EVROPSKÁ UNIE
Evropské strukturální a investiční fondy
Operační program Výzkum, vývoj a vzdělávání

MINISTERSTVO ŠKOLSTVÍ, MLÁDEŽE A TĚLOVÝCHOVY

JAT – Java Technology

David Ježek
david.jezek@vsb.cz

Tel: **597 325 874**
Office: **EA406**

Something from history…

- 500 BC – Pythagoras founded a society of peoples that believe among other things, that beans contains souls of dead peoples, therefor eating of beans was forbidden.
- 850 AC – Arabian goatherd Khalid found, that his herd behaves in a strange way and discovered plant “Coffea arabica” and the effect of caffeine.
- End of the 17th century – cultivation of coffee beans on the island Java, name Java was connected with coffee in general
- 1995 – Sun’s programing language Oak has problem with the name, therefore was chosen name Java and language was connected with coffee.

500 BC – Pythagoras founded a society of peoples that believe among other things, that beans contains souls of dead peoples, therefor eating of beans was forbidden.
850 AC – Arabian goatherd Khalid found, that his herd behaves in a strange way and discovered plant “Coffea arabica” and the effect of caffeine.
End of the 17th century – cultivation of coffee beans on the island Java, name Java was connected with coffee in general
1995 – Sun’s programing language Oak has problem with the name, therefore was chosen name Java and language was connected with coffee.

Java Platforms

- **Standard Edition (Java SE)**
  - Java Applet, Java WebStart
- **Java FX**
  - JavaFX 2.1 - desktop, browser and mobile phones.
  - Planed: TV set-top boxes, gaming consoles, Blu-ray players and other
- **Enterprise Edition (Java EE)**
  - Servlets, JavaServerFaces, JavaServerPages
- **Micro Edition (Java ME)**
  - Mobile phone, Java TV, Java Card, other devices

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.

Java SE Platform

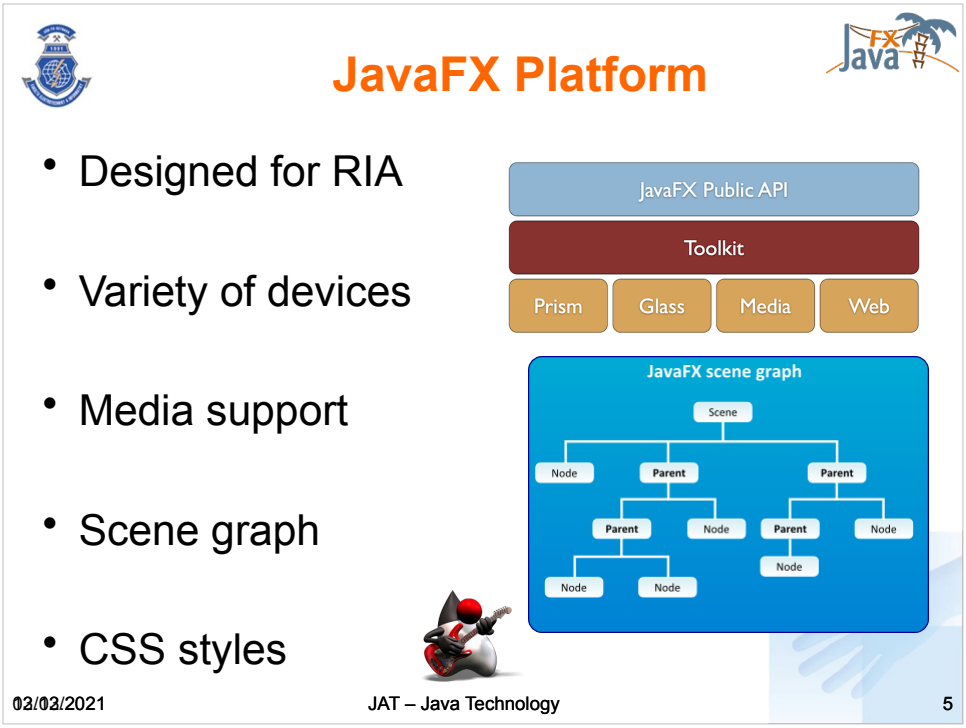
JDK							
		JRE					
Java Language	Tools and Utilities	Runtime	Java SE API				Java VM
			Base Library	Other Base Packages	Integration Libraries	User Intrface Libraries	
javac	javadoc JAR javah javap JPDA JConsole VisualVM java DB Security Internationalization RMI IDL Deployment Monitoring Troubleshooting Scripting	Java Java Web Start Applet/Plug-in	Lang and Util Collections Concurrency Utilities JAR Logging Management Preferences API Reference Objects Reflection Regular Expressions Versioning ZIP Instrumentation	Beans I18N Support I/O JMX Math Networking Override Mechanism Security Object Serialization Extension Mechanism XML	IDL JDBC JNDI RMI RMI-IIOP Scripting JNI	AWT Swing Java 2D Accessibility Drag and Drop Input Methods Image I/O Print Service Sound	HotSpot
	JVM TI						

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:

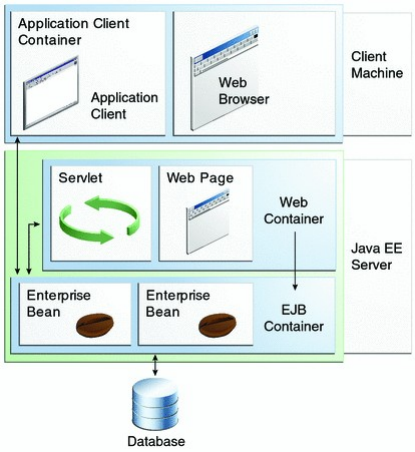
- Java SE API (Application Programming Interface) contains libraries of classes that provide various functionalities. From basic types and object to high level classes for networking, security, database access, GUI, XML and others.
- Virtual Machine that provide unified virtual environment to run Java 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



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.

Java EE Platform

- Multi-tier architecture
- Java EE API
  - Servlets
  - Java Server Faces
  - Java Server Pages
  - Persistence API
  - …
- Runtime environment



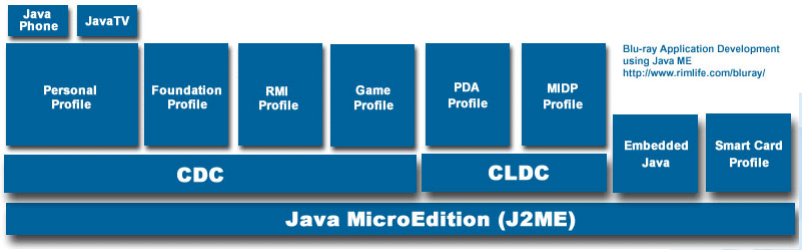
The Java EE platform is built on top of the Java SE platform. The Java EE platform provides:

- 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

- Designed to run java application on devices with limited resources
- Java ME API is subset of Java SE API with some extra special class libraries
- Define profiles and configuration for different types of devices



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

3. Component Technology – Motivation

- **Development**
  - Reusability
  - Easy testing
  - The possibility of specialization of producers
- **Distribution**
  - Short time to market
  - Vendor independence
- **Service**
  - Reduced maintenance costs
  - Interchangeability – customer pressure to standardization

Now we know something about Java platforms. Java is object oriented language and lot of Java principles are based on software components. But what is and why use software components.

Lets start with why use software components:

In phase of software development we gain:

Reusability – It help us use one component many times in different software application that will improve quality and decrease price of development.

Easy testing – It much easiest test several small component then test one huge software.

The possibility of specialization of producers – A producer of components can specialize on one kind of component to improve quality, reliability, performance and reduce cost of development.

In phase of distribution of software we gain:

Short time to market – If we reuse completed component instead of write whole software from scratch, we will reduce effort of software development significantly.

Vendor independence – If we use components with standardized interface, we could change vendor of individual components without rewriting whole application.

I phase of service distributed software we gain:

Reduced maintenance costs – If we use components, we could easily remove found errors or change behavior of system because we change or modify only small component instead of huge software.

3.1. Application of component in IT

- **Hardware**
  - Memory, processors, motherboards
  - Peripherals – PnP, drivers
  - Communication elements
- **Software**
  - Graphical User Interface – Swing
  - Distributed application –CORBA, EJB, .NET, COM, DCOM, …
  - Databases
  - Information systems

Using components is nothing new for many industries such as construction, automotive and more. In the field of information technology components have long been used in the fields:

Hardware

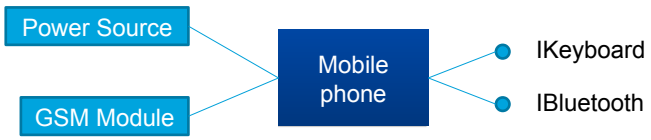
- Memory, processors, motherboards all these part of computer has standardized hardware interface (slots, ports) and communication protocols. So you can buy different part from different producer and assembly everything together. Many of producer are specialized only of one or few type of parts
- Some of computer peripherals has plug and play interface, that means you can connect different peripherals with standardized interface to any computer. And again drivers for different peripherals communicate with operating system throw standardized interface so you can install driver for different peripherals form different producer to your operating system.
- Communication elements – same as others

Software

- Graphical User Interface – Nowadays, almost all technology for creating GUI are based on stacking individual elements (components) such as text field, button, label and others, together. In Java exist component technology for GUI called Swing or older AWT.
- Distributed application – Many distributed application work with technologies for defining and creating components that can run on different computers sometime different platforms. Example of

3.2. What is a component? 1/3

- Building unit with contractually defined:
  - interfaces
  - explicit contextual links

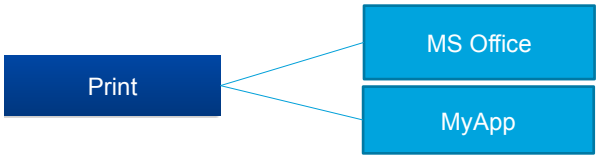


Building unit with contractually defined:
interfaces
explicit contextual links

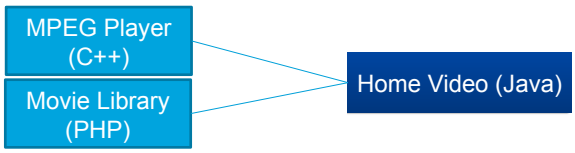
For example mobile phones has interface Keybord and interface Bluetooth and of course all mobile phones has link to power source and some GPS module.

What is a component? 2/3

- It can be used independently of
  - the environment for which it was created.



  - the environment in which it was created.



It can be used independently of the environment for which it was created.
For example component for printing can be original created for MS Office application, but it has defined and standardized well know interface so any other application can use that component to print data to printer.

It can be used independently of the environment in which it was created.
For example component home video player written in java can use different component written in different languages to decode video streams.

What is a component? 3/3

It is designed to integrate third party.

- **Author of component**
  - He does not know who and for what will others use the components.
  - He must meet specified interface.
- **Author of application**
  - He does not know who will supply components.
  - Communicates with component through the specified interface.
- **Integrator**
  - Link the application with the appropriate components.

It is designed to integrate by third party. Because o that we distinguish three roles in process of component development and deployment.

Roles:

Author of component

He does not know who and for what will others use the components.

He must meet specified interface

Author of application

He does not know who will supply components.

Communicates with component through the specified interface.

Integrator

Link the application with the appropriate

Requirements for components

- Complete documentation
- Thorough testing
- Robust control of validity of inputs
- Return of sufficiently informative error messages
- Assumed that the component will be used for unexpected purposes.

Using components brings many advantages, but on the other hand, it is necessary to keep certain principles:

- Complete documentation - Without full documentation cannot others effectively use component.

Thorough testing - Because the component is used for many different purposes in many applications, it is necessary to ensure its maximal reliability.

- Robust control of validity of inputs - Component is used by other programmers, and they may not know all restrictions on components, so we need all the input values to be checked and verified their accuracy before use.
- Return of sufficiently informative error messages - If the inputs of the component are incorrect , or any other error arise within the component , returned error message must contains sufficient information about where the error occurred and what is the cause.
- Assumed that the component will be used for unexpected purposes.

Component specification

- State
  - **Properties** – read, write
- Behavior
  - **Methods** – invocation, parameters, results
- Interaction with the environment
  - **Events** – registration, notification

Component is specified by its state, behavior and Interaction with the environment.

State is determined by the values of individual properties, which can be read (get value) and some of them set (changing value).

Behavior is determined by individual methods and their parameters and return values. Each method determines the behavior by its source code that changes the state of the component itself or a interacts with the surrounding environment.

Interaction with the environment is often realized through a mechanism of events that

The life cycle of component 1/2

- **Creation of component**
  - Standards – defined by used technology CORBA, COM+, DCOM, EJB, .NET, JavaBeans
  - Binary compatibility - independent of language
- **Publication of interface**
  - Documentation - for humans
  - Introspection - part of the components from which the client application can read the component's metadata

Creation of component

Standards – defined by used technology
CORBA, COM+, DCOM, EJB, .NET

Binary compatibility - independent of language
(valid for some technologies)

Publication of interface

Documentation - for humans

Introspection - part of the components from which the client application can read the component's metadata

The life cycle of component 2/2

- **Distribution of component**
  - Publication of the component or registration of the component in one of the directory services
  - LDAP(standard from IETF), JNDI (Java), UDDI (XML web services)
  - Library distribution
- **Search for components**
  - Identification of components, realization of late binding
- **Making an application**
  - IDE support –access to the component such as to internal objects

Distribution of component

Publication of the component or registration of the component in one of the directory services

LDAP(standard from IETF), JNDI (Java), UDDI (XML web services)

Library distribution

Search for components

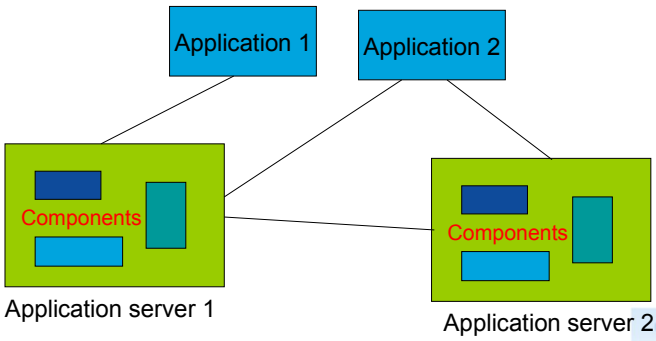
Identification of components, realization of late binding

Making an application

IDE support –access to the component such as to internal objects

Architecture of component-oriented systems

Architecture of system with distributed components – component can run on different servers.



Architecture of system with distributed components – component can run on different servers.

Application server

- Environment for running applications and software components
  - Distributed environment
  - Resources
  - Security
  - Transaction
- Examples of Java application servers
  - JBoss, Jakarta Tomcat, BEA Weblogic, Citrix Meta Frame, IBM WebSphere, Oracle AS, Glassfish, Sun Java System AS, …

Environment for running applications and software components. Often provide the following services for application and components:

Distributed environment - Several servers are combined into one virtual unit.

Resources - Provides searching and management of resources such as database connection, other components, files, memory, ….

Security – Provides security mechanisms.

Transaction

Examples of Java application servers

JBoss, Jakarta Tomcat, BEA Weblogic, Citrix Meta Frame, IBM WebSphere, Oracle AS, Glassfish, Sun Java System AS, …

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

Major component technologies

- JavaBeans, EJB
  - Only Java language
  - Enterprise Java Beans – for large systems, distributed
- COM, COM+, DCOM, ActiveX
  - Binary compatible
  - The basic technology for Windows
- .NET
  - Compatibility at the language level – C++, C#, Jscript, VB.NET
- CORBA
  - Binary Compatible, Platform compatible
  - Many languages

JavaBeans, EJB – JavaBeans technology for all application is used very often in Java application. EJB technology for distributing Java application, mainly used for web application.
Only Java language
Enterprise Java Beans – for large systems, distributed

COM, COM+, DCOM, ActiveX – COM, COM+, DCOM was replaced by ActiveX technology.
Binary compatible
The basic technology for Windows

.NET
Compatibility at the language level – C++, C#, Jscript, VB.NET

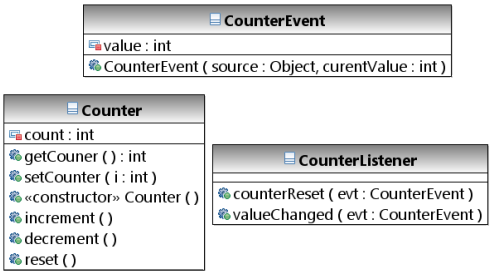
CORBA – Technology defined independently to language, hardware or software system. Component on different system written in different languages can communicate each other. Support for object oriented programing.
Binary Compatible, Platform compatible
Many languages

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

2. JavaBeans

JavaBean component is a Java class meet a specified condition.

- Sources
  - http://java.sun.com/docs/books/tutorial/javabeans/TOC.htm
  - http://java.sun.com/javase/technologies/desktop/javabeans/docs/spec.html
  - http://www.cs.vsb.cz/behalek/vyuka/pte/prednasky/index.php



In fact there is only one necessary condition. That the class could be considered a component must have a public constructor without parameters.

Note that due to the characteristics of Java, a class with no constructor in source code is a JavaBean component, because if the class has no constructor in source code, the Java compiler creates a public constructor with no parameters automatically.

JavaBeans - fundamentals

"A Java Bean is a reusable software component that can be manipulated visually in a builder tool."

- Component granularity
  - Small building blocks
  - „whole application"
- Portability
- A uniform, high-quality API
- Simplicity
- Beans v. Class Libraries
- Design time vs. run-time
- Security Issues
- Internationalization
- Component persistency
  - Serialization or Externalization
- Local activation
- Multi-Threading

"A Java Bean is a reusable software component that can be manipulated visually in a builder tool."

Component granularity
Small building blocks
„whole application"
Portability
A uniform, high-quality API
Simplicity
Beans v. Class Libraries
Design time vs. run-time
Security
Internationalization
Component persistency
Serialization or Externalization
Local activation
Multi-Threading

Cited from: http://java.sun.com/javase/technologies/desktop/javabeans/docs/spec.html :

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.

A uniform, high-quality API
Different platforms will vary in their ability to support the full JavaBean APIs. However whenever
a platform is unable to provide the full functionality it must provide some reasonable,
harmless default instead.
For example, if a platform doesn't support menubar-merging then when a Bean provides a
menubar it may pop up above the component, rather than being merged into the containing document's
menubar.
This means that JavaBeans component writers can program to a consistent set of APIs and trust
them to work everywhere. We don't want bean implementors to have to do checks to discover
which APIs are supported on their current platform.

Simplicity
We would like to keep the JavaBeans APIs relatively simple. We will focus on making small
lightweight components easy to implement and use, while making heavyweight components
possible.
As a general rule we don't want to invent an enormous java.beans.Everything class that people
have to inherit from. Instead we'd like the JavaBeans runtimes to provide default behaviour for
all objects, but to have a mechanism to override a given piece of default behaviour by inheriting
from some specific java.beans.something interface.
One of our goals is that people should be able to learn the basic JavaBeans concepts very quickly
so that they can start writing and using simple components with very little effort and then
slowly progress to using the more sophisticated features of the API.

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.

Internationalization
As part of the JDK 1.1, various internationalization APIs are being added to the core Java API
set. These internalization APIs can be used by individual Java Beans

What should be saved
Beans should be 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)
and any cached state that can be regenerated on demand.
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.
Similarly a bean should not store away pointers to transient objects such as beans or
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".

Local activation
The standard security model for java Bean components is that they run within the same address
space as their container.
So for example, if the container is a java application, then the contained bean is run in the same
java virtual machine as its container. If the container is a non-java application, then the java Bean will run in a java virtual machine that is directly associated with the application. (Normally
this virtual machine will be running in the same address space as the application.)

Multi-Threading
Java Beans should assume that they are running in a multi-threaded environment and that several
threads may be simultaneously delivering events and/or calling methods and/or
setting properties.
It is the responsibility of each java bean developer to make sure that their bean behaves properly
under multi-threaded access. for simple beans this can generally be handled by simply
making all the methods "synchronized".

Java Beans – Component types

- **Visual components**
  - They have visual representation and occupying space in the window. Therefore, they must be subclass of class java.awt.Component
  - Examples: button, table, scrolling lists, HTML viewer
  - Support in visual tools
- **Invisible components**
  - Example: timer, spell checker, …
  - Support in visual tools

Visual components
They have visual representation and occupying space in the window. Therefore, they must be subclass of class java.awt.Component
Examples: button, table, scrolling lists, HTML viewer
Support in visual tools
Invisible components – Any component without a visual representation that provides primarily functionality.
Example: timer, spell checker, …
Support in visual tools - Component can be inserted and connected to other component (visual/invisible).

Cited from http://java.sun.com/javase/technologies/desktop/javabeans/docs/spec.html :

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.

Java Beans – Component Structure

- **Properties**
  - Value of properties can be read or written through access method (get/set methods). Could not be accessed directly.
- **Methods**
  - Operation over components
- **Events**
  - Communication link between components

The structure of JavaBean component corresponds to a general structure of the components defined in the previous slides.

Properties

Value of properties can be read or written through access method (get/set methods). Could not be accessed directly.

Methods

Operation over components

Events

Communication link between components

Implementation of the individual structures of the components are described in the following slides.

Java Beans – Events (1)

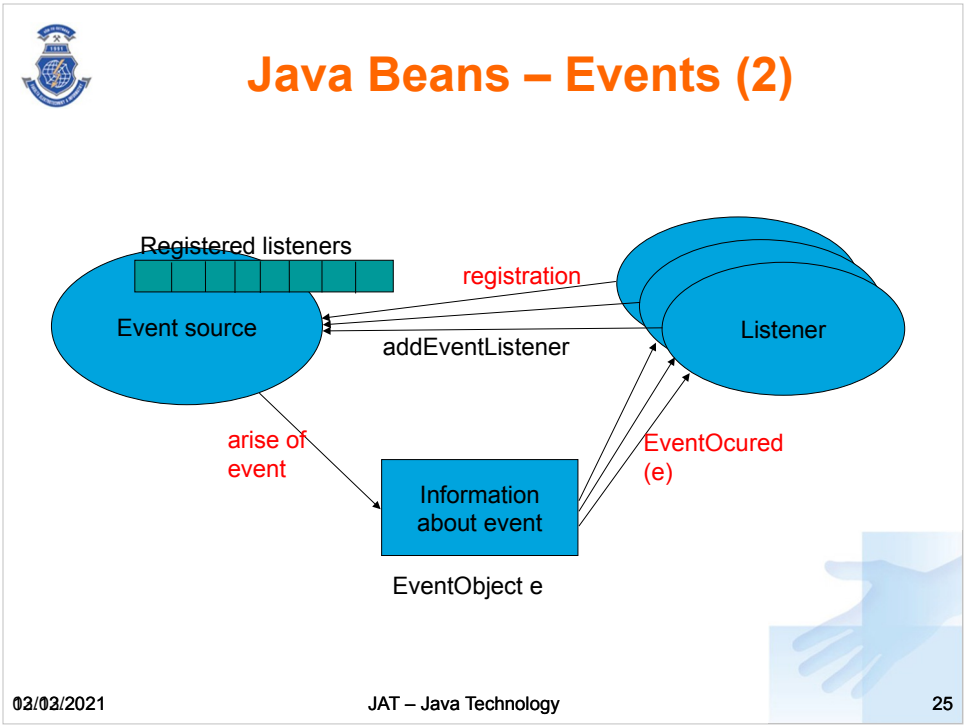
- **Source of events**
  - An object that generates events
  - Manages the list of registered listeners
- **Listener**
  - An object that wants to be informed about the event
  - Must be registered at the event source
  - Must implement the agreed interface

The mechanism of events in the Java technology is not used only for the JavaBean, but as a general mechanism. This method is not exceptional, is somehow implemented in all modern programming languages, and in principle It is based on the Observer design pattern.

Description of the source mechanism speaks about events and listeners, in the context of Java Beans technology, the source of events will be some component and the listener will be some component or another object.

Source of events

An object that generates events and manages



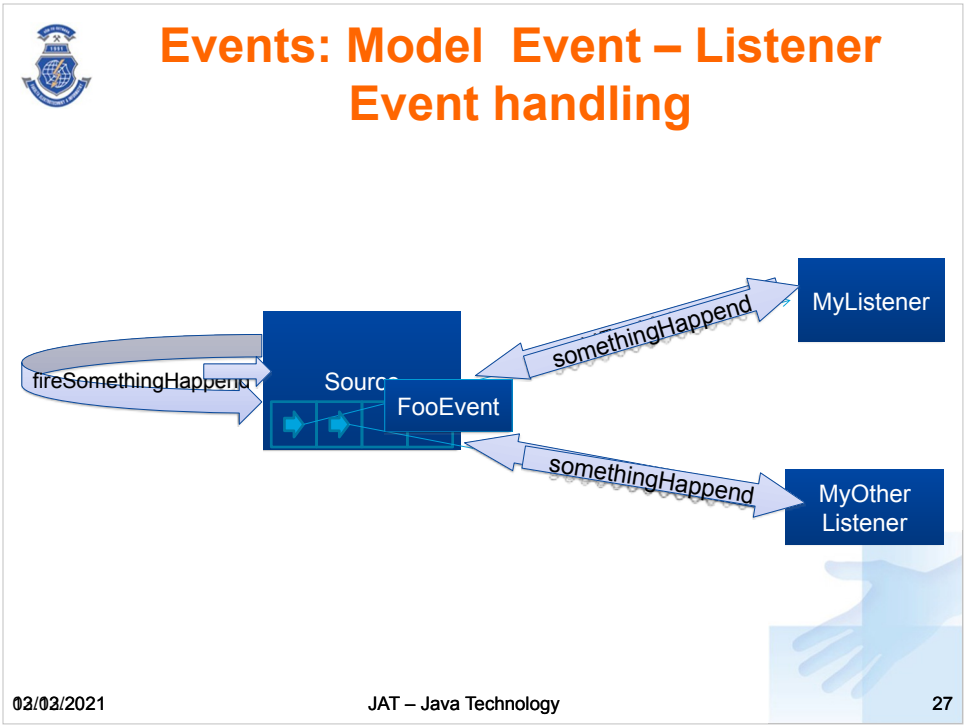
The whole method consists in that that we have a event source that contains an array or collection of listeners, which is initially empty. Individual listeners will then register at the event source using method addEventListener, it means that the event source stores a pointer to each listener into a prepared array.

If an event occurs, the event source creates new object which representing the event and distribute it to all registered listeners. Which means that the event source passes through the array of listeners and to all stored pointers of listeners calls the method that represents the event.

Java Beans – Events Handling

- The listener register to the event source (eg buttons, that we waiting to be pressed)
- The user press a button - an event occurred
- Event Source (button) pass through the list of registered listeners, and notify each one about occurred event:
  - Call the agreed method of listener interface
  - Pass event information to method as parameter (object of an subclass of java.util.EventObject)

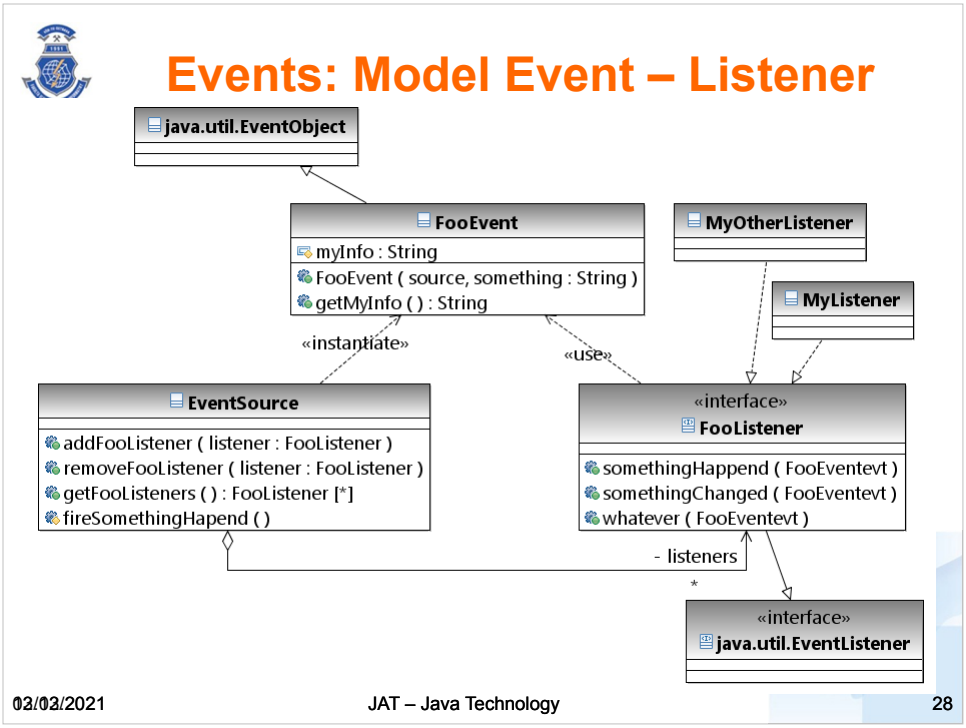
The listener register to the event source (eg buttons, that we waiting to be pressed)
The user press a button - an event occurred
Event Source (button) pass through the list of registered listeners, and notify each one about occurred event:
Call the agreed method of listener interface
Pass event information to method as parameter (object of an subclass of java.util.EventObject)



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.

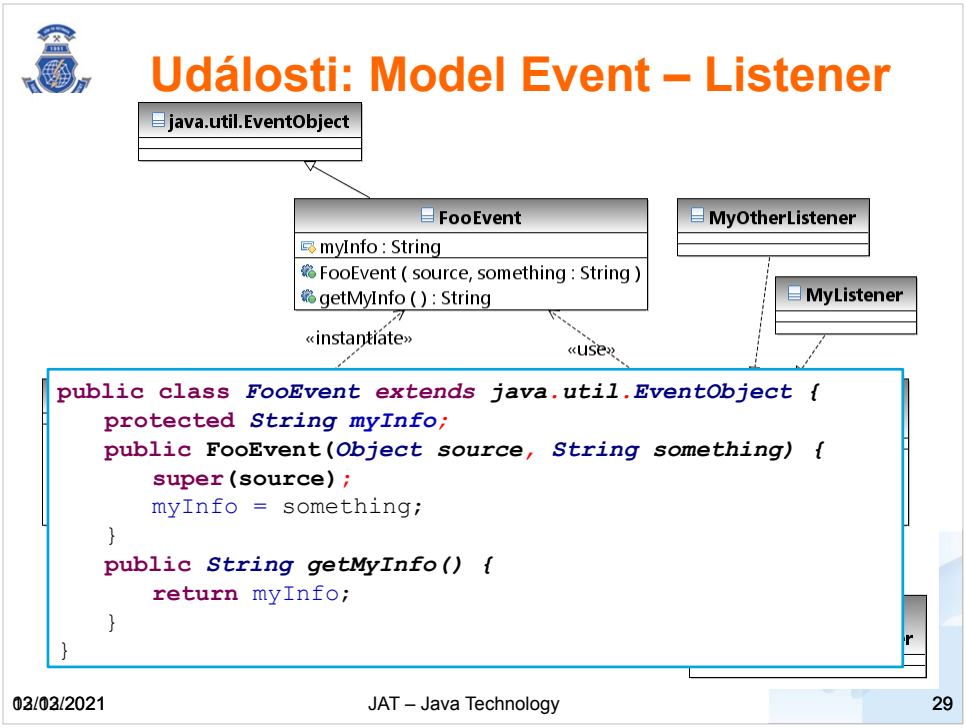
If an event occurs, the object of class Source calls himself method fireSomethingHappend. This method creates a object of class FooEvent that sends as a parameter of method somethingHappend to all listeners whose pointers are stored in the list of



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.

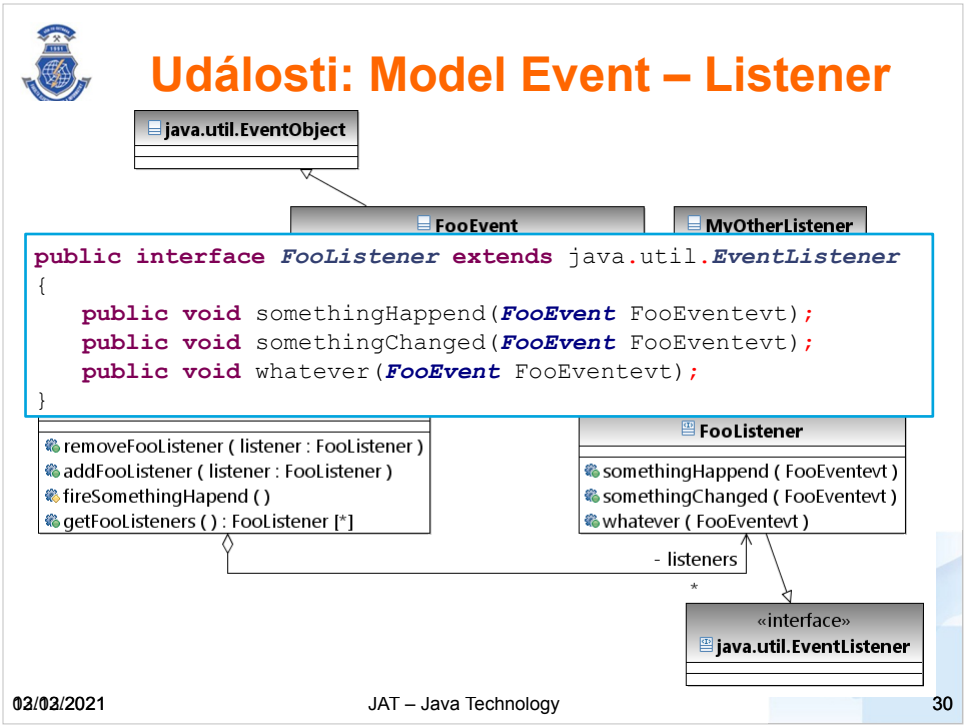
Furthermore, there must be FooListener interface, whose methods determine what events listeners will receive. These methods must have a first parameter of class FooEvent. This interface must derive from interface java.util.EventListener. Specific listeners (MyListener, MyOtherListener) must then implement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



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.

Události: Model Event – Listener

```
import java.util.ArrayList;
private ArrayList<FooListener> listeners = new
ArrayList<FooListener>();
public class EventSource {
    public FooListener[] getFooListeners() {
        return listeners.toArray(new FooListener[listeners.size()]);
    }
    public synchronized void removeFooListener(FooListener listener)
{
        listeners.remove(listener);
    }
    public synchronized void addFooListener(FooListener listener){
        listeners.add(listener);
    }
    protected synchronized void fireSomethingHapend() {
        FooEvent e = new FooEvent(this, "something");
        for (FooListener l : listeners) {
            l.somethingHapend(e);
        }
    }
}
```


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.

JavaBeans – Events - Summary

- Class, that represent event
  - It have to be subclass of `java.util.EventObject`
  - The name must have the form `<something>Event`
  - It have to have constructor with at least one parameter (source of event)
- Interface for listeners (listener type)
  - It have to implements interface `java.util.EventListener`
  - The name must have the form `<something>Listener`
  - Each method should have one parameter of type `<something>Event`
  - The methods should return nothing (`void`)
- Class that represent source of events (JavaBean) must have methods:

```
public void add<ListenerType>(<ListenerType> listener)
  throws java.util.TooManyListenersException;
public void remove<ListenerType>(<ListenerType> listener)
public <ListenerType>[] get<ListenerType>s();
```


Class, that represent event

It have to be subclass of

`java.util.EventObject`

The name must have the form

`<something>Event`

It have to have constructor with at least one parameter (source of event)

Interface for listeners (listener type)

It have to implements interface

`java.util.EventListener`

The name must have the form

`<something>Listener`

Each method should have one

JavaBeans – Adapter(1)

- **EventListener** interface for a particular component can contain many methods.
- If we want to respond only to certain events:
  - We have to implement either a blank response to all other events
  - Or we use the adapter as the base class and implement only the chosen method
- The adapter implements the default response to all events

EventListener interface for a particular component can contain many methods.

If we want to respond only to certain events:

We have to implement either a blank response to all other events

Or we use the adapter as the base class and implement only the chosen method

The adapter implements the default response to all events

Because the interface EventListener of one component

JavaBeans – Adapter(2)

```
public interface CounterListener {
   public void counterReset(CounterEvent evt);
   public void valueChanged(CounterEvent evt);
}


public class CounterAdapter implements CounterListener{
   public void counterReset(CounterEvent evt) {
   }
   public void valueChanged(CounterEvent evt) {
   }
}
```


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.

Java Beans – Adapter (3)

```
Counter counter = new Counter();
//use of anonymous inner classes
counter.AddCounterListener(new CounterAdapter() {
   @Override
   public void counterReset(CounterEvent e) {
      System.out.println("Reset");
   }
});


Counter counter = new Counter();
//use of anonymous inner classes
counter.AddCounterListener(new CounterListener() {
   public void valueChanged(CounterEvent evt) {
   }
   public void counterReset(CounterEvent e) {
      System.out.println("Reset");
   }
});
```


If you use the adapter CounterAdatpet instead of interface CounterListener the source code is a bit shorter and at first glance it is clear that there are only processes one type of events. If used directly interfaces these benefits are lost.

Importance of the adapter increases with the number of methods in interfaces.

JavaBeans Properties

- Property with name **counter**

```
public int getCounter()
```

```
private int counter;
```

```
public void setCounter(int
i)
```

```
public class TickCounter {
    private int counter;
    public int getCounter() {
        return this.counter;
    }
    public void setCounter(int i)
{
        this.counter = i;
    }
```


Features of JavaBean components are not defined by any special language structures or keyword, but are defined by the naming convention. The property of component is made up of methods that comply with certain principles.

There must be a method with name starting with prefix “get” and continuing with name of the property, in our case name of property is “counter”. The method must return a value whose type matches the type of property. If property can be changed, there must be method with name starting with prefix “set” and continuing with name of the property. The method must have exactly one

JavaBeans – Properties

- **Simple properties**
  - Getter method

```
public <PropertyType> get<PropertyName>()
public boolean is<PropertyName>()
```

  - Setter method

```
public void set<PropertyName>(<PropertyType> value)
```


These methods with prefix “get” and “set” are called getter and setter.

Naming convention is then straightforward. Method name corresponds with the prefix connected with name of property. The return type of getter methods correspond with type of property and type of single parameter of setter methods correspond with type of property too.

There is one exception for the properties of type “boolean” can be a getter method prefix "is“ instead of “get”. The name of such method is more clearly for humans.

JavaBeans – Properties

- **Indexed properties**
  - indexed setter

```
void set<PropertyName>(int index, <PropertyType> value)
```

  - indexed getter

```
public <PropertyType> get<PropertyName> (int index)
```

  - array setter

```
public void set<PropertyName> (PropertyType values[])
```

  - array getter

```
public <PropertyType>[] get<PropertyName> ()
```


For JavaBean components, there exist indexed properties, that represent an array of values as opposed to ordinary properties that represent only one value.

For these properties exists getter and setter methods that return the entire array of values (as for ordinary properties), but also are added indexed getter and setter methods used to access individual elements of this property.

JavaBeans – Properties

```
protected ArrayList<JButton> buttons;
void setButtons(int index, JButton b){
 buttons.set(index, b);
}
public JButton getButtons(int index){
 return buttons.get(index);
}
public void setButtons(JButton values[]){
 buttons.clear();
 for(JButton b : values){
 buttons.add(b);
 }
}
public JButton[] getButtons(){
 return buttons.toArray(new JButton[buttons.size()]);
}
```


This example of source code shows indexed property “buttons”. Values of this property are stored in collection buttons (type of collection is ArrayList<JButton>). So indexed getter and setter method access directly to individual elements stored in collection. But getter and setter method have to work with array of buttons, so there must be a conversion between array and collection.

JavaBeans – Special properties

- Bound properties
  - Generate event PropertyChange from interface java.beans.PropertyChangeListener, if their value is changed.
- Constrained properties
  - Generate event VetoableChange from interface java.beans.VetoableChangeListener, if their value is changing.
  - Change can be rejected.

Bound properties
Generate event PropertyChange from interface java.beans.PropertyChangeListener, if their value is changed.

Constrained properties
Generate event VetoableChange from interface java.beans.VetoableChangeListener, if their value is changing.
Change can be rejected.

If property is bound the component guarantee generation of PropertyChange event every

JavaBeans – Properties

Bound properties

```
public void addPropertyChangeListener(PropertyChangeListener x);
public void removePropertyChangeListener(PropertyChangeListener x);

public interface PropertyChangeListener extends java.util.EventListener {
    void propertyChange(PropertyChangeEvent evt);
}

public class PropertyChangeEvent extends java.util.EventObject {
    public PropertyChangeEvent(Object source, String propertyName,
     Object oldValue, Object newValue);

public String getPropertyName();
public Object getNewValue();
public Object getOldValue();
}
```


If some components wants to have a bounded properties, it must meet the following rules:

There have to be methods addPropertyChangeListener and removePropertyChangeListener, that register listeners of type PropertyChangeListener. The interface java.beans.PropertyChangelistener define only type of events, the propertyChange event. That event pass event object of type java.beans.PropertyChangeEvent, which handle information about arose event. Listener object could read information as name of changed property, old value of property and new value of property.

Cited from:

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*

JavaBeans Properties

Sun Microsystems 42 10/8/97

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.

JavaBeans – Properties

Bound properties

```
public void addPropertyChangeListener(String propertyName,
            PropertyChangeListener listener);
public void removePropertyChangeListener(String propertyName,
            PropertyChangeListener listener);

public void add<PropertyName>Listener(
            PropertyChangeListener p);
public void remove<PropertyName>Listener(
            PropertyChangeListener p);
```


In the previous slides, the listeners registered to listen all changes of all bond properties of the component. But component can provide registration to the selected pound properties only. In JavaBeans specification exist two way to provide such as functionality. One of way accept name of selected bound property as a additional parameter. Second way provide special methods add/remove<PropertyName>Listener that register listener only for bound property that name is contained in name of method.

Cited from:

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*

JavaBeans Properties

Sun Microsystems 42 10/8/97

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.

JavaBeans – Properties

Constrained properties

```
<PropertyType> get<PropertyName>();
void set<PropertyName>(<PropertyType> value)
        throws PropertyVetoException;

public void addVetoableChangeListener(
        VetoableChangeListener x);
public void removeVetoableChangeListener(
        VetoableChangeListener x);

public interface VetoableChangeListener extends
                EventListener {
    public void vetoableChange(PropertyChangeEvent evt)
            throws PropertyVetoException;
}
```


Another extension of properties is possibility to create constrained property. Component with constrained property allow register listener that can reject change of property. Such as component have to have method called addVetoableChangeListener and removeVetoableChangeListener for registering listeners that implements interface java.beans.VetoableChangeListener and for each constrained property is setter method defined with “throws PropertyVetoException”, that means, that during changing value of this property can arise exception PropertyVetoException.

If someone call setter method of the constrained property, component have to first arise event vetoableChange to all registered listeners, if any one of that listeners arise exception PropertyVetoException setter method is canceled and change of property is rejected.

JavaBeans – Properties

Constrained properties

```
void addVetoableChangeListener(String propertyName,
      VetoableChangeListener listener);
void removeVetoableChangeListener(String propertyName,
      VetoableChangeListener listener);

void add<PropertyName>Listener(
         VetoableChangeListener p);
void remove<PropertyName>Listener(
         VetoableChangeListener p);
```


Previous slide describe registration of VetoableChangeListener for all constrained property, but there exist two ways for registration VetoableChangeListener only for selected constrained property. That mechanism is same as in case of bound properties.

JavaBeans – Properties

```
public void setPriceInCents(int newPriceInCents){

int oldPriceInCents = ourPriceInCents;
//First tell the vetoers about the change. If anyone
//objects, we let the PropertyVetoException propagate
//back to our caller.
vetos.fireVetoableChange("priceInCents", new
Integer(oldPriceInCents), new Integer(newPriceInCents));
//No one vetoed, so go ahead and make the change.
ourPriceInCents = newPriceInCents;
changes.firePropertyChange("priceInCents", new
Integer(oldPriceInCents), new Integer(newPriceInCents));

}
```


This source code example shows implementation for setter of constrained property “priceInCents”. For notifying of veto listeners is used object “vetos” that is instance of class VetoableChangeSupport. Similar object of type PropertyChangeSupport is uses for notifying ProprtyChangeListeners. These object ensure sending of event notification only to listener that are registered to current property.

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

JavaBeans – Using the properties of component

- Attributes of objects in scripting languages
  - JSP, JSF
- Programmatic access through public access methods
- Access via forms (property sheets) in design tools
- Reading and writing to persistent memory

Properties of JavaBeans component are used in:

- Many scripting languages use properties without “get” and “set” prefixes as properties of object.
- Programmatically can by properties accessed through public getter and setter methods.
- Java design tools visualize set of properties of selected object in GUI, frequently called property sheet.
- Also persistent frameworks often work only with JavaBean properties, not with class fields.

JavaBeans – Methods

- All public methods that are not getter or setter method of any property or add/remove method for registering listeners, are considered as component methods.

```
public void clear() {
   val=0;
}
public void increment() {
   val++;
}
```


All public methods that are not getter or setter method of any property or add/remove method for registering listeners, are considered as component methods.

Java GUI Toolkit – Introduction 1

- 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

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

Java GUI Toolkit – Introduction 2

- 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 contents visually impaired people).

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

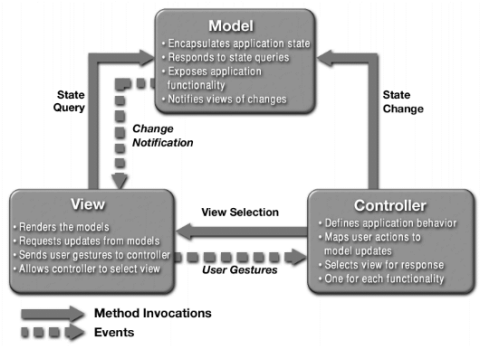
Java GUI Toolkit – Introduction 3

- 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.

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.

Design Pattern Model-View-Controller

- Exist many variants of MVC pattern but in Java SE is used following one:



- Properties:
  - It is possible to have
  - many views to one
  - model.
  - Reusability of model.

- http://java.sun.com/blueprints/patterns/MVC-detailed.html

Beware, do not confuse with the MVC architecture.

MVC design patter consist of three elements. The Mode, the View and the Controller.

Model is responsible for data maintenance. Data are encapsulated inside model component (class) and represent state of the application. Model also provide interface for data querying and manipulation. If some data are changed model is responsible for notification of all registered views.

View is responsible for data presentation in user interface. Can ask model about actual values of model data. Receive notification about changed data from model. After that

Java GUI

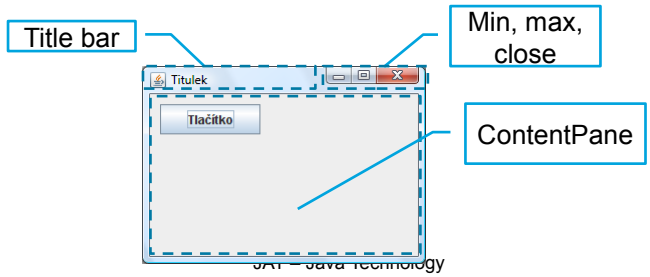
- Fundamental components
  - java.awt.Component, javax.swing.JComponent
    - addMouseListener
    - addKeyListener
    - get/set[Preferred/Minimum/Maximum]Size
- Panels jawa.awt.Panel, javax.swing.JPanel
  - setLayout(LayoutManager l)
  - add(Component c, Object constraints)

All visible elements of java user interface have to be a subclasses of java.awt.JComponent even base component class for swing javax.swing.JComponent is subclass of java.awt.JComponent. In fact it is subclass of abstract class java.awt.Container that is subclass of java.awt.JComponent. The base class java.awt.JComponent provide method for registering mouse listeners and key listeners, setters and getters for properties like maximum size, minimum size, preferred size and many others properties and methods common for all elements of user interface.

Panels are elements, that in general are not

The Most Frequently Used Components of Swing

- Root component for application UI: JFrame (JDialog and JApplet for application embedded in web page)
  - Application window contain standard elements, that can be hidden.
  - Communicate with operating system.
  - Contains container called ContentPane for other elements of user interface (for example JButton, JLabel, JPanel).
  - Methods for make window visible
    - pack(), setVisible(Boolean b)



Root component for application UI: JFrame (JDialog and JApplet for application embedded in web page)

Application window contain standard elements, that can be hidden.

Communicate with operating system.

Contains container called ContentPane for other elements of user interface (for example JButton, JLabel, JPanel).

Methods for make window visible

pack(), setVisible(Boolean b)

Cited from:

2.17.9 Virtual Machine Exit

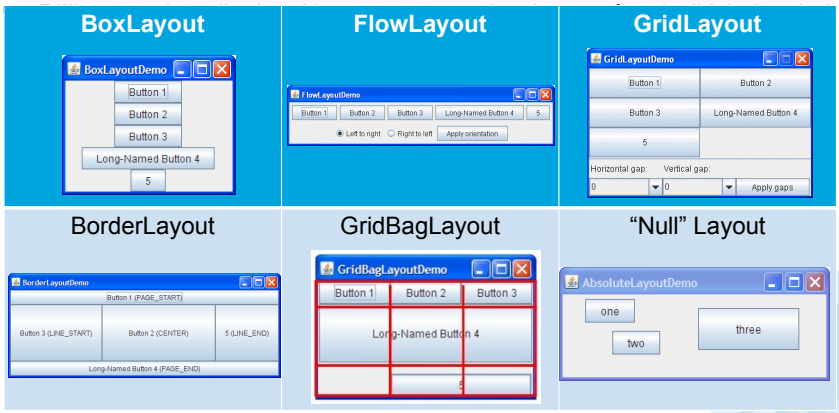
The Java virtual machine terminates all its activity and exits when one of two things happens:

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

Layout Management

- Dynamic GUI
- Layout managers controls the placement of components within the container.



03.12.2021 JAT – Java Technology 54

BoxLayout – Arrange components horizontally or vertically. This layout manager respect system locales setting and change direction and orientation base on direction and orientation of text for current culture. So order and orientation for component with same setting will be different for European users and Chinese users, Arabian users.

FlowLayout – Arrange components like characters in text, so in order from left to right and if there is no space, components are wrapped to next line.

GridLayout – Arrange components to grid with fixed count of rows and columns. Each row and each column has same height and witdh

Examples of Swing GUI

Examples many Swing components and their possibilities can be found in demo applications call SwingSet2 and SwingSet3. These demo applications also include source codes, that helps understand way of using swing components in custom Java code.

- <JAVA_JDK_HOME>\demo\jfc\SwingSet2\
- https://swingset3.dev.java.net/ (online version – runnable through Java WebStart technology)

Examples many Swing components and their possibilities can be found in demo applications call SwingSet2 and SwingSet3. These demo applications also include source codes, that helps understand way of using swing components in custom Java code.
<JAVA_JDK_HOME>\demo\jfc\SwingSet2\
https://swingset3.dev.java.net/ (online version – runnable through Java WebStart technology)

JavaAPI for XML

- JAXP (SAX, DOM, XSLT, StAX)
  - Apache Xerces
- JAXB
  - JavaEE

XML – history

- **SGML** (*Standard Generalized Markup Language*) is a standard for defining generalized markup languages for documents. Which allows define markup language as oven subsets. SGML is a complex langugage which allows many markup syntaxes. That complexity is disadvantage for common usage.
- SGML is ISO standard called *ISO 8879:1986 Information processing—Text and office systems—Standard Generalized Markup Language (SGML)*

XML – history

- Language XML is created as profile (specialized subset) of SGML and become very popular.
- XML can be easy parsed and processed because of simplicity.

- XHTML, GML, SVG, MathML, DocBook

XML – example

```
<math xmlns=" h t t p : / /www.w3. org /1998/Math /MathML">
<mrow>
    <msup>
        <mfenced open=" [ " close=" ] ">
            <mrow>
                <mi>a< / mi>
                <mo>+< /mo>
                <mi>b< / mi>
            < /mrow>
        < /mfenced>
        <mn>260< /mn>
    < /msup>
< /mrow>
```

$[a+b]^{260}$

XML

View: Data model

- XML document is modeled as tree (XML tree).
- Notice: This data model was alsow presented in SGML language and in database community is known as weakly structured data.

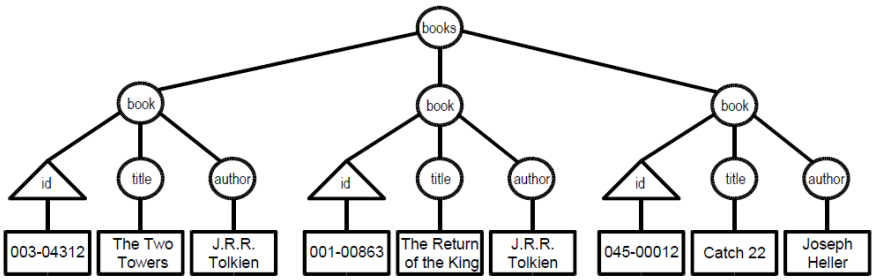
XML – well formed document

- Element has type identified by name (known as tag). Exampel: <book>...</book>.
- Element can contains set of pairs attribute=’value’.
- In text form XML document can be identified start (start-tag) and end mark(end-tag) of element (<name>...</name>).
- Text between start and end mark is called element content.

XML – well formed document

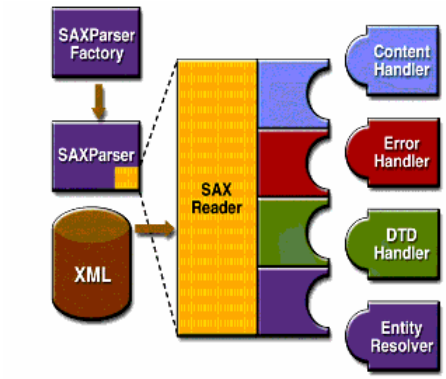
- If element contains others tags and characters contents is called mixed content. For example:
  <a>Hi, Mike.
- Elements with no content are called empty. Short syntax:
  .
- First lien contains XML declaration, for example:
  <?xml version="1.0" ?>
- Document is called well formed if fullfil all these rules.

XML - tree


books
book
book
book
id
title
author
id
title
author
id
title
author
003-04312
The Two Towers
J.R.R. Tolkien
001-00863
The Return of the King
J.R.R. Tolkien
045-00012
Catch 22
Joseph Heller

JavaAPI for XML – SAX

- Event model
- SAXParser inform about found start or end tags, …



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.

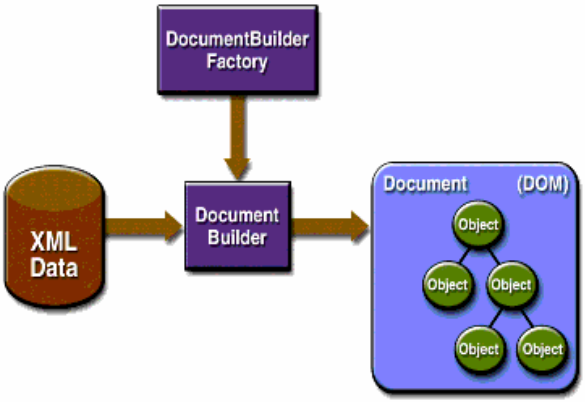
JavaAPI for XML – SAX

```
try {
   SAXParserFactory saxFactory =
               SAXParserFactory.newInstance();
   SAXParser parser = saxFactory.newSAXParser();
   XMLReader xmlReader = parser.getXMLReader();
   MyHandler h = new MyHandler();
   xmlReader.setContentHandler(h);
   xmlReader.setErrorHandler(h);
   InputSource source = new InputSource(
         new FileReader("pom.xml"));
   xmlReader.parse(source);
} catch (ParserConfigurationException | SAXException |
                           IOException e) {
   e.printStackTrace();
}
```

```java
public class MyHandler extends DefaultHandler {

    @Override
    public void startDocument() throws SAXException {
        //do something
    }
    @Override
    public void endDocument() throws SAXException {
        //do something
    }
    @Override
    public void startElement(String uri, String localName,
                String qName, Attributes atts) throws SAXException {
        //do something
    }
    @Override
    public void endElement(String uri, String localName,
                            String qName) throws SAXException {
        //do something
    }
    @Override
    public void characters(char[] ch, int start, int length)
                                            throws SAXException {
        //do something
    }
}
```

JavaAPI for XML – DOM

- Whole document is loaded to memory and DOM tree is created.



JavaAPI for XML – DOM

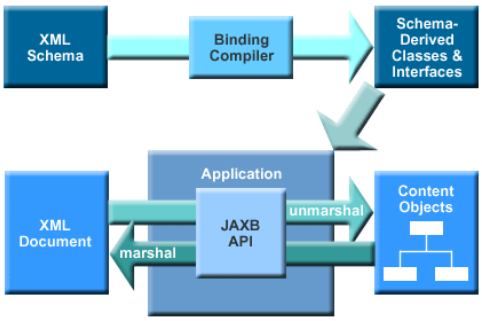
```
DocumentBuilderFactory dbfactory =
DocumentBuilderFactory.newInstance();
try {
   DocumentBuilder builder =
            dbfactory.newDocumentBuilder();
   Document doc = builder.parse(new File("pom.xml"));
   Element root = doc.getDocumentElement();
   NodeList nl = root.getChildNodes();
   for (int i = 0; i < nl.getLength(); i++) {
      String name = nl.item(i).getNodeName();
      System.out.println(name);
   }
   xpathSearch(doc);
} catch (Exception e) {
   e.printStackTrace();
}
```

JavaAPI for XML – DOM XPath

```
try {
   XPathFactory factory = XPathFactory.newInstance();
   XPath xPath = factory.newXPath();
   Object list = xPath.evaluate(
         "//plugin|//dependency", doc,
         XPathConstants.NODESET);
   NodeList nl = (NodeList) list;
   for (int i = 0; i < nl.getLength(); i++) {
      System.out.println(nl.item(i).getNodeName() +
            "='" + nl.item(i).getNodeValue() + "' :"
            + nl.item(i).getTextContent());
   }
} catch (XPathExpressionException e) {
   e.printStackTrace();
}
```

JAXB – Java Architecture for XML binding

- Easy access to data in XML file and storage of data into XML
- https://jaxb.java.net/
- Included in JavaSE 7 and newer



JAXB

- Binding a schema
  - xjc.bat -p book book.xsd -d d:\Temp
- Load of XML document

```
try {
File file = new File("file.xml");
JAXBContext jaxbContext =
JAXBContext.newInstance(Setting.class);
Marshaller jaxbMarshaller =
jaxbContext.createMarshaller();
// output pretty printed
jaxbMarshaller.setProperty(Marshaller.JAXB_FORMATTED_OU
TPUT, true);
jaxbMarshaller.marshal(setting, file);
jaxbMarshaller.marshal(setting, System.out);
} catch (JAXBException e) {}
```

JAXB

- Save of XML document

```
try {
  File file = new File("file.xml");
  JAXBContext jaxbContext =
JAXBContext.newInstance(Setting.class);
  Unmarshaller jaxbUnmarshaller =
jaxbContext.createUnmarshaller();
  Setting setting2 = (Setting)
jaxbUnmarshaller.unmarshal(file);
  System.out.println(setting2);
} catch (JAXBException e) {
  e.printStackTrace();
}
```

JAXB

```
@XmlRootElement
public class Setting {…

@XmlElement(name="file")
@XmlElementWrapper(name="files")
public void setLastUsedFiles(List<String>
lastUsedFiles) {…

@XmlAttribute
public void setPort(int port) {…
```

Directory services

- Directory service
  - It is a software system, that store, organize and provide information in directory structure (tree structure)
- X500 series of standards for directory services
- Examples of directory services
  - DNS, LDAP, UDDI
- http://en.wikipedia.org/wiki/Directory_service

Directory service
It is a software system, that store, organize and provide information in directory structure (tree structure)
X500 series of standards for directory services
Examples of directory services
DNS, LDAP, UDDI
http://en.wikipedia.org/wiki/Directory_service

Cited from:
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*)

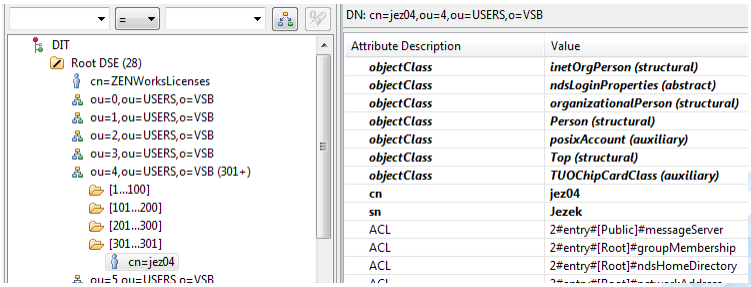
- **LDAP** is defined protocol for storing and access to data in directory server. Base on this protocol are individual entries stored on server and organized to tree structure (like in real directory architecture).
- Lightweight „version“ of X500
- http://cs.wikipedia.org/wiki/LDAP

LDAP is defined protocol for storing and access to data in directory server. Base on this protocol are individual entries stored on server and organized to tree structure (like in real directory architecture).
Lightweight „version“ of X500
http://cs.wikipedia.org/wiki/LDAP

Cited from:
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

LDAP – fundamental terms

- Attribute – has type and name
- Class – set of attributes
- Schema – define concrete structure of classes
- Object(Entry) – instance of one class or more then one class



Attribute – has type and name, it is basic structure for holding information. Type of attribute can be class.
Class – set of attributes.
Schema – define possible content of the entries in a subtree.
Object(Entry) – instance of one class or more then one class.

The tree defined by LDAP consist from entries. Each node of tree is one entry, that is a instance of one or more classes. These classes define which attributes (and how many) entry have to have and which can have. Each node (entry) of LDAP tree can have sub nodes (entries), the rules that restrict which entry can have which subentries are defined in schema.

Cited from:
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).

LDAP – fundamental terms

- DN = Distinguished Name
  - DN: cn=jez04,ou=4,ou=USERS,o=VSB
- RDN = Relative Distinguished Name
  - RDN: cn=jez04

  cn – common name

  dc – domain component

  mail – e-mail address and

  sn – surname

  o – organization

  ou – organization unit

DN = Distinguished Name – It is unique identifier of entry. It is consist from relative distinguished name (RDN) followed by parent (entry/node of LDAP tree that contains this entry/node as sub entry/node) entry.
Example of distinguished name:
DN: cn=jez04,ou=4,ou=USERS,o=VSB
RDN = Relative Distinguished Name – Is distinguish name in current context (subtree). It is constructed from some attribute of given entry.
Our example of DN consist of RDN cn=jez04 (constructed from attribute cn-common name) and DN of parent entry which is DN: ou=4,ou=USERS,o=VSB (an organization unit from organization VSB)

The type of attributes in LDAP are predefined. It is possible to create new attribute type, but its require get world unique identification called OID from LDAP OID vendors that is very uncommon.
Most used type of attributes:
cn – common name
dc – domain component
mail – e-mail address and
sn – surname
o – organization
ou – organization unit

Cited from:
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.

Java Naming and Directory Interface (JNDI)

- JNDI is part of Java API that provide unified access to various naming and directory services for java application.
- http://java.sun.com/products/jndi/tutorial/index.html

JNDI is part of Java API that provide unified access to various naming and directory services for java application.

Cited from:
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.

JNDI – Naming Concepts

- Name – object naming
  - Name convention
    - Directory path in UNIX system: /home/fei
    - DNS: www.vsb.cz
    - LDAP: cn=jez04,ou=4,ou=USERS,o=VSB
- Bindings
  - Association of name with object
- References and Addresses
- Context
  - Context is set of bindings (name-to-object bindings)
  - Has defined name convention.
  - Subcontext
- Naming System

Naming concept of JNDI generalize concepts of many directory services and can be summarized as follow:
Name – each object in directory structure has unique name, that can be divided into parts, that define position of object in the directory structure.
Name convention – Each directory service has own naming convention. For example:
- Directory pathname (name of file together with path to file) in UNIX system use character slash to separate individual parts of name and name start with root element of tree and continue with sub-nodes: /home/fei
  - Domain Name System (DNS) use for separation character dot and start with leaf node of tree structure and continue with parents elements and end with root node: www.vsb.cz
  - LDAP protocol use character comma as a separator and start with leaf node and continue with parents as DNS: cn=jez04,ou=4,ou=USERS,o=VSB

Bindings – The name is only one part of directory services second is stored object. Association between names and stored object is called bindings. For example file pathname is connected with data contained in file, domain name of server is connected with IP address of server and LDAP distinguish name is connected with entry containing stored attributes.
References and Addresses – some object cannot be stored directly so each directory service provide system for storing references to object or addresses to object.
Context – Context is set of bindings (name-to-object bindings) that have defined name convention. Name in one context object can be bound to another context object called subcontext, that have same naming convention. For example one directory in file system ("/home") is an context and subdirectory ("/home/fei") of that directory is subcontext.
Naming System – A naming system is a connected set of contexts of the same type (they have the same naming convention) and provides a common set of operations. Examples of naming systems: UNIX file system, DNS for whole internet.

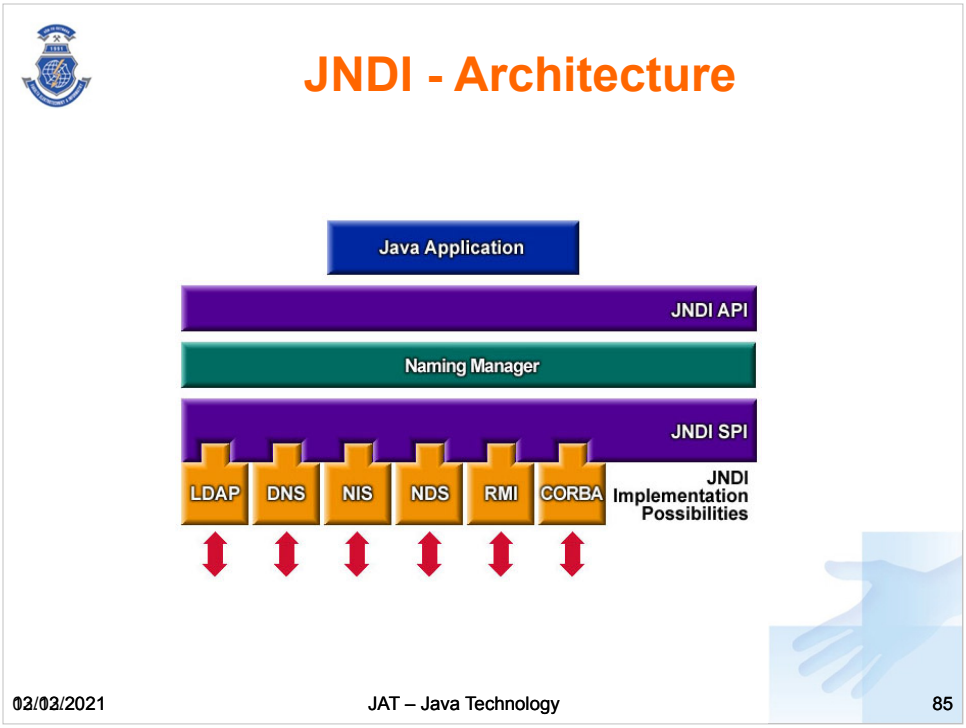
Cited from:
Naming Concepts
A fundamental facility in any computing system is the *naming service*--the means by which names are associated with objects and objects are found based on their names. When using almost any computer program or system, you are always naming one object or another. For example, when you use an electronic mail system, you must provide the name of the recipient to whom you want to send mail. To access a file in the computer, you must supply its name. A naming service allows you to look up an object given its name. A naming service's primary function is to map people-friendly names to objects, such as addresses, identifiers, or objects typically used by computer programs. For example, the Internet Domain Name System (DNS) maps machine names (such as www.sun.com) to IP addresses (such as 192.9.48.5). A file system maps a filename (for example, c:\bin\autoexec.bat) to a file handle that a program can use to access the contents of the file. These two examples also illustrate the wide range of scale at which naming services exist--from naming an object on the Internet to naming a file on the local file system.
Names
To look up an object in a naming system, you supply it the *name* of the object. The naming system determines the syntax that the name must follow. This syntax is sometimes called the naming system's *naming convention*.
For example, the UNIX™ file system's naming convention is that a file is named from its path relative to the root of the file system, with each component in the path separated from left to right using the forward slash character ("/"). The UNIX pathname, /usr/hello, for example, names a file hello in the file directory usr, which is located in the root of the file system.
The DNS naming convention calls for components in the DNS name to be ordered from right to left and delimited by the dot character (".."). Thus the DNS name sales.Wiz.COM names a DNS entry with the name sales, relative to the DNS entry Wiz.COM. The DNS entry Wiz.COM, in turn, names an entry with the name Wiz in the COM entry.
The Lightweight Directory Access Protocol (LDAP) naming convention orders components from right to left, delimited by the comma character (","). Thus the LDAP name cn=Rosanna Lee, o=Sun, c=US names an LDAP entry cn=Rosanna Lee, relative to the entry o=Sun, which in turn, is relative to c=us. The LDAP has the further rule that each component of the name must be a name/value pair with the name and value separated by an equals character ("=").
Bindings
The association of a name with an object is called a *binding*. For example, a file name is *bound* to a file. The DNS contains bindings that map machine names to IP addresses. An LDAP name is bound to an LDAP entry.
References and Addresses
Depending on the naming service, some objects cannot be stored directly; that is, a copy of the object cannot be placed inside the naming service. Instead, they must be stored by reference; that is, a *pointer* or *reference* to the object is placed inside the naming service. A reference is information about how to access an object. Typically, it is a much more compact representation that can be used to communicate with the object, while the object itself might contain more state information. Using the reference, you can contact the object and obtain more information about the object.
For example, an airplane object might contain a list of the airplane's passengers and crew, its flight plan, and fuel and instrument status, and its flight number and departure time. By contrast, an airplane object reference might contain only its flight number and departure time. The reference is a much more compact representation of information about the airplane object and can be used to obtain additional information. A file object, for example, is accessed using a *file reference*, also called a *file handle*. A printer object, for example, might contain the state of the printer, such as its current queue and the amount of paper in the paper tray. A printer object reference, on the other hand, might contain only information on how to reach the printer, such as its print server name and printing protocol.
Although in general a reference can contain any arbitrary information, it is useful to refer to its contents as *addresses* (or communication end points): specific information about how to access the object.
For simplicity, this tutorial uses "object" to refer to both objects and object references when a distinction between the two is not required.
Context
A *context* is a set of name-to-object bindings. Every context has an associated naming convention. A context provides a lookup (*resolution*) operation that returns the object and may provide operations such as those for binding names, unbinding names, and listing bound names. A name in one context object can be bound to another context object (called a *subcontext*) that has the same naming convention.
For example, a file directory, such as /usr, in the UNIX file system is a context. A file directory named relative to another file directory is a subcontext (some UNIX users refer to this as a *subdirectory*). That is, in a file directory /usr/bin, the directory bin is a subcontext of usr. In another example, a DNS domain, such as COM, is a context. A DNS domain named relative to another DNS domain is a subcontext. For example, in the DNS domain Sun.COM, the DNS domain Sun is a subcontext of COM.
Finally, an LDAP entry, such as c=us, is a context. An LDAP entry named relative to another LDAP entry is a subcontext. For example, in the an LDAP entry o=sun,c=us, the entry o=sun is a subcontext of c=us.
Naming Systems and Namespaces
A *naming system* is a connected set of contexts of the same type (they have the same naming convention) and provides a common set of operations.
For example, a system that implements the DNS is a naming system. A system that communicates using the LDAP is a naming system.
A naming system provides a *naming service* to its customers for performing naming-related operations. A naming service is accessed through its own interface. For example, the DNS offers a naming service that maps machine names to IP addresses. The LDAP offers a naming service that maps LDAP names to LDAP entries. A file system offers a naming service that maps filenames to files and directories.
A *namespace* is the set of names in a naming system. For example, the UNIX file system has a namespace consisting of all of the names of files and directories in that file system. The DNS namespace contains names of DNS domains and entries. The LDAP namespace contains names of LDAP entries.

JNDI – Directory Concepts

- Attributes
  - Identifier
  - Value
- Directory
  - Set of objects
- Directory Service
  - Service provided basic operation such as add, delete, modify, search.

While the name concept of JNDI provide only the names (in tree structure) and their connection with objects, directory concept of JNDI provide extension that allow functionality very similar to LDAP.
Directory concept provide:
- **Attributes** – each attribute has identifier and value.
- **Directory** - A directory is a connected set of directory objects.
- **Directory service** - Directory service provided basic operation such as add, delete, modify, search

Cited from:
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 is not naming or directory service, but it is common API for accessing different naming and directory service. Architecture of JNDI consist from three layers:

1. JNDI API that provide application interface to java applications.
2. NamingManager – implements JNDI functionality.
3. JNDI SPI – service provider interface, that provide interface for connecting provider, that connect JNDI to custom naming or directory service. There exist many provider for different services.

Cited from:

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

JNDI – Usage

Basic steps

- Download SPI library to connect to the service provider
  - Oracle provides library for the most widely used naming and directory services (LDAP, DNS, File system, Novell, …)
    - http://java.sun.com/products/jndi/serviceproviders.html
- Setting of environment variable for base context
- Obtain the context
- Using the context

Basic steps

Download SPI library to connect to the service provider

Oracle provides library for the most widely used naming and directory services (LDAP, DNS, File system, Novell, …)

http://java.sun.com/products/jndi/serviceproviders.html

Setting of environment variable for base context

Obtain the context

Using the context

JNDI – Usage: Setting

Setting of Environment Variables – Connection to LDAP

```
   Hashtable<String, String> env =
                                 new Hashtable<String,
String>();
   env.put(Context.INITIAL_CONTEXT_FACTORY,

"com.sun.jndi.ldap.LdapCtxFactory");
   env.put(Context.PROVIDER_URL,
                                 "ldap://
pca1035a.vsb.cz:10389/o=jat");
   env.put(Context.SECURITY_CREDENTIALS, "secret");
   env.put(Context.SECURITY_PRINCIPAL,
                                 "uid=admin, ou=system");
   env.put(Context.SECURITY_AUTHENTICATION, "simple");
```


Setting of environment variables is first step, there are two fundamental variables:

INITIAL_CONTEXT_FACTORY – Contains full canonical name of java class that implements interface javax.naming.spi.InitialContextFactory. The name of that class provide author of SPI library.

PROVIDER_URL - Contains URL address to server where real directory server is running.

Next three variables are specific for selected SPI library (LDAP in this case).

SECURITY_PRINCIPAL – Contains user name for user that authenticate to LDAP server.

SECURITY_CREDENTIALS – Contains password for user.

SECURITY_AUTHENTICATION – Contains name of method of authentication defined by LDAP protocol

JNDI – Usage: Setting

Setting of Environment Variables – Connection to File System

```
   Hashtable<String, String> env =
                                 new Hashtable<String,
String>();
   env.put(Context.INITIAL_CONTEXT_FACTORY,

"com.sun.jndi.fscontext.RefFSContextFactory");
   env.put(Context.PROVIDER_URL,
                "file:/tmp/tutorial/");

   Context ctx = new InitialContext(env);
```


Setting of variables is similar as in previous case. There are two common variables

INITIAL_CONTEXT_FACTORY and PROVIDER_URL, that refer to file system provider class and directory of file system which will be used as root of JNDI context.

No other setting are necessary.

JNDI – Usage: Obtaining Context

```
env.put(Context.INITIAL_CONTEXT_FACTORY,
   "com.sun.jndi.ldap.LdapCtxFactory");
env.put(Context.PROVIDER_URL,
   "ldap://pca1035a.vsb.cz:10389/o=jat");
...

try {
   Context ctx = new InitialContext(env);

   Object r  = ctx.lookup(
   "cn=homedir,cn=Jon Ruiz,ou=People/Windows");
   System.out.println(r);

   } catch (NamingException e) {
   }
```


Once we have set the variables, we simply create an object of class IntialContext. If we don't have environment variables set in operating system, we have to pass object of class Hashtable with definition of these variables in constructor.

If we have environment variables set in operating system it is better, because code doesn't contains any login specific information and whole code is more reusable. Changing user login information even whole provider of service doesn't influence the code. For example it is very useful for testing, because application can be run in test environment with different set

JNDI – Usage: Using the Context

```
env.put(Context.INITIAL_CONTEXT_FACTORY,
   "com.sun.jndi.ldap.LdapCtxFactory");
env.put(Context.PROVIDER_URL,
   "ldap://pca1035a.vsb.cz:10389/o=jat");

try {
   Context ctx = new InitialContext(env);

   Object r  = ctx.lookup(
   "cn=homedir,cn=Jon Ruiz,ou=People/Windows");
   System.out.println(r);

   } catch (NamingException e) {
   }
```


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.

JNDI – Usage: Obtaining and Using the Directory Context

- Obtain of directory context

```
//Setting of environment variables
DirContext ctx = new InitialDirContext(env);
```

- Read the attributes from an entry

```
Attributes attrs = ctx.getAttributes(
        "cn = Ted Geisel, ou=People");
attrs.get("sn").get();
```


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

JNDI – Listing

- `Context.list()`
  - Only list of name
- `Context.listBindings()`
  - List of names and objects that are bind to them
- `NamingEnumeration interface`
  - *`NamingEnumeration<NameClassPair>`*
  - *`NamingEnumeration<Binding>`*

  Closure:
  - `hasMore()` return `false`
  - `NamingEnumeration.close()`
  - `NamingException`

JNDI API provide two of listing entries in the context. It can list only names of object or names and bound object. These two ways are important due the performance and effectiveness of code.

If application need only list of names, from them select only few of them and after that, for the selected names application need obtain bound object, it is recommended use listing of names only. If application need list of names and for all or almost all need bound object it is recommended use list of bindings.

Both methods return NamingEnumeration

JNDI – Operation with Object

- **`Context.bind()`**
  - Insert new specified object and bound it with specified new name
- **`Context.rebind()`**
  - Insert new object and bound it with existing name
- **`Context.unbind()`**
  - Delete name and bound object
- **`Context.rename()`**
  - Change the name for specified entry

JNDI provide these operation over entry names and bound object:

Context.bind()

Insert new specified object and bound it with specified new name

Context.rebind()

Insert new object and bound it with existing name

Context.unbind()

Delete name and bound object

Context.rename()

Change the name for specified entry

JNDI – Operation with Context

- **`Context.createSubcontext()`**
  - Create new subcontext (node in tree structure)
- **`Context.destroySubcontext()`**
  - Delete existing subcontext
- **`Context.rename()`**
  - Change the name for specified entry even context

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 Operation

Obtaining attributes collection

- `Attributes answer = ctx.getAttributes("some dn");`
  - `answer.getAll(); answer.getIDs();`

Modification of collection of attributes

- `ctx.modifyAttributes(name, DirContext.REPLACE_ATTRIBUTE, atributes);`
- `DirContext.ADD_ATTRIBUTE`
- `DirContext.REMOVE_ATTRIBUTE`

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.

JNDI – Directory Operation

Modification of attributes using ***ModificationItem***

- Create object of class ***ModificationItem***

```
new
  ModificationItem( DirContext.REPLAC
  E_ATTRIBUTE,
      new BasicAttribute("mail",
      "geisel@wizards.com"));
```

- Pass collection of ***ModificationItem*** object to method **modifyAttributes**

```
ctx.modifyAttributes(name, modItems);
```


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 Operation

- Searching using attribute

```
// ignore attribute name case
Attributes matchAttrs = new BasicAttributes(true);
matchAttrs.put(new BasicAttribute("sn", "Geisel"));
matchAttrs.put(new BasicAttribute("mail"));
NamingEnumeration<SearchResult> answer =
ctx.search("ou=People", matchAttrs);
```

- Searching using filters

```
SearchControls ctls = new SearchControls();
String filter = "(&(sn=Geisel)(mail=*))";
NamingEnumeration<SearchResult> answer = ctx.search(
            "ou=People", filter, ctls);
```


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.

JNDI – Directory Operation

- Searching filter syntax
  - Prefix notation of logical formulas
  - & (and), | (or), ! (not), = , ~=, >=, <=, =*, *, \
- Example

```
(| (& (sn=Geisel) (mail=*)) (sn=L*))
```

- Control of search operation `SearchControls`
  - Scope of searching
  - Required attributes
  - Time limit
  - Limitation for number of returned entry

Filter formula for advance search condition has prefix notation, that means formula A + B in infix notation is rewrite to + A B in prefix notation. Generally instead triple operand, operator, operand in infix notation, prefix notation just change order to operator, operand, operand. Filter syntax contains standard operators like and, or, equals, not equals. But there exist one uncommon operator ~=, that means approximate equality according to the matching rule of the attribute, but on some system for names can have meaning “sound like”.

Another important thing in search using filters, are object of type SearchControls, that can modify scope of searching, list of required attribute, time limit or maximum number of result entries.

JNDI - Names

- Can be used two types of names
  - Strings
  - Interface **Name**
- **CompoundName** implements interface **Name**
  - Name is composed from individual parts of tree nodes
  - Can be easily generated programmatically
  - Construction of name is independent of context system provider.
    - C:\Windows\system\
    - /home/staff/
    - cn=jez04, ou=4, ou=USERS, o=VSB
    - floreon.vsb.cz

JNDI support two types of names

Name defined in string, that must respect context system provider name syntax

Interface **Name**

CompoundName implements interface **Name**

Name is composed from individual parts of tree nodes.

Can be easily generated programmatically.

Construction of name is independent of context system provider:

C:\Windows\system\

/home/staff/

cn=jez04, ou=4, ou=USERS, o=VSB

JNDI – Names: CompoundName

- Name is composed from individual parts of tree nodes.
- Collection of parts of name contains root at position with index 0 and continue with subcontext until reach current entry.
- That means construction of name is always same even it is used file system provider or DNS provider, where name syntax has different direction (from root to leaf and from leaf to root) and different name parts separator.
- Parsing of string name to **CompoundName**

```
NameParser parser = ctx.getNameParser("");
Name cn = parser.parse(compoundStringName);
```

- Methods for **CompoundName** construction

```
getAll();
get(int posn);
getPrefix(int posn);
getSuffix(int posn);
add(String comp);
add(int posn, String comp);
addAll(Name comps);
addAll(Name suffix);
addAll(posn, Name suffix);
remove(posn);
```


Name is composed from individual parts of tree nodes.

Collection of parts of name contains root at position with index 0 and continue with subcontext until reach current entry.

That means construction of name is always same even it is used file system provider or DNS provider, where name syntax has different direction (from root to leaf and from leaf to root) and different name parts separator.

Parsing of string name to **CompoundName**

```
NameParser parser = ctx.getNameParser("");
Name cn = parser.parse(compoundStringName);
```

Methods for **CompoundName** construction

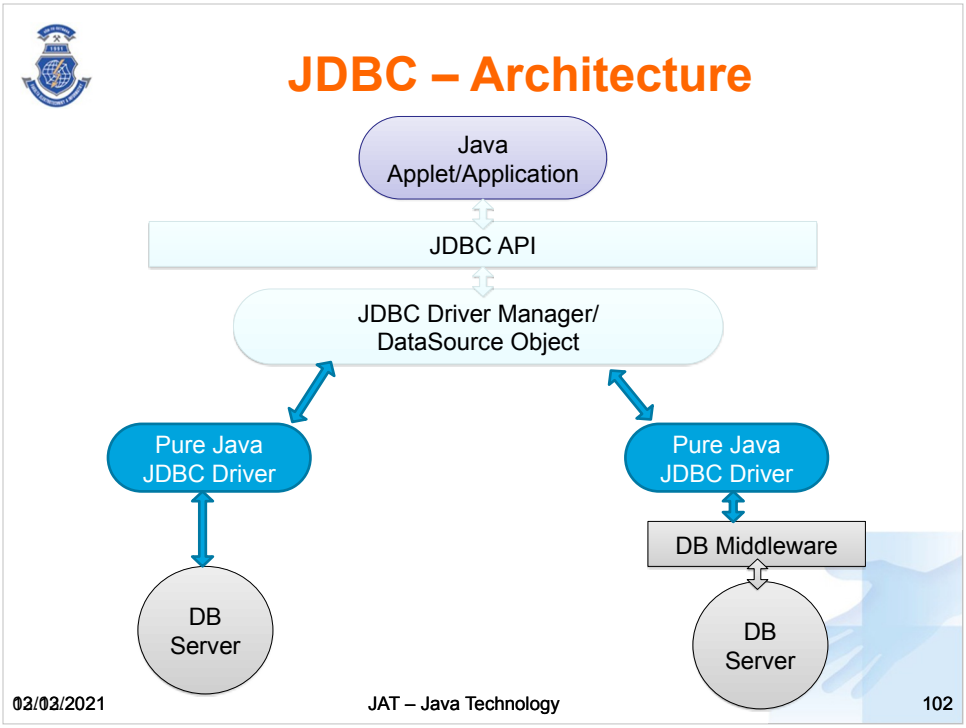
JDBC – Java DataBase Connection

- Technology for unified access to database form Java.
- JDBC API
  - Establishing connection with database
  - Creating and sending SQL statement to database
  - Retrieving and process result of SQL query
- http://java.sun.com/docs/books/tutorial/jdbc/basics/index.html

Technology for unified access to database form Java.

JDBC API provide functionality for three fundamental steps for communication with database server.

1. Establishing connection with database.
2. Creating and sending SQL statement to database.
3. Retrieving and process result of SQL query.



JDBC Architecture consist form:
JDBC API that provide unified interface for java application.
JDBC DriverManager that provide association between JDBC API and Pure Java JDBC driver.
Pure Java JDBC driver that is provided by producer of database server and communicate with database server throw proprietary protocol and provide JDBC API for java application.

Cited from:
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 Drivers

- Types:
  - JDBC – ODBC bridge driver
  - Java and native code Driver
  - Pure Java Driver – communication with database server
  - Pure Java Driver – communicate with middleware server

- Concrete drivers for common database systems
  - MySQL Connector/JDBC
    - http://www.mysql.com/downloads/connector/j/
      - mysql-connector-java-X.X.XX-bin.jar
  - Oracle Database 11g R2 JDBC driver
    - http://www.oracle.com/technetwork/database/enterprise-edition/jdbc-112010-090769.html
  - JavaDB – distribution of Apache Derby database
    - Included in JDK from version 7
    - Written completely in Java
  - MSSQL, …
- **Obtained driver library must be linked to Java application in same way as other used libraries.**

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.

JDBC – Driver Initialization

- Using **DriverManager**

```
try {
   Class.forName("com.mysql.jdbc.Driver")
         .newInstance();
   Class.forName("oracle.jdbc.OracleDriver")
         .newInstance();

Class.forName("org.apache.derby.jdbc.EmbeddedDriver")
         .newInstance();

} catch (ClassNotFoundException e) {
   e.printStackTrace();
} catch (InstantiationException e) {
   e.printStackTrace();
} catch (IllegalAccessException e) {
   e.printStackTrace();
}
```


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.

JDBC – Establishing Connection

- Using **DriverManager**

```
try {
   Connection conn = DriverManager.getConnection(
         "jdbc:mysql://localhost/shop" +
         "?user=shop&password=shop");
   Statement stm = conn.createStatement();
   ResultSet rs = stm.executeQuery("show tables");
   while(rs.next()){
      System.out.println(rs.getString(1));
   }
} catch (SQLException e) {
   e.printStackTrace();
}
```

- Connection string syntax
  - jdbc:oracle:thin:[user/password]@[host][:port]:SID
  - jdbc:derby:database name[;create=true]
  - jdbc:mysql://[host][,failoverhost...][:port]/[database][?propertyName1][=propertyValue1][&propertyName2][=propertyValue2]...

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.

JDBC – Establishing Connection

- Obtaining of **DataSource**

```
Context ctx = new InitialContext(env);
DataSource ds = (DataSource) ctx.lookup(
            "java:comp/env/jdbc/myDB");
Connection con = ds.getConnection();
```

- Creation of **DataSource**

```
com.mysql.jdbc.jdbc2.optional.MysqlDataSource ds =
   new MysqlDataSource();
ds.setPort(3306);
ds.setDatabaseName("jat_example");
ds.setServerName("localhost");
ds.setUser("admin");
ds.setPassword(„heslo");
ctx.bind("java:comp/env/jdbc/myDB", ds);
```


Another way of obtaining connection to database is by using DataSource. Object that implements interface DataSource holds all necessary information for connecting to database. But proposed way how to use DataSource is not only create object that implements interface DataSource, but it is more sophisticate:

- DataSource is only interface with method getConnection(). Provider of JDBC library should provide class that implements DataSource interface.
- First step is create instance of such as class and fulfill all necessary information into it.
- Fulfilled DataSource is stored on some server accessible via JNDI

JDBC – SQL Query

- Creating and sending SQL query to database

```
con = factory.getConnection();
Statement stmt = con.createStatement(
                    ResultSet.TYPE_SCROLL_SENSITIVE,
                    ResultSet.CONCUR_READ_ONLY);
ResultSet rs = stmt.executeQuery(
            "SELECT cof_name, price FROM coffes");
```

- Other methods of class ***Statement***
  - `boolean execute(String)`
  - `ResultSet getResultSet()`
  - `int executeUpdate(String)` (for SQL INSERT, UPDATE, DELETE)
  - `close()`

If application has connection to database, it can create Statement using method createSatatment. That method vas two parameters that can set type of ResultSet generated by this Statement.

First parameter:

ResultSet.TYPE_FORWARD_ONLY – cursor of the result set may move only forward.

ResultSet.TYPE_SCROLL_SENSITIVE – cursor of the result set may move forward and backward and Result set is sensitive to changes to the data.

ResultSet.TYPE_SCROLL_INSENSITIVE – cursor of the result set may move forward and backward and Result set is insensitive to changes to the data.

JDBC – Process Results

- Process results – read data and ResultSet metadata

```
ResultSetMetaData meta = rs.getMetaData();
while(rs.next()){
for(int i=1; i<=meta.getColumnCount(); i++){
System.out.println(meta.getColumnName(i) +
": " + rs.getString(i));
}
```

- Moving cursor
  - When result set is returned cursor point before first row
  - `next()` return `false` if cursor move after last row
  - `last()`, `first()` , `previous()` , `relative(int)` , `absolute(int)`

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.

JDBC – Process Results

- Data reading
  - get**XXX**()
    - byte, double, float, int, long, string, short, BigDecimal, Blob, Date, Time
- Meta-information
  - `ResultSetMetaData getMetaData()`

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).

JDBC – Update of Tables

- Using SQL query (`UPDATE myTable SET column1='value' WHERE …`)
- Using `ResultSet`

```
Statement stmt = con.createStatement(
ResultSet.TYPE_SCROLL_SENSITIVE,
   ResultSet.CONCUR_UPDATABLE);
ResultSet uprs = stmt.executeQuery(
   "SELECT COF_NAME, PRICE FROM COFFEES");
uprs.next();
uprs.updateString("COF_NAME", "Foldgers");
uprs.updateRow();
```

- `cancelRowUpdates();`
- `updateXXX()` - double, float, int, string, Time, …
- `deleteRow()`
- `moveToInsertRow()`

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 .

JDBC – Automatically Generated Keys

- 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.

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.

JDBC – Prepared Statements

- 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)

```
PreparedStatement prepStm = con.prepareStatement(
"UPDATE COFFEES SET SALES = ? WHERE COF_NAME LIKE ?");

prepStm.setInt(1, 75);
prepStm.setString(2, "Colombian");
prepStm.executeUpdate();
```


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

JDBC – 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()**

```
con.setAutoCommit(false);
updateSales.executeUpdate("UPDATE COFFEES SET SALES = 50 WHERE
COF_NAME LIKE 'Colombia'");
updateTotal.executeUpdate("UPDATE COFFEES SET TOTAL = TOTAL + 50
WHERE COF_NAME LIKE 'Colombia'");
con.commit();
con.setAutoCommit(true);
```


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

JDBC – Transaction

- rollback() – cancel transaction and change values in DB into state before transaction begin.
- SavePoint – allow rollback transaction to this point (SavePoint)

```
//Process some SQL statements
Savepoint svpt1 = con.setSavepoint("SAVEPOINT_1");
// Process some SQL statements
con.rollback(svpt1);
// Process some SQL statements
con.commit();
```

- `con.releaseSavepoint(svpt1);`

JDBC API also provide support for canceling transaction (rollback) and from version 3.0 has support of savepoints. Savepoints is method how to implement nested transaction. If application create save point within transaction, that transaction can be rollback whole or to specified savepoint.

Don't forgot release save point by method releaseSavepoint() that free system resource associated with specified savpoint.

JDBC – Stored Procedures

- Stored procedure sreation

```
String createProcedure = "proprietary code to creating
gprocedur";
Statement stmt = con.createStatement();
stmt.executeUpdate(createProcedure);
```

- Stored procedure call

```
CallableStatement cs = con.prepareCall(
        "{call SHOW_SUPPLIERS(?, ?)}");

cs.setXXX(int, String);

ResultSet rs = cs.executeQuery();
```


Stored procedure are SQL code, that is stored and executed on database server. JDBC support for stored procedures is limited. Creation of stored procedure is not unified and developer has to write SQL code in specific SQL dialect for the database server. That code can be executed using standard statement and execute() or executeUpdate() method. Support for creation of stored procedures is not so important, because in most cases application don't create stored procedure at runtime. SQL stored procedures are in most case already deployed on database server and application just call them.

Invocation of stored procedures is

JDBC – Exceptions, Warnings

- SQL erroes and exception threw by database server are wrap to java exception with class **SQLException**
- Warnings get be get from object of class **Connection**, **Statement**, **ResultSet**

```
SQLWarning warning = stmt.getWarnings();
while (warning != null) {
   System.out.println("Message: "+
warning.getMessage());
   System.out.println("SQLState: "+
warning.getSQLState());
   System.out.print("Vendor error code: ");
   System.out.println(warning.getErrorCode());
   warning = warning.getNextWarning();
}
```


SQL exception threw by database server are wrap to java exception with class **SQLException**

Warnings get be get from object of class **Connection, Statement, ResultSet.**

SQL statement can generate more then one warning and class SQLWarning provide method getNextWarning() to obtain next warning that occur dunning statement execution.

JDBC – Closing the Connection

- Don’t forget close the connection if you don’t need it anymore.

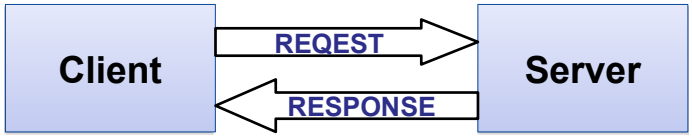
```
con.close();
```


Don’t forget close the connection if you don’t need it anymore.

Use method close() form interface Connection.

Hypertext Transfer Protokol (HTTP)

- 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.
- Methods: **GET, PUT, POST**, HEAD, DELETE, OPTIONS, TRACE, CONNECT
- Actual version 1.1, can use persistent connection (HTTP keep-alive) – one TCP connection is used for sending more then one requests and responses.

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

HTTP - Request

```
method URL_document version_HTTP
head
empty_line
request_body
```

- **Example:**

```
GET /papers/content.html HTTP/1.1
User-Agent: Mozilla/4.0 (compatible; MSIE
  5.0; Windows NT)
Host: www.server.cz

```

Method

Head

Empty line

No body

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.

HTTP Response

```
Protocol status_code status_message
Head
Empty_line
Response_body
```

- **Example:**

```
HTTP/1.1 200 OK
Server: Microsoft-IIS/5.0
Content-type: text/html
...

<!DOCTYPE HTML PUBLIC '-//W3C//DTD HTML 4.0
  Transitional//EN'>
...
```

Protocol

Head

Empty line

Body

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

HTTP and security

- Protocol contains only plain text data
  - Readability, simplicity
  - It is dangerous send data (password) through method GET:

```
GET /do.login?login=jez04&password=myPasswd ...
```

  - 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 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.

Restriction of HTTP – stateless 1/2

- 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.
    - Transferring all identification data in all request is dangerous.
  - Cookies – Mechanism for storing data sent by server in browser. That data are automatically send to server in each request.
    - Storing and transferring all identification data in all request is dangerous even with cookies.

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.
Transferring all identification data in all request is dangerous.
Cookies – Mechanism for storing data sent by server in browser. That data are automatically send to server in each request.
Storing and transferring all identification data in all request is dangerous even with cookies.

Restriction of HTTP – stateless 2/2

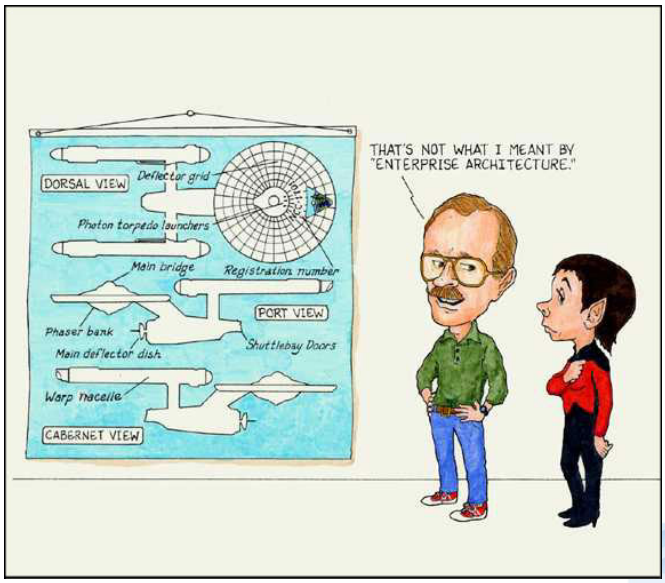
- Described disadvantages led to the introduction of sessions:
  - An identifier (called session id) is assigned to each new client and on server is stored pair of information the session id and client identification.
  - Session id is transferred to server with each request using cookies, parameter 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.

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.

HTTP – Reference

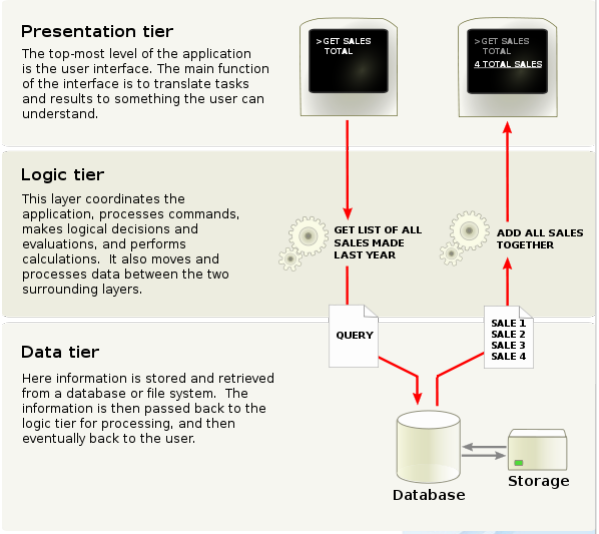
- World Wide Web Consortium: http://www.w3.org/, http://www.w3.org/Protocols/

Example of „Good“ Architecture



IS Architecture

- Multi-tier architecture
  - Subsystems are logically structured to tiers.
    - Typical tiers: presentation, application logic, data persistency.
  - Each tier represents abstraction with own responsibility. Advantages:
    - Tier can be easily understood and used.
    - Development inside tier is easier..
  - Tiers are isolated form changes in implementation of other tiers.



Application Servers

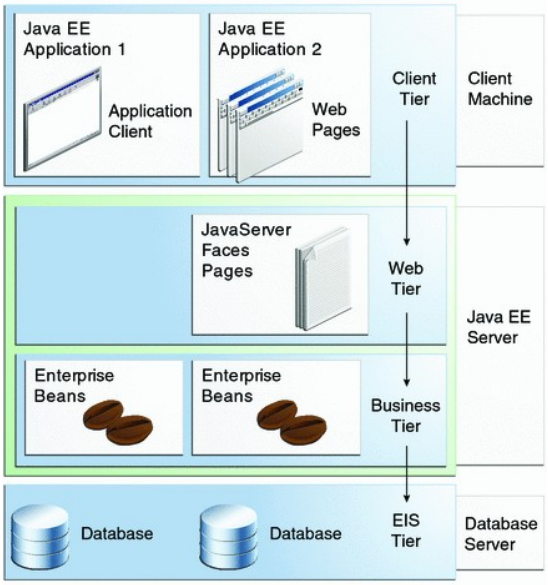
- Application server is software framework that provides an environment in which application can run.
- Application servers support:
  - Clustering
  - Fail-over – Automatic switching to a backup server, if primary server collapses.
  - Load balancing
- Application server for multi-tier architecture provide API to expose:
  - Presentation tier – Web tier used by web browsers.
  - Application logic tier – Business logic tier used by client applications.
- Application server is often integrated with web server.

Java EE Application Servers

- Java EE Application server implements application container specified by Java EE platform. That container provides interface between Java EE applications and low-level platform-specific services.

- Application server GlassFish
  - Application server implementation from company Sun (now days Oracle). GalshFish is reference implementation of Java EE specifications and is free for download (Open Source).
- Apache Tomcat
  - Apache Tomcat application server do not implements fully Java EE specification. This server implements only Java Servlet and Java Server Pages (JSP) technologies, but is often included like core module in application servers that fully implements Java EE specification.
- JBoss AS
  - JBoss Application Server fully implements Java EE specification and use Tomcat as core module. This server is one of most used open source Java EE application server.

Java EE – Application Model



The Java EE platform uses a distributed multitiered application model for enterprise applications. Application logic is divided into components according to function, and the application components are assigned to specific tier based on functionality of the component.

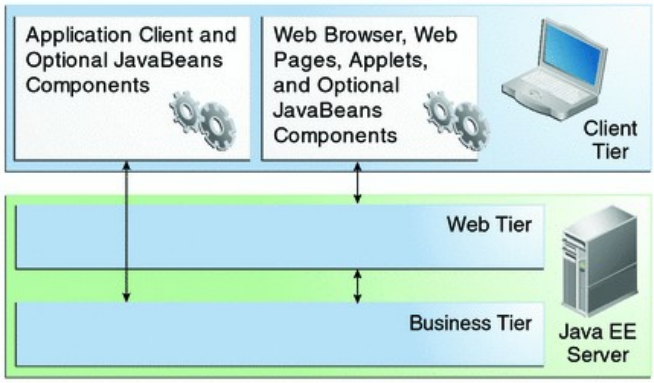
Client-tier components run on the client machine, in most cases client components run inside web browser.

Web-tier components run on the Java EE server.

Business-tier components run on the Java EE server.

Enterprise information system (EIS)-tier software runs on the EIS server.

Java EE – Communication with Server



Java EE Clients

A Java EE contains two basic types of clients:
a web client
an application client.

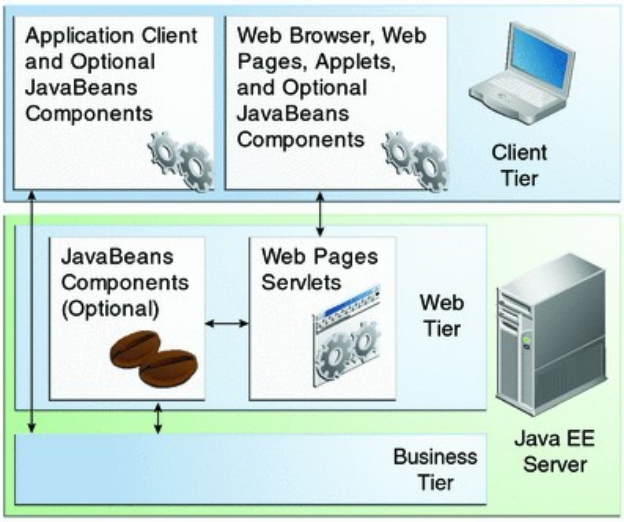
Web Clients

A **web client** consists of two parts. First is Dynamic web pages containing various types of markup language (HTML, XML, CSS, JavaScript and so on). And second is a web browser, which renders the pages received from the server.

Application Clients

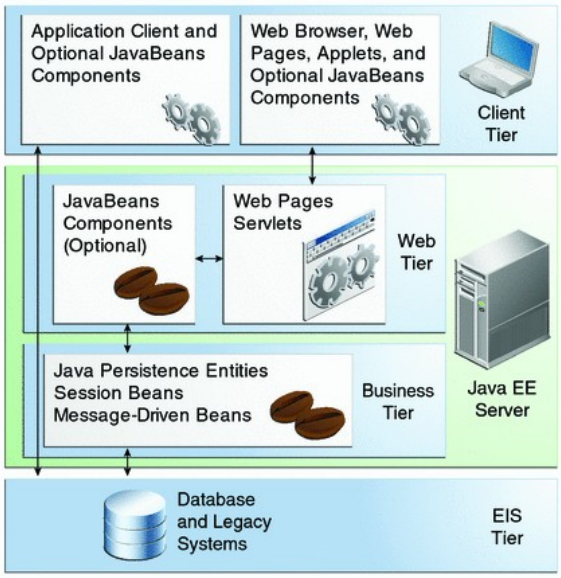
An application client is written in Java language, runs on a client machine and has often graphical user interface (GUI) created in the Swing or AWT API, but a command-line interface is certainly possible. Application clients directly access enterprise beans running in the business tier.

Java EE – Web Components



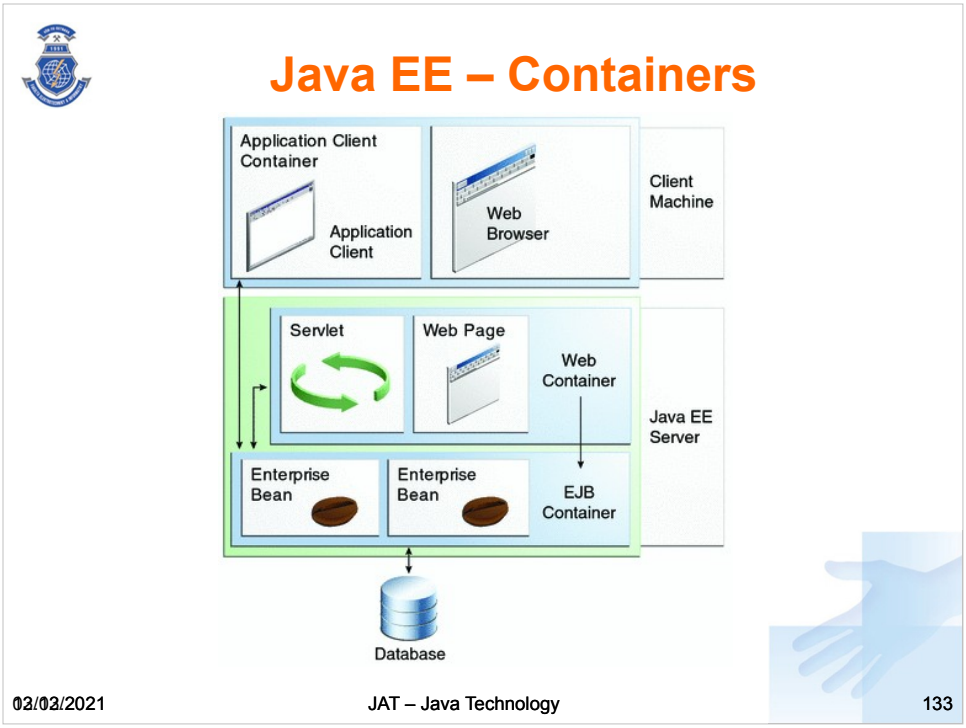
Java EE web components are either servlets or web pages created using JavaServer Faces technology or JSP technology (JSP pages). Web components generate dynamical web pages using markup languages (HTML, XHTML, XML and so on) and process response from web client (web browser). Standard web components often collaborate with user JavaBeans components.

Java EE – Business Components



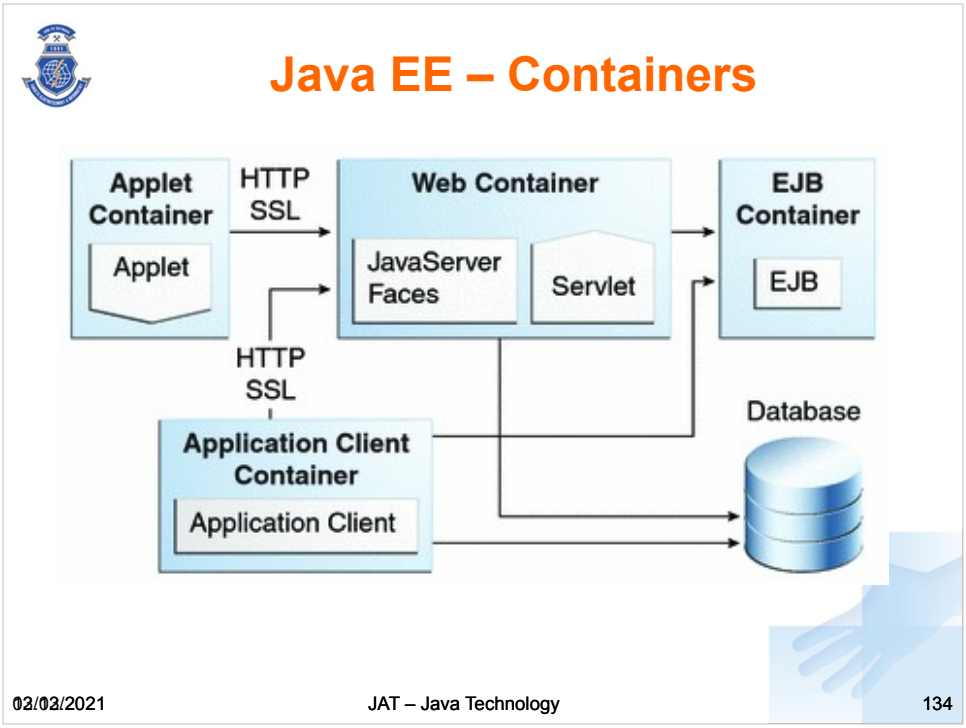
Business tire contains components that perform “business logic”. It is functionality of application independent of presentation logic that meets application requirements.

Business components are used by JavaBean components from web tire or directly by Client application.



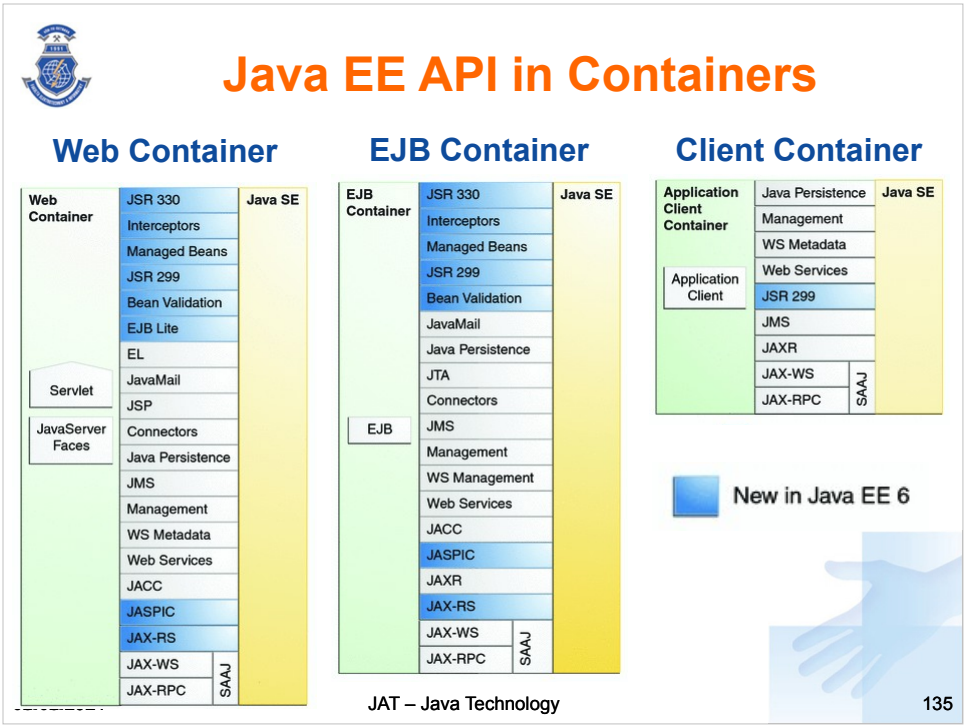
Containers are the interface between a component and the low-level platform-specific functionality that supports the component. Each type of component must be deployed to specific container.
Java EE application server provides EJB and web containers.

All containers from Java EE technology:
Enterprise JavaBeans (EJB) container: Manages the execution of enterprise beans for Java EE applications. Enterprise beans and their container run on the Java EE server.
Web container: Manages the execution of web pages, servlets, and some EJB components for Java EE applications. Web components and their container run on the Java EE server.
Application client container: Manages the execution of application client components. Application clients and their container run on the client.
Applet container: Manages the execution of applets. This container consists of a web browser and Java Plug-in running on the client together.



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.



Each container has defined specific set of provided functionality.

- Web container is focused on dynamic web page generation (JSP, JavaServerFaces, Servlet, EL, ManagedBeans …).
- EJB container is focused on functionality and data manipulation (Java Persistence, JTA - Java Transaction API, JAXB – Java Architecture for XML Binding …).
- Client container is focused on communication with EJB or web container.

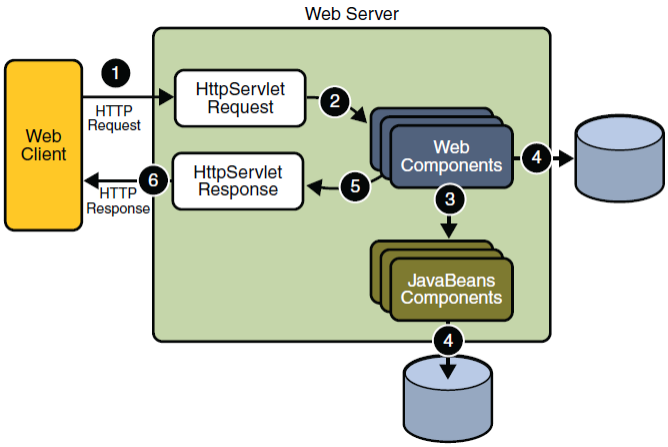
Web container and EJB container are quite similar because smaller application can use only web container that provide some functionality from EJB Container (EJB lite).

Java EE – Web Application Technologies

Java Servlet technology is the foundation of all the web application technologies. All other web technologies (either technologies from Java EE or the other – Apache Struts 2, JBoss SEAM, Wicket, Vaadin, Spring ...) are built over Java servlet technology.

There exist two separated technologies in Java EE platform for web development JavaServer Faces and JavaServer Pages.
JavaServer Faces is component like technology. Web page is generated by composition of components like Swing GUI.
JavaServer Pages is technology partially similar to PHP, because combine plain HTML markup language with pieces of java code or JavaBean properties.

Java EE – Web Application



Web applications provide the dynamic extension capabilities for a web server.

The client sends an HTTP request to the web server. A web server 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. After that the web component generates an HTTPServletResponse or it can pass the request to another web component. Web server sends data from HTTPServletResponse back to client.

Java Servlet

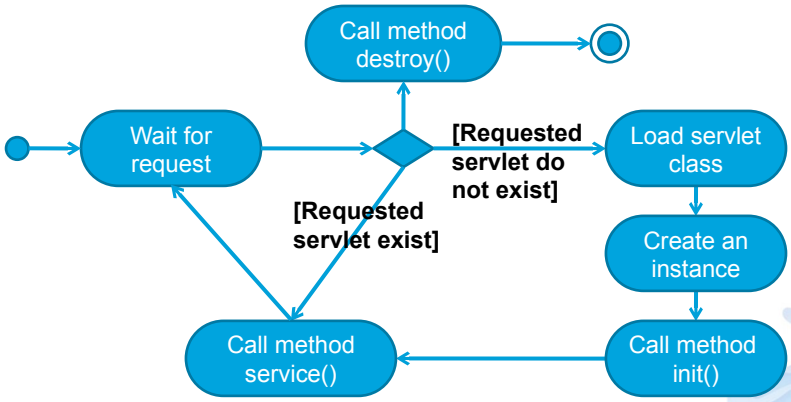
A Servlet

is a Java programming language class used to extend the capabilities of servers that can be accessed by a host application via a request-response programming model.

- Servlets can respond to any type of request, not only HTTP requests but HTTP requests are the most common requests.

Servlet – Lifecycle

- The Web container is responsible for managing the lifecycle of servlets and mapping a URL to a particular servlet.



Servlet vs. HTTPServlet

- Interface **Servlet** is a general and is not bind with protocol HTTP. Code for handling requests should be contained in method **service()**.
- Class **HTTPServlet** implements interface **Servlet**. Method **service()** is already implemented and parse attributes from requests and call one of the method **doPost()**, **doGet()**, … according to http request method (GET, POST, …).

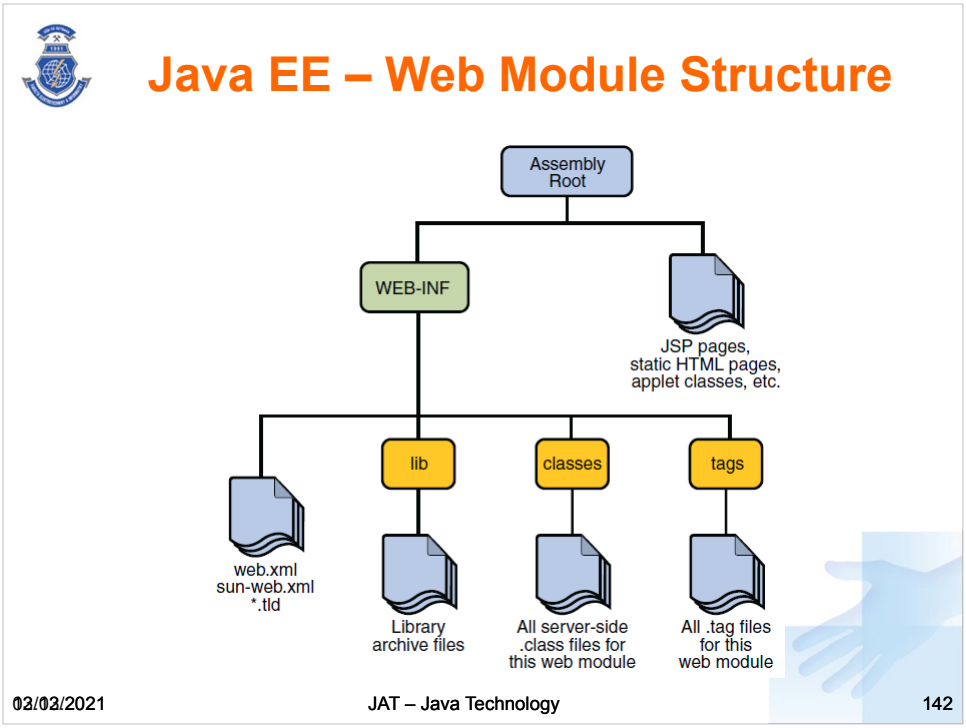
HTTPServlet - example

```
@WebServlet(description = "desc", urlPatterns = {"/MyServlet"})
public class MyFirstServlet extends HttpServlet {
    public MyFirstServlet() {
        super();
    }
    public String getServletInfo() {
   return "My first servlet";
   }
   @Override
   protected void doGet(HttpServletRequest request, HttpServletResponse
response) throws ServletException, IOException {
   doPost(request, response);
   }
   @Override
   protected void doPost(HttpServletRequest request,
HttpServletResponse response) throws ServletException, IOException {
   PrintWriter pw = response.getWriter();
   pw.println("<html><body>Hello world!</body></html>");
   pw.close();
   }
}
```


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!”



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.

Servlet Deployment

- WEB-INF\web.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app …"— namespace specification --">
  <display-name>JSPExample</display-name>
  <welcome-file-list>
    <welcome-file>index.html</welcome-file>
  </welcome-file-list>
 <servlet>
    <display-name>MyFirstServlet</display-name>
    <description>pokus pokus</description>
    <servlet-name>MyFirstServlet</servlet-name>
    <servlet-class>MyFirstServlet</servlet-class>
  </servlet>
  <servlet-mapping>
    <servlet-name>MyFirstServlet</servlet-name>
    <url-pattern>/MyFirstServlet</url-pattern>
  </servlet-mapping>
</web-app>
```


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.

Servlet – Request Information

- **String request.getParameter("parameter name")**
- HTTP request
  - http://[*host]:[port][request-path]?[query-string]*
  - Request-path: /MyServletApp/MyFristServlet/something
    - Context path: /MyServletApp
    - Servlet path: /MyFirstServlet
    - Path info: /something

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.

Servlet – Scope Objects

- javax.servlet.ServletContext
  - getServletConfig().getServletContext(),
  - Contains: attributes, context path
- javax.servlet.http.HttpSession
  - HTTPServletRequest.getSession()
  - Contains: attributes
- javax.servlet.ServletRequest
  - Parameter of methods service() a doGet (), doPost (), …
  - Contains: attributes
- javax.servlet.jsp.PageContext

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 – Response

- response.setContentType("text/html");
- setCharacterEncoding()
- response.getWriter()
- response.getOutputStream()
- HTTP
  - addCookie()

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.

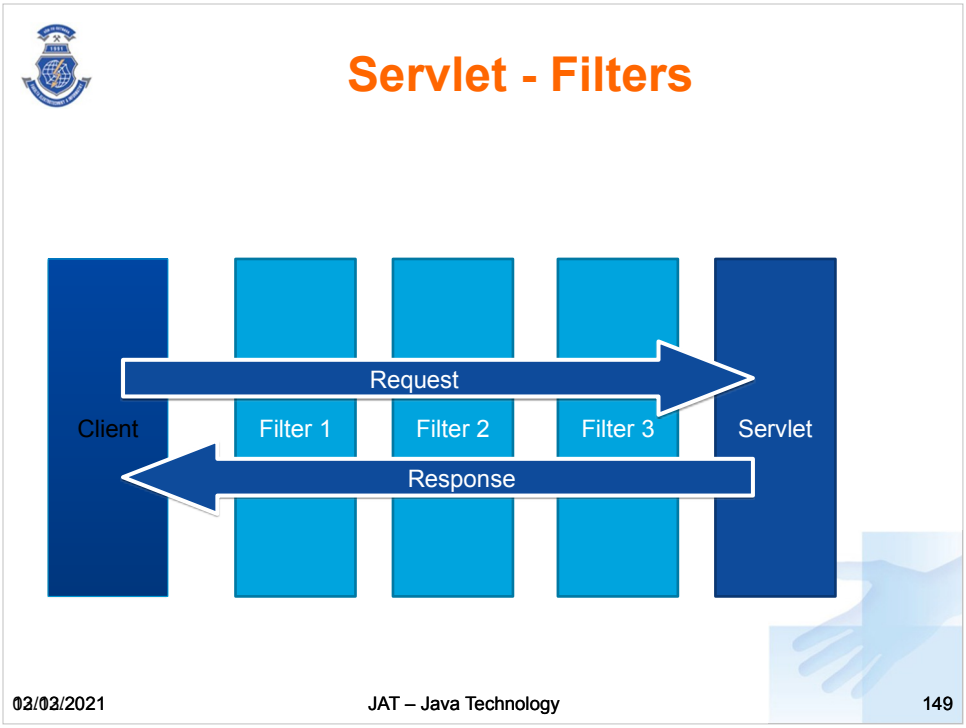
Servlet – Initialization and Destruction

- **Servlet.init()**
  - Web container create object of servlet class (constructor is performed automatically), inject required resources to specified properties and call method **init()**. Method **init()** can initialize other resource like images, database connections and others.
- **Servlet.destroy()**
  - If web container decide based on internal mechanisms that servlet will be destroyed and removed from memory, then call method **destroy()** before remove servlet from memory.

Servlet – Lifecycle Monitoring

- Web container generates events if initialize or destroy servlet context, session and request. Listener of such events have to implements following interfaces:

javax.servlet.ServletContextListener
javax.servlet.ServletContextAttributeListener
javax.servlet.http.HttpSessionListener
javax.servlet.http.HttpSessionActivationListener
javax.servlet.http.HttpSessionAttributeListener
javax.servlet.ServletRequestListener
javax.servlet.ServletRequestAttributeListener



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.

Servlet - Filters

- Filter can change request before and response after servlet processing.
- Each filter have to implements interface **javax.servlet.Filter**
  - Filter method **doFilter()** is most important, because this method performs filtering.
- Interface **javax.servlet.FilterChain** is a parameter of method **Filter.doFilter()** and each filter should call method **FilterChain.doFilter()** to pass control to next filter in chain.

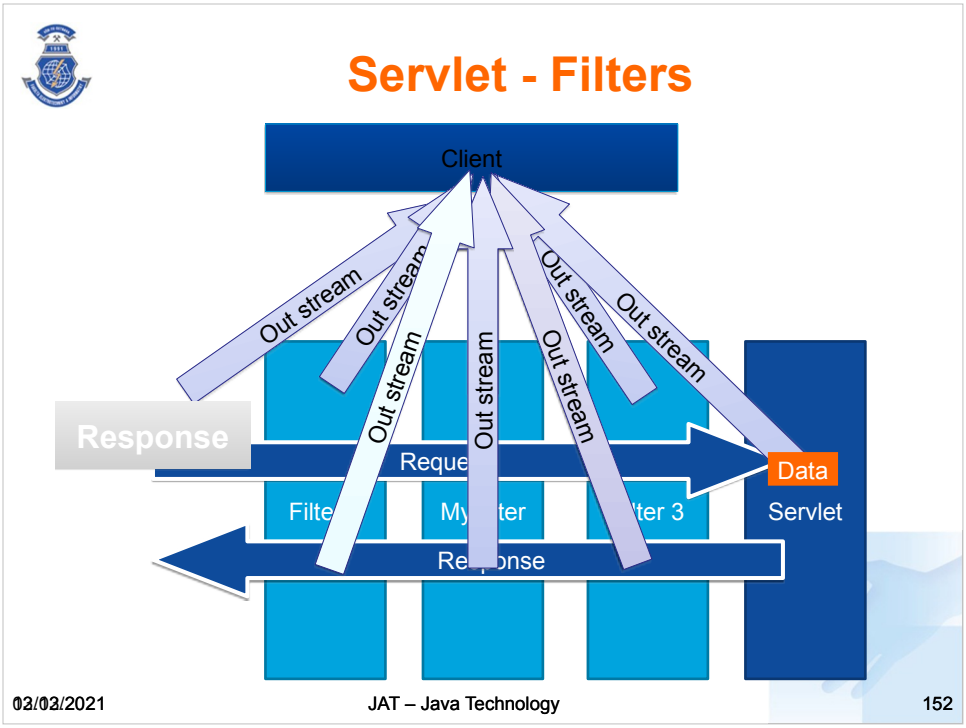
Servlet – Filter Example

```
@WebFilter(filterName="/MyFilter",
urlPatterns={"",""})
public class MyFilter implements Filter {
   public MyFilter() {
   }
   public void doFilter(ServletRequest request,
         ServletResponse response, FilterChain chain)
            throws IOException, ServletException {

      MyWrappedHttpResponse wrapper = new

MyWrappedHttpResponse( (HttpServletResponse)
            response);

      chain.doFilter(request, wrapper);

      response.getWriter().write(wrapper.toString()
         .toUpperCase());
   }
```


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.



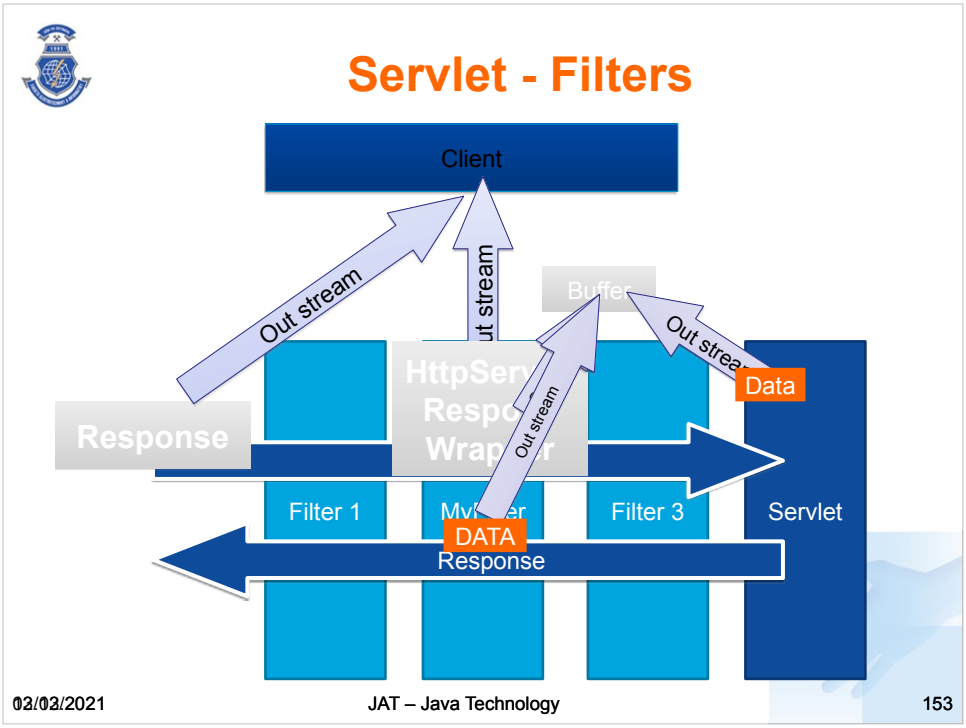
If filter want process data for client from servlet or other filter it need response wrapper.

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.

Servlet - Filter

```
public class MyWrappedHttpResponse extends
            HttpServletResponseWrapper {
    private CharArrayWriter buffer;
    public MyWrappedHttpResponse(
         HttpServletResponse response) {
        super(response);
        buffer = new CharArrayWriter();
    }
    public String toString() {
        return buffer.toString();
    }
    public PrintWriter getWriter() {
        return new PrintWriter(buffer);
    }
}
```


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

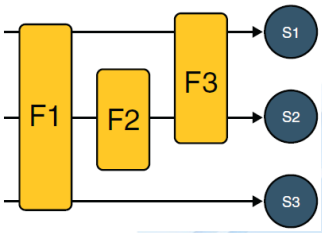
Our class add private field “buffer” of type CharArrayWriter, initialize the filed in constructor and override two methods getWriter() and toString().

Method getWriter() return output stream connected to buffer.

Method toString() return content of buffer as string.

Servlet – Filter Mapping

```
<filter>
   <display-name>MyFilter</display-name>
   <filter-name>MyFilter</filter-name>
   <filter-class>MyFilter</filter-class>
 </filter>
 <filter-mapping>
   <filter-name>MyFilter</filter-name>
   <url-pattern>/*</url-pattern>
   <dispatcher>REQUEST</dispatcher>
   <dispatcher>FORWARD</dispatcher>
   <dispatcher>INCLUDE</dispatcher>
   <dispatcher>ERROR</dispatcher>
</filter-mapping>
```



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.

Servlet - Include

- Implementation of include

```
RequestDispatcher dispatcher =
   getServletContext().getRequestDispatcher("/banner");
if (dispatcher != null){
   dispatcher.include(request, response);
}
```

- Included servlet can write data to response.
- Included servlet cannot change HTTP header setting (encoding, response type, etc.) and cannot create HTTP Cookies.

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.

Servlet – Transfer Control

- Implementation of forward

```
RequestDispatcher dispatcher =
   request.getRequestDispatcher("/another_ser");
if (dispatcher != null)
   dispatcher.forward(request, response);
```

- The request URL is changed to new one.
- Origin request URL is stored to the request attributes with names:
  - javax.servlet.forward.*request-uri*
  - javax.servlet.forward.*context-path*
  - javax.servlet.forward.*servlet-path*
  - javax.servlet.forward.*path-info*
  - javax.servlet.forward.*query-string*
- Non one can write data to response before forward or exception arise during forwarding.

Servlet - Session

- Session is created automatically when method **getSession()** is called.
- Session can store object between client requests. It is realized by methods **getAttribute()** and **setAttribute()**.

```
HttpSession s = request.getSession();
Object o = s.getAttribute("counter");
if(o == null){
   o = new Counter();
   s.setAttribute("counter", o);
}
Counter c = (Counter)o;
```

- Session is terminated when method **Session.invalidate()** is called or if it is not used during timeout period.

Servlet – Session

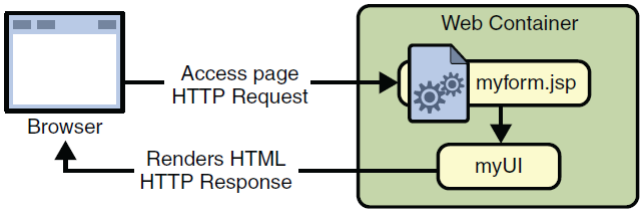
- Session ID is stored in Cookies.
- If cookies are switch off, session ID have to be stored as request parameter.

```
out.println("<p>   <p><strong><a href=\"" +
   response.encodeURL(request.getContextPath() +
   "/bookcatalog") +
   "\">ContinueShopping </a>");
```

- Special method HttpServletResponse.encodeURL() exist to add session ID as parameter to encoded URL. The method determine if cookies are switch off and if yes add parameter with session ID to encoded URL

JSF – Java Server Faces

- JSF – component technology for creating server-side user interface of web applications.
- Main configuration file: faces-config.xml



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.

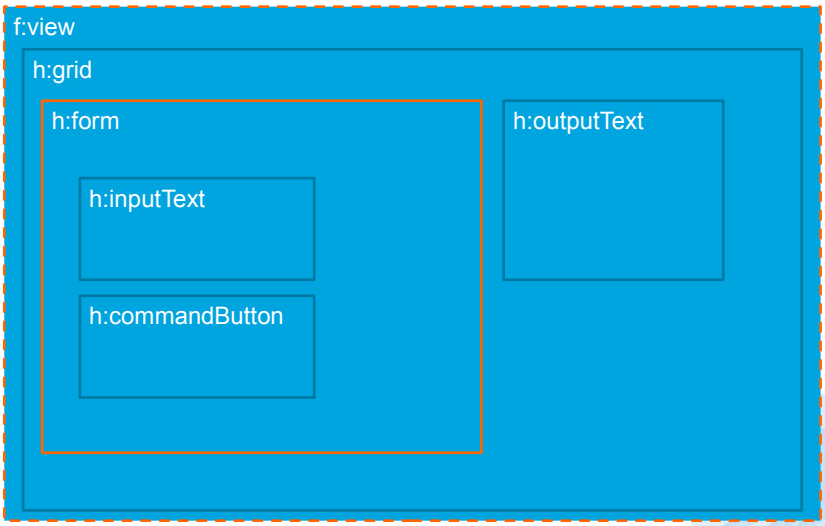
JSF – FacesServlet mapping

Programmer has to add following lines to web.xml to use JSF

```
 <servlet>
    <servlet-name>Faces Servlet</servlet-name>
    <servlet-
class>javax.faces.webapp.FacesServlet</servlet
-class>
    <load-on-startup>1</load-on-startup>
  </servlet>
  <servlet-mapping>
    <servlet-name>Faces Servlet</servlet-name>
    <url-pattern>/guess/*</url-pattern>
  </servlet-mapping>
```


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 – page structure



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.

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
    <html xmlns="http://www.w3.org/1999/xhtml"
        xmlns:f="http://java.sun.com/jsf/core">
        <h:head>
            <meta http-equiv="Content-Type" content="text/html;
charset=UTF-8" />
            <title>Facelet Title</title>
        </h:head>
        <h:body>
            <h:panelGrid border="1" columns="2">
                <h:form>
                    <h:inputText
value="#{myBean.name}"></h:inputText>
                    <h:commandButton type="submit" value="Send"
                        action="ok"></h:commandButton>
                </h:form>
                <h:outputText value="#{myBean.age}"></h:outputText>
            </h:panelGrid>
        </h:body>
</html>
```


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 - Namespaces

```
<html xmlns="http://www.w3.org/1999/xhtml"
  xmlns:f="http://java.sun.com/jsf/core
  xmlns:h="http://java.sun.com/jsf/html"

xmlns:ui="http://java.sun.com/jsf/facelets
"
  >
```


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.

JSF – GUI - View and Form

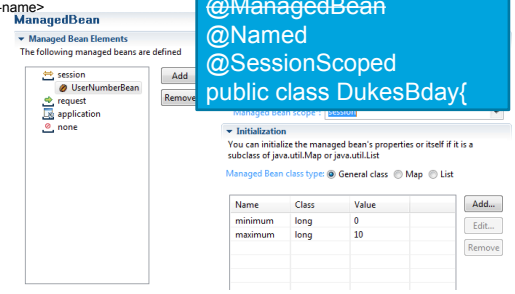
- JSF components are organized in components tree. The structure of component tree corresponds with structure of tags.
- All JSF components have to be inside component view.
  - From version JavaEE 6 component view is not required. Whole page automatically represent one view component.
- All JSF components that handle inputs form user have to by inside component form

```
<f:view>
   <h:form id="helloForm1">
   </h:form>
</f:view>
```

JSF – Managed Bean

- JavaBeans which creation and state are managed by JavaEE container. Can be defined in configuration file (faces-config.xml) or by annotations.

```
<managed-bean>
<managed-bean-name>UserNumberBean</managed-bean-name>
<managed-bean-class>
guessNumber.UserNumberBean
</managed-bean-class>
<managed-bean-scope>session</managed-bean-scope>
<managed-property>
<property-name>minimum</property-name>
<property-class>long</property-class>
<value>0</value>
</managed-property>
<managed-property>
<property-name>maximum</property-name>
<property-class>long</property-class>
<value>10</value>
</managed-property>
</managed-bean>
```




```
~~@ManagedBean~~
@Named
@SessionScoped
public class DukesBday{
```

JSF – Managed Bean (CDI)

- Class UserBean is JavaBean (has public constructor without parameters)
- Name of instance ManagedBean is taken from anotation, if not specified is derived from name of class with small letter at begining
- Class UserBean – name ManagedBean userBean
- Properties:
- name, logged (read only), users (read only)
- Methods for buttons: logout() (return String)

```
import
javax.enterprise.context.SessionScoped;
import javax.inject.Named;
@Named("userBean")
@SessionScoped
public class UserBean {
private String name;private String passwd;private User
loggedUser;private User edit;
private ArrayList<User> users = new ArrayList<User>();
public UserBean() {users.add(new User(1, "admin", "admin"));}
public String getName() {return
name;}
public void setName(String name) {this.name
= name;}
public ArrayList<User> getUsers() {return
users;}
public boolean isLogged(){return loggedUser != null;}
public String logout(){loggedUser = null;return "";}
```

```
<h:panelGroup rendered="#{userBean.logged}">
  <h:commandButton id="c5" value="Logout"
    action="#{userBean.logout()}" />
</h:panelGroup>
```

JSF – Managed Bean

- Class UserBean is JavaBean (has public without parameters)
- Name of instance ManagedBean is taken from anotation, if not specified is derived from name of class with small letter at begining
- Class UserBean – name ManagedBean userBean
- Properties:
  - name, logged (read only), users (read only)
- Methods for buttons: logout() (return String)

```
import javax.faces.bean.ManagedBean;
import javax.faces.bean.SessionScoped;
@ManagedBean(name="userBean")
@SessionScoped
public class UserBean {
private String name;private String passwd;private User loggedUser;private User edit;
private ArrayList<User> users = new ArrayList<User>();
public UserBean() {users.add(new User(1, "admin", "admin"));}
public String getName() {return name;}
public void setName(String name) {this.name = name;}
public ArrayList<User> getUsers() {return users;}
public boolean isLogged(){return loggedUser != null;}
public String logout(){loggedUser = null;return "";}
```

```
<h:panelGroup rendered="#{userBean.logged}">
  <h:commandButton id="c5" value="Logout"
    action="#{userBean.logout()}" />
</h:panelGroup>
```

JSF – Managed Bean

- Application (@ApplicationScoped)
- Session (@SessionScoped)
- **View (@ViewScoped)**
- Request (@RequestScoped)
- None (@NoneScoped)
- Custom (@CustomScoped)

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.

EL – Unified Expression Language

- Immediate expressions
  - `${expression}`
  - Example: `${sessionScope.cart.total}`
- Deferred expressions
  - `#{expression}`
  - Example:

```
<h:inputText id="name"
value="#{customer.name}"/>
```

- Expressions can be used to generate dynamic content in text document (often HTML – JSP technology)
- Can be used as parameters of tags (JSF technology)

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.

EL – Value Expression

- Expressions can access to:
  - JavaBeans, collections, enum types, implicit objects
- Expression can access JavaBean properties in two ways
  - ${customer.name}
  - ${customer["name"]}
- Both access methods can be combined
  - ${customer.address["street"]}
-

EL – Value Expression

- Expression ${customer} process search of properties with name “customer” in context of page, request, session and application.

- If program define enum type Animals and property myAnimal of type Animals

```
public enum Animals{dog, cat, fish, bird}
```

- Expression can use construction like this:

```
${ myAnimal == "dog"}
```

EL – Value Expression

- Expression can access to collections
  - Expression access to any element

```
${customer.orders[1]}
```

  - Expression access to first element

```
${customer.orders.orderNumber}
```

- Expressions can access to maps

```
${customer.favourite["computers"]}
```

- Expressions allow use constants

```
${"text"}
${customer.age + 20}
${true}
${57.5}
```

EL – Method Expression

- Expression can call methods
  - No paramaters

```
<h:inputText id="name"
value="#{customer.name}"
validator="#{customer.validateName()}"/>
```

  - With parameters

```
<h:inputText
value="#{userNumberBean.userNumber('5')}">
```

EL – Operators

- **Arithmetic:** +, - (binary), *, / and div, % and mod, - (unary)
- **Logical:** and, &&, or, ||, not, !
- **Relational:** ==, eq, !
  =, ne, <, lt, >, gt, <=, ge, >=, le
- **Empty:** The `empty` operator is a prefix operation that can be used to determine whether a value is null or empty.
- **Conditional:** A ? B : C

JSF – GUI Components

- `<h:outputText lang="en_US" value="#{UserNumberBean.minimum}"/>`
- `<h:graphicImage id="waveImg" url="/wave.med.gif" />`
- `<h:inputText id="userNo" label="User Number" value="#{UserNumberBean.userNumber}"></h:inputText>`

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.

JSF - GUI Components

```
<h:panelGroup style="border-bottom-style: double; border-top-style: double; border-left-style: double; border-right-style: double">

<h:outputLink value="somePage.xhtml">

</h:panelGroup>
```


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.

JSF - GUI Components

```
<h:panelGrid border="1" columns="2">
  <h:outputText value="item1"></h:outputText>
  <h:outputText value="item2"></h:outputText>
  <h:outputText value="item3"></h:outputText>
  <h:outputText value="item4"></h:outputText>
</h:panelGrid>
```


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 - GUI Components

```
<h:commandButton value="Send"
outcome="success"></h:commandButton>

<h:commandLink>
  <h:outputText value="CommandLink">
  </h:outputText>
</h:commandLink>
```


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.

JSF – GUI - Table

```
<h:dataTable var="p"
value="#{personAgenda.allPositions}">
<h:column>
<h:outputText value="#{p.description}" />
</h:column>
<h:column>
<h:commandButton value="Edit"
action="#{personAgenda.editPosition(p)}"
/>
</h:column>
</h:dataTable>
```


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.

JSF – GUI - Table

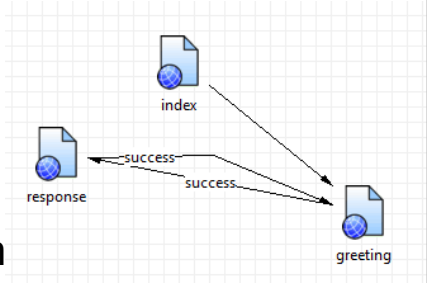
```
<h:dataTable var="p"
value="#{personAgenda.allPositions}"
binding="#{personAgenda.positionTable}">
<h:column>
  <f:facet name="header">
    <h:outputText value="Name"/>
  </f:facet>
  <h:outputText value="#{p.description}" />
</h:column>
<h:column>
  <h:commandButton value="Edit"
    action="#{personAgenda.editPosition}" />
</h:column>
</h:dataTable>
```

JSF - buttons

```
<h:button value="Send"
outcome="success"></h:commandButto
n>

<h:commandLink
action="#{myBean.doSomething()}">
  <h:outputText
value="CommandLink">
  </h:outputText>
</h:commandLink>
```

JSF – Navigation Model

- Navigation through rules and action



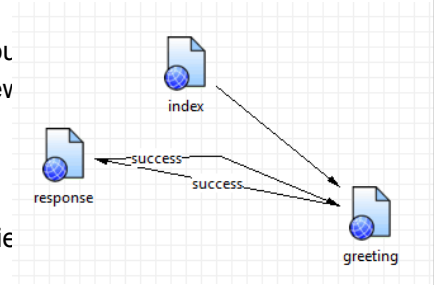
- Method of BackingBean
  - <h:commandButton id="submit" action="#{userNumberBean.deliveryOrder()}" value="Submit" />

Method must return String and usually has no parameters

Method expresion

JSF - navigation

```
<navigation-rule>
    <from-view-id>/greeting.jsp</from-view-id>
    <navigation-case>
        <from-outcome>success</from-ou
        <to-view-id>/response.jsp</to-view
    </navigation-case>
</navigation-rule>
<navigation-rule>
    <from-view-id>/response.jsp</from-vie
    <navigation-case>
        <from-outcome>success</from-outcome>
        <to-view-id>/greeting.jsp</to-view-id>
    </navigation-case>
</navigation-rule>
```



JSF – navigation rules

```
<navigation-rule>
  <from-view-id>*</from-view-id>
  <navigation-case>
   <from-outcome>editPosition</from-outcome>
   <to-view-id>/editPosition.xhtml</to-view-id>
  </navigation-case>
  <navigation-case>
   <from-outcome>logout</from-outcome>
   <to-view-id>/Logout</to-view-id>
   <redirect/>
  </navigation-case>
<navigation-rule>
  <from-view-id>/login.xhtml</from-view-id>
  <navigation-case>
   <to-view-id>/index.xhtml</to-view-id>
   <redirect/>
  </navigation-case>
```

Code to get ManagedBean

- Using DI:

```
@ManagedProperty("#{personMB}")
private PersonMB personMB;
```

- Resolving EL expresion:

```
StarShipDB controller = (StarShipDB)
arg0.getApplication().getELResolver().
      getValue(arg0.getELContext(), null, "starShipDB");
```

- Using JNDI (Session Bean):

```
try {
   starShipSB = InitialContext.doLookup(
   "java:/global/jat2017cv11v1/StarShipSB");
   System.out.println("starshipSB in converter class: " +
starShipSB.getClass().getCanonicalName());
} catch (NamingException e) {e.printStackTrace();}
```

JSF – Facelets - Template

```
<h:body>
<div id="top" class="top">
  <ui:insert name="top">Top Section</ui:insert>
</div>
<div>
  <div id="left">
   <ui:insert name="left">Left Section</ui:insert>
  </div>
  <div id="content" class="left_content">
    <ui:insert name="content">Main
Content</ui:insert>
  </div>
</div>
</h:body>
```


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.

JSF – Facelets – Usage of Template

```
<html xmlns=…
<h:body>
<ui:composition template="./template.xhtml">
<ui:define name="top">
    Welcome to Template Client Page
</ui:define>
<ui:define name="left">
    <h:outputLabel value="You are in the Left Section"/>
</ui:define>
<ui:define name="content">
    <h:graphicImage value="#{resource['images:wave.med.gif']}"/>
    <h:outputText value="You are in the Main Content Section"/>
</ui:define>
</ui:composition>
</h:body>
</html>
```

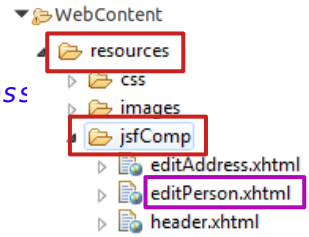

Any page can use template if use component ui:composition. The page should contains components ui:define with name that corresponded with name of components ui:insert form used template.

JSF – composite components

```
<?xml version='1.0' encoding='UTF-8' ?>
<!DOCTYPE html …>
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:cc="http://java.sun.com/jsf/composite"
      xmlns:h="http://java.sun.com/jsf/html">
  <cc:interface>
    <cc:attribute name="addressObject"
required="true"/>
  </cc:interface>
  <cc:implementation>
    <h:inputText
value="#{cc.attrs.addressObject.street}"/>
    <h:inputText
value="#{cc.attrs.addressObject.city}"/>
  </cc:implementation>
</html>
```

JSF – composite components

```
<?xml version='1.0' encoding='UTF-8' ?>
<!DOCTYPE composition …>
<ui:composition
xmlns:ui="http://java.sun.com/jsf/facelets"
   xmlns:h="http://java.sun.com/jsf/html"
   xmlns:f="http://java.sun.com/jsf/core"
   xmlns:my="http://java.sun.com/jsf/composite/jsfComp"
   template="/template_secured.xhtml">
 <ui:define name="content">
  <h:form>
<my:editPerson addressObject=
"#{personAgendaMB.editedCustomer.address
/>
  </h:form>
 </ui:define>
</ui:composition>
```



JSF – Conversion Model

- Component connection to server-site JavaBeans needs data type conversion
  - Model view: data are represented in java data types (int, long, java.util.Date, …)
  - Presentation view: data represented in human readable form in HTML (text)
- Example:
  - Java.util.Date vs. „28.9.2009“
- Implementation of users converters – interface `javax.faces.convert.Converter`

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 - conversion

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

```
@ApplicationScope
@FacesConverter(value = "shipConverter", managed = true)
public class StarShipConverter implements Converter<StarShip>{
....
```

JSF - conversion

```
import javax.enterprise.context.ApplicationScoped;
import javax.faces.annotation.FacesConfig;

@ApplicationScoped
@FacesConfig
public class Jsf23Activator {
}

@ApplicationScoped
@FacesConverter(value = "shipConverter", managed = true)
public class StarShipConverter implements Converter<StarShip>{

@Inject
private StarShipDB starShipDB;

@Override
public StarShip getAsObject(FacesContext context, UIComponent component,
String value) {
}

@Override
public String getAsString(FacesContext context, UIComponent component,
StarShip value) {
}
```

JSF – Conversion Model

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

Expresion have to return object which implements interface **Converter**

```
<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
```

JSF - Conversion Model

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

</h:inputText>
```

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

JSF – Validation Model

- Validation is performed before data are set into properties connected with component through EL expression.
- Validation is performed after data is converted into java data type coresponding with bean property
- Set of standard validtors:
  - validateDoubleRange, validateLength, validateLongRange, validateRegex
- Method of BackingBean

```
public void validate(FacesContext context, UIComponent
component, Object value) throws ValidatorException
```

- Implementation of interface `javax.faces.validator.Validator`

JSF - validations

- ```
  <h:inputText id="userNo" label="User Number"
  value="#{userNumberBean.userNumber}"
  validatorMessage="message"><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 - validations

- ```
  <h:inputText id="userNo" label="User Number"
  value="#{userNumberBean.userNumber}"
  validatorMessage="message">
  <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 - validations

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

  </h:inputText>
  ```

Reference to **method**

```
@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 – in code evaluation of EL expresion

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


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

JSF – custom validation messages of standard validators

```
<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=In
valid 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 - localization

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

File:
jat/messages.properties

userNoConvert=The value you entered is not a number.

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

JSF - localization

```
<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 – localization based on language of web browser

```
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 component model

- Coplet set of UI component
- Extensibility
- Base class UIComponentBase
  - UIColumn, UICommand, UIData, UIForm, UIGraphic, UIInput, UIMessage, UIMessages, UIOutput, UIPanel, UIParameter, UISelectBoolean, UISelectItem, UISelectItems, UISelectMany, UISelectOne, UIViewRoot
- Behavioral interfaces
  - ActionSource, ActionSource2, EditableValueHolder, NamingContainer, StateHolder, ValueHolder

JSF – Model of component rendering

- Separation of component beghavior from rendering
- One component can be represented by several diferend TAGs
- UISelectOne
  - Radio buttons
  - Combo box
  - List box

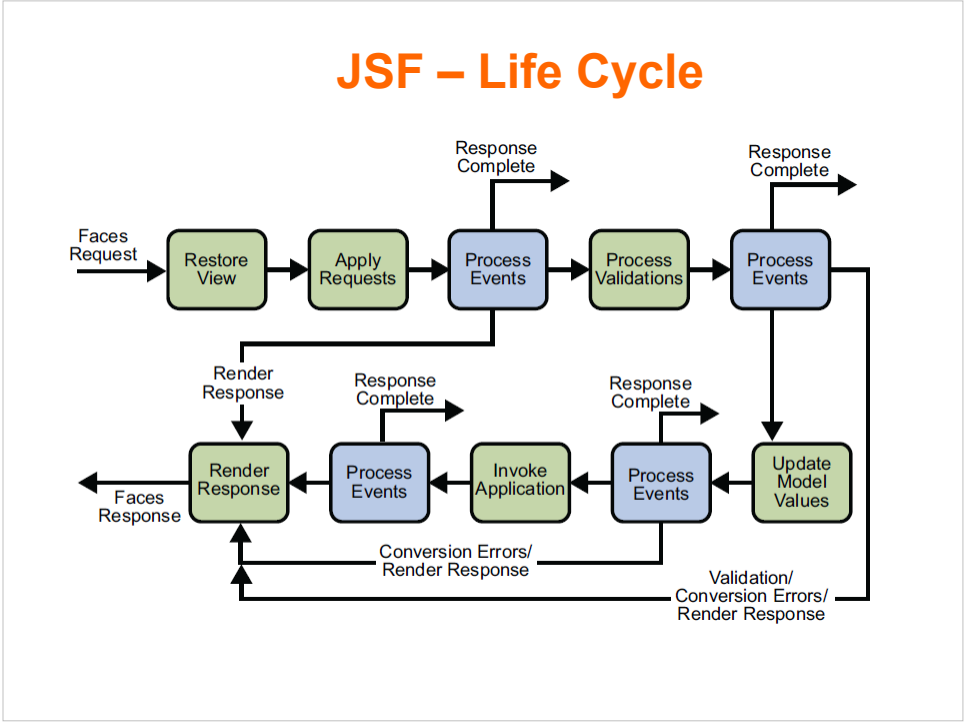
JSF – Event – Listener model

- Like Evenl-Lisener model from JavaBeans
- 3 type of events: value-change events, action events, data-model events
  - Implementation of method in BackingBean and usage of EL expression for method reference in atribute of component TAG.
  - Implementation of listener

Class which implement interface **listener**

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

Expresion have to return object which implements interface of **listener**



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 FacesContext, 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.

P6

- EJB - Enterprise JavaBeans

Enterprise Java Beans (EJB)

- Specification of architecture for development and deployment of distributed transactional object component on server side
- Conventions + set of interfaces(EJB API)
- Target = ensure compatibility between products from diferent suppliers
  - components
  - Container
- EJB 3.0

Enterprise JavaBeans

- EnterpriseBean are components implementing technology Enterprise JavaBeans (EJB)
- EnterpriseBean runs inside EJB container or web container
- EnterpriseBean is server side component encapsulating business logic
- EnterpriseBean can be invoked remotely

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

EJB container

- Environment for component
  - Remote access
  - security
  - transaction
  - Parallel access
  - Access to resources and their sharing
- Isolation of component from application
  - Independent of container producer
  - Development of application is easier

When toUse Enterprise Beans

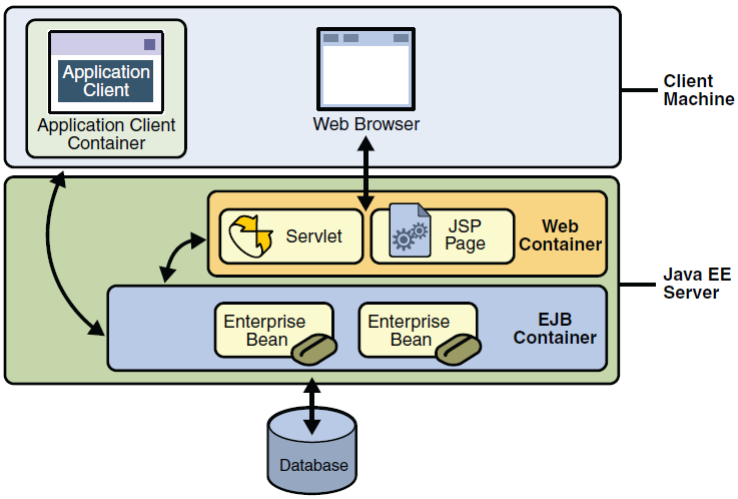
You should consider using enterprise beans if your application has any of the following requirements:

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 – usage


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

Types of EJB components

- Session Bean:
  - Stateless session bean
  - Statefull session bean
  - Singelton
- Message-Driven Beans
  - Stateless service which can be call asynchronously

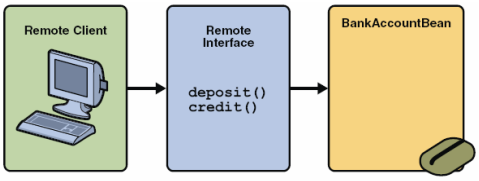
What Is a Session Bean?
A *session bean represents a single client inside the Application Server. To access an application* that is deployed on the server, the client invokes the session bean's methods. The session bean performs work for its client, shielding the client from complexity by executing business 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 - usage

- To concrete instance access only one client at time
- Sate is not persistent, onlz short time (hours)
- Web services
- Statefull
  - Interaction between SB and client, hold information between calls SB
- Stateless
  - No information are stored for concrete client
  - General tasks

SessionBean - Interface

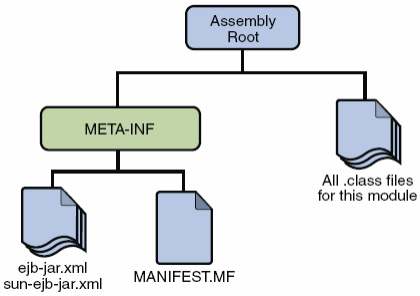
- Clienat access using interface (business interface)
- One bean can have more then one business interfaces




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

SessionBean - implementation

- Interface: `interface NameOfInterface`
- Class of component
  `class NameOfInterfaceBean implements NameOfInterface`
- Support classes



SessionBean – Remote vs. Local

Remote client

- Can run on remote JVM (other JVM)
- Client can be:
  - Web component
  - Application
  - Another EJB
- Big isolation of method's parameters
  - Client and bean work with diferent copies of objects
  - Beter security
- Data granularity

Local client

- Must run in same JVM
- Client can be:
  - Web component
  - Another EJB
- Weak isolation
  - Client and bean work with same object
  - Change done in bean has efect in client
  - Lower security

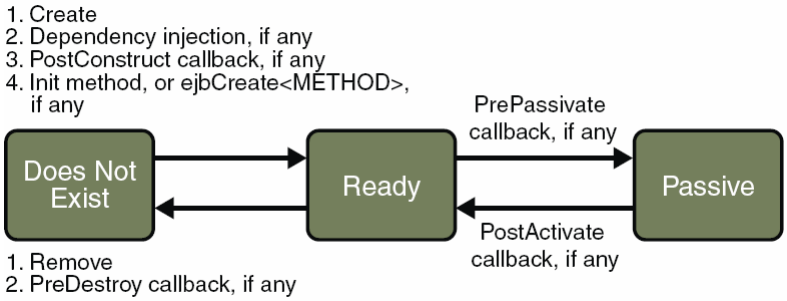
Deciding on Remote or Local Access
Whether to allow local or remote access depends on the following factors.
Tight or loose coupling of related beans: Tightly coupled beans depend on one another.
For example, if a session bean that processes sales orders calls a session bean that emails a confirmation 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.

Statefull SessionBean – life cycle

@PostConstruct
@PreDestroy
@Remove – client call method with this annotation to mark conversation as finished

@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.

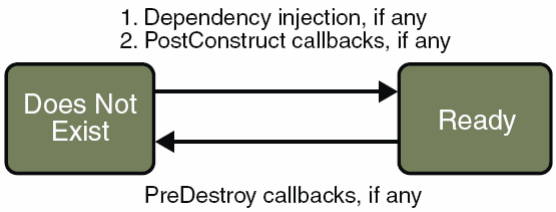
While in the ready stage, the EJB container may decide to deactivate, or *passivate, the bean by* moving it from memory to secondary storage. (Typically, the EJB container uses a least-recently-used algorithm to select a bean for passivation.) The EJB container invokes the method annotated @PrePassivate, 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 – life cycle

@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 – server implementation

```
@Remote
public interface Account {
   public void deposite(String accNum, float amounght);
   public void remove(String accNum, float amounght);
}
```

<projectName>EAR/<BeanName>Bean/remote

```
@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) {
   }
}
```

SessionBean – interfaces

```
@Remote

@Local

@LocalBean
```

SessionBean – server implementation

```
@Remote
public interface Account {
   public void deposite(String accNum, float amounght);
   public void remove(String accNum, float amounght);
}
```

<projectName>EAR/<BeanName>Bean/remote

```
@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) {
   }
}
```

SessionBean – client implementation

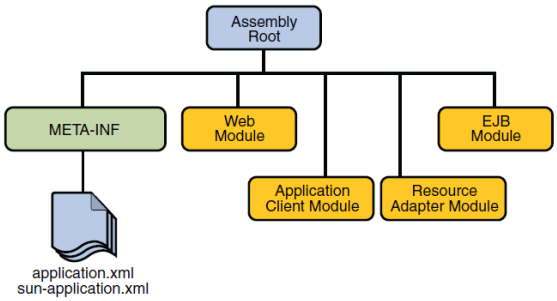
- Client have to have access to Business interface (for example as *.jar)
- Client use libraries from JavaEE
- Types of clients
  - Application Client Container
  - Normal application
  - Web client (war)

SessionBean – client implementation of normal application

```
public class Main {

   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("Amount:" +
                a.deposite("1234/0300", 10.25f));
   }
```

SessionBean – client implementation ACC

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



SessionBean – client implementation web application - 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

Session Bean - Injection

- Resource injection can be done only for objects where for their instantiation is in responsibility of JavaEE container
  - Servlets
  - ServletContextListener
  - Managed Backing Beans in JSF

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".

Session Bean - JSF

- Can be easily initialized using dependency injection in Managed Bean
- Managed Bean call method of EJB
- Old version of JSF has no direct support for session invalidation

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

Stateful SessionBean

- Bean hold state (values of instance variables) between individual calls
- Bean has only one client (that guarantee). The lifetime of bean is same as lifetime of variable with annotation @EJB

Stateful Session Bean - JSF

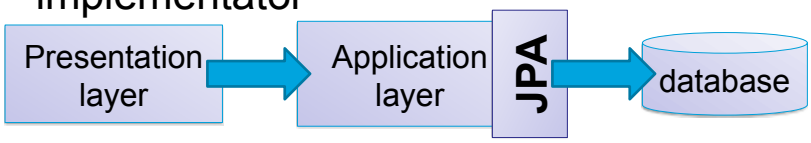
```
public String logout()
{
  FacesContext.getCurrentInstance().getExternalContext().
    invalidateSession();
  return "/home.xhtml?faces-redirect=true";
}

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

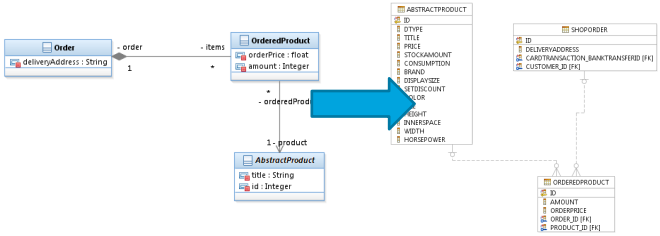
P7

- Java Persistence API
  - Language 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 for persistance using object-relation mapping
- Only interface, it is necessary add implementator





JPA - Entity

- **Entity** – it is light-weight object from persistence domain. Typically are connected with database table.
  - Eache object is related to one record in database table.
- **Persistent state of entity** is reprezented by instance variables and class properties.
  - Mapping between database and properties is defined by annotations.

JPA – Entity class

- Class have to has annotation `javax.persistence.Entity`
- Class have to has public or protected constructor with no parameter (can have another constructors)
- Class and methods and instance variables cannot be declared as `final`

JPA – Entity class

- If is entity used in remote EJB interface have to implemented interface `Serializable`
- Entity class can be descendant of entity class or non-entity class. Non-entity classes can by descendat of entity class.
- Persistance instance variables have to be declared as private, protected or package-private. They should be accessed through set and get methods.

JPA – Entity class - example

```
@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 – persistence properties, instance variables

- Instance variables – persistence provider access directly to them
- Properties – Persistence access properties using get, set method
  - Can be used: Collection, Set, List, Map even generic versions
- **`hashcode() equals()`**
- Types: Java primitive data types
  - java.lang.String, other serializable types (classes represented primitive data tzpes, java.math.BigInteger, java.math.BigDecimal, java.util.Date, java.util.Calendar, java.sql.Date, java.sql.Time, java.sql.TimeStamp, user serialzable types, byte[], Byte[], char[], Character[], enum types, other entities, collections of entities

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

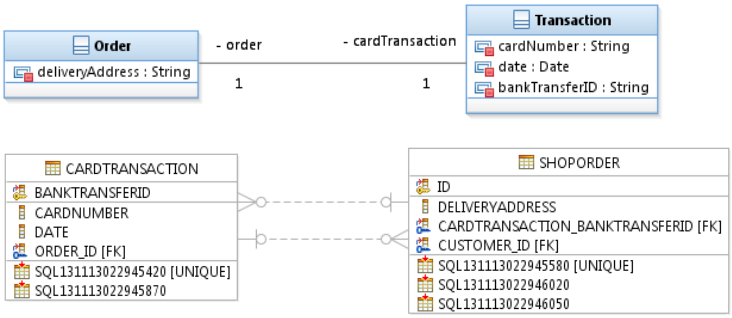
JPA – primary keys

- Each entity have to have owen primary key.
- javax.persistence.Id
- Composite Primary Key
  - Have to exisit class which define composite key
  - javax.persistence.EmbeddedId
  - javax.persistence.IdClass
  - Have to be composed from types:
    - Java Primitive data types (and coresponding embedded classes)
    - java.lang.String
    - java.util.Date (DATE), java.sql.Date

JPA – relation 1-1

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

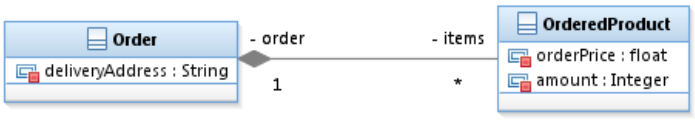
```
@Entity
public class Transaction {
@OneToOne
private Order order;
…
```

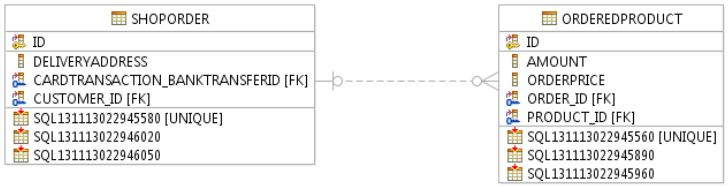


JPA – relation 1-N

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

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

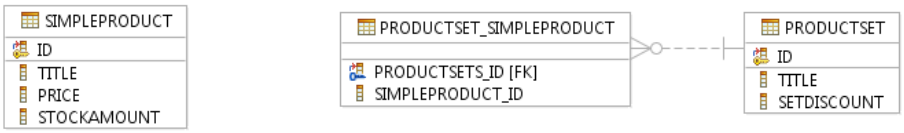




JPA – relation M-N

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

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



JPA – inheritance

- **Entity can be extended from non entity calss**
- **Entita – can be extend from abstract class**

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

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

JPA – inheritance mapping strategy

- One table peer class hierarchy
- One table for non-abstract entity
- Join strategy

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

JPA – inheritance mapping strategy

One table peer class hierarchy

@Inheritance(strategy=SINGLE_TABLE)

@DiscriminatorColumn(

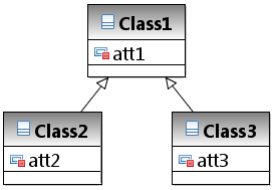
String name

DiscriminatorType discriminatorType

String columnDefinition

String length)


```
public enum DiscriminatorType {
    STRING,
    CHAR,
    INTEGER
};
```

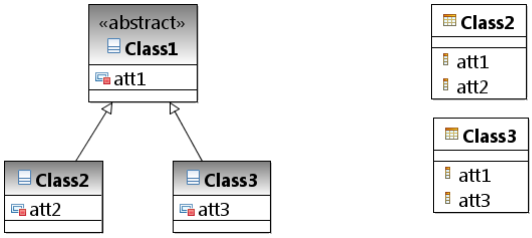


@DiscriminatorValue

JPA – inheritance mapping strategy

One table for non-abstract entity

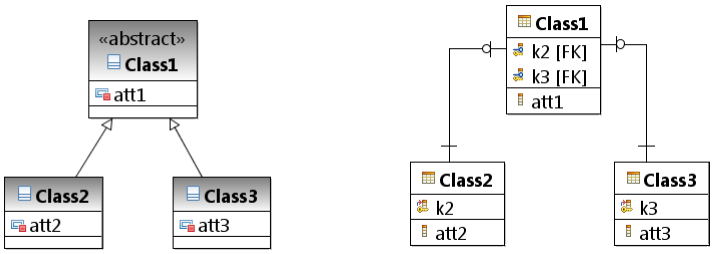
@Inheritance(strategy=TABLE_PER_CLASS)



JPA – inheritance mapping strategy

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=GenerationType.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 – entity manager

- **Persistence context**: set of entities which exisit in concrete data storage
- EntityManager
  - Create, delete entities, find entities, execute queries
- Container managed entity manager

```
@PersistenceContext
EntityManager em;
```

JPA – find entities

```
@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 – entity life cycle

- 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 - queries

```
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 – named queries

Class annotation - 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 – parameters in queries

- Named

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

- Numbered

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

JPA – Persistence Units

- Package contains all entiy classes mapped into one database storage (DB).
- Have to contains file persistence.xml
- Can be part of 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 – queries - examples

- 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 – queries - examples

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

JPA – query - LIKE

- SELECT p FROM Player p WHERE p.name LIKE ’Mich%’
- _ - pattern matches exactly one character
- % - pattern can match zero or more characters
- ESCAPE – can define another escape character
  - LIKE ’_%’ ESCAPE ’\’
- NOT LIKE

JPA – gueries – 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

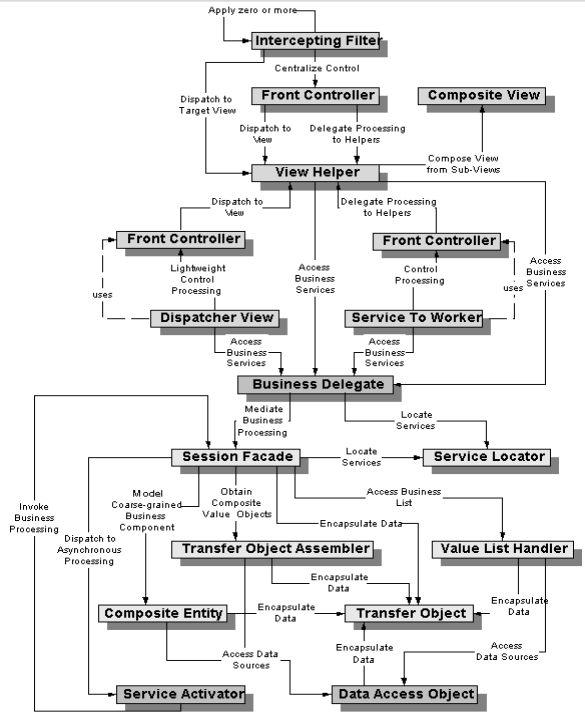
JPA – queries between, in

- 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’)

P8

- Design patterns JavaEE
  - DAO
- http://java.sun.com/blueprints/corej2eepatterns/Patterns/

Design patterns JavaEE



DAO – Data Access Object

Problem

- Different types of data storage need different access methods.
- Change data another storage lead to big code rework.

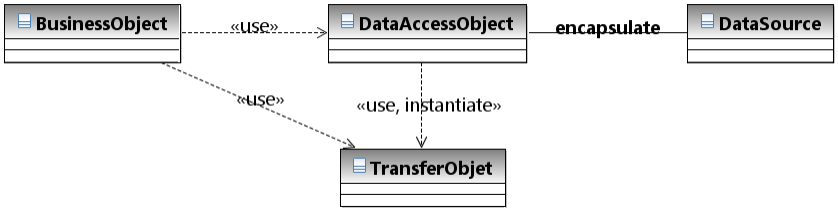
Forces

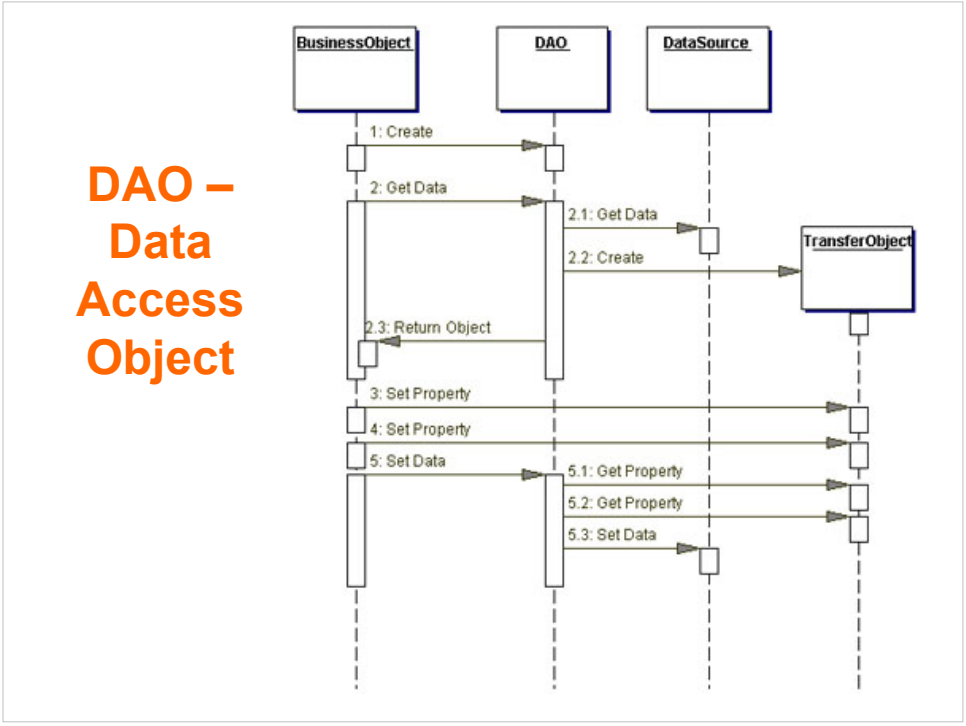
- Components need store data
- Access to each storage is different
- Components usualy use proprietar API to access storage
- Reduced portability of components
- Components should be transparents to storage implementation and should allow simple migration

DAO – Data Access Object

Solution

- Usage of DAO object for encapsulation all accesses to storage. DAO take over responsibility for connection to storage and store or retrive data.
- DAO provides simple and storage-independent interface.





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

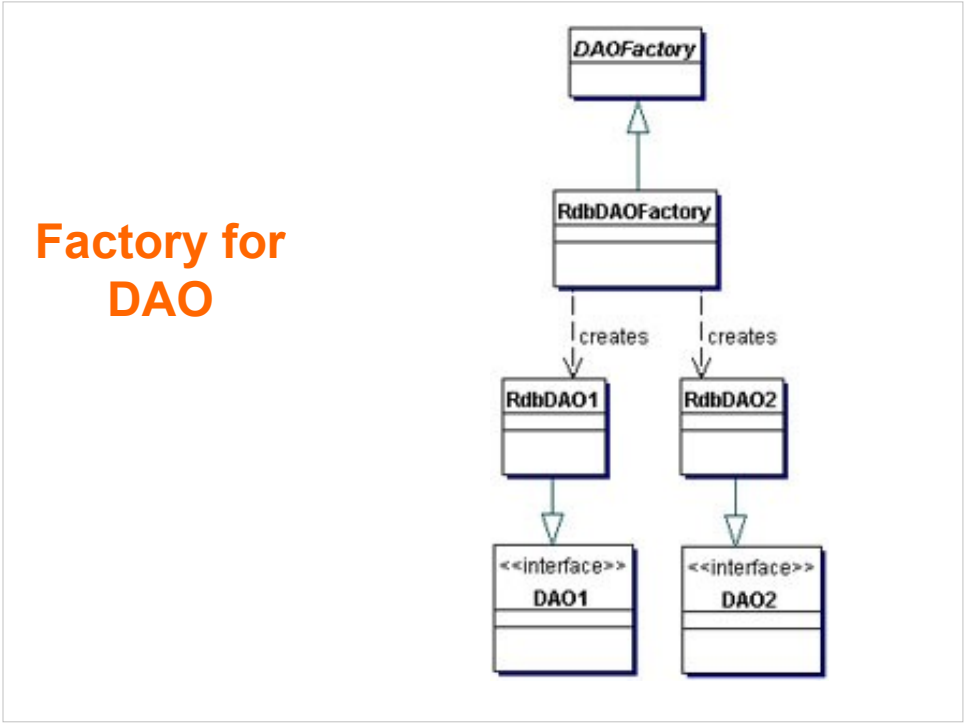
The DataAccessObject is the primary object of this pattern. The DataAccessObject abstracts the underlying data access implementation for the BusinessObject to enable transparent access to the data source. The BusinessObject also delegates data load and store operations to the DataAccessObject.

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.



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

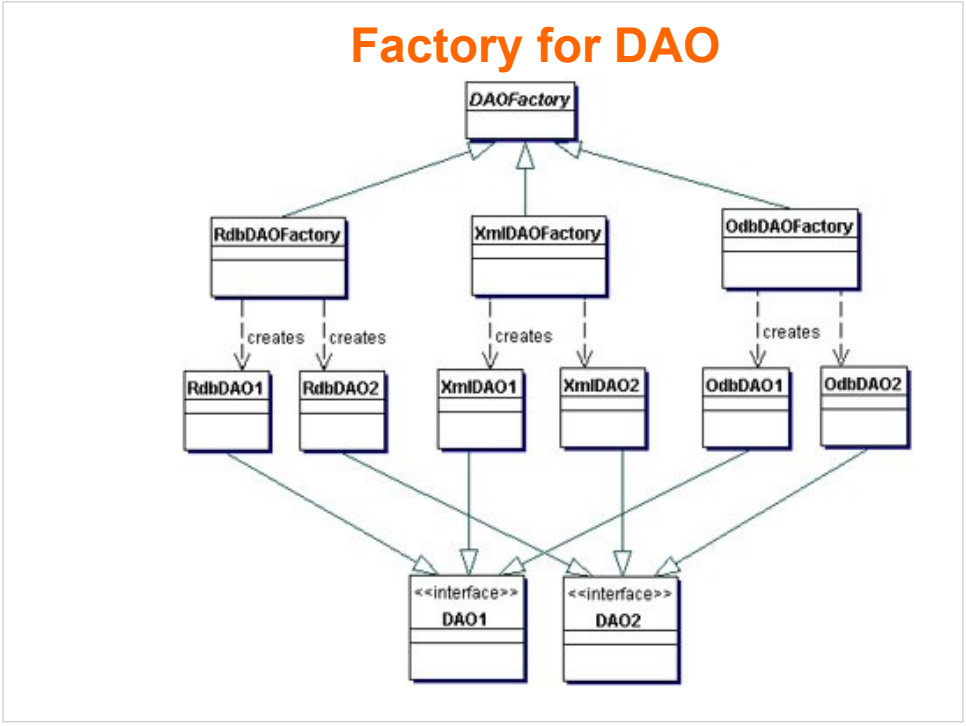
The DataAccessObject is the primary object of this pattern. The DataAccessObject abstracts the underlying data access implementation for the BusinessObject to enable transparent access to the data source. The BusinessObject also delegates data load and store operations to the DataAccessObject.

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.



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

- Web services

What are web services

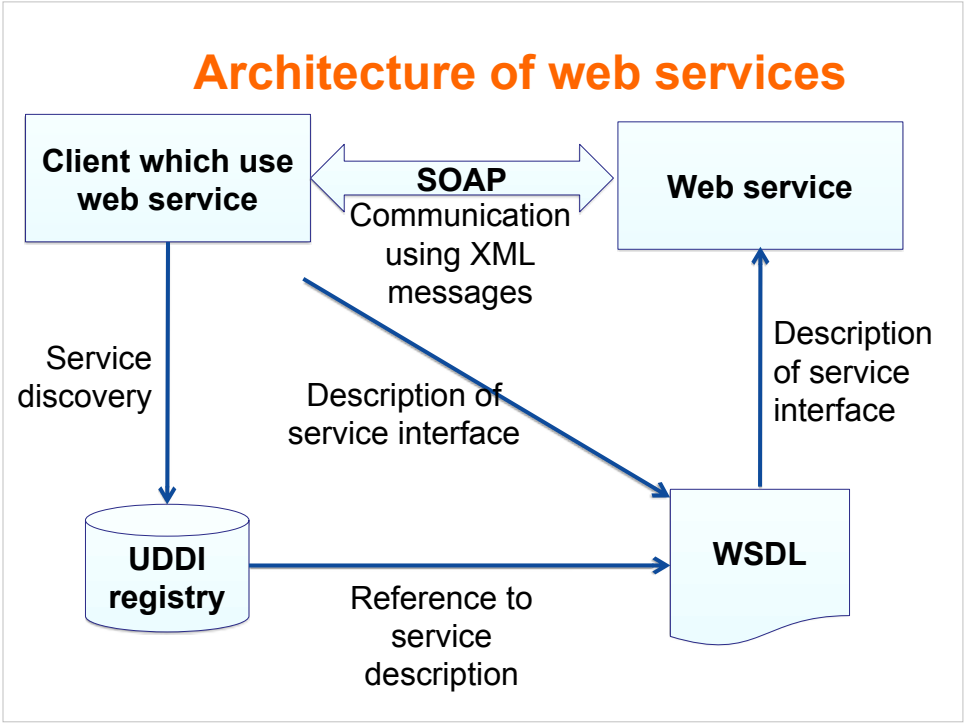
- Interface to application which is accesible throught computer network, based on standard internet technologies.
- Generally: if is application accessibel through network by protocols like HTTP, XML, SMTP or Jabber, it is web service.
- Layer between server side application program and program on client side.

What are web services

- Functionality of service is not dependent on programming language of client or server (Java, C++, PHP, C#, ...).
- Example: HTML pages:
  - server=WWW server, client=browser
- Nowadays, we do not understand web services like this in general, a web service is a set of specific specifications from W3C.
- Available services: exchange rates, stock exchange, search services (Google), maps, weather.
- Components of distributed application?

Architecture of web services

- Set of protocols, http://www.w3.org/2002/ws/:
  - Message transfer – SOAP,
    - http://www.w3.org/2000/xp/Group/.
  - Description of service – WSDL,
    - http://www.w3.org/2002/ws/desc/.
  - Service discovery – UDDI.


Architecture of web services
Client which use web service
SOAP
Communication using XML messages
Web service
Service discovery
Description of service interface
Description of service interface
UDDI registry
WSDL
Reference to service description

Web Services Description Language (WSDL)

- Description of web service based on XML.
- IBM, Microsoft, nowadays W3C.
- WSDL file with definition of service interface, XML document, contains definition of:
  - Methods,
  - Parameters (data types).

Example of 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)

- Standard protocol for messages (envelope + set of rules for data representation in XML).
- Message SOAP can be used in different protocols for example HTTP or RPC (Remote Procedure Call).
- It is copmposed from three parts:
  - envelope – define what message contains and how should be processed.
  - Set of coding rules – for exampel serialization of primitives data types for RPC or message passing through HTTP.
  - Konvention for calling remote procedures.

Simple Object Access Protocol (SOAP)

- SOAP based on XML.
- SOAP is quite simple
- It does not deal with transactions and security.
- Message contains element Envelope, which contains :
  - header – meta-information,
  - body – information.

Example 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">
```

Example 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>
```

Example 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">
```

Example 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)

- Registration and discovery of web services.
- Offers a public database (registry). Two biggest databases was managed by ~~IBM a Microsoft.~~
- UDDI regiostry contains four types of entities:
  - business entity.
  - business service.
  - binding template, description by WSDL.
  - service type.

Java web services

- Standard JavaEE web appiucation
- Definition of class:

```
@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 – old way

- WEB-INF/web.xml (only 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 - client

Client code generation:

- Eclipse UI
- Old way

  <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 - client

- Usage of generated code in JavaSE application:

```
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

P11

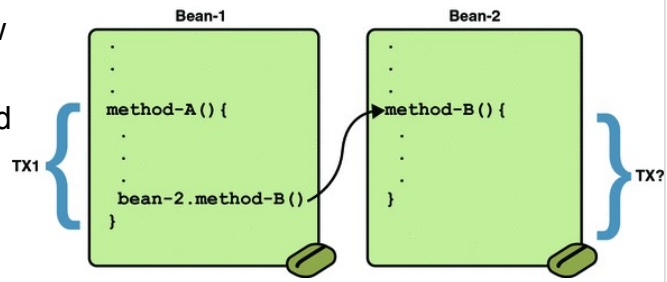
- JTA – Java Transaction
- JMS – Java Message Services
- Message-Driven Beans

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

JTA – Java Transaction

Conatiner managed transaction

- Inside one EJB method is not allowed more then one transactions or nested transactions.
- Transaction interface
  - 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

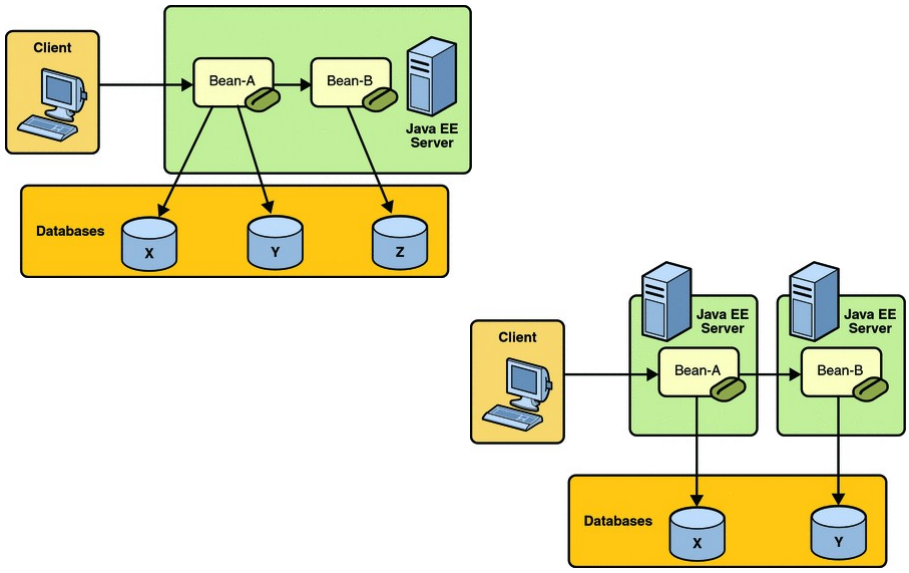
Table 33-1 summarizes the effects of the transaction attributes. Both the T1 and the T2 transactions are controlled by the container. A T1 transaction is associated with the client that calls a method in the enterprise bean. In most cases, the client is another enterprise bean. A T2 transaction is started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

Transaction managed by application

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

- Allowe more ten one transaction in method
- More lines of code

```
@Resource
SessionContext context;


UserTransaction utx = context.getUserTransaction();

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