the specified UIComponent, into a model data object that is appropriate for being stored during the *Apply Request Values* phase of the request processing lifecycle. java.lang.String **getAsString**(FacesContext context, UIComponent component, java.lang.Object value)

Convert the specified model object value, which is associated with the specified UIComponent, into a String that is suitable for being included in the response generated during the *Render Response* phase of the request processing lifeycle.

Method Detail

getAsObject

java.lang.Object **getAsObject**(FacesContext context, UIComponent component, java.lang.String value) Convert the specified string value, which is associated with the specified UIComponent, into a model data object that is appropriate for being stored during the *Apply Request Values* phase of the request processing lifecycle.

Parameters: context - FacesContext for the request being processed component - UIComponent with which this model object value is associated value - String value to be converted (may be null) **Returns:** null if the value to convert is null, otherwise the result of the conversion **Throws:** ConverterException - if conversion cannot be successfully performed java.lang.NullPointerException - if context or component is null **getAsString**

java.lang.String **getAsString**(FacesContext context, UIComponent component, java.lang.Object value) Convert the specified model object value, which is associated with the specified UIComponent, into a String that is suitable for being included in the response generated during the *Render Response* phase of the request processing lifeycle.

Parameters: context - FacesContext for the request being processed component - UIComponent with which this model object value is associated value - Model object value to be converted (may be null) **Returns:** a zero-length String if value is null, otherwise the result of the conversion **Throws:** ConverterException - if conversion cannot be successfully performed java.lang.NullPointerException - if context or component is null

BigDecimalConverter
BigIntegerConverter
BooleanConverter
ByteConverter
CharacterConverter
DateTimeConverter
DoubleConverter
EnumConverter
FloatConverter
IntegerConverter
LongConverter
NumberConverter
ShortConverter

JSF - konverze

```
<h:inputText id="valueEdit"
   value="#{counterHolder.counter.value}" label="Counter value"
   converter="#{converterFactory.employeeConverter}">
      <f:converter binding="#{counterHolder}"/>
</h:inputText>

<managed-bean>
<managed-bean-name>counterHolder</managed-bean-name>
<managed-bean-class>bean.CounterHolder</managed-bean-class>
<managed-bean-scope>session</managed-bean-scope>
</managed-bean>

@ManagedBean
public class CounterHolder implements
   javax.faces.convert.Converter
```

Výraz musí vždy vracet objekt implementující rozhraní **Converter**

JSF - konverze

```
<h:inputText id="valueEdit"
  value="#{counterHolder.currentCompany}" label="Counter
  value">

</h:inputText>
```

Výraz musí vždy vracet objekt implementující rozhraní **Converter**

```
@ApplicationScoped
@FacesConverter(forClass=Company.class, managed = true)
public class CounterHolder implements
  javax.faces.convert.Converter
```

JSF – Validační model

- Validace probíhá před uložením dat do proměně dané příslušným EL výrazem
- Validace probíhá po konverzi dat z prezentační podoby do datového typu modelu
- Sada standartních validátorů
  - validateDoubleRange, validateLength, validateLongRange
- Metoda BackingBean (atribut validator)
- Implementace rozhraní `javax.faces.validator.Validator`

JSF - validátory

- ```
  <h:inputText id="userNo" label="User Number"
  value="#{userNumberBean.userNumber}"
  validatorMessage="zpráva"><f:validateLongRange
  minimum="#{userNumberBean.minimum}"
  maximum="#{userNumberBean.maximum}"
  /></h:inputText>
  ```

- ```
  <h:message showSummary="true" showDetail="false"
  style="color: red; font-family: 'New Century
  Schoolbook', serif; font-style: oblique; text-
  decoration: overline" id="errors1" for="userNo"/>
  ```


Converter
Id musí seďět mezi komponentou a message

JSF - validátory

- ```
  <h:inputText id="userNo" label="User Number"
  value="#{userNumberBean.userNumber}"
  validatorMessage="zpráva">
  <f:validator validatorID="myValidatorID" />
  </h:inputText>
  ```

```
@FacesValidator(value="myValidatorID")
public class CounterVelidator implements
  javax.faces.validator.Validator
```


Converter
Id musí seďět mezi komponentou a message
```

JSF - validátory

- ```
  <h:inputText id="userNo" label="User Number"
  value="#{UserNumberBean.userNumber}"
  validatorMessage="zpráva"
  validator="#{counterVelidator.validate}">

  </h:inputText>
  ```

Reference na **metodu**

```
@ManagedBean
public class CounterVelidator implements
  javax.faces.validator.Validator
public void validate(FacesContext context, UIComponent
component, Object value) throws ValidatorException
```


Converter
Id musí seďět mezi komponentou a message
```

JSF – vyhodnocení EL výrazu

```
public Object getAsObject(FacesContext facesContext,
UIComponent component, String value) {

PersonMB controller = (PersonMB)
facesContext.getApplication().getELResolver().
   getValue(facesContext.getELContext(), null, "personMB");
```

JSF – vlastní zprávy validátorů

```
<application>
  <message-bundle>jat.validation-message</message-bundle>
</application>
```

```
javax.faces.converter.DateTimeConverter.DATE={2}: ''{0}''
could not be understood as a date.
javax.faces.converter.DateTimeConverter.DATE_detail=Invalid
date format.

javax.faces.validator.LengthValidator.MINIMUM=Minimum length
of ''{0}'' is required.
```

JSF – face messages

```
FacesContext ctx = FacesContext.getCurrentInstance();

FacesMessage msg = new FacesMessage
(FacesMessage.SEVERITY_INFO, errorMessage, detailMessage);

ctx.addMessage(null, msg);
```

JSF – GUI - CombBox

```
<h:selectOneMenu value="#{personMB.editedEmployee}"
converter="#{converterFactory.employeeConverter}">
<f:selectItems value="#{personMB.allEmployeesASSelectItem}"/>
</h:selectOneMenu>

public List<SelectItem> getAllEmployeesASSelectItem(){
  Collection<Employee> allEmp = getAllEmployees();
  ArrayList<SelectItem> selItems = new
ArrayList<SelectItem>(allEmp.size());
  for(Employee e : allEmp){
   selItems.add(new SelectItem(e, e.getName() + " " +
e.getSurename()));
  }
  return selItems;
}
```

JSF – GUI - CombBox

```
 <h:selectOneMenu
value="#{personAgendaMB.editedEmployee}"
converter="#{converterFactory.employeeConverter}">
  <f:selectItem noSelectionOption="true"
itemValue="#{null}" itemLabel="None" />
  <f:selectItems value="#{personAgendaMB.allEmployees}"
var="p" itemLabel="#{p.name}" itemValue="#{p}"/>
</h:selectOneMenu>
```

```
public Collection<Person> getAllEmployees(){
  Collection<Employee> allEmp = getAllEmployees();
  return allEmp;
}
```

JSF - lokalizace

```
<application>
<resource-bundle>
  <base-name>jat.messages</base-r
  <var>msg</var>
</resource-bundle>
<locale-config>
  <default-locale>en</default-locale>
  <supported-locale>cs</supported-locale>
</locale-config>
</application>
```

Soubor:
jat/messages.properties

userNoConvert=The value you entered is not a number.

```
<h:inputText id="userNo" label="User Number" value="#{…}"
   validatorMessage="#{msg.userNoConvert}">
```

JSF - lokalizace

```
<f:view locale="#{languageMB.locale}">
public String setENLocale(){
FacesContext.getCurrentInstance().
getViewRoot().setLocale(Locale.ENGLISH);
return "";
}
public String setCZLocale(){
FacesContext.getCurrentInstance().
getViewRoot().setLocale(new Locale("cs"));
return "";
}
```

JSF – lokalizace na základě jazyka prohlížeče

```
private String locale;
public String getLocale() {
  if(locale == null) {
  String languages =FacesContext.getCurrentInstance().
getExternalContext().getRequestHeaderMap().
    get("Accept-Language");
  if(languages != null) {
   return languages.split(",")[0];
    }
  }
 return locale;
}
```

JSF – UI komponentní model

- Kompletní sada UI komponent
- Rozšiřitelnost
- Základní třída UIComponentBase
  - UIColumn, UICommand, UIData, UIForm, UIGraphic, UIInput, UIMessage, UIMessages, UIOutput, UIPanel, UIParameter, UISelectBoolean, UISelectItem, UISelectItems, UISelectMany, UISelectOne, UIViewRoot
- Rozhraní chování
  - ActionSource, ActionSource2, EditableValueHolder, NamingContainer, StateHolder, ValueHolder

JSF – Model renderování komponent

- Oddělení chování komponent od renderování
- Jedna komponenta může být reprezentována několika různými tagy
- UISelectOne
  - Radio buttony
  - Combo box
  - List box

JSF – Event – Listener model

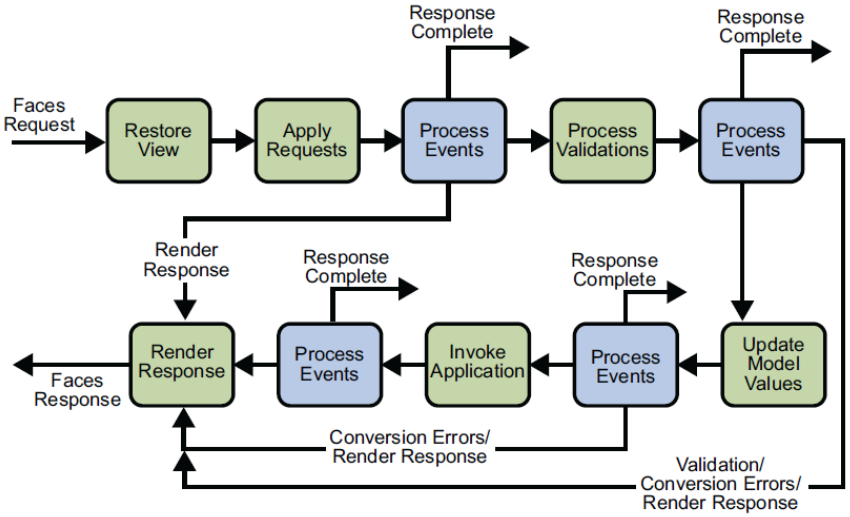
- Obdoba Evenl-Lisener modelu z JavaBeans
- 3 typy událostí: value-change events, action events, data-model events
  - Implementace metody v BackingBean a využití EL výrazu pro metody v příslušném atributu tagu komponenty
  - Implementace posluchače

Třída implementující rozhraní **posluchače**

```
<h:inputText id="name" size="50"
         value="#{cashier.name}" required="true">
  <f:valueChangeListener type="listeners.NameChanged"/>
  <f:valueChangeListener binding="#{mBean.property}"/>
</h:inputText>
```

Výraz musí vždy vracet objekt implementující rozhraní **posluchače**

JSF – životní cyklus



The Lifecycle of a JavaServer Faces Application

The lifecycle of a JavaServer Faces application begins when the client makes an HTTP request for a page and ends when the server responds with the page, translated to HTML.

The lifecycle can be divided into two main phases, execute and render. The execute phase is further divided into sub-phases to support the sophisticated component tree. This structure requires that component data be converted and validated, component events be handled, and component data be propagated to beans in an orderly fashion.

A JavaServer Faces page is represented by a tree of components, called a **view**. During the lifecycle, the JavaServer Faces implementation must build the view while considering the state saved from a previous submission of the page. When the client requests a page, the JavaServer Faces implementation performs several tasks, such as validating the data input of components in the view and converting input data to types specified on the server side.

The JavaServer Faces implementation performs all these tasks as a series of steps in the JavaServer Faces request-response lifecycle. Figure 10-1 illustrates these steps.

Figure 10-1 JavaServer Faces Standard Request-Response Lifecycle

The lifecycle handles two kinds of requests: **initial requests** and **postbacks**. An initial request occurs when a user makes a request for a page for the first time. A postback request occurs when a user submits the form contained on a page that was previously loaded into the browser as a result of executing an initial request.

When the lifecycle handles an initial request, it executes only the Restore View and Render Response phases, because there is no user input or actions to process. Conversely, when the lifecycle handles a postback, it executes all of the phases.

Usually, the first request for a JavaServer Faces page comes in from a client, as a result of clicking a link or button component on a JavaServer Faces page. To render a response that is another JavaServer Faces page, the application creates a new view and stores it in the FacesContext instance, which represents all of the information associated with processing an incoming request and creating a response. The application then acquires object references needed by the view and calls FacesContext.renderResponse method, which forces immediate rendering of the view by skipping to the Render Response Phase of the lifecycle, as is shown by the arrows labelled Render Response in the diagram.

Sometimes, an application might need to redirect to a different web application resource, such as a web service, or generate a response that does not contain JavaServer Faces components. In these situations, the developer must skip the Render Response phase by calling the FacesContext.responseComplete method. This situation is also shown in the diagram, this time with the arrows labelled Response Complete.

The most common situation is that a JavaServer Faces component submits a request for another JavaServer Faces page. In this case, the JavaServer Faces implementation handles the request and automatically goes through the phases in the lifecycle to perform any necessary conversions, validations, and model updates, and to generate the response.

There is one exception to the lifecycle described in this section. When a component's immediate attribute is set totrue, the validation, conversion, and events associated with these components are processed during the Apply Request Values Phase rather than in a later phase.

The details of the lifecycle explained in the following sections are primarily intended for developers who need to know information such as when validations, conversions, and events are usually handled and what they can do to change how and when they are handled. For more information on each of the lifecycle phases, download the latest JavaServer Faces Specification documentation from https://javaserverfaces.java.net/.

The JavaServer Faces application lifecycle **execute** phase contains the following sub-phases:

Restore View Phase
Apply Request Values Phase
Process Validations Phase
Update Model Values Phase
Invoke Application Phase
Render Response Phase

Restore View Phase

When a request for a JavaServer Faces page is made, usually by an action such as when a link or a button component is clicked, the JavaServer Faces implementation begins the Restore View phase.

During this phase, the JavaServer Faces implementation builds the view of the page, wires event handlers and validators to components in the view, and saves the view in the FacesContext instance, which contains all the information needed to process a single request. All the application's components, event handlers, converters, and validators have access to the FacesContext instance.

If the request for the page is an initial request, the JavaServer Faces implementation creates an empty view during this phase and the lifecycle advances to the Render Response phase, during which the empty view is populated with the components referenced by the tags in the page.

If the request for the page is a postback, a view corresponding to this page already exists in the FacesContextinstance. During this phase, the JavaServer Faces implementation restores the view by using the state information saved on the client or the server.

Apply Request Values Phase

After the component tree is restored during a postback request, each component in the tree extracts its new value from the request parameters by using its decode (processDecodes()) method. The value is then stored locally on each component. If the conversion of the value fails, an error message that is associated with the component is generated and queued on FacesContext. This message will be displayed during the Render Response phase, along with any validation errors resulting from the Process Validations phase.

If any decode methods or event listeners have called the renderResponse method, on the current FacesContextinstance, the JavaServer Faces implementation skips to the Render Response phase.

If any events have been queued during this phase, the JavaServer Faces implementation broadcasts the events to interested listeners.

If some components on the page have their immediate attributes (see The immediate Attribute) set to true, then the validations, conversions, and events associated with these components will be processed during this phase.

At this point, if the application needs to redirect to a different web application resource or generate a response that does not contain any JavaServer Faces components, it can call the FacesContext.responseComplete method.

At the end of this phase, the components are set to their new values, and messages and events have been queued.

If the current request is identified as a partial request, the partial context is retrieved from the FacesContext, and the partial processing method is applied.

Process Validations Phase

During this phase, the JavaServer Faces implementation processes all validators registered on the components in the tree, by using its validate (processValidators) method. It examines the component attributes that specify the rules for the validation and compares these rules to the local value stored for the component.

If the local value is invalid, the JavaServer Faces implementation adds an error message to the FacesContextinstance, and the lifecycle advances directly to the Render Response phase so that the page is rendered again with the error messages displayed. If there were conversion errors from the Apply Request Values phase, the messages for these errors are also displayed.

If any validate methods or event listeners have called the renderResponse method on the currentFacesContext, the JavaServer Faces implementation skips to the Render Response phase.

At this point, if the application needs to redirect to a different web application resource or generate a response that does not contain any JavaServer Faces components, it can call the FacesContext.responseComplete method.

If events have been queued during this phase, the JavaServer Faces implementation broadcasts them to interested listeners.

If the current request is identified as a partial request, the partial context is retrieved from the Faces Context, and the partial processing method is applied.

Update Model Values Phase

After the JavaServer Faces implementation determines that the data is valid, it traverses the component tree and sets the corresponding server-side object properties to the components' local values. The JavaServer Faces implementation updates only the bean properties pointed at by an input component's value attribute. If the local data cannot be converted to the types specified by the bean properties, the lifecycle advances directly to the Render Response phase so that the page is re-rendered with errors displayed. This is similar to what happens with validation errors.

If any updateModels methods or any listeners have called the renderResponse method on the currentFacesContext instance, the JavaServer Faces implementation skips to the Render Response phase.

At this point, if the application needs to redirect to a different web application resource or generate a response that does not contain any JavaServer Faces components, it can call the FacesContext.responseComplete method.

If any events have been queued during this phase, the JavaServer Faces implementation broadcasts them to interested listeners.

If the current request is identified as a partial request, the partial context is retrieved from the FacesContext, and the partial processing method is applied.

Invoke Application Phase

During this phase, the JavaServer Faces implementation handles any application-level events, such as submitting a form or linking to another page.

At this point, if the application needs to redirect to a different web application resource or generate a response that does not contain any JavaServer Faces components, it can call the FacesContext.responseComplete method.

If the view being processed was reconstructed from state information from a previous request and if a component has fired an event, these events are broadcast to interested listeners.

Finally, the JavaServer Faces implementation transfers control to the Render Response phase.

Render Response Phase

During this phase, JavaServer Faces builds the view and delegates authority to the appropriate resource for rendering the pages.

If this is an initial request, the components that are represented on the page will be added to the component tree. If this is not an initial request, the components are already added to the tree, so they need not be added again.

If the request is a postback and errors were encountered during the Apply Request Values phase, Process Validations phase, or Update Model Values phase, the original page is rendered again during this phase. If the pages containmessage or messages tags, any queued error messages are displayed on the page.

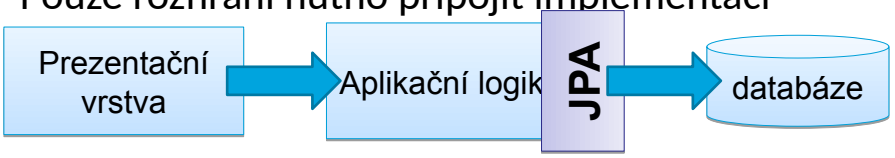
After the content of the view is rendered, the state of the response is saved so that subsequent requests can access it. The saved state is available to the Restore View phase.

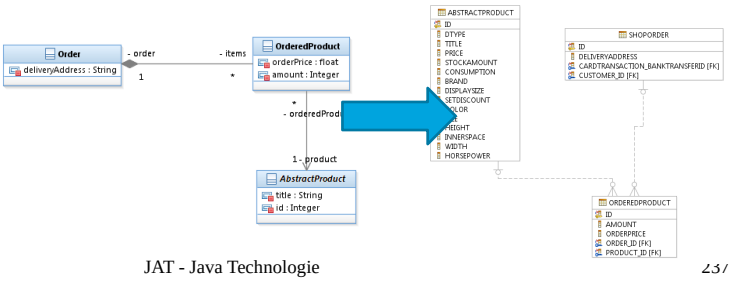
P7

- Java Persistenc API
  - Jazyk QL
- Hibernate
  - HQL
  - http://docs.jboss.org/hibernate/stable/core/reference/en/html/tutorial.html
  - http://www.manning.com/bauer2/chapter2.pdf

JPA – Java Persistent API

- API pro perzistenci při využití objektově relační mapování
- Pouze rozhraní nutno připojit implementaci





JPA - Entity

- **Entity** – jsou lehké objekty z perzistentní domény. Typicky představují tabulku v databázi.
  - Každý jeden objekt odpovídá jednomu záznamu v databázi.
- **Perzistentní stav entity** je reprezentován perzistentními třídními proměnnými nebo perzistentními vlastnostmi.
  - Mapování mezi databázi (tabulkami/sloupci) a vlastnostmi je určeno anotacemi

JPA - Entitní třída

- Třída musí být oanotovaná anotací `javax.persistence.Entity`
- Třída musí mít public nebo protected konstruktor bez parametrů (může mít jiné konstruktory)
- Třída ani žádné metoda nebo třídní proměná nesmějí být deklarovány jako `final`

JPA - Entitní třída

- Pokud je entita použita ve vzdáleném rozhraní (EJB) musí třída implementovat rozhraní `Serializable`
- Entitní třídy mohou být potomky entitních i ne-entitních tříd. Ne-entitnní třídy mohou být podomky entitních tříd.
- Perzistentní třídní proměnné musí být definovány jako private, protected nebo package-private. Mnělo by se k nim přistupovat jen pomocí set, get metod.

JPA - Entitní třída - ukázka

```
@Entity
@Table(name="ShopOrder")
public class Order {
  @Id
  @GeneratedValue(strategy=GenerationType.IDENTITY)
  private int id;
  @OneToOne
  private Transaction cardTransaction;
  @ManyToOne()
  private Customer customer;
  @OneToMany(mappedBy="order")
  private Set<OrderedProduct> items;
  private String deliveryAddress;
  …
}
```

JPA – perzistentní vlastnosti, třídní proměnné

- Třídní proměnné – perzistence přistupuje přímo k těmto proměnným
- Vlastnosti – Perzistence přistupuje k proměnným pomocí get, set metod.
  - Lze využít kolekce: Collection, Set, List, Map i generické verze
- **`hashcode() equals()`**
- Typy:primitivní datové typy jazyka Java
  - java.lang.String, ostatní serializovatelné typy (třídy reprezentující primitivní typy, java.math.BigInteger, java.math.BigDecimal, java.util.Date, java.util.Calendar, java.sql.Date, java.sql.Time, java.sql.TimeStamp, uživatelské serializovatelné typy, byte[], Byte[], char[], Character[], výčtové typy, ostatní entity, kolekce entit

Problém kde dát anotaci – logika v set /get metodách - přístup k set/get

JPA – primární klíče

- Každá entita musí mít svůj primární klíč.
- javax.persistence.Id
- Složené primární klíče
  - Musí existovat třída definující složený klíč
  - javax.persistence.EmbeddedId
  - javax.persistence.IdClass
  - Musí být těchto typů:
    - primitivní typy jazyka Java (a příslušné obalující třídy)
    - java.lang.String
    - java.util.Date (DATE), java.sql.Date

JPA – primární klíče

- Každá entita musí mít svůj primární klíč.
- javax.persistence.Id
- Složené primární klíče
  - Musí existovat třída definující složený klíč
  - javax.persistence.IdClass
  - javax.persistence.EmbeddedId
  - Musí být těchto typů:
    - primitivní typy jazyka Java (a příslušné obalující třídy)
    - java.lang.String
    - java.util.Date (DATE), java.sql.Date
- Desetinná čísla by neměla být použita jako klíč

JPA – třída pro primární klíč

Třída musí:

- být public
- mít public třídní proměnné nebo protected ale musí existovat public set get metody.
- public default konstruktor bez parametrů
- implementovat metody hashcode() a equals(Object)
- implementovat rozhraní Serializable

JPA – třída pro primární klíč

Třída musí:

- být mapována na několik třídních proměnných nebo vlastností entity
- názvy proměnných/vlastností klíče musí odpovídat názvům v entitě

JPA – třída pro složený primární klíč

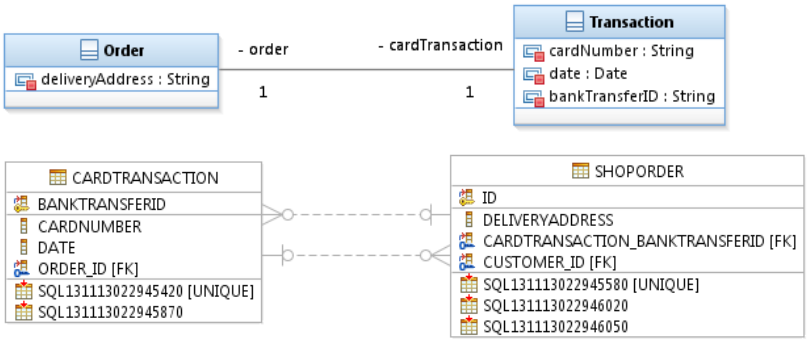
```
public final class LineItemKey implements Serializable {
   public Integer orderId;
   public int itemId;
   public LineItemKey() {
   }
   public LineItemKey(Integer orderId, int itemId) {
      this.orderId = orderId;
      this.itemId = itemId;
   }
   public boolean equals(Object otherOb) {
      if (this == otherOb) {
         return true;
      }
      if (!(otherOb instanceof LineItemKey)) {
         return false;
      }
```

```
        LineItemKey other = (LineItemKey) otherOb;
        return ((orderId == null ? other.orderId == null :
orderId
              .equals(other.orderId)) && (itemId ==
other.itemId));
    }
    public int hashCode() {
        return ((orderId == null ? 0 : orderId.hashCode()) ^
((int) itemId));
    }
    public String toString() {
        return "" + orderId + "-" + itemId;
    }
}
```

JPA – násobnost vazeb 1-1

```
@Entity
public class Order {
@OneToOne
private Transaction
cardTransaction;
…
```

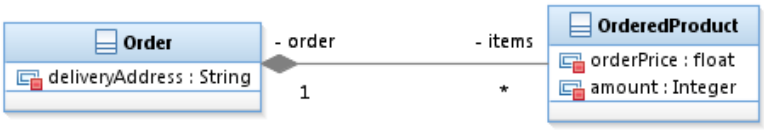
```
@Entity
public class Transaction {
@OneToOne
private Order order;
…
```

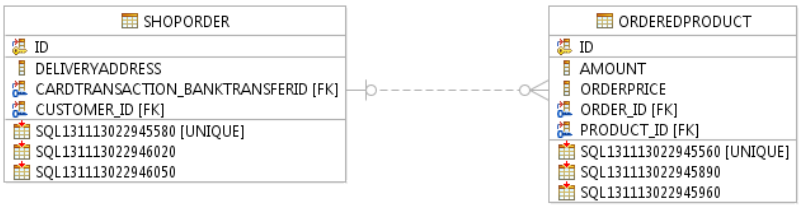


JPA – násobnost vazeb 1-N

```
@Entity
public class Order {
@OneToMany(mappedBy="order")
private Set<OrderedProduct>
items;
…
```

```
@Entity
public class OrderedProduct {
@ManyToOne
private Order order;
…
```

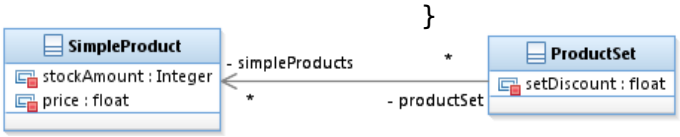




JPA – násobnost vazeb M-N

```
@Entity
public class SimpleProduct
extends AbstractProduct {
@ManyToMany(mappedBy="simpleP
roduct")
private List<ProductSet>
productSets;
}
```

```
@Entity
public class ProductSet
extends AbstractProduct
@ManyToMany
private List<SimpleProduct>
simpleProduct;
private float setDiscount;
}
```



JPA – dědičnost

- **Entita může dědit z neentitní třídy**
- **Entita – předek může být abstraktní**

```
@Entity
public abstract class Employee {
@Id
protected Integer employeeId;
}
```

```
@Entity
public class PartTimeEmployee extends
    Employee {
protected Float hourlyWage;
}
```

```
@Entity
public class FullTimeEmployee extends
    Employee {
protected Integer salary;
}
```

JPA – mapovací strategie dědičnosti

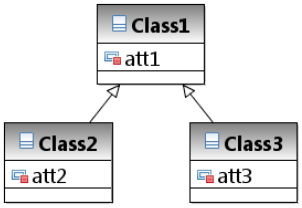
- Jedna tabulka na hierarchii tříd
- Jedna tabulka pro konkrétní entitu
- Join strategy

```
public enum InheritanceType {
   SINGLE_TABLE,
   JOINED,
   TABLE_PER_CLASS
};
@Inheritance(strategy=JOINED)
```

JPA – mapovací strategie dědičnosti

Jedna tabulka na hierarchii tříd

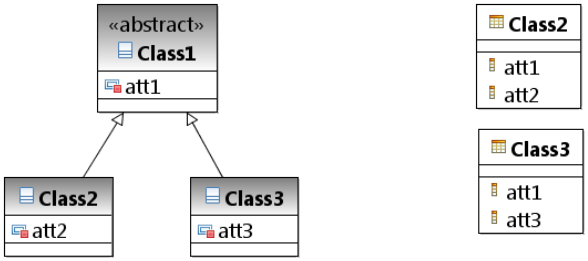
```
@Inheritance(strategy=SINGLE_TABLE)
@DiscriminatorColumn(
    String name
    DiscriminatorType discriminatorType
    String columnDefinition
    String length)
public enum DiscriminatorType {
    STRING,
    CHAR,
    INTEGER
};
@DiscriminatorValue
```



JPA – mapovací strategie dědičnosti

Jedna tabulka pro konkrétní entitu

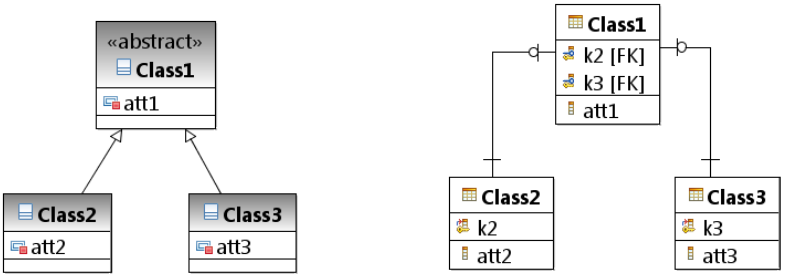
@Inheritance(strategy=TABLE_PER_CLASS)



JPA – mapovací strategie dědičnosti

Join strategy

@Inheritance(strategy=JOINED)



JPA – MappedSuperclass

```
@MappedSuperclass
public class Person {
@Column(length=50)
private String name;
@Column(length=50)
private String surename;
@Column(length=50)
private String email;
@Column(length=50)
private String password;
}
```

```
@Entity
public class Customer extends Person
{
@Id
@GeneratedValue(strategy=GenerationTy
pe.IDENTITY)
private int id;
@OneToMany(mappedBy="customer")
private Set<Order> orders;
```

```
@Entity
public class Employee extends Person {
@Id
@Column(length=50)
private String login;
private float sallary;
@Column(length=50)
private String depsrtment;
}
```


Mapped Superclasses

Entities may inherit from superclasses that contain persistent state and mapping information, but are not entities. That is, the superclass is not decorated with the @Entity annotation, and is not mapped as an entity by the Java Persistence provider. These superclasses are most often used when you have state and mapping information common to multiple entity classes.

Mapped superclasses are specified by decorating the class with the javax.persistence.MappedSuperclass annotation.

Mapped superclasses are not queryable, and can't be used in EntityManager or Query operations. You must use entity subclasses of the mapped superclass in EntityManager or Query operations.Mapped superclasses can't be targets of entity relationships.Mapped superclasses can be abstract or concrete.

Mapped superclasses do not have any corresponding tables in the underlying datastore. Entities that inherit from the mapped superclass define the table mappings. For instance, in the code sample above the underlying tables would be FULLTIMEEMPLOYEE and PARTTIMEEMPLOYEE, but there is no EMPLOYEE table.

JPA – správa entit

- **Perzistentní kontext**: množina entit existujících v konkrétním datovém úložišti
- EntityManager
  - Vytváří, maže entity, hledá entity, provádí dotazy
- Kontejnerem řízený správce entit

```
@PersistenceContext
EntityManager em;
```

JPA – správa entit

- Aplikací řízený správce entit
  - @PersistenceUnit
  - EntityManagerFactory emf;

  - EntityManager em = emf.createEntityManager();

JPA – nalezení entity

```
@PersistenceContext
EntityManager em;

public void enterOrder(int custID, Order newOrder) {
   Customer cust = em.find(Customer.class, custID);
   cust.getOrders().add(newOrder);
   newOrder.setCustomer(cust);
}
```

JPA – životní cyklus entity

- New
- Managed
- Detached
- Removed

```
@PersistenceContext
EntityManager em;
...
public LineItem createLineItem(Order
    order, Product product, int quantity) {
    LineItem li = new LineItem(order,
    product, quantity);
    order.getLineItems().add(li);
    em.persist(li);
    return li;
}

em.remove(order);
em.flush();
```

JPA - dotazy

```
public List findWithName(String name) {
   return em.createQuery(
   "SELECT c FROM Customer c WHERE c.name
   LIKE :custName")
   .setParameter("custName", name)
   .setMaxResults(10)
   .getResultList();
}
.setFirstResult(100)
```

JPA – pojmenované dotazy

Anotace třídy - entity

```
@NamedQuery(
    name="findAllCustomersWithName",
    query="SELECT c FROM Customer c WHERE c.name LIKE :custName"
)

@PersistenceContext
public EntityManager em;
...
customers = em.createNamedQuery("findAllCustomersWithName")
    .setParameter("custName", "Smith")
    .getResultList();
```

JPA – parametry v dotazech

- Pojmenované

```
return em.createQuery(
    "SELECT c FROM Customer c WHERE c.name LIKE :custName")
    .setParameter("custName", name)
    .getResultList();
```

- Číslované

```
return em.createQuery(
    “SELECT c FROM Customer c WHERE c.name LIKE ?1”)
    .setParameter(1, name)
    .getResultList();
```

JPA – Persistence Units

- Balík obsahující všechny entitní třídy mapované na jedno datové úložiště (DB).
- Musí obsahovat soubor persistence.xml
- Může být součástí EAR, WAR, EJB JAR

JPA – persistence.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="2.1" xmlns="http://xmlns.jcp.org/xml/ns/persistence" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/persistence http://xmlns.jcp.org/xml/ns/persistence/persistence_2_1.xsd">
<persistence-unit name="Slaids">
<provider>org.hibernate.ejb.HibernatePersistence</provider>
<provider>org.eclipse.persistence.jpa.PersistenceProvider</provider>
<jta-data-source>java:/jdbc/slaids</jta-data-source>
<properties>
<property name="javax.persistence.schema-generation.database.action"
value="create"/>
<property name="hibernate.hbm2ddl.auto" value="create"/>
<property name="hibernate.dialect"
value="org.hibernate.dialect.DerbyTenSevenDialect"/>
• <property name="eclipselink.ddl-generation" value="create-tables" />
•     <property name="eclipselink.ddl-generation.output-mode" value="database" />
• <property name="eclipselink.target-database" value="Derby"/>
</properties>
</persistence-unit>
</persistence>
```



```
<persistence>
<persistence-unit name="OrderManagement">
<description>This unit …..</description>
<jta-data-source>jdbc/MyOrderDB</jta-data-source>
<jar-file>MyOrderApp.jar</jar-file>
<class>com.widgets.Order</class>
<class>com.widgets.Customer</class>
</persistence-unit>
</persistence>
```

JPA – Query Language

Select Statement

- SELECT, FROM, WHERE, GROUP BY, HAVING, ORDER BY

Update, Delte Statement

- UPDATE Player p SET p.status = ’inactive’ WHERE p.lastPlayed < :inactiveThresholdDate
- DELETE FROM Player p WHERE p.status = ’inactive’ AND p.teams IS EMPTY

JPA – dotazy - příklady

- SELECT p FROM Player AS p
- SELECT DISTINCT p FROM Player AS p WHERE p.position = ?1
- SELECT DISTINCT p FROM Player AS p, IN(p.teams) AS t
- SELECT DISTINCT p FROM Player AS p JOIN p.teams AS t
- SELECT DISTINCT p FROM Player AS p WHERE p.team IS NOT EMPTY
- SELECT t FROM Team AS t JOIN t.league AS l WHERE l.sport = ’soccer’ OR l.sport =’football’
- SELECT DISTINCT p FROM Player AS p, IN (p.teams) AS t WHERE t.city = :city

JPA – dotazy - příklady

- SELECT DISTINCT p FROM Player AS p, IN (p.teams) AS t WHERE t.league.sport = :sport

JPA – dotazy - LIKE

- SELECT p FROM Player p WHERE p.name LIKE ’Mich%’
- _ - jakýkoliv jeden znak
- % - nula nebo výce jakýchkoliv znaků
- ESCAPE – říká jaký znak je použit jako escape
  - LIKE ’_%’ ESCAPE ’\’
- NOT LIKE

JPA – DOTAZY – NULL, IS EMPTY

- SELECT t FROM Team t WHERE t.league IS NULL
- SELECT t FROM Team t WHERE t.league IS NOT NULL
- Nelze použít WHERE t.league = NULL

- SELECT p FROM Player p WHERE p.teams IS EMPTY
- SELECT p FROM Player p WHERE p.teams IS NOT EMPTY

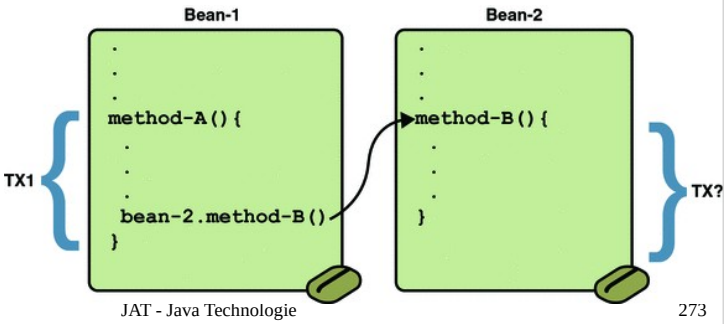
JPA – dotazy between, in

- SELECT DISTINCT p FROM Player p WHERE p.salary BETWEEN :lowerSalary AND :higherSalary
- p.salary >= :lowerSalary AND p.salary <= :higherSalary
- o.country IN (’UK’, ’US’, ’France’)

JTA – Java Transaction

Kontejnerem řízené transakce

- V rámci jedné EJB metody není povoleno vnořování nebo více transakcí.
- Rozsah transakcí
  - Required
  - RequiresNew
  - Mandatory
  - NotSupported
  - Supports
  - Never



JTA

Transaction Attribute	Client's Transaction	Business Method's Transaction
Required	None	T2
	T1	T1
RequiresNew	None	T2
	T1	T2
Mandatory	None	error
	T1	T1
NotSupported	None	None
	T1	None
Supports	None	None
	T1	T1
Never	None	None
	T1	Error

Required Attribute

If the client is running within a transaction and invokes the enterprise bean's method, the method executes within the client's transaction. If the client is not associated with a transaction, the container starts a new transaction before running the method.

The Required attribute is the implicit transaction attribute for all enterprise bean methods running with container-managed transaction demarcation. You typically do not set the Required attribute unless you need to override another transaction attribute. Because transaction attributes are declarative, you can easily change them later.

RequiresNew Attribute

If the client is running within a transaction and invokes the enterprise bean's method, the container takes the following steps:

Suspends the client's transaction

Starts a new transaction

Delegates the call to the method

Resumes the client's transaction after the method completes

If the client is not associated with a transaction, the container starts a new transaction before running the method.

You should use the RequiresNew attribute when you want to ensure that the method always runs within a new transaction.

Mandatory Attribute

If the client is running within a transaction and invokes the enterprise bean's method, the method executes within the client's transaction. If the client is not associated with a transaction, the container throws the TransactionRequiredException.

Use the Mandatory attribute if the enterprise bean's method must use the transaction of the client.

NotSupported Attribute

If the client is running within a transaction and invokes the enterprise bean's method, the container suspends the client's transaction before invoking the method. After the method has completed, the container resumes the client's transaction.

If the client is not associated with a transaction, the container does not start a new transaction before running the method.

Use the NotSupported attribute for methods that don't need transactions. Because transactions involve overhead, this attribute may improve performance.

Supports Attribute

If the client is running within a transaction and invokes the enterprise bean's method, the method executes within the client's transaction. If the client is not associated with a transaction, the container does not start a new transaction before running the method.

Because the transactional behavior of the method may vary, you should use the Supports attribute with caution.

Never Attribute

If the client is running within a transaction and invokes the enterprise bean's method, the container throws a RemoteException. If the client is not associated with a transaction, the container does not start a new transaction before running the method.

Summary of Transaction Attributes

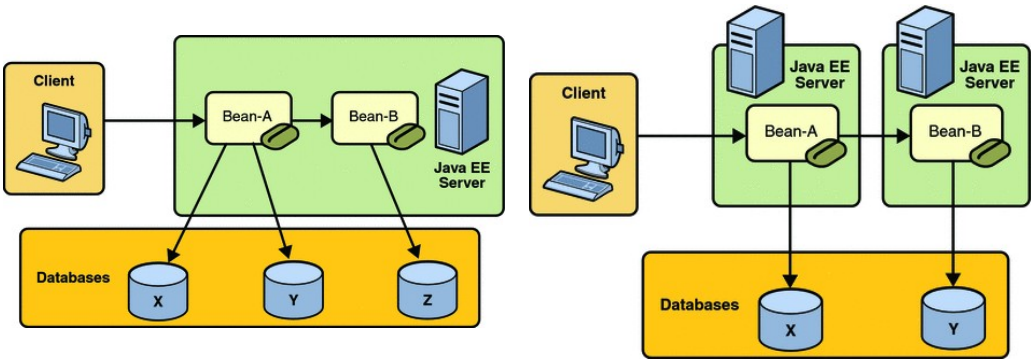
Table 33-1 summarizes the effects of the transaction attributes. Both the T1 and the T2 transactions are controlled by the container. A T1 transaction is associated with the client that calls a method in the enterprise bean. In most cases, the client is another enterprise bean. A T2 transaction is started by the container just before the method executes.

In the last column of Table 33-1, the word **None** means that the business method does not execute within a transaction controlled by the container. However, the database calls in such a business method might be controlled by the transaction manager of the DBMS.

JTA

```
@TransactionAttribute(NOT_SUPPORTED)
@Stateful
public class TransactionBean{
    @TransactionAttribute(REQUIRES_NEW)
    public void firstMethod() {…}
    @TransactionAttribute(REQUIRED)
    public void secondMethod() {…}
    public void thirdMethod() {…}
    public void fourthMethod() {…}


@Resource
private SessionContext sctx;

sctx.setRollbackOnly();
```

JTA


Client
Bean-A
Bean-B
Java EE Server
Databases
X
Y
Z
Client
Java EE Server
Bean-A
Java EE Server
Bean-B
Databases
X
Y

JTA

Transakce řízené aplikací

<non-jta-data-source>jdbc/pokus3</non-jta-data-source>

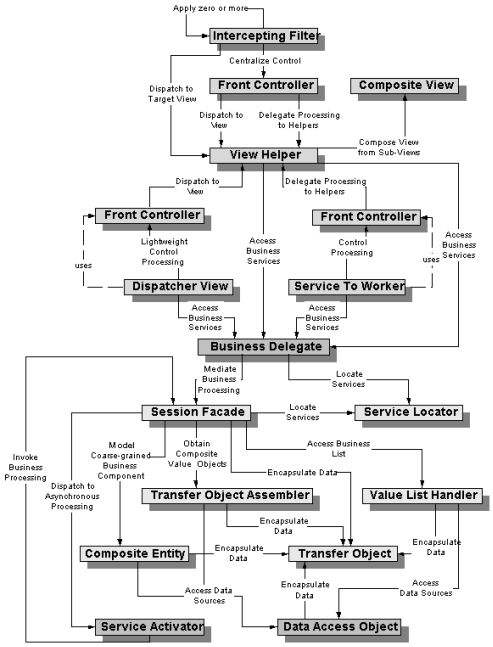
- Umožňuje více transakcí v metodě
- Více práce

```
@Resource
SessionContext context;


UserTransaction utx = context.getUserTransaction();

utx.begin();
// Do work
utx.commit();
```

P8

- Návrhové vzory JavaEE
  - DAO
- http://java.sun.com/blueprints/corej2eepatterns/Patterns/

Návrhové vzory JavaEE



DAO – Data Access Object

Problém

- Různé typy datových uložišť vyžadují různé metody přístupu
- Obtížnost při změně datového uložiště

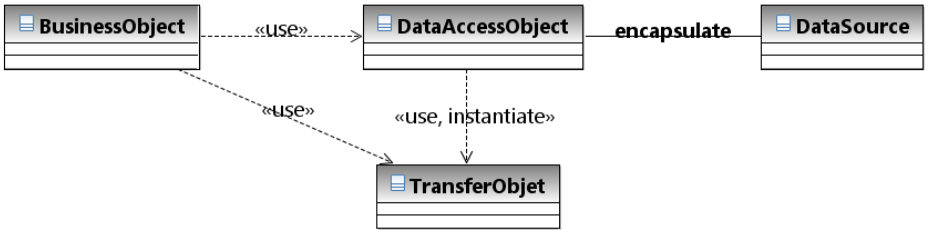
Důvody

- Komponenty potřebují ukládat data
- Přístup k úložištím je rozdílný
- Komponenty obvykle používají proprietární API pro přístup do úložiště
- Snížení přenositelnosti komponent
- Komponenty by mněli být transparentní k implementaci úložiště a mněli by umožňovat snadnou migraci

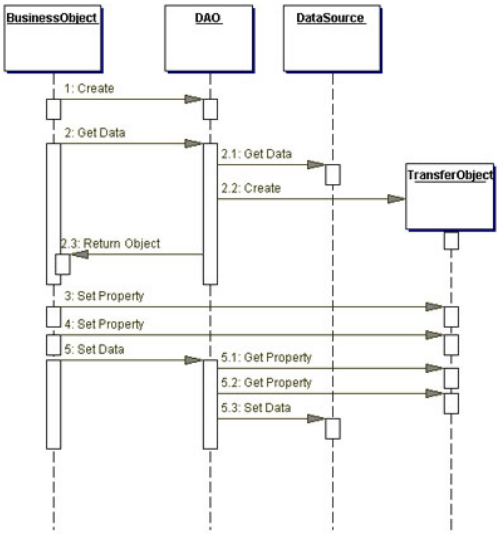
DAO – Data Access Object

Řešení

- Použití DAO objektu pro obalení veškerého přístupu k úložišti. DAO má na starost připojení k úložišti a uložení nebo získání dat.
- DAO poskytuje jednoduché a naprosto na implementaci úložiště nezávislé rozhraní.



DAO – Data Access Object



BusinessObject

The BusinessObject represents the data client. It is the object that requires access to the data source to obtain and store data. A BusinessObject may be implemented as a session bean, entity bean, or some other Java object, in addition to a servlet or helper bean that accesses the data source.

DataAccessObject

The DataAccessObject is the primary object of this pattern. The DataAccessObject abstracts the underlying data access implementation for the BusinessObject to enable transparent access to the data source. The BusinessObject also delegates data load and store operations to the DataAccessObject.

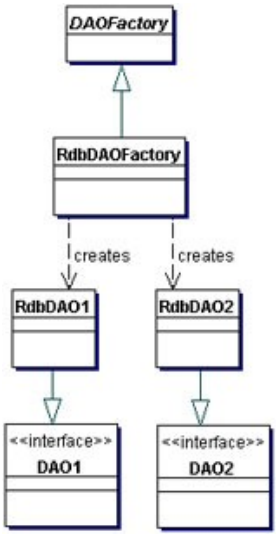
DataSource

This represents a data source implementation. A data source could be a database such as an RDBMS, OODBMS, XML repository, flat file system, and so forth. A data source can also be another system (legacy/mainframe), service (B2B service or credit card bureau), or some kind of repository (LDAP).

TransferObject

This represents a Transfer Object used as a data carrier. The DataAccessObject may use a Transfer Object to return data to the client. The DataAccessObject may also receive the data from the client in a Transfer Object to update the data in the data source.

Továrna pro DAO



BusinessObject

The BusinessObject represents the data client. It is the object that requires access to the data source to obtain and store data. A BusinessObject may be implemented as a session bean, entity bean, or some other Java object, in addition to a servlet or helper bean that accesses the data source.

DataAccessObject

The DataAccessObject is the primary object of this pattern. The DataAccessObject abstracts the underlying data access implementation for the BusinessObject to enable transparent access to the data source. The BusinessObject also delegates data load and store operations to the DataAccessObject.

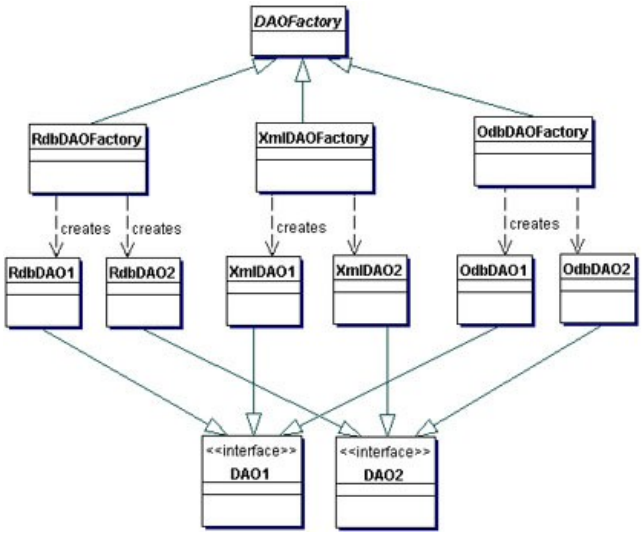
DataSource

This represents a data source implementation. A data source could be a database such as an RDBMS, OODBMS, XML repository, flat file system, and so forth. A data source can also be another system (legacy/mainframe), service (B2B service or credit card bureau), or some kind of repository (LDAP).

TransferObject

This represents a Transfer Object used as a data carrier. The DataAccessObject may use a Transfer Object to return data to the client. The DataAccessObject may also receive the data from the client in a Transfer Object to update the data in the data source.

Továrna pro DAO



BusinessObject

The BusinessObject represents the data client. It is the object that requires access to the data source to obtain and store data. A BusinessObject may be implemented as a session bean, entity bean, or some other Java object, in addition to a servlet or helper bean that accesses the data source.

DataAccessObject

The DataAccessObject is the primary object of this pattern. The DataAccessObject abstracts the underlying data access implementation for the BusinessObject to enable transparent access to the data source. The BusinessObject also delegates data load and store operations to the DataAccessObject.

DataSource

This represents a data source implementation. A data source could be a database such as an RDBMS, OODBMS, XML repository, flat file system, and so forth. A data source can also be another system (legacy/mainframe), service (B2B service or credit card bureau), or some kind of repository (LDAP).

TransferObject

This represents a Transfer Object used as a data carrier. The DataAccessObject may use a Transfer Object to return data to the client. The DataAccessObject may also receive the data from the client in a Transfer Object to update the data in the data source.

P10

- XML
- Java XML API
  - JAXP (SAX, DOM, XSLT, StAX)
    - http://java.sun.com/javase/6/docs/technotes/guides/xml/jaxp/index.html
  - JAXB
    - http://java.sun.com/developer/technicalArticles/WebServices/jaxb/
- Webové služby

Co jsou to webové služby

- Rozhraní k aplikaci dostupné prostřednictvím počítačové sítě, založené na standardních internetových technologiích.
- Obecně: je-li aplikace dostupná přes síť pomocí protokolů jako HTTP, XML, SMTP, nebo Jabber, je to webová služba.
- Vrstva mezi aplikačním programem a klientem.

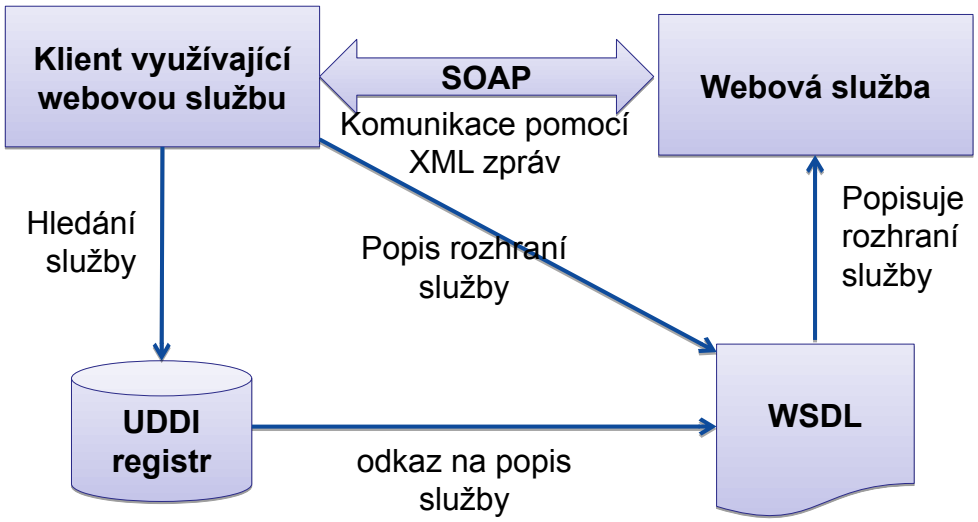
Co jsou to webové služby

- Funkčnost služby není závislá na jazyku v jakém je klient nebo server implementován (Java, C++, PHP, C#, ...).
- Příklad: HTML stránky:
  - server=WWW server, klient=prohlížeč
- V dnešní době nechápeme webové služby takto obecně, webová služba je množina konkrétních specifikací (W3C).
- Dostupné služby: kurzovní lístky, burza, vyhledávací služby (Google), mapy, počasí.
- Komponenty distribuované aplikace?

Architektura webových služeb

- Množina protokolů, http://www.w3.org/2002/ws/:
  - Přenos zpráv – SOAP,
    - http://www.w3.org/2000/xp/Group/.
  - Popis služeb – WSDL,
    - http://www.w3.org/2002/ws/desc/.
  - Hledání služeb – UDDI.

Architektura webových služeb



Web Services Description Language (WSDL)

- Popis webové služby založený na XML.
- IBM, Microsoft, dnes W3C.
- WSDL soubor s definicí rozhraní služby je XML dokument, obsahuje definici:
  - Metod,
  - Parametrů.

Příklad, webová služba pro přístup ke zdroji XML dat

- Metody:
  - Index – vytvoření nové databáze a vložení kolekce XML dokumentů
  - Query – dotaz
  - DatabaseList – seznam databází.
  - ResourceList – seznam odkazů ve stránce

Příklad WSDL

```
<wsdl:definitions
    targetNamespace=" http://tempur i.org / ">
<wsdl : types>
    <s:schema elementFormDefault="qualified "
        targetNamespace=" http://tempuri.org / ">
. . .
    <s:element name="Query">
        <s:complexType><s:sequence>
            <s:element minOccurs="1" maxOccurs="1"
                name="dbId" type="s:int" / >
            <s:element minOccurs="0" maxOccurs="1"
                name="query" type="s:string"/>
        </s:sequence></s:complexType>
    </s:element>
```

Simple Object Access Protocol (SOAP)

- Standardní protokol pro obalování zpráv sdílených mezi aplikacemi (obálka + sada pravidel pro reprezentaci dat v XML).
- Zprávy SOAP je možné zabalit do různých protokolů, např. HTTP. Můžeme jej ovšem použít pro RPC (Remote Procedure Call).
- Skládá se ze tří částí:
  - envelope - definuje co zpráva obsahuje a jak ji zpracovat.
  - Množina kódovacích pravidel – např. serializace primitivních datových typů pro RPC, zasílání zpráv pomocí HTTP.
  - Konvence pro reprezentaci volání vzdálených procedur.

Simple Object Access Protocol (SOAP)

- SOAP je založen na XML.
- SOAP je poměrně jednoduchý
- Neřeší transakce a bezpečnost.
- Zpráva obsahuje element Envelope, který obsahuje:
  - hlavičku – informace,
  - tělo – metainformace.

Příklad SOAP 1.2, request 1/2

```
POST /AmphorAWS/AmphorAWS.asmx HTTP/1.1
Host : localhost
Content−Type: application/soap+xml;charset=utf−8
Content−Length: length

<?xml version="1.0" encoding="utf−8" ?>
<soap12:Envelope
xmlns:xsi="http://www.w3.org/2001/XMLSchema−instance"
xmlns:xsd="http://www.w3. org/2001/XMLSchema"
xmlns:soap12="http://www.w3.org/2003/05/
  soap−envelope">
```

Příklad SOAP 1.2, request 2/2

```
<soap12:Body>
    <Query xmlns="http://tempuri.org/">
        <dbId>1</dbId>
        <query>
            doc('books.xml')/books/book[author/last='Fernadez']
        </query>
    </Query>
</soap12:Body>
</soap12:Envelope>
```

Příklad SOAP 1.2, response 1/2

```
HTTP/1.1 200OK
Content−Type: application/soap+xml ; charset=utf−8
Content−Length: length

<?xml version="1.0" encoding="utf−8" ?>
<soap12:Envelope
xmlns:xsi="http://www.w3.org/2001/XMLSchema−instance"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:soap12="http://www.w3.org/2003/05/soap−envelope">
```

Příklad SOAP 1.2, response 2/2

```
<soap12:Body>
    <QueryResponse xmlns="http://tempuri.org/">
        <QueryResult>string</QueryResult>
    </QueryResponse>
</soap12:Body>
</soap12:Envelope>
```

Universal Description, Discovery and Integration (UDDI)

- Registrace a vyhledávání webových služeb.
- Nabízí veřejnou databázi (registry). Např. dvě největší databáze spravují ~~IBM a Microsoft.~~
- UDDI registr obsahuje čtyři druhy entit:
  - podnikatelské entity (business entity).
  - služby (business service).
  - šablony vazeb (binding template), např. popis pomocí WSDL.
  - typy služeb (service type).

Java web services

- Standardní JavaEE webová aplikace
- Definice třídy:

```
@WebService(name="TestWS")
public class MyWebService {
   @WebMethod
   public String sayHallo(int nTimes){
      String ret = "";
      for(int i=0; i<nTimes; i++){
         ret += "Ahoj ";
      }
      return ret;
   }
}
```

Java web services

- WEB-INF/web.xml (pouze JBoss server)

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns:xsi="http… >
  <display-name>EnterpriseWeb</display-name>
  <servlet>
    <description></description>
    <display-name>Hello</display-name>
    <servlet-name>Hello</servlet-name>
    <servlet-class>webService.MyWebService</servlet-class>
  </servlet>
  <servlet-mapping>
    <servlet-name>Hello</servlet-name>
    <url-pattern>/Hello</url-pattern>
  </servlet-mapping>
</web-app>
```

Java WS - klient

- Vygenerování kódu klienta:
- <jboss-install-dir>/bin
  - wsconsume.bat -v -k -p ws -o "\EnterpriseWebClient\src"
  - wsimport.bat –keep WSDL_URI

http://localhost:8080/EnterpriseWeb/Hello?wsdl

~~http://localhost:8080/EnterpriseWeb/Hello?Tester~~

Java WS - klient

- Použití v JavaSE aplikaci:

```
public class WebServiceClient {
   public static void main(String[] args){
      MyWebService ws = new
               MyWebServiceLocator().getMyWebServicePort();
      String response = ws.sayHallo(5);
      System.out.println("Web service response:" +

            response);
   }
}
```


Jboss server nesmí být spuštěn z eclipse ale z příkazové řádky, aby nechyběla definice -Djava.endorsed.dirs=/<JBOSS_HOME>/lib/endorsed

P6

- EJB - Enterprise JavaBeans

Enterprise Java Beans (EJB)

- Specifikace architektury pro vývoj a nasazení distribuovaných transakčních objektových komponent na straně serveru
- Konvence + sada rozhraní (EJB API)
- Cíl = zajištění kompatibility mezi produkty různých výrobců
  - komponenty
  - Kontejner
- EJB 3.0

Enterprise JavaBeans

- EnterpriseBean jsou komponenty implementující technologii Enterprise JavaBeans (EJB)
- EnterpriseBean běží v EJB kontejneru
- EnterpriseBean je serverová komponenta obalující aplikační logiku
- EnterpriseBean lze volat vzdáleně

„Webová služba obalená do objektu bez XML“, ale může udržovat stav.

EJB kontejner

- prostředí, v němž běží komponenty
  - vzdálený přístup
  - bezpečnost
  - transakce
  - souběžný přístup
  - přístup ke zdrojům a jejich sdílení
- izolace komponent od aplikací
  - nezávislost na dodavateli kontejneru
  - zjednodušení tvorby aplikací

When toUse Enterprise Beans

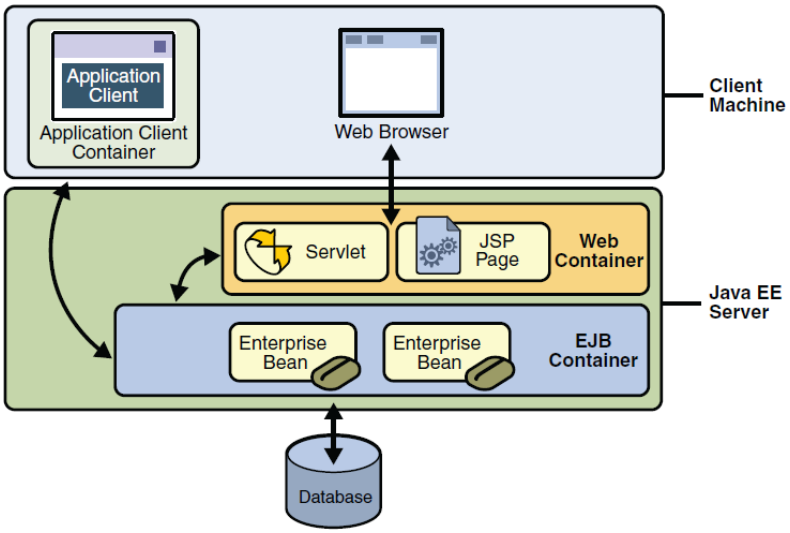
You should consider using enterprise beans if your application has any of the following requirements:

The application must be scalable. To accommodate a growing number of users, you may need to distribute an application's components across multiple machines.Not only can the enterprise beans of an application run on different machines, but also their location will remain transparent to the clients.

Transactions must ensure data integrity. Enterprise beans support transactions, the mechanisms that manage the concurrent access of shared objects.

The application will have a variety of clients. With only a few lines of code, remote clients can easily locate enterprise beans. These clients can be thin, various, and numerous.

EJB – použití


Application Client
Application Client Container
Web Browser
Client Machine
Servlet
JSP Page
Web Container
Java EE Server
Enterprise Bean
Enterprise Bean
EJB Container
Database

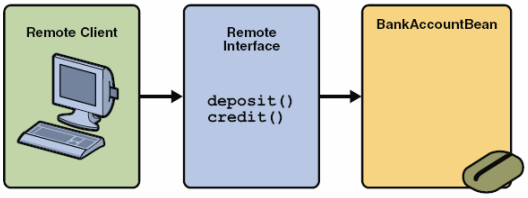
Typy komponent EJB

- Session Bean:
  - Stateless session bean: bezstavové služby
  - Statefull session bean: stavové objekty v rámci session
  - Singelton
- Message-Driven Beans
  - bezstavové služby volané asynchronně

What Is a Session Bean?
A *session bean represents a single client inside the Application Server. To access an application* that is deployed on the server, the client invokes the session bean's methods. The session bean performs work for its client, shielding the client from complexity by executing business tasks inside the server.
As its name suggests, a session bean is similar to an interactive session. A session bean is not shared; it can have only one client, in the same way that an interactive session can have only one user. Like an interactive session, a session bean is not persistent. (That is, its data is not saved to a database.) When the client terminates, its session bean appears to terminate and is no longer associated with the client.
For code samples, see Chapter 22, "Session Bean Examples."
State Management Modes
There are two types of session beans: stateful and stateless.
Stateful Session Beans
The state of an object consists of the values of its instance variables. In a *stateful session bean,* the instance variables represent the state of a unique client-bean session. Because the client interacts ("talks") with its bean, this state is often called the *conversational state.*
The state is retained for the duration of the client-bean session. If the client removes the bean or terminates, the session ends and the state disappears. This transient nature of the state is not a problem, however, because when the conversation between the client and the bean ends there is no need to retain the state.
Stateless Session Beans
A *stateless session bean does not maintain a conversational state with the client. When a client* invokes the methods of a stateless bean, the bean's instance variables may contain a state specific to that client, but only for the duration of the invocation. When the method is finished, the client-specific state should not be retained. Clients may, however, change the state of instance variables in pooled stateless beans, and this state is held over to the next invocation of the pooled stateless bean. Except during method invocation, all instances of a stateless bean are equivalent, allowing the EJB container to assign an instance to any client. That is, the state of a stateless session bean should apply accross all clients.
Because stateless session beans can support multiple clients, they can offer better scalability for applications that require large numbers of clients. Typically, an application requires fewer stateless session beans than stateful session beans to support the same number of clients.
A stateless session bean can implement a web service, but other types of enterprise beans cannot.

SessionBean - Rozhraní

- Klient přistupuje přes rozhraní (business interface)
- Jedna bean může mít více byznys rozhraní




```
@Remote
public interface Account {
}
@Stateless
public class AccountBean implements Account {
    public AccountBean() {
    }
}
```

SessionBean - použití

- Ke konkrétní instanci přistupuje ve stejný čas pouze jeden klient
- Stav není perzistentní, pouze krátkou dobu (hodiny)
- Webové služby
- Statefull
  - Interakce mezi SB a klientem, udržování informací mezi voláním SB
- Stateless
  - Není třeba udržovat data pro konkrétního klienta
  - Obecné úlohy

SessionBean – Remote vs. Local

Vzdálený klient

- Může běžet na vzdáleném počítači (jiná JVM)
- Klient může být:
  - Webová komponenta
  - Aplikace
  - jiná EJB
- Velká izolace parametrů metod
  - Klient a bean pracují s rozdílnými kopiemi objektů
  - Zvýšená ochrana
- Granularita dat

Lokální klient

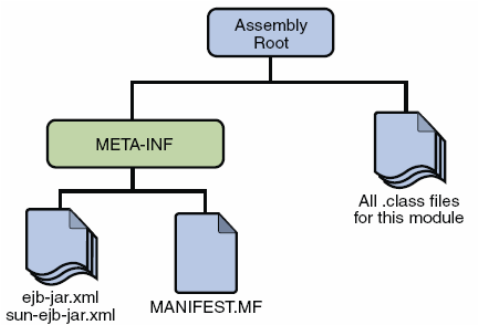
- Musí běžet ve stejné JVM
- Klient může být:
  - webová komponenta
  - Jiná EJB
- Slabá izolace
  - Klient a bean pracují se stejnými objekty
  - Změna provedená bean-ou se projeví i u klienta
  - Nižší ochrana

Deciding on Remote or Local Access
Whether to allow local or remote access depends on the following factors.
Tight or loose coupling of related beans: Tightly coupled beans depend on one another.
For example, if a session bean that processes sales orders calls a session bean that emails a confirmation message to the customer, these beans are tightly coupled. Tightly coupled beans are good candidates for local access. Because they fit together as a logical unit, they typically call each other often and would benefit from the increased performance that is possible with local access.
Type of client: If an enterprise bean is accessed by application clients, then it should allow remote access. In a production environment, these clients almost always run on different machines than the Application Server. If an enterprise bean's clients are web components or other enterprise beans, then the type of access depends on how you want to distribute your components.
Component distribution: Java EE applications are scalable because their server-side components can be distributed across multiple machines. In a distributed application, for example, the web components may run on a different server than do the enterprise beans they access. In this distributed scenario, the enterprise beans should allow remote access.
Performance:Due to factors such as network latency, remote calls may be slower than local calls. On the other hand, if you distribute components among different servers, you may improve the application's overall performance. Both of these statements are generalizations; actual performance can vary in different operational environments.Nevertheless, you should keep in mind how your application design might affect performance.
If you aren't sure which type of access an enterprise bean should have, choose remote access. This decision gives you more flexibility. In the future you can distribute your components to accommodate the growing demands on your application.
Defining Client Access with Interfaces
640 The Java EE 5Tutorial • October 2008
Although it is uncommon, it is possible for an enterprise bean to allow both remote and local access. If this is the case, either the business interface of the bean must be explicitly designated as a business interface by being decorated with the @Remote or @Local annotations, or the bean class must explicitly designate the business interfaces by using the @Remote and @Local annotations. The same business interface cannot be both a local and remote business interface.

Method Parameters and Access
The type of access affects the parameters of the bean methods that are called by clients. The following topics apply not only to method parameters but also to method return values.
Isolation
The parameters of remote calls are more isolated than those of local calls. With remote calls, the client and bean operate on different copies of a parameter object. If the client changes the value of the object, the value of the copy in the bean does not change. This layer of isolation can help protect the bean if the client accidentally modifies the data.
In a local call, both the client and the bean can modify the same parameter object. In general, you should not rely on this side effect of local calls. Perhaps someday you will want to distribute your components, replacing the local calls with remote ones.
As with remote clients, web service clients operate on different copies of parameters than does the bean that implements the web service.
Granularity of Accessed Data
Because remote calls are likely to be slower than local calls, the parameters in remote methods should be relatively coarse-grained. A coarse-grained object contains more data than a fine-grained one, so fewer access calls are required. For the same reason, the parameters of the methods called by web service clients should also be coarse-grained.

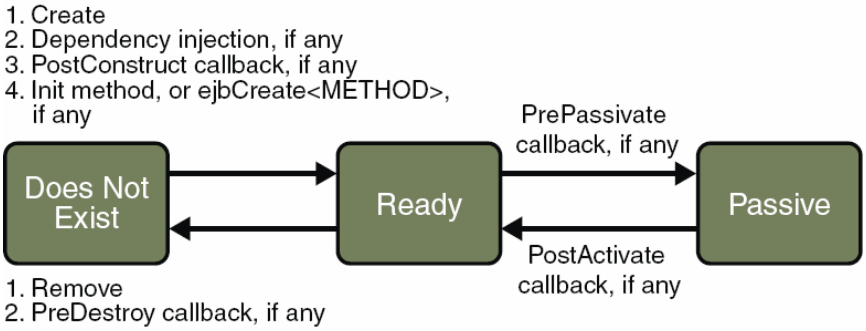
SessionBean - implementace

- Rozhraní `interface JménoRozhraní`
- Třída komponenty
  `class JménoRozhraníBean implements JménoRozhraní`
- Pomocné třídy



Statefull SessionBean – životní cyklus

@PostConstruct
@PreDestroy
@Remove
@PrePassivate
@PostActivate

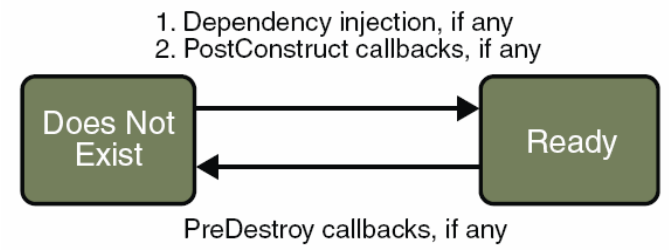


The Life Cycles of Enterprise Beans
An enterprise bean goes through various stages during its lifetime, or life cycle. Each type of enterprise bean (stateful session, stateless session, or message-driven) has a different life cycle. The descriptions that follow refer to methods that are explained along with the code examples in the next two chapters. If you are new to enterprise beans, you should skip this section and run the code examples first.
The Life Cycle of a Stateful Session Bean
Figure 20–3 illustrates the stages that a session bean passes through during its lifetime. The client initiates the life cycle by obtaining a reference to a stateful session bean. The container performs any dependency injection and then invokes the method annotated with @PostConstruct, if any. The bean is now ready to have its business methods invoked by the client.
While in the ready stage, the EJB container may decide to deactivate, or *passivate, the bean by* moving it from memory to secondary storage. (Typically, the EJB container uses a least-recently-used algorithm to select a bean for passivation.) The EJB container invokes the method annotated @PrePassivate, if any, immediately before passivating it. If a client invokes a business method on the bean while it is in the passive stage, the EJB container activates the bean, calls the method annotated @PostActivate, if any, and then moves it to the ready stage.
At the end of the life cycle, the client invokes a method annotated @Remove, and the EJB container calls the method annotated @PreDestroy, if any. The bean's instance is then ready for garbage collection.
Your code controls the invocation of only one life-cycle method: the method annotated @Remove. All other methods in Figure 20–3 are invoked by the EJB container. See Chapter 34, "Resource Connections," for more information.

Stateless SessionBean – životní cyklus

@PostConstruct

@PreDestroy



The Life Cycle of a Stateless Session Bean

Because a stateless session bean is never passivated, its life cycle has only two stages: nonexistent and ready for the invocation of business methods. Figure 20–4 illustrates the stages of a stateless session bean.

The client initiates the life cycle by obtaining a reference to a stateless session bean. The container performs any dependency injection and then invokes the method annotated @PostConstruct, if any. The bean is now ready to have its business methods invoked by the client.

At the end of the life cycle, the EJB container calls the method annotated @PreDestroy, if any. The bean's instance is then ready for garbage collection.

SessionBean – implementace server

```
@Remote
public interface Account {
    public void deposite(String accNum, float amounght);
    public void remove(String accNum, float amounght);
}
@Stateless(mappedName = "comp/env/ejb/Account")
public class AccountBean implements Account {
    public AccountBean() {
    }
    @Override
    public void deposite(String accNum, float amounght) {
    }
    @Override
    public void remove(String accNum, float amounght) {
    }
}
```

<projectName>EAR/<BeanName>Bean/remote

SessionBean – interfaces

```
@Remote

@Local

@LocalBean
```

SessionBean – implementace server

```
@Remote
public interface Account {
    public void deposite(String accNum, float amounght);
    public void remove(String accNum, float amounght);
}
@Stateless(mappedName = "comp/env/ejb/Account")
public class AccountBean implements Account {
    public AccountBean() {
    }
    @Override
    public void deposite(String accNum, float amounght) {
    }
    @Override
    public void remove(String accNum, float amounght) {
    }
}
```

<projectName>EAR/<BeanName>Bean/remote

SessionBean – implementace klient

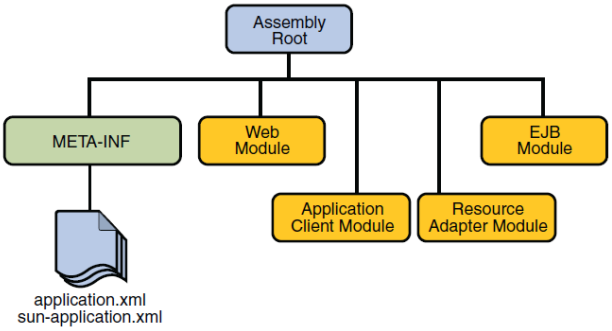
- Klient musí mít k dispozici Business interface (např. jako *.jar)
- Klient využívá knihovny JavaEE
- Typy klientů
  - Application Client Container
  - Běžná aplikace
  - Webový klient

SessionBean – implementace klient běžná aplikace

```
public class Main {
    @EJB
    protected static Account a;
    public static void main(String[] args) {
        Properties props = new Properties();
        props.setProperty(Context.INITIAL_CONTEXT_FACTORY,
                "org.jnp.interfaces.NamingContextFactory");
        props.setProperty(Context.URL_PKG_PREFIXES,
                "org.jboss.naming.client");
        props.setProperty(Context.PROVIDER_URL,
                "jnp://localhost:1099");
        InitialContext ctx = new InitialContext(props);
        a = (Account)ctx.lookup("comp/env/ejb/Account");
        System.out.println("Zustatek: " +
                a.deposite("1234/0300", 10.25f));
    }
}
```

SessionBean – implementace klient ACC

```
public class RunClient {
    @EJB
    static protected Account ac;
    public static void main(String[] args) {
        System.out.println("Zustatek: " +
                    ac.deposite("1234/0300", 10.25f));
    }
}
```



SessionBean – implementace klient ACC

Spouštění JBOSS:

org.jboss.client.AppClientMain
-jbossclient RunClient
-launchers
org.jboss.ejb3.client.ClientLauncher
-j2ee.clientName EnterpriseEJBClient
CLLASSPATH

Nastavení proměnných pro JNDI

- java.naming.factory.initial=org.jnp.interfaces.NamingContextFactory
- java.naming.provider.url=jnp://localhost:1099

- Spouštění GlassFish

com.sun.enterprise.appclient.Main
CLLASSPATH

Nastavení proměnných pro JNDI

- appserv-launch.jar
- java.naming.factory.state=com.sun.corba.ee.impl.presentation.rmi.JNDIStateFactoryImpl

SessionBean – implementace klient ACC

EAR soubor musí obsahovat
/META-INF/aplication.xml

```
<module>   <ejb>EnterpriseEJB.jar</ejb>  </module>
<module>   <java>EnterpriseEJBClient.jar</java>  </module>
```

Pokud chybí neprovede se zveřejnění klientského modulu

JAR soubor s klientem musí obsahovat
/META-INF/application-client.xml

```
<application-client version="5" xmlns=….>
<display-name> EnterpriseEJBClient</display-name>
</application-client>
```

/META-INF/MANIFEST.MF

```
Manifest-Version: 1.0
Class-Path:
Main-Class: MyMain
```

Pokud chybí neprovede se Injection, nebo se vůbec nespustí

SessionBean – implementace klient ACC

- JBOSS server pro výpis na konzoli vyžaduje pro klientskou aplikaci nastavení proměnných prostředí pro log4j
- log4j.properties
  - log4j.rootLogger=DEBUG, stdout
  - log4j.appender.stdout=org.apache.log4j.ConsoleAppender
  - log4j.appender.stdout.layout=org.apache.log4j.PatternLayout
  - log4j.appender.stdout.layout.ConversionPattern=%d %p [%t][%c] - <%m>%n

SessionBean – implementace klient webová aplikace - servlet

```
public class EnterpriseServlet extends HttpServlet {
    @EJB
    protected Account ac;
    public EnterpriseServlet() {
        super();
    }
    protected void doPost(HttpServletRequest request,
HttpServletResponse response) throws ServletException,
IOException {
        response.setContentType("text/html");
        PrintWriter pw = response.getWriter();
        pw.write("<html><body>" + ac.deposite("", 17.5f) +
                "</body></html>");
    }
}
```


Unfortunately for the web application developer, resource injection using annotations can only be used with classes that are managed by a Java EE compliant container. Because the web container does not manage JavaBeans components, you cannot inject resources into them. One exception is a request-scoped JavaServer Faces managed bean. These beans are managed by the container and therefore support resource injection. This is only helpful if your application is a JavaServer Faces application.

You can still use resource injection in a web application that is not a JavaServer Faces application if you can do it in an object that is managed by the container. These objects include servlets and ServletContextListener objects. These objects can then give the application’s beans access to the resources.

In the case of Duke’s Bookstore

SessionBean – implementace klient webová aplikace - JSP

```
<%@ page language="java" contentType="text/html; charset=ISO-8859-1" pageEncoding="ISO-8859-1"%>
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c"%>
<%@page import="javax.ejb.EJB"%>
<%@page import="bank.Account"%>

<%!@EJB
Account account;%>
<html><body>
<%= account.deposite("1234/0300", 125.4f) %>

<% request.setAttribute("account", account); %>
<jsp:setProperty property="ac" name="wrapper" value="${account}"/>

</body></html>
```


Unfortunately for the web application developer, resource injection using annotations can only
be used with classes that are managed by a Java EE compliant container. Because the web
container does not manage JavaBeans components, you cannot inject resources into them. One
exception is a request-scoped JavaServer Faces managed bean. These beans are managed by the
container and therefore support resource injection. This is only helpful if your application is a
JavaServer Faces application.
You can still use resource injection in a web application that is not a JavaServer Faces
application if you can do it in an object that is managed by the container. These objects include
servlets and ServletContextListener objects. These objects can then give the application’s
beans access to the resources.
In the case of Duke’s Bookstore

Statefull SessionBean

- Bean si pamatuje stav (hodnoty proměnných) mezi jednotlivými dotazy
- Klinet má výhradní přístup k jedné beaně po dobu platnosti proměnné oanotované anotací @EJB

Session Bean - Injection

- Injection zdrojů lze pouze u objektů za jejihž vytvoření je zodpovědný JavaEE kontejner
  - Servlety
  - ServletContextListener
  - Managed Backing Beans v JSF aplikacích
- Pokud inicializujete Stateful session bean v JSP/servletu a uložíte odkaz do sessionScope aplikace bude s beanou komunikovat, ale také s ní může komunikovat kdokoliv jiný

Unfortunately for the web application developer, resource injection using annotations can only
be used with classes that are managed by a Java EE compliant container. Because the web
container does not manage JavaBeans components, you cannot inject resources into them. One
exception is a request-scoped JavaServer Faces managed bean. These beans are managed by the
container and therefore support resource injection. This is only helpful if your application is a
JavaServer Faces application.
You can still use resource injection in a web application that is not a JavaServer Faces
application if you can do it in an object that is managed by the container. These objects include
servlets and ServletContextListener objects. These objects can then give the application's
beans access to the resources.
In the case of Duke's Bookstore

Dependency Injection
A session bean may use dependency injection mechanisms to acquire references to resources or other
objects in its environment (see Chapter 16, "Enterprise Bean Environment"). If a session bean makes
use of dependency injection, the container injects these references after the bean instance is created, and
before any business methods are invoked on the bean instance. If a dependency on the SessionContext
is declared, or if the bean class implements the optional SessionBean interface (see Section
4.3.5), the SessionContext is also injected at this time. If dependency injection fails, the bean
instance is discarded.
*Under the EJB 3.0 API, the bean class may acquire the SessionContext interface through
dependency injection without having to implement the SessionBean interface. In this case,
the Resource annotation (or resource-env-ref deployment descriptor element) is
used to denote the bean's dependency on the SessionContext. See Chapter 16, "Enterprise
Bean Environment".*

Stateful Session Bean - JSF

- Lze jednoduše inicializovat EJB pomocí injection v Managed Bean
- Managed Bean pak musí poskytovat rozhraní pro práci s EJB
- Problém je uvolnění EJB – JSF nemá podporu pro ukončení session

```
counter.release();
HttpSession s = (HttpSession)(FacesContext.
      getCurrentInstance().getExternalContext().
      getSession(false));
s.invalidate();
```

Stateful Session Bean - JSF

```
public String logout()
{
  FacesContext.getCurrentInstance().getExternalContext().
    invalidateSession();
  return "/home.xhtml?faces-redirect=true";
}

public void logout() throws IOException {
  ExternalContext ec =
    FacesContext.getCurrentInstance().getExternalContext();
  ec.invalidateSession(); ec.redirect(ec.getRequestContextPath() +
    "/home.xhtml");
}
```

P11

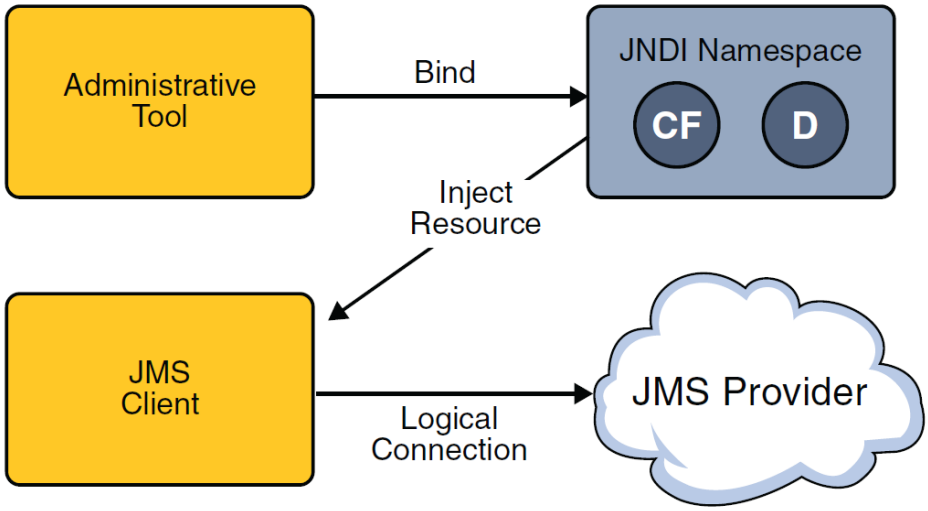
- JTA – Java Transaction
- JMS – Java Message Services
- Message-Driven Beans

- Reference: Java EE Tutorial
  - http://java.sun.com/javaee/5/docs/tutorial/doc/

JMS – Java Message Services

- Asynchronní zasílání zpráv mezi dvěma komponentami
- loosely coupled
- Zaručuje doručení zprávy

JMS – Architektura



A JMS application is composed of the following parts.

■ A *JMS provider is a messaging system that implements the JMS interfaces and provides* administrative and control features. An implementation of the Java EE platform includes a JMS provider.

■ *JMS clients are the programs or components, written in the Java programming language,* that produce and consume messages. Any Java EE application component can act as a JMS client.

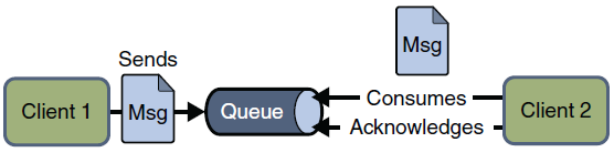
■ *Messages are the objects that communicate information between JMS clients.*

■ *Administered objects are preconfigured JMS objects created by an administrator for the use* of clients. The two kinds of JMS administered objects are destinations and connection factories, which are described in “JMS Administered Objects” on page 903.

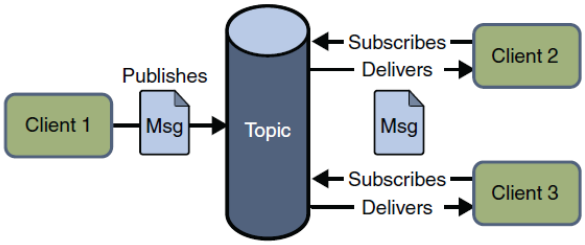
Figure 31–2 illustrates the way these parts interact. Administrative tools allow you to bind destinations and connection factories into a JNDI namespace. A JMS client can then use resource injection to access the administered objects in the namespace and then establish a logical connection to the same objects through the JMS provider.

JMS - domény

- **Point-to-Point Messaging Domain**



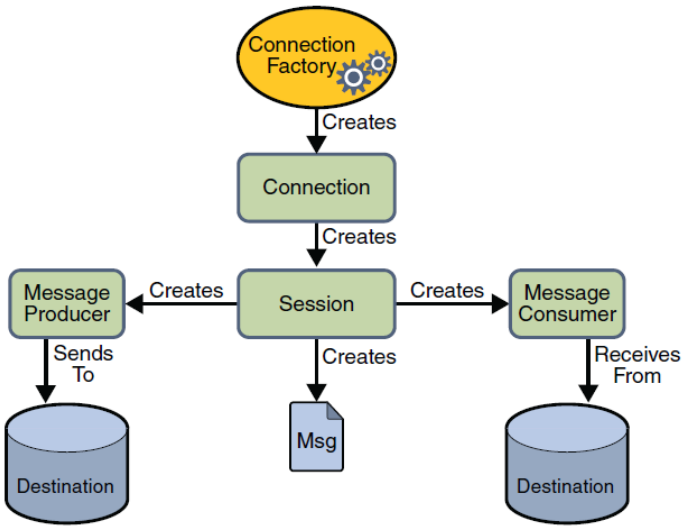
- **Publish/Subscribe Messaging Domain**



Point-to-Point Messaging Domain
A point-to-point (PTP) product or application is built on the concept of message queues, senders, and receivers. Each message is addressed to a specific queue, and receiving clients extract messages from the queues established to hold their messages. Queues retain all messages
sent to them until the messages are consumed or until the messages expire.
PTP messaging has the following characteristics and is illustrated in Figure 31–3.
■ Each message has only one consumer.
■ A sender and a receiver of a message have no timing dependencies. The receiver can fetch the message whether or not it was running when the client sent the message.
■ The receiver acknowledges the successful processing of a message.
Use PTP messaging when every message you send must be processed successfully by one consumer.

Publish/Subscribe Messaging Domain
In a publish/subscribe (pub/sub) product or application, clients address messages to a *topic,* which functions somewhat like a bulletin board. Publishers and subscribers are generally anonymous and can dynamically publish or subscribe to the content hierarchy. The system takes care of distributing the messages arriving from a topic's multiple publishers to its multiple subscribers. Topics retain messages only as long as it takes to distribute them to current subscribers.
Pub/sub messaging has the following characteristics.
■ Each message can have multiple consumers.
■ Publishers and subscribers have a timing dependency. A client that subscribes to a topic can consume only messages published after the client has created a subscription, and the subscriber must continue to be active in order for it to consume messages.
The JMS API relaxes this timing dependency to some extent by allowing subscribers to create *durable subscriptions, which receive messages sent while the subscribers are not active. Durable*
subscriptions provide the flexibility and reliability of queues but still allow clients to send messages to many recipients. For more information about durable subscriptions, see "Creating Durable Subscriptions" on page 944.
Use pub/sub messaging when each message can be processed by zero, one, or many consumers.
Figure 31–4 illustrates pub/sub messaging.

JMS – Model


Connection Factory
Creates
Connection
Creates
Message Producer
Creates
Session
Creates
Message Consumer
Sends To
Creates
Receives From
Destination
Msg
Destination

JMS – Administrativní objekty

- JMS Connection Factories

```
@Resource(mappedName="jms/ConnectionFactory")
private static ConnectionFactory connectionFactory;

Connection con = connectionFactory.createConnection();
con.close();
```

- JMS Destinations

```
@Resource(mappedName="jms/Queue")
private static Queue queue;

@Resource(mappedName="jms/Topic")
private static Topic topic;
```

JMS – session

- Session je jednovláknový kontext pro odesílání a přijímání zpráv

```
Session session = connection.createSession(false,
      Session.AUTO_ACKNOWLEDGE);
```

- Vytváření zpráv

```
MessageProducer produc = session.createProducer(queue);
produc.send(message);
```

- Příjem zpráv

```
MessageConsumer consum = session.createConsumer(queue);
connection.start();
Message m = consum.receive();
```

JMS – message listener

- Posluchač slouží k asynchronímu příjmu zpráv

```
public abstract interface MessageListener
{
  public abstract void onMessage(Message paramMessage);
}
```

- Metoda onMessage „nesmí" vyhazovat žádnou vyjímku

```
MessageListener myListener = new MyMessageListener();
consumer.setMessageListener(myListener);
```

JMS - zprávy

- Každá zpráva má tři části
  - Hlavičku:
    - JMSDestination, JMSDeliveryMode, JMSExpiration, JMSPriority, JMSMessageID, JMSTimestamp (nastavují se automaticky při odeslání)
    - **JMSCorrelationID, JMSReplyTo, JMSType (nastavuje klient)**
    - JMSRedelivered (nastavuje JMS provider)
  - Vlastnosti:
    - Uživatelem definované vlastnosti, které se přidají do hlavičky

JMS - zprávy

- Tělo zprávy:
  - **TextMessage** – tělo je řetězec
  - **MapMessage** – mapa jmen(retězců) a hodnot (primitivní typy i objekty)
  - **BytesMessage** – pole bytů, koresponduje s datovým streamem (primitivní typy)
  - **StreamMessage** – stream
  - **ObjectMessage** – tělem je objekt, který implementuje rozhraní Serializable
  - **Message** – prázdné tělo

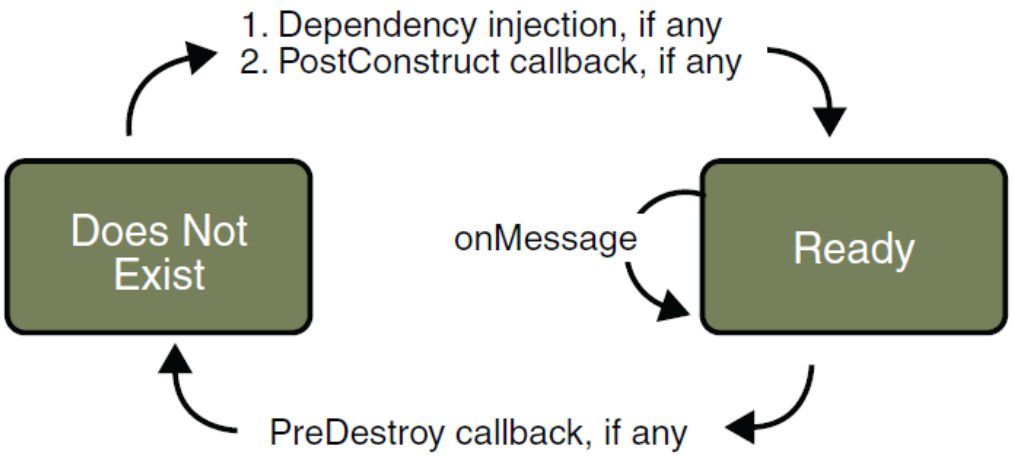
JMS – příklad

```
TextMessage message = session.createTextMessage();
message.setText("text");
producer.send(message);

Message m = consumer.receive();
if (m instanceof TextMessage) {
   TextMessage message = (TextMessage) m;
   System.out.println("Read: " + message.getText());
} else {
   // Handle error
}
```

Message – Driven Beans

- Java EE bean dovolující asynchronní zpracování událostí
- Zpracovává JMS zprávy
- Činnost je vyvolána příchodem nové zprávy
- Nemá rozhraní
- Nejvíce se podobá Stateless session beaně
- Jedna bean může zpracovávat zprávy od více klientů

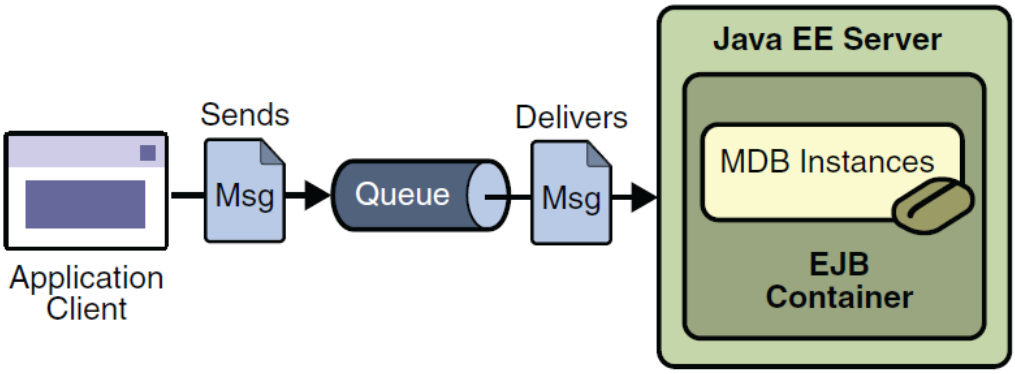
MDB – životní cyklus



The EJB container usually creates a pool of message-driven bean instances. For each instance,
the EJB container performs these tasks:
1. If the message-driven bean uses dependency injection, the container injects these references
before instantiating the instance.
2. The container calls the method annotated @PostConstruct, if any.

Like a stateless session bean, a message-driven bean is never passivated, and it has only two
states: nonexistent and ready to receive messages.
At the end of the life cycle, the container calls the method annotated @PreDestroy, if any. The
bean’s instance is then ready for garbage collection.

MDB - příklad


Sends
Delivers
Msg
Queue
Msg
Application Client
Java EE Server
MDB Instances
EJB Container

MDB – příklad client

```
public class SimpleMessageClient {
   @Resource(mappedName = "jms/ConnectionFactory")
   private static ConnectionFactory connectionFactory;
   @Resource(mappedName = "jms/Queue")
   private static Queue queue;

   public static void main(String[] args) {
      Connection connection = null;
      Session session = null;
      MessageProducer messageProducer = null;
      TextMessage message = null;
      final int NUM_MSGS = 3;
      try {
         connection = connectionFactory.createConnection();
         session = connection.createSession(false,
                                            Session.AUTO_ACKNOWLEDGE);
         messageProducer = session.createProducer(queue);
         message = session.createTextMessage();
```

```
            for (int i = 0; i < NUM_MSGS; i++) {
                message.setText("This is message " + (i + 1));
                System.out.println("Sending message: " +

    message.getText());
                messageProducer.send(message);
            }
        } catch (JMSException e) {
            System.out.println("Exception occurred: " + e.toString());
        } finally {
            if (connection != null) {
                try {
                    connection.close();
                } catch (JMSException e) {
                }
            } // if
            System.exit(0);
        } // finally
    } // main
} // class
```

MDB – příklad MDB

```
@MessageDriven(mappedName = "jms/Queue")
public class SimpleMessageBean implements MessageListener {
    static final Logger logger =
                        Logger.getLogger("SimpleMessageBean");
    @Resource
    private MessageDrivenContext mdc;
    public SimpleMessageBean() {
    }
    public void onMessage(Message inMessage) {
        TextMessage msg = null;
        try {
            if (inMessage instanceof TextMessage) {
                msg = (TextMessage) inMessage;
                logger.info("MESSAGE BEAN: Message received: " +

        msg.getText());
            }
```

```
            else {
                logger.warning("Message of wrong type: "
                        + inMessage.getClass().getName());
            }
        } catch (JMSException e) {
            e.printStackTrace();
        } catch (Throwable te) {
            te.printStackTrace();
        }
    }
}
```

P12

- Portlet
  - http://jcp.org/aboutJava/communityprocess/final/jsr286/index.html
  - http://portals.apache.org/pluto/
  - Java Portlet Specification (JSR168,JSR286)
- RMI – Remote Method Invocations
  - http://java.sun.com/javase/technologies/core/basic/rmi/index.jsp
- CORBA - Common Object Request Broker Architecture
  - http://java.sun.com/javase/technologies/core/corba/index.jsp
  - http://queue.acm.org/detail.cfm?id=1142044

Portal

Co je to portál?

- Portále je webová aplikace běžně poskytující personalizaci, autentizaci, agregování obsahu z více zdrojů a poskytuje prezentační vrstvu pro IS.
- Agregace je integrace obsahu z více zdrojů do webové stránky.
- Portál může mít sofistikovanou personalizaci pro poskytnutí osobního obsahu.
  - Portálová stránka může mít rozdílné sady portletů pro rozdílné uživatele

Portlet

Co je to portlet?

- Portlet poskytuje specifický kousek obsahu, jako součást portálové stránky.
- Portlet připojitelná je UI komponenta, řízená a zobrazovaná na webovém portálu.
- Portlet produkuje část (fragment) HTML (XHTML, WML), která je agregována do portálové stránky.

Portlet

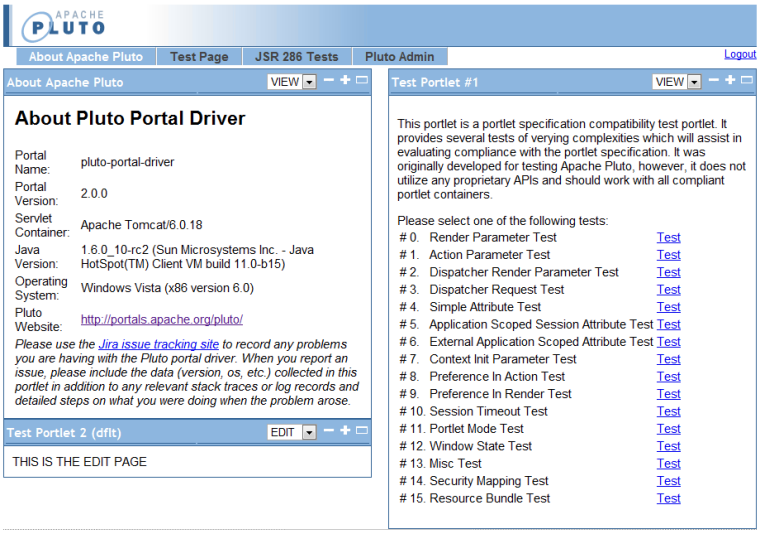
Portálová stránka

- Portálová stránka je kolekce nepřekrývajících se oken portletů

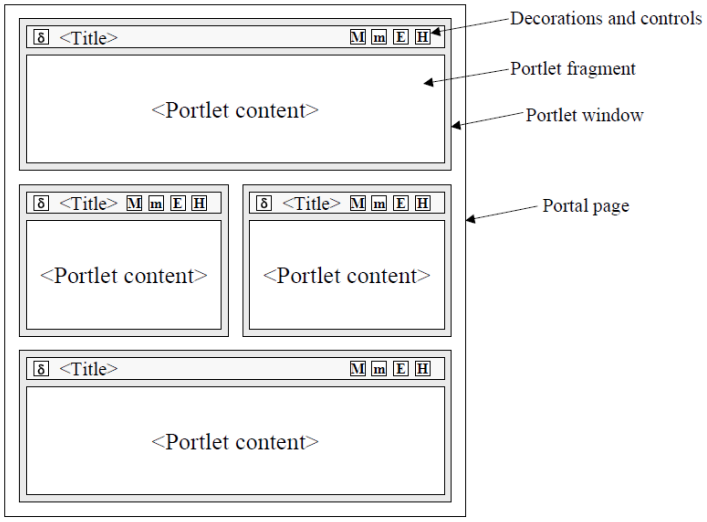
Co je to Portlet kontejner

- Portletový kontejner spouští portlety a poskytuje jim perostředí a řídí jejich životní cyklus.
- Poskytuje perzistentní úložiště nastavení portletů.
- Kontejner neagreguje obsah portletů do stránky, to je zodpovědnost portálu.

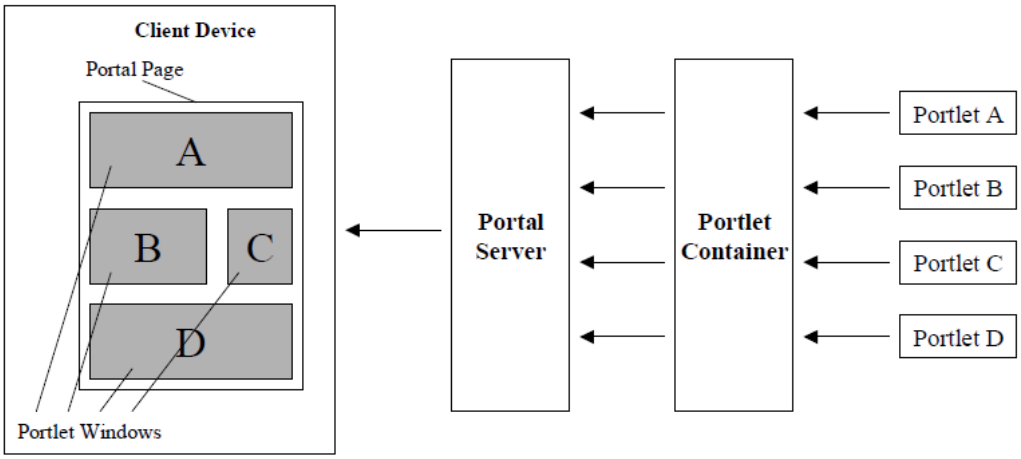
Portlet


APACHE PLUTO
About Apache Pluto
Test Page
JSR 286 Tests
Pluto Admin
Logout
About Apache Pluto
VIEW
About Pluto Portal Driver
Portal Name: pluto-portal-driver
Portal Version: 2.0.0
Servlet Container: Apache Tomcat/6.0.18
Java Version: 1.6.0_10-rc2 (Sun Microsystems Inc. - Java HotSpot(TM) Client VM build 11.0-b15)
Operating System: Windows Vista (x86 version 6.0)
Pluto Website: http://portals.apache.org/pluto/
Please use the Jira issue tracking site to record any problems you are having with the Pluto portal driver. When you report an issue, please include the data (version, os, etc.) collected in this portlet in addition to any relevant stack traces or log records and detailed steps on what you were doing when the problem arose.
Test Portlet 2 (dflt)
EDIT
THIS IS THE EDIT PAGE
Test Portlet #1
VIEW
This portlet is a portlet specification compatibility test portlet. It provides several tests of verying complexities which will assist in evaluating compliance with the portlet specification. It was originally developed for testing Apache Pluto, however, it does not utilize any proprietary APIs and should work with all compliant portlet containers.
Please select one of the following tests:
0. Render Parameter Test Test
1. Action Parameter Test Test
2. Dispatcher Render Parameter Test Test
3. Dispatcher Request Test Test
4. Simple Attribute Test Test
5. Application Scoped Session Attribute Test Test
6. External Application Scoped Attribute Test Test
7. Context Init Parameter Test Test
8. Preference In Action Test Test
9. Preference In Render Test Test
10. Session Timeout Test Test
11. Portlet Mode Test Test
12. Window State Test Test
13. Misc Test Test
14. Security Mapping Test Test
15. Resource Bundle Test Test

Portlet


<Title>
<Portlet content>
Decorations and controls
Portlet fragment
Portlet window
Portal page

Portlet

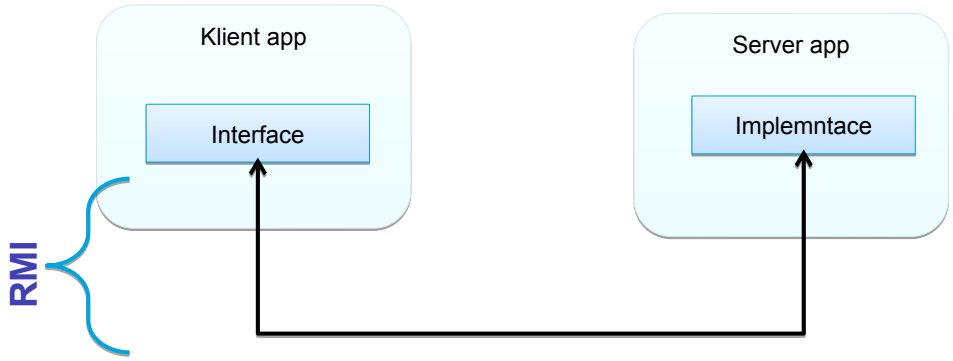

Client Device
Portal Page
A
B
C
D
Portlet Windows
Portal Server
Portlet Container
Portlet A
Portlet B
Portlet C
Portlet D

RMI – Remote Method Invocation

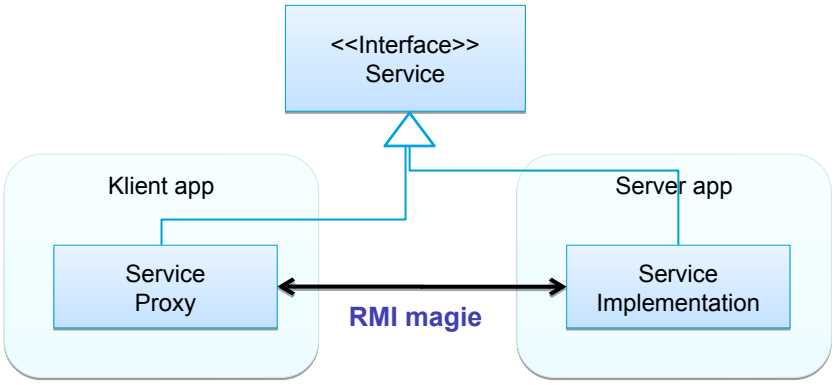
- Vzdálené volání metod (předchůdce webových služeb)
- Vychází z RPC (Remote Procedure Call)

- Primárním cílem RMI bylo vytvořit systém, který umožní vytvářet distribuované aplikace stejně snadno jako nedistribuované aplikace.

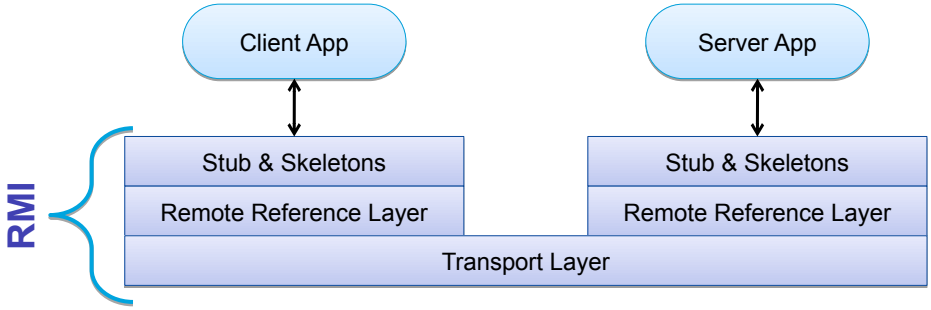
RMI - Architektura



RMI - Architektura



RMI – Architektura



RMI – Server

```
public interface Calculator extends java.rmi.Remote {
   public long add(long a, long b) throws
            java.rmi.RemoteException;
}
public class CalculatorImpl
      extends java.rmi.server.UnicastRemoteObject
      implements Calculator {
   public CalculatorImpl() throws
         java.rmi.RemoteException {
      super();
   }
   public long add(long a, long b) throws
            java.rmi.RemoteException {
      return a + b;
   }
}
```

RMI – Server

- Příkaz

```
<JDK_HOME>\bin\rmic.exe -keep CalculatorImpl
```

- Vytvoří třídu CalculatorImpl_Stub

```
public class CalculatorServer {
   public static void main(String args[]) {
      try {
         System.setSecurityManager(new

RMISecurityManager());
         Calculator c = new CalculatorImpl();
         Naming.rebind(
                  "rmi://localhost:1099/CalculatorService", c);
      } catch (Exception e) {
      }
   }
}
```

rmi://<host_name> [:<name_service_port>] /<service_name>

RMI - Klient

```
public class CalculatorClient {
   public static void main(String[] args) {
      try {
         System.setSecurityManager(new
                                 java.rmi.RMISecurityManager());
         Calculator c = (Calculator) Naming.lookup(

"rmi://localhost/CalculatorService");
         System.out.println(c.add(4, 5));
      } catch (Exception e) {
      }
   }
}
```

Na servrové straně nutno spustit:

`<JDK_HOME>\bin\rmiregistry.exe`

CORBA – Common Object Request Broker Architecture

IDL – Interface Definition Language

- Na platformě nezávislý jazyk pro popis rozhraní a datových typů

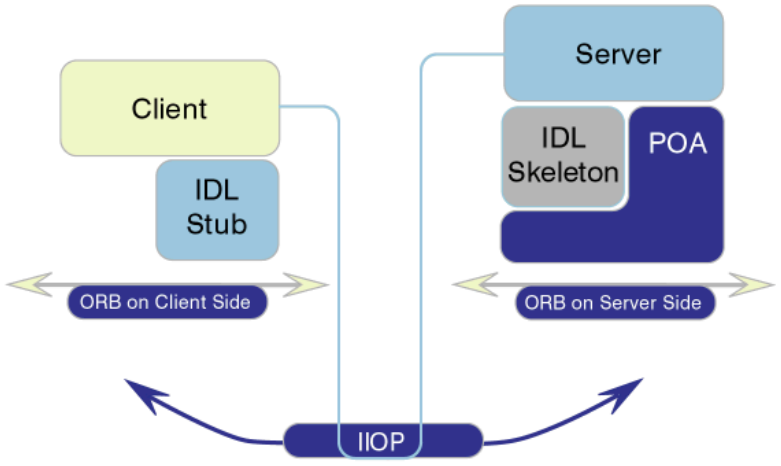
```
module StockObjects {
  struct Quote {
   string symbol;
   long at_time;
   double price;
   long volume;
 };
 exception Unknown{};
```

```
interface Stock {
Quote get_quote() raises(Unknown);
void set_quote(in Quote stock_quote);
// Provides the stock description,
readonly attribute string description;
  };
interface StockFactory {
     Stock create_stock(
      in string symbol,
      in string description
    );  };};
```

CORBA

- Distribuovaná objektová architektura
- Nezávislá na platformě a jazyce
- Pouze specifikace
- Neexistuje referenční implementace

CORBA – Architektura



CORBA 1.0 (October 1991)

Implementace objektu: Kód objektu, který implementuje zveřejněné služby objektu. Implementace může být napsána v libovolném podporovaném jazyce (obvykle C, C++ nebo Java). Rozhraní služeb objektu je definováno v jazyce IDL (*Interface Definition Language*).

Klient: Program využívající vzdálené objekty. Pro použití objektu musí mít dostupnou definici rozhraní objektu v jazyce IDL buď vv době překladu nebo při *dynamickém volání* za chodu programu a jednoznačnou adresu objektu (IOR).

IDL stubs (spojky): Kód vygenerovaný kompilátorem jazyka IDL, který propojuje uživatelský kód s agentem ORB. V jazyce C++ má spojka formu *zástupné třídy, jejíž metody může klientský kód přímo volat.*

DII (Dynamic Invocation Interface): Klient může používat také objekty, ke kterým získá definici rozhraní za běhu programu. Rozhraní pro *dynamické volání* metod dovoluje generovat dynamické požadavky.

ORB (Object Request Broker): *Zprostředkovatel objektových služeb zahrnuje veškeré vnitřní mechanismy pro vyhledání požadovaného objektu, generování a přenos požadavků, parametrů a výsledků na úrovni komunikace mezi systémy. ORB může používat různé met*ody komunikace, včetně přímé aktivace objektů v rámci jednoho adresového prostoru.

Přenosný Objektový adaptér (POA): Objektový adaptér propojuje implementaci objektu se agentem ORB, demultiplexuje přicházející požadavky, aktivuje objekty a předává jim požadavky prostřednictvím volání metod kostry objektu.

Kostra objektu: Je vygenerována kompilátorem jazyka IDL, slouží jako bázová třída odpovídající definici objektu v jazyce IDL.

DSI (Dynamic Skeleton Interface): Dynamicky vytvořená kostra objektu, obdoba DII na straně klienta. Typickým použitím je *most* pro transformaci požadavků z jednoho komunikačního protokolu do jiného nebo *firewall*.

GIOP (General Inter ORB Protocol): Protokol komunikace mezi různými ORB. Je definován nad běžným spojovaným transportním protokolem. Konkrétní implementace nad protokolem TCP/IP je definována jako IIOP (Internet Inter-ORB Protocol).