Java Management Extensions
JMX
Overview

- part of JDK since version 5
  - previously an external set of jar archives
- MBean = Managed Java Bean
  - beans intended for managing something (device, application, anything)
  - provides an interface like std. beans
    - properties (get and set methods)
    - regular methods
    - notifications via events
  - several types
    - standard
    - dynamic
    - open
    - model
- (not only) universal client – JConsole
Architecture

- JMX-compliant Management Application
- Web Browser
- Proprietary Management Application
- JMX Manager
- Connectors and Protocol Adaptors

- Distributed Services Level
- Agent Level
- Instrumentation Level

Java virtual machine (host1)
Resource 1 (MBean)
Resource 2 (MBean)

Current JMX specification
Separate JSRs
Future phases of the JMX specification

Image source: JMX Specification, version 1.4
Types of MBeans

- **Standard**
  - the simplest type
  - its interface = all methods
- **Dynamic**
  - must implement a particular interface
  - more flexible
  - can be changed at runtime
- **Open**
  - dynamic
  - but can use only basic types
    - no need for a special descriptor
- **Model**
  - dynamic
  - fully configurable at run-time
Standard MBean

• defined explicitly by its interface and implementation (class)
  – the interface must have the same name as the class plus extension MBean
  – all methods in the MBean interface are provided
    • methods of the class but not in the interface are not visible via JMX
  – rules for naming properties and methods are the same as for regular beans
  – the interface is at run-time obtained via reflection
Example of a std. MBean

package example.mbeans;

public interface MyClassMBean {
    public int getState();
    public void setState(int s);
    public void reset();
}

package example.mbeans;

public class MyClass implements MyClassMBean {
    private int state = 0;
    private String hidden = null;
    public int getState() {
        return(state);
    }
    public void setState(int s) {
        state = s;
    }
    public String getHidden() {
        return(hidden);
    }
    public void setHidden(String h) {
        hidden = h;
    }
    public void reset() {
        state = 0;
        hidden = null;
    }
}
package example.mbeans;

import java.lang.management.*;
import javax.management.*;

public class Main {

    public static void main(String[] args) throws Exception {

        MBeanServer mbs = ManagementFactory.getPlatformMBeanServer();

        ObjectName name = new ObjectName("example.mbeans:type=MyClass");

        MyClass mbean = new MyClass();
        mbs.registerMBean(mbean, name);

        System.out.println("Waiting forever...");
        Thread.sleep(Long.MAX_VALUE);
    }
}
Dynamic MBean

- intended for a changing interface
- implements the DynamicMBean interface
  - the bean's interface is obtained at run-time via calling methods of this interface

interface DynamicMBean {
    MBeanInfo getMBeanInfo();
    Object getAttribute(String attribute);
    AttributeList getAttributes(String[] attributes);
    void setAttribute(Attribute attribute);
    AttributeList setAttributes(AttributeList attributes);
    Object invoke(String actionName, Object[] params, String[] signature);
}
Dynamic MBean

- MBeanInfo
  - describes the MBean interface
  - for each call, a result of getMBeanInfo can be different
  - then, universal JMX clients cannot be (usually) used
Identification

- the class **ObjectName**
  - represent the name of a mbean or a pattern for searching
  - composed of a domain and properties
  - domain
    - string
    - must not contain colon and //
  - properties
    - name-value pairs
      - type – type of mbean
      - name
      - ...

JMX notification

- MBean can generate events
  - e.g. after change of its state
  - similar to regular beans
- the Notification class
  - represents an event
  - extends java.util.EventObject
  - can be used directly
    - but typically via its children (again as with regular beans)
- the NotificationListener interface
  - registering for event listening
- the NotificationBroadcaster interface
  - MBeans generating events must implement this interface
  - it is better to implement NotificationEmitter
    - extends NotificationBroadcaster
JMX notifikace

- the NotificationFilter interface
  - filtering notifications
  - a listener registers it
- types of event
  - it is not the class
  - a property of the event (String)
  - hierarchical
    - JMX.<something> reserved for JMX
- properties of the event (of the class Notification)
  - type
  - sequence number
  - timestamp (when the event was generated)
  - message
  - user data
JMX notification

- NotificationEmitter
  - void addNotificationListener(NotificationListener listener, NotificationFilter filter, Object handback)
  - handback
    - a utility object
    - the emitter does not use it
    - it is passed during event delivery
  - void removeNotificationListener(NotificationListener listener)
  - void removeNotificationListener(NotificationListener listener, NotificationFilter filter, Object handback)
  - MBeanNotificationInfo[] getNotificationInfo()
JMX notification

- NotificationListener
  - void handleNotification(Notification notification, Object handback)

- NotificationFilter
  - boolean isNotificationEnabled(Notification notification)

- support for notifying field changes
  - AttributeChangeNotification
  - AttributeChangeNotificationFilter

- the NotificationBroadcasterSupport class
  - a prepared implementation of NotificationBroadcaster
public class Hello extends NotificationBroadcasterSupport implements HelloMBean {
    ...
    public synchronized void setCacheSize(int size) {
        int oldSize = this.cacheSize;
        this.cacheSize = size;
        Notification n = new AttributeChangeNotification(this,
            sequenceNumber++, System.currentTimeMillis(), "CacheSize changed", "CacheSize", "int", oldSize, this.cacheSize);
        sendNotification(n);
    }

    public MBeanNotificationInfo[] getNotificationInfo() {
        String[] types = new String[]{AttributeChangeNotification.ATTRIBUTE_CHANGE};
        String name = AttributeChangeNotification.class.getName();
        String description = "An attribute of this MBean has changed";
        MBeanNotificationInfo info = new MBeanNotificationInfo(types, name, description);

        return new MBeanNotificationInfo[]{info};
    }
}
Open MBean

- dynamic MBean
- uses only a limited set of data types
  - basic data types
    - primitive types (wrapper types)
    - String
    - BigDecimal, BigInteger
    - Date
    - javax.management.openbean.CompositeData
    - javax.management.openbean.CompositeTabular
    - arrays of these types
- can be used with universal clients
  - no need to recompile clients after the interface change
Open MBean

- `javax.management.openbean.CompositeData`
  - interface
  - represents composed types
  - “structures”
  - similar to a hash table
- `javax.management.openbean.CompositeTabular`
  - interface
  - represents arrays

- OpenMBeanInfo
  - extends MBeanInfo
  - plus other “Open” descriptors
    - OpenMBeanOperationInfo,..s.
Model MBean

- dynamic
- generic and fully configurable at run-time
  - no static interface, but elements are dynamically added
Model MBean example

MBEANServer mbs = ...

HashMap map = new HashMap();

Method getMethod = HashMap.class.getMethod("get", new Class[]{
    Object.class});
ModelMBeanOperationInfo getInfo =
    new ModelMBeanOperationInfo("Get value for key", getMethod);
ModelMBeanInfo mmibi =
    new ModelMBeanInfoSupport(HashMap.class.getName(),
        "Map of keys and values",
        null, // no attributes
        null, // no constructors
        new ModelMBeanOperationInfo[]{getInfo},
        null); // no notifications

ModelMBean mmb = new RequiredModelMBean(mmibi);
mmb.setManagedResource(map, "ObjectReference");

ObjectName mapName = new ObjectName(":\type=Map,name=whatever");
mbs.registerMBean(mmb, mapName);

mbs.invoke(mapName, "get", new Object[] {"key"}, new String[]
    {Object.class.getName()});
MXBean

- a new type of MBean
  - since JDK 6 (partially also in 5)
- a standard MBean
- plus rules for Open MBean
  - i.e. uses only a limited set of data types
- MXBean is a class implementing a `<something>`MXBean interface
  - the class can have any name
- instead of the extension MXBean the annotation @MXBean can be used
  - also @MXBean(false) can be used to set that the given interface is not a JMX interface even it has the MXBean extension
Architecture (recap.)

JMX-compliant Management Application

Distributed Services Level

Web Browser

Proprietary Management Application

Connectors and Protocol Adaptors

Agent Level

MBean Server

Agent Services (as MBeans)

Instrumentation Level

Resource 1 (MBean)

Resource 2 (MBean)

Java virtual machine (host1)

Current JMX specification

Separate JSRs

Future phases of the JMX specification

image source JMX Specification, version 1.4
JMX Remote

- remote access to JMX
- via *connectors*
  - composed of
    - connector client
    - connector server
- connectors can be created over (almost) anything
- the specification defines 2 particular connectors
  - RMI
  - generic
    - JMX Messaging Protocol (JMXMP)
      - directly over TCP
    - its implementation is optional
A connection creation

connect "service:jmx:jmxmp://host1:9876"

1. connection request

2. create server end

3. connection response

4. create client end

image source JMX Specification, version 1.4
creating a MBean, registration,... are as previously
plus creating the connector server

```java
MBeanServer mbs = MBeanServerFactory.createMBeanServer();
...

JMXServiceURL url = new JMXServiceURL("service:jmx:rmi:///jndi/rmi://localhost:9999/server");

JMXConnectorServer cs = JMXConnectorServerFactory.newJMXConnectorServer(url, null, mbs);

cs.start();
...

cs.stop();
```
JMX Remote

• JMXServiceURL
  - url of the connector server
  - depends on the type of a connector
  - common structure
    service:jmx:<protocol>::...
  - for own connectors it is not necessary to follow the structure
    • but it is recommended

• the JMX specification defines
  - message buffering
  - rules for parallel usage
  - how to deal with communication errors
  - dynamic class loading
  - security
  - ...
JMX Remote – RMI connector

- mandatory
  - every JMX implementation must contain it
- uses regular RMI
- usage of JRMP or IIOP can be specified
- using the RMI connector
  - service:jmx:rmi://host:port
    - the connector server creates a RMI server and returns a URL in a form service:jmx:rmi://host:port/stub/XXXX
      - XXXX is the serialized RMI server
  - service:jmx:iiop://host:port
    - the connector server creates a CORBA object and returns a URL in a form service:jmx:iiop://host:port/ior/IOR:XXXX
      - XXXX is std. ior
    - creates a server and registers it in the naming service
    - iiop can be written instead of rmi
JMX Remote – Generic connector

- **optional**
  - JMX implementations need not to contain it
- **configurable**
  - goal – a simple specification of transport protocols and wrapper objects for communication
- **defines communication using messaging**
  - a connection initialization
  - messages
  - ...
- **JMXMP connector**
  - a configuration of the generic connector for JMXMP
creating a connection to the server

```java
JMXServiceURL url = new JMXServiceURL("service:jmx:rmi:///jndi/rmi://localhost:9999/server");
JMXConnector jmxc = JMXConnectorFactory.connect(url, null);

MBeanServerConnection mbsc = jmxc.getMBeanServerConnection();
```

usage

```java
mbsc.queryMBeans(ObjectName name, QueryExp query)
mbsc.getAttribute(ObjectName name, String attrName)
mbsc.setAttribute(ObjectName, Attribute attr)
```
**JMX Remote – client**

- creating a proxy object for direct access
  - it is necessary to know the interface
  - works for standard mbeans

```java
t JMX.newMBeanProxy(MBeanServerConnection connection, ObjectName objectName, Class<T> interfaceClass)

t JMX.newMBeanProxy(MBeanServerConnection connection, ObjectName objectName, Class<T> interfaceClass, boolean notificationBroadcaster)
```
Aspects (AOP)
AspectJ
AOP

• Aspect-oriented programming
• „separation of concerns“
  - concern ~ a part of program code related to a particular functionality

• typically understood as an extension of OOP
• solves the problem that it is not always possible to put a code for a single functionality to a single (or several) classes
  - contrary, code is through the all application
Application modularity

XML parsing in Tomcat

logging

URL manipulating in Tomcat
AspectJ

- a Java extension
  - 1 concept – **joinpoint**
    - a place in a program for adding code
  - several constructs
    - **pointcut**
      - definition of joinpoint(s)
    - **advice**
      - code to be added
    - **inter-type declaration**
      - extending a class declaration
    - **aspect**
      - a “class” that can contain the above mentioned constructs
Pointcut

• call(void Point.setX(int))
• call(void Point.setX(int)) ||
call(void Point.setY(int))
• call(void FigureElement.setXY(int,int)) ||
call(void Point.setX(int)) || call(void Point.setY(int)) ||
call(void Line.setP1(Point)) ||
call(void Line.setP2(Point))
• pointcut move():
call(void FigureElement.setXY(int,int)) ||
call(void Point.setX(int)) || call(void Point.setY(int)) ||
call(void Line.setP1(Point)) ||
call(void Line.setP2(Point));
• call(public * Figure.* (..))
Advice

- before(): move() {
  System.out.println("about to move");
}

- after() returning: move() {
  System.out.println("just successfully moved");
}
aspect PointObserving {
    private Vector Point.observers = new Vector();
    ...
}
aspect PointObserving {
    private Vector Point.observers = new Vector();
    public static void addObserver(Point p, Screen s) {
        p.observers.add(s);
    }
    public static void removeObserver(Point p, Screen s) {
        p.observers.remove(s);
    }
    pointcut changes(Point p): target(p) && call(void Point.set*(int));
    after(Point p): changes(p) {
        Iterator iter = p.observers.iterator();
        while ( iter.hasNext() ) {
            updateObserver(p, (Screen)iter.next());
        }
    }
    static void updateObserver(Point p, Screen s) {
        s.display(p);
    }
}
• aspect SimpleTracing {
  pointcut tracedCall():
      call(void FigureElement.draw(GraphicsContext));

  before(): tracedCall() {
      System.out.println("Entering: " + thisJoinPoint);
  }
}

• aspect SetsInRotateCounting {
  int rotateCount = 0;
  int setCount = 0;

  before(): call(void Line.rotate(double)) {
      rotateCount++;
  }

  before(): call(void Point.set*(int))
      && cflow(call(void Line.rotate(double))) {
      setCount++;
  }
}
Aspects can be defined directly in Java via annotations.

```java
@Aspect
public class Foo {

    @Pointcut("call(* *.*(..))")
    void anyCall() {}

    @Before("call(* org.aspectprogrammer..*(..))
              && this(Foo)")
    public void callFromFoo() {
    }
}
```
JAVA

JEE
Java Enterprise Edition
Overview

- JEE
- JSE
- JME
- Java Card API
“Enterprise” applications

• “big enterprise” applications
• required features
  – re-usability
  – loosely coupled
  – transactions
  – declarative interface
  – persistence
  – security
  – distributed applications
  – ...

Java, summer semester 2018
3-tier architecture

**Presentation tier**
The top-most level of the application is the user interface. The main function of the interface is to translate tasks and results to something the user can understand.

**Logic tier**
This layer coordinates the application, processes commands, makes logical decisions and evaluations, and performs calculations. It also moves and processes data between the two surrounding layers.

**Data tier**
Here information is stored and retrieved from a database or file system. The information is then passed back to the logic tier for processing, and then eventually back to the user.

EJB
(first, briefly EJB 2, i.e. old EJB)
Overview

- Enterprise Java Beans
- components
- runs in a server
  - an EJB container
- local and remote access
- the container offers many services
  - persistence
  - security
  - transactions
  - scalability
  - concurrency
EJB

• kinds of beans
  – session beans
    • stateless
      – no state kept
    • statefull
      – a state is kept
  – message-driven beans
  – entity beans
Session beans

• implement business logic
  – logic tier
• not persistent
Session beans

- 2 interfaces
  - business interface
    - extends EJBOBJECT
    - declares provided methods
  - home interface
    - extends EJBHome
    - defines “create” methods
- implementation
  - a class extending SessionBean
  - has methods from the business interface
    *but does not implement it!*
  - must have following methods
    - public void ejbRemove()
    - public void ejbActivate()
    - public void ejbPassivate()
    - public void setSessionContext(SessionContext sc)
    - public void ejbCreate()
Entity beans

• access to persistent data (database)
  - data tier
• persistence
  - container managed
  - bean managed
Entity Beans

- 2 interfaces
  - home
    - create
    - findByPrimaryKey
  - business
    - get/set methods for data

- implementation
  - extends EntityBean
  - does not implement the business interface
  - get/set methods are abstract
Message-driven beans

- implementing the MessageListener interface
  - onMessage()
EJB container

- EJB container “ties” the interface and implementation
  - generates stubs and skeletons

image source: B.Eckel: Thinking in Enterprise Java
Deployment

- deployment descriptor
- EAR
Overview

• 2002
• critique of EJB
  – to complex
  – hard to be used
  – hard to be tested
  – RemoteException everywhere
  – …
• Rod Johnson: Expert One-on-One
  J2EE Design and Development
  – critique of EJB +
    proposal of a better architecture
  • Spring foundations
Overview

- Spring
  - http://www.spring.io/
  - based on POJO
    - plain old Java objects
    - but can be integrated with EJB
  - “lightweight” solution
    - the smallest possible dependency of application code on Spring
    - no server necessary
      - suitable for any type of application
  - effort for integration with other frameworks
    - not to “reinvent the wheel”
    - to use proven existing solutions
Spring core

- the org.springframework.beans package
- an “inversion of control” container
  - Dependency Injection
  - Hollywood Principle: "Don't call me, I'll call you."

- objects are not interconnected in code but in a configuration file
- an object is not responsible for searching its dependencies
- dependencies are declared
  - a container “provides” them – sets them via setters
    - common naming conventions setXxx()
    - or via parameters of constructors
- no special requirements on objects
Spring core

- objects created via a “factory”
  - the interface
    org.springframework.beans.factory.BeanFactory
  - the most used factories
    - DefaultListableBeanFactory
public class nameBean {
    String name;

    public void setName(String a) {
        name = a;
    }

    public String getName() {
        return name;
    }
}

<bean id="bean1" class="nameBean">
    <property name="name">
        <value>Tom</value>
    </property>
</bean>

• interconnecting objects

<bean id="bean" class="beanImpl">
    <property name="conn">
        <ref bean="bean2"/>
    </property>
</bean>

<bean id="bean2" class="bean2impl"/>
Spring and data tier

- anything can be used
  - JDBC
  - ORM
    - Hibernate
    - ...
- can be used separately
  - simplified DB usage
  - unified exceptions
  - ...
Spring and data tier

- JdbcTemplate template = new JdbcTemplate(dataSource);
  List names = template.query("SELECT USER.NAME FROM USER",
   new RowMapper() {
     public Object mapRow(ResultSet rs, int rowNum) throws SQLException {
       return rs.getString(1);
     }
   });

- int youngUserCount = template.queryForInt("SELECT COUNT(0) FROM USER WHERE USER.AGE < ?", new Object[] { new Integer(25) });

- class UserQuery extends MappingSqlQuery {
  public UserQuery(DataSource datasource) {
    super(datasource, "SELECT * FROM PUB_USER_ADDRESS WHERE USER_ID = ?");
    declareParameter(new SqlParameter(Types.NUMERIC));
    compile();
  }
  protected Object mapRow(ResultSet rs, int rownum) throws SQLException{
    User user = new User();
    user.setId(rs.getLong("USER_ID"));  user.setForename(rs.getString("FORENAME"));
    return user; }
  public User findUser(long id) { return (User) findObject(id); }
}
User user = userQuery.findUser(25);
Spring AOP

- implemented in plain Java
  - can be integrate with AspectJ
- intended for functionality for which aspects are ideal
  - originally for adding JEE services to Spring
    - transactions
    - logging
    - ...

Other Spring parts

• Spring MVC
  - a web MVC framework
  - inspired by the Struts framework
  - does not prescribe what should be used for generating pages
    • JSP
    • template systems (Velocity,...)
    • ...

• EJB
  - instead of POJO, EJBs can be used

• ...

Spring Roo

- framework easy generation of enterprise applications
  - roughly
  - creation of an application using a “wizard” in several steps
Overview

- inspired by Springem
- instead of implementing interfaces, annotations are used
- using “dependency injection”
- no need to use descriptors
- …
- entity beans replaced by Java Persistence API
  - “mapping” classes to tables in relational database
  - JPQL query language
    - “SQL over objects”
@Remote
public interface Converter {
    public BigDecimal dollarToYen(BigDecimal dollars);
}

@Stateless
public class ConverterBean implements converter.ejb.Converter {
    private BigDecimal euroRate = new BigDecimal("0.0070");

    public BigDecimal dollarToYen(BigDecimal dollars) {
        BigDecimal result = dollars.multiply(yenRate);
        return result.setScale(2, BigDecimal.ROUND_UP);
    }
}
@MessageDriven(mappedName="MDBQueue")
public class MDB implements MessageListener {
    public void onMessage(Message msg) {
        System.out.println("Got message!");
    }
}
@Entity
@Table(name = "phonebook")
public class PhoneBook implements Serializable {
    @Column(name="number") private String number;
    @Column(name="name") private String name;

    public PhoneBook() {} 

    public PhoneBook(String name, String number) {
        this.name = name;
        this.number = number;
    }

    @Id public String getName() { return name; }
    public void setName(String name) { this.name = name; }
    public String getNumber() { return number; }
    public void setNumber(String number) { this.number = number; }
}
JPQL

- inspired by HQL - a subset of HQL

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

- DELETE FROM ...
  WHERE ...

- UPDATE ... SET ...
  WHERE ...

- SELECT a FROM Author a ORDER BY a.firstName, a.lastName
- SELECT DISTINCT a FROM Author a INNER JOIN a.books b WHERE b.publisher.name = 'MatfyzPress'