Java Management Extensions (JMX)

JMX Overview (1/2)

- **The Java Management Extensions (JMX)** - standard way of managing resources:
  - applications,
  - devices,
  - services.
- JMX can be used to monitor and manage the JVM.
- JMX defines:
  - architecture,
  - design patterns,
  - APIs,
  - services.

JMX Overview (2/2)

- Resource is instrumented by one or more Java objects known as **Managed Beans** (MBeans).
- MBeans are registered in a core-managed object known as an **MBean server**.
- **JMX agent** consists of:
  - MBean server, in which MBeans are registered,
  - set of services for handling the MBeans.
- Resource instrumentation is independent from the management infrastructure (can be rendered manageable regardless of how their management applications).
- JMX defines standard **connectors** (JMX connectors) that enable to access JMX agents from remote management applications. JMX connectors using different protocols provide the same management interface.

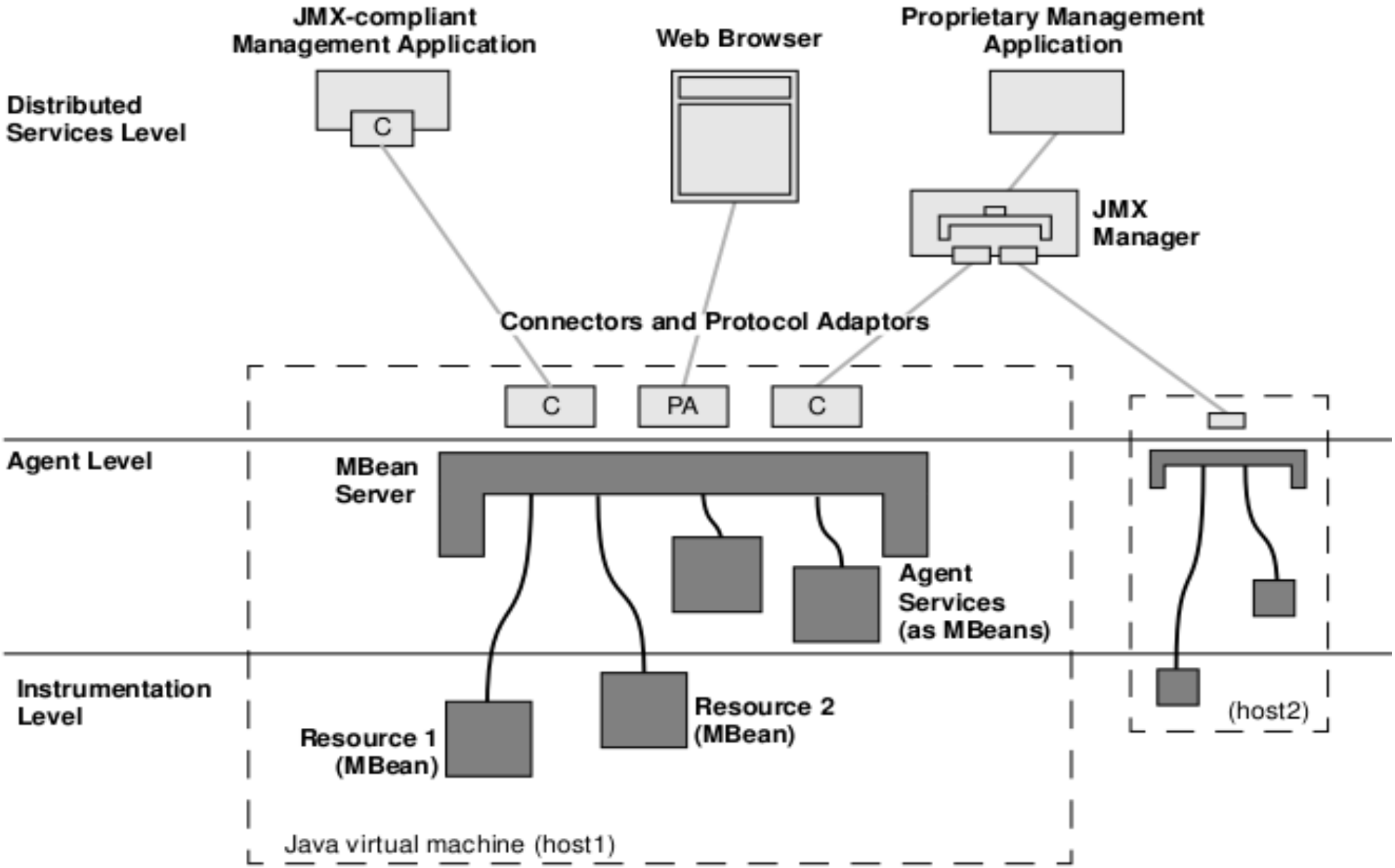
Benefits of JMX Technology (1/2)

- Java EE Application Server conforms to the JMX architecture and consequently can be managed by using JMX technology.
- Java VM is highly instrumented using the JMX technology. You can start a JMX agent to access the built-in Java VM instrumentation, and thereby monitor and manage a Java VM remotely.
- Every JMX agent service is an independent module that can be plugged into the management agent, depending on the requirements.
- The JMX specification provides a set of core agent services. Additional services can be developed and dynamically loaded, unloaded, or updated in the management infrastructure.

Benefits of JMX Technology (2/2)

- JMX specification references existing Java specifications, for example, the Java Naming and Directory Interface (J.N. D.I.) API.
- The JMX technology-based applications (JMX applications) can be created from a NetBeans IDE module.
- The JMX technology integrates with existing management solutions and emerging technologies.
- JMX solutions can use lookup and discovery services and protocols such as Jini network technology and the Service Location Protocol (SLP).

Architecture of the JMX (1/3)



Architecture of the JMX (2/3)

- The JMX technology is divided into three levels, as follows:
  - Instrumentation,
  - JMX agent,
  - Remote management,
- **Instrumentation** - to manage resources using the JMX technology, resources must be instrumented.
- MBeans implements the access to the resources' instrumentation.
- **MBeans** (standard) must follow the design patterns and interfaces defined in the JMX specification.
- Other types of MBean: Standard MBeans, Dynamic MBeans, Open MBeans, Model MBeans, MXBeans.

Architecture of the JMX (3/3)

- Once a resource has been instrumented by MBeans, it can be managed through a **JMX agent**. MBeans do not require knowledge of the JMX agent with which they will operate.
- Developers of applications, systems, and networks can make their products manageable in a standard way without having to understand.
- Instrumentation level of the JMX specification provides a **notification mechanism**. This mechanism enables MBeans to generate and propagate notification events to components of the other levels.

Remote Management

- JMX instrumentation can be accessed either through existing management protocols such as the Simple Network Management Protocol (SNMP) or through proprietary protocols.
- Each **adaptor** provides a view through a specific protocol of all MBeans that are registered in the MBean server. For example, an HTML adaptor could display an MBean in a browser.
- **Connectors** provide a manager-side interface that handles the communication between manager and JMX agent. Each connector provides the same remote management interface through a different protocol.
- The JMX technology provides a standard solution for exporting JMX technology instrumentation to remote applications based on Java RMI.

Monitoring and Management of JVM

- The Java VM has built-in (out-of-the-box) instrumentation that enables you to monitor and manage it by using the JMX technology.
- **The platform MXBeans** are a set of MXBeans that is provided with the Java SE platform for monitoring and managing the Java VM and other components of the Java Runtime Environment (JRE).
- Each platform MXBean encapsulates a part of Java VM functionality: class-loading system, just-in-time (JIT) compilation system, garbage collector etc.
- The Java SE platform provides a **standard platform MBean server** in which these platform MXBeans are registered. The platform MBean server can also register any other MBeans.

Platform MXBeans

- *ClassLoadingMXBean* - class loading system of the JVM.
- *CompilationMXBean* - compilation system of the JVM.
- *MemoryMXBean* - memory system of the JVM.
- *ThreadMXBean* - threads system of the JVM.
- *RuntimeMXBean* - runtime system of the JVM.
- *OperatingSystemMXBean* - operating system on which the JVM is running.

GarbageCollectorMXBean - garbage collector in the JVM.

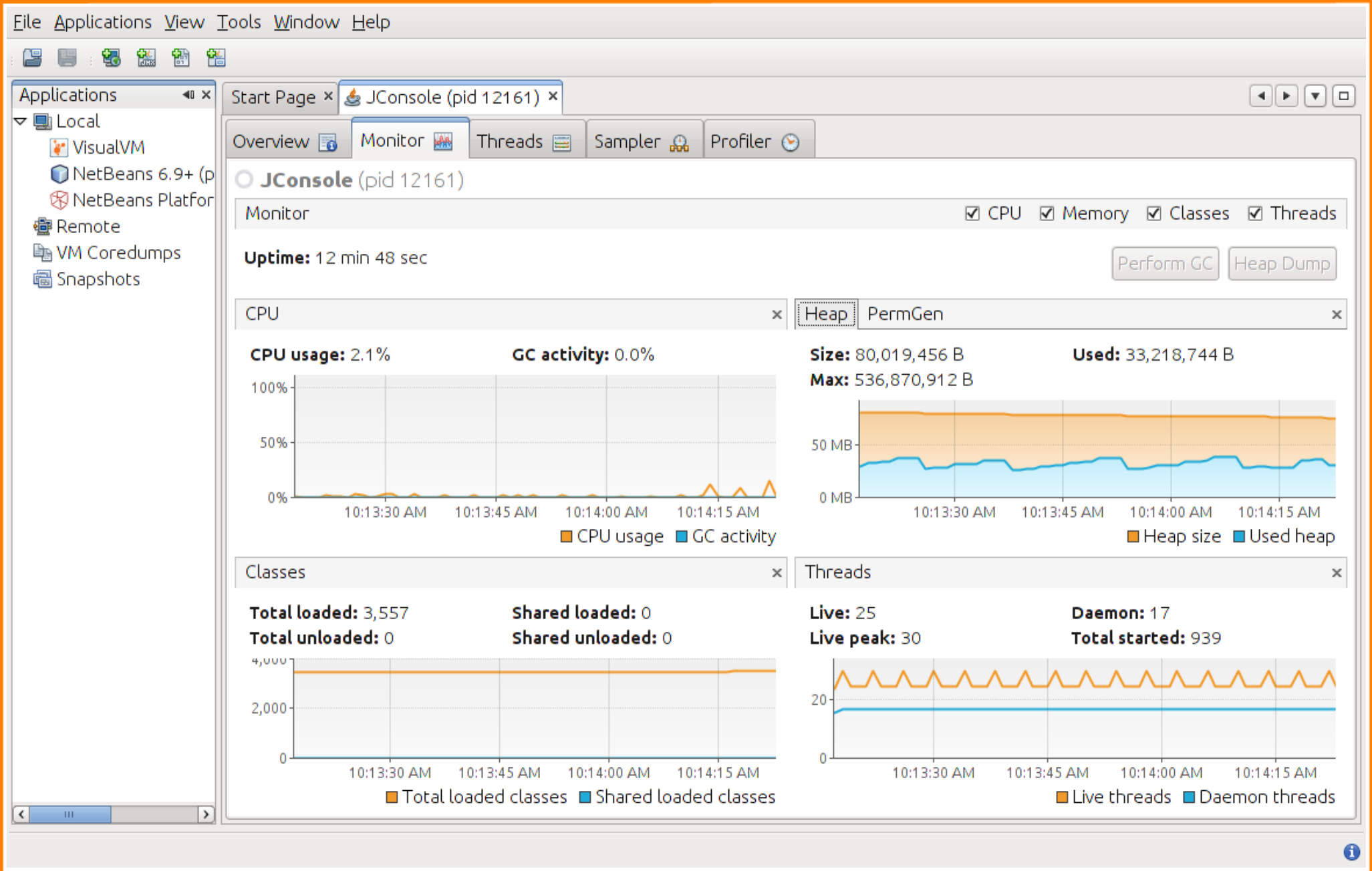
- *MemoryManagerMXBean* - memory manager in the JVM.
- *MemoryPoolMXBean* - memory pool in the JVM.
- Example:

```
RuntimeMXBean mxbean =
ManagementFactory.getRuntimeMXBean();
// Get the standard attribute "VmVendor"
String vendor = mxbean.getVmVendor();
```

VisualVM (1/2)

- **Java VisualVM** - provides a visual interface for viewing detailed information about Java applications and for troubleshooting and profiling these applications.
- Substitute most of the previously standalone tools: JConsole, jstat, jinfo, jstack, jmap
- Enable to view different data about multiple Java applications uniformly, whether they are running locally or on remote machines.
- Java VisualVM can allow developers to generate and analyse:
  - heap dumps,
  - track down memory leaks,
  - browse the platform's MBeans and perform operations on those MBeans,
  - perform and monitor garbage collection,
  - and perform lightweight memory and CPU profiling.

VisualVM (2/2)


File Applications View Tools Window Help
Applications
Local
VisualVM
NetBeans 6.9+ (p
NetBeans Platfor
Remote
VM Coredumps
Snapshots
Start Page
JConsole (pid 12161)
Overview
Monitor
Threads
Sampler
Profiler
JConsole (pid 12161)
Monitor
CPU
Memory
Classes
Threads
Uptime: 12 min 48 sec
Perform GC
Heap Dump
CPU
CPU usage: 2.1%
GC activity: 0.0%
100%
50%
0%
10:13:30 AM
10:13:45 AM
10:14:00 AM
10:14:15 AM
CPU usage
GC activity
Heap
PermGen
Size: 80,019,456 B
Used: 33,218,744 B
Max: 536,870,912 B
50 MB
0 MB
Heap size
Used heap
Classes
Total loaded: 3,557
Shared loaded: 0
Total unloaded: 0
Shared unloaded: 0
2,000
0
Total loaded classes
Shared loaded classes
Threads
Live: 25
Daemon: 17
Live peak: 30
Total started: 939
20
Live threads
Daemon threads

MBeans

- An **MBean** is a managed Java object that follows the design patterns set forth in the JMX specification. An MBean can represent a device, an application, or any resource that needs to be managed.
- MBeans expose a management interface that consists of:
  - a set of readable and/or writable **attributes**,
  - a set of invokable **operations**,
  - a self-description.
- MBeans can also emit **notifications** when certain predefined events occur.
- The JMX specification defines five types of MBean:
  - Standard MBeans
  - MXBeans
  - Dynamic MBeans
  - Open MBeans
  - Model MBeans

Standard MBeans

- A standard MBean is defined by:
  - interface called *Something*MBean and a Java
  - Java class called *Something* that implements that interface.
- Every method in the interface defines either an attribute or an operation in the MBean:

```
public interface HelloMBean {
  public void sayHello();
  public int add(int x, int y);
  public String getName();
  public int getCacheSize();
  public void setCacheSize(int size);
}
```

- Getter and setter methods are declared to allow the managed application to access and possibly change the attribute values.

MBean Implementation

```
public class Hello implements HelloMBean {
  public void sayHello() {
    System.out.println("hello, world");
  }
  public int add(int x, int y) {
    return x + y;
  }
  public String getName() { return this.name; }
  public int getCacheSize() { return this.cacheSize; }
  public synchronized void setCacheSize(int size) {
    this.cacheSize = size;
    System.out.println(
        "Cache size now " + this.cacheSize);
  }
  private final String name = "Reginald";
  private int cacheSize = DEFAULT_CACHE_SIZE;
  private static final int DEFAULT_CACHE_SIZE = 200;
}
```

JMX Agent

- A JMX technology-based agent (**JMX agent**) is a standard management agent that directly controls resources and makes them available to remote management applications. JMX agents are usually located on the same machine as the resources they control
- The core components of a JMX agent:
  - **MBean server -** a managed object server in which MBeans are registered.
  - set of **services** to manage MBeans, and
  - at least one **communications adaptor** or **connector** to allow access by a management application.
- JMX agent does not need to know:
  - which resources it will serve (any instrumented resource can use any JMX agent),
  - functions of the management applications that will access it.

Creating a JMX Agent 2

- If no MBean server has been created by the platform already, then *getPlatformMBeanServer()* creates an MBean server automatically by calling the method *MBeanServerFactory.createMBeanServer()*.
- Every JMX MBean must have an **object name**. The object name is an instance of the class *ObjectName.*
- ObjectName must conform to the syntax defined by the JMX specification. Must contain:
  - domain,
  - list of key-properties.

JMX Agent Example

```
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("com.example:type=Hello");
    Hello mbean = new Hello();
    mbs.registerMBean(mbean, name);
    ...
  }
}
```

MXBeans

- **MXBean** - a type of MBean that references only a predefined set of data types. MXBean is usable by any client (client needn't to access classes representing the types of MXBeans).
- MXBean is given by **@MXBean** annotation.
- Example: type *MemoryUsage* that is referenced in the MXBean interface *MemoryMXBean,* is mapped into a standard set of types, the so-called **Open Types** that are defined in the package *javax.management.openmbean*.
- Mapping rules:
  - simple types such as int or String to remain unchanged,
  - complex types (i.e. *MemoryUsage*) get mapped to the standard type *CompositeDataSupport*.

MXBean Example (1/4)

- MXBean interface is declared the same way as you declare a standard MBean interface.

```
public interface QueueSamplerMXBean {
 public QueueSample getQueueSample();
 public void clearQueue();
}
```

- In the *QueueSample* class, the MXBean framework calls the getters in *QueueSample* to convert the given instance into a *CompositeData* instance and uses the **@ConstructorProperties** annotation to reconstruct a *QueueSample* instance from a *CompositeData* instance.

MXBean Example (2/4)

```
import java.util.Date;
import java.util.Queue;
public class QueueSampler implements QueueSamplerMXBean {
  private Queue queue;
  public QueueSampler(Queue queue) {
    this.queue = queue;
  }
  public QueueSample getQueueSample() {
    synchronized (queue) {
      return new QueueSample(new Date(), queue.size(),
                             queue.peek()); }
  }
  public void clearQueue() {
    synchronized (queue) { queue.clear(); }
  }
}
```

MXBean Example (3/4)

```
import java.beans.ConstructorProperties;
import java.util.Date;
public class QueueSample {
  private final Date date;
  private final int size;
  private final String head;
  @ConstructorProperties({"date", "size", "head"})
  public QueueSample(Date date, int size, String head) {
    this.date = date; this.size = size; this.head = head;
  }
  public Date getDate() { return date; }
  public int getSize() { return size; }
  public String getHead() { return head;}
}
```

MXBean Example (4/4)

```
import java.lang.management.ManagementFactory;
import javax.management.MBeanServer;
import javax.management.ObjectName;
public class Main {
  public static void main(String[] args)
                        throws Exception {
    MBeanServer mbs =
      ManagementFactory.getPlatformMBeanServer();
    ObjectName mxbeanName =
      new ObjectName("com.example:type=QueueSampler");
    Queue queue = new ArrayBlockingQueue(10);
    queue.add("Request-1");
    queue.add("Request-2");
    QueueSampler mxbean = new QueueSampler(queue);
    mbs.registerMBean(mxbean, mxbeanName);
    ...
  }
}
```

Notifications (1/3)

- MBeans can generate **notifications**, for example, to signal a state change, detect problem etc.
- To generate notifications, an MBean must implement the interface *NotificationEmitter* or extend *NotificationBroadcasterSupport*.
- To send a notification, you need to construct an instance of the class *javax.management.Notification* or a subclass (such as *AttributeChangedNotification*), and pass the instance to *NotificationBroadcasterSupport.sendNotification*.
- Notification has:
  - source - tification is the object name of the MBean that generated the notification.
  - sequence number - can be used to order notifications coming from the same source when order matters.

Notifications (2/3)

```
import javax.management.*;
public class Hello extends
  NotificationBroadcasterSupport implements HelloMBean {
  private final String name = "Reginald";
  private int cacheSize = DEFAULT_CACHE_SIZE;
  private static final int DEFAULT_CACHE_SIZE = 200;
  private long sequenceNumber = 1;

  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);
  }
```

Notifications (3 / 3)

```
@Override
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};
 }
}
```

Remote Management

- **JMX connector** makes an MBean server accessible to remote Java technology-based clients. MBean server.
- JMX connector consists of:
  - **connector server** - is attached to an MBean server and listens for connection requests from clients.
  - **connector client** - is responsible for establishing a connection with the connector server.
- A connector client is usually in a different Java Virtual Machine (Java VM) from the connector server and is often running on a different machine.
- The client end of a connector exports essentially the same interface as MBean server so enables JMX client to perform operations on the MBean, exactly as if the operations were being performed locally.
- The JMX API defines a standard connection protocol based on RMI.

RMI Connector Client

- The *Client* class creates an RMI connector client that is configured to connect to an RMI connector server that you will launch when you start the JMX agent.

```
import javax.management.remote.JMXConnector;
import javax.management.remote.JMXConnectorFactory;
import javax.management.remote.JMXServiceURL;
public class Client {
  public static void main(String[] args)
                          throws Exception {
  JMXServiceURL url = new  JMXServiceURL(
    "service:jmx:rmi:///jndi/rmi://:9999/jmxrmi");
  JMXConnector jmxc =
          JMXConnectorFactory.connect(url, null);
```

Creating a Notification Listener

```
public static class ClientListener
                implements NotificationListener {
  public void handleNotification(
        Notification notification, Object handback) {
    String message = notification.getMessage();
    if (notification instanceof
                AttributeChangeNotification) {
      AttributeChangeNotification acn =
        (AttributeChangeNotification) notification;
      String attributeName = acn.getAttributeName());
      String attributeType = acn.getAttributeType());
      Object newValue = acn.getNewValue());
      Object oldValue = acn.getOldValue());
    }
  }
}
```

Connecting to the MBean Server

```
// connection to the remote MBean server
  MBeanServerConnection mbsc =
    jmxc.getMBeanServerConnection();
// installing NotificationListener
  ClientListener listener = new ClientListener();
  mbsc.addNotificationListener(mbeanName, listener,
                        null, null);

// discovering information about the MBeans
// found in the agent's MBean server
  String domains[] = mbsc.getDomains();
  String defaultDomain = mbsc.getDefaultDomain());
  int count = mbsc.getMBeanCount());
  Set mbeanNames =
              new TreeSet(mbsc.queryNames(null, null));
```

Remote MBeans via Proxies

- **MBean proxy** is local to the client, and emulates the remote MBean:

```
ObjectName mbeanName =
  new ObjectName("com.example:type=Hello");
HelloMBean mbeanProxy =
  JMX.newMBeanProxy(mbsc, mbeanName,
                    HelloMBean.class, true);
mbsc.addNotificationListener(mbeanName, listener,
                             null, null);
mbeanProxy.setCacheSize(150);
sleep(2000);
int cacheSize = mbeanProxy.getCacheSize();
mbeanProxy.sayHello();
int add23 = mbeanProxy.add(2, 3);
waitForEnterPressed();
```

Remote MXBeans via Proxies

```
ObjectName mxbeanName =
   new ObjectName("com.example:type=QueueSampler");
QueueSamplerMXBean mxbeanProxy =
    JMX.newMXBeanProxy(mbsc, mxbeanName,
                       QueueSamplerMXBean.class);
QueueSample queue = mxbeanProxy.getQueueSample();
Date date = queue.getDate();
String head = queue.getHead();
int size = queue.getSize();
mxbeanProxy.clearQueue();
```