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

Connectors and Protocol Adaptors

Distributed Services Level

Agent Level

Instrumentation Level

Current JMX specification
Separate JSRs
Future phases of the JMX specification

Added Management Protocol APIs

JMX Manager

SNMP Manager API
CIM/WBEM API
TMN Manager API

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

```java
text
package example.mbeans;

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

text
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;
    }
}
```
Using MBean

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

```java
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 mmbi =
    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(mmbi);
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.)

The diagram illustrates the JMX-compliant architecture with components such as the JMX-compliant Management Application, Web Browser, Proprietary Management Application, JMX Manager, and connectors and protocol adaptors. The figure also indicates distributed services level, agent level, instrumentation level, and additional management protocol APIs. The diagram includes symbols for currently specified JMX, separate JSRs, and future phases of the JMX specification.

Image source: JMX Specification, version 1.4

Java, summer semester 2017
3.4.2017
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

connector client

4. create client end

client connection

2. create server end

connector server

service:jmx:jmxmp://host1:9876

3. connection response

server connection

image source JMX Specification, version 1.4
JMX Remote

- 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
**JMX Remote – client**

- 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)
```

```java
T JMX.newMBeanProxy(MBeanServerConnection connection, ObjectName objectName, Class<T> interfaceClass, boolean notificationBroadcaster)
```
Unit testing
Introduction

• unit testing
  – testing “small” units of functionality
  – a unit – independent on other ones
    • tests are separated
    • creating helper objects for tests
      – context
  – typically in OO languages
    • unit ~ method
  – ideally – unit tests for all units in a program
    • typically in OO languages
      – for all public methods
JUnit

- support for unit testing in Java
- http://www.junit.org/
- current version 4
  - usage based on annotations
- older versions
  - based on inheritance and naming conventions
Usage

- test methods marked by the @Test annotation
- JUnit is run on a set of classes
  - searches in them all @Test methods
  - executes them

- other annotations
  - @Before
    - a method run before each test
    - intended for “environment” preparation
  - @After
    - a method run after each test
    - intended for “cleaning”
  - @BeforeClass
    - a method run before all tests in the given class
  - @AfterClass
    - a method run after all tests in the given class
package junitfaq;

import org.junit.*;
import static org.junit.Assert.*;
import java.util.*;

public class SimpleTest {

    private Collection collection;

    @BeforeClass
    public static void oneTimeSetUp() {
        // one-time initialization code
    }

    @AfterClass
    public static void oneTimeTearDown() {
        // one-time cleanup code
    }

    @Before
    public void setUp() {
        collection = new ArrayList();
    }

    @After
    public void tearDown() {
        collection.clear();
    }

    @Test
    public void testEmptyCollection() {
        assertTrue(collection.isEmpty());
    }

    @Test
    public void testOneItemCollection() {
        collection.add("itemA");
        assertEquals(1, collection.size());
    }
}
Assert

- `assertTrue`
- `assertFalse`
- `assertEquals`
- `assert...`
  - static methods of `org.junit.Assert`
  - testing conditions in tests
  - test fails if `assert...` fails
    - `assert...()` throws `AssertionError`

- in general
  - test is successful if the method terminates regularly
  - test fails if the method throws an exception
Testing exceptions

• how to test “correctly” thrown exceptions?

```java
@Test(expected= IndexOutOfBoundsException.class) public void empty() {
    new ArrayList<Object>().get(0);
}
```
Running tests

- from code
  ```java
  org.junit.runner.JUnitCore.runClasses(TestClass1.class, ...);
  ```
- from command line
  ```shell
  java org.junit.runner.JUnitCore TestClass1 [other třídy...]
  ```
- from Ant
  - the task junit
    ```xml
    <junit printsummary="yes" fork="yes" haltonfailure="yes">
      <formatter type="plain"/>
      <test name="my.test.TestCase"/>
    </junit>
    ```
- from IDE
TestNG

- http://testng.org/
- inspired by JUnit
- more feature than JUnit
  - e.g. parameters for test methods
- basic usage is the same
Maven
Overview

- http://maven.apache.org/
- a tool for managing projects
  - roughly, Maven can be seen as an Ant extension
    - but it is not an Ant extension
- provides
  - dependency management
  - project building
  - usage of “best practices”
  - extensibility by new modules
  - ...
Usage

- a project generation
  mvn archetype:generate
    -DarchetypeGroupId=org.apache.maven.archetypes
    -DgroupId=com.mycompany.app
    -DartifactId=my-app

  - archetype ~ a project template
  - generates the following structure
Project structure

my-app
|-- pom.xml
 `-- src
   |-- main
     `-- java
       `-- com
         `-- mycompany
           `-- app
             `-- App.java
 `-- test
  `-- java
    `-- com
      `-- mycompany
        `-- app
          `-- AppTest.java
POM – Project Object Model

- a project definition

```xml
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
                     http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.mycompany.app</groupId>
  <artifactId>my-app</artifactId>
  <packaging>jar</packaging>
  <version>1.0-SNAPSHOT</version>
  <name>Maven Quick Start Archetype</name>
  <url>http://maven.apache.org</url>
  <dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>3.8.1</version>
      <scope>test</scope>
    </dependency>
  </dependencies>
</project>
```
Build lifecycle

- mvn “phase”
  - previous phases are also executed

1. process-resources
2. compile
3. process-test-resources
4. test-compile
5. test
6. package
7. install
8. deploy
Others

- generating different project types
  mvn archetype:generate \
    -DarchetypeGroupId=org.apache.maven.archetypes
    -DarchetypeArtifactId=maven-archetype-webapp
    -DgroupId=com.mycompany.app
    -DartifactId=my-webapp

- generating documentation
  mvn archetype:generate
    -DarchetypeGroupId=org.apache.maven.archetypes
    -DarchetypeArtifactId=maven-archetype-site
    -DgroupId=com.mycompany.app
    -DartifactId=my-app-site
Apache Ivy
Overview

- http://ant.apache.org/ivy
- dependency management
- an Ant extension
  - can be used standalone
- a “lightweight” maven
Usage

• dependencies in the ivy.xml file

```xml
<ivy-module version="2.0">
  <info organisation="org.apache" module="hello-ivy"/>
  <dependencies>
    <dependency org="commons-lang" name="commons-lang" rev="2.0"/>
    <dependency org="commons-cli" name="commons-cli" rev="1.0"/>
  </dependencies>
</ivy-module>
```

• build.xml

```xml
<project xmlns:ivy="antlib:org.apache.ivy.ant" name="hello-ivy" default="run">
  <target name="resolve" description="retrieve dependencies with ivy">
    <ivy:retrieve />
  </target>
</project>
```
Dependencies

• by default, maven repositories are used
• own one can be defined
  – local
  – shared
  – public
Overview

- https://gradle.org/
- managing projects
  - as Maven
- creating a project
  - gradle init --type java-library
- types
  - java
  - groovy
  - scala
  - ...
- dependencies ~ Maven repositories
- build file
  - gradle
apply plugin: 'java'
apply plugin: 'application'

repositories {
    jcenter()
}

dependencies {
    compile 'com.google.guava:guava:20.0'
    testCompile 'junit:junit:4.12'
}

mainClassName = 'App'