Java SE 5 Features

Sang Shin
sang.shin@sun.com
www.javapassion.com
Sun Microsystems Inc.
Agenda – Java SE 5 Features

• Generics
• Annotation
• Concurrency
• JMX (Management & Monitoring)
Generics
Sub-topics of Generics

- What is and why use Generics?
- Usage of Generics
- Generics and sub-typing
- Wildcard
- Type erasure
- Interoperability
- Creating your own Generic class
Generics:
What is it?
How do define it?
How to use it?
Why use it?
What is Generics?

• Generics provides abstraction over Types
  > Classes, Interfaces and Methods can be Parameterized by Types (in the same way a Java type is parameterized by an instance of it)

• Generics makes type safe code possible
  > If it compiles without any errors or warnings, then it must not raise any unexpected ClassCastException during runtime

• Generics provides increased readability
  > Once you get used to it
Example Definition of a Generic Class: LinkedList<E>

- Definitions: LinkedList<E> has a type parameter E that represents the type of the elements stored in the linked list

```java
public class LinkedList<E>
    extends AbstractSequentialList<E>
    implements List<E>, Queue<E>, Cloneable, java.io.Serializable{
    private transient Entry<E> header = new Entry<E>(null, null, null);
    private transient int size = 0;

    public E getFirst() {
        if (size==0) throw new NoSuchElementException();
        return header.next.element;
    }
```
Example Usage of Generic Class: LinkedList<Integer>

- Usage: Replace type parameter <E> with concrete type argument, like <Integer> or <String> or <MyType>
  > LinkedList<Integer> can store only Integer or sub-type of Integer as elements

```java
LinkedList<Integer> li =
    new LinkedList<Integer>();
li.add(new Integer(0));
Integer i = li.iterator().next();
```
Example Definition and Usage of Parameterized List interface

// Definition of the Generic'ized List interface
//
interface List<E>{
    void add(E x);
    Iterator<E> iterator();
    ...
}

// Usage of List interface with // concrete type parameter, String //
List<String> ls = new ArrayList<String>(10);
Why Generics? Non-genericized Code is not Type Safe

// Suppose you want to maintain String entries in a Vector. By mistake,
// you add an Integer element. Compiler does not detect this. And
// ClassCastException occurs during runtime.
// This is not type safe code.

Vector v = new Vector();
v.add(new String("valid string")); // intended
v.add(new Integer(4));           // unintended

// ClassCastException occurs during runtime
String s = (String)v.get(0);
Why Generics?

- **Problem:** for Collection element types
  - Compiler is unable to verify types of the elements
  - Assignment must have type casting
  - ClassCastException can occur during runtime

- **Solution:** Generics
  - Tell the compiler the type of the collection and let compiler do the type checking (during compile time)
  - Example: Compiler will check if you are adding Integer type entry to a String type collection
    - **Compile time detection of type mismatch**
  - Let the compiler do the casting
Generics:
More Usage Examples of Generics
Using Generic Classes: Example 1

• Instantiate a generic class to create type specific object
• In J2SE 5.0, all collection classes are rewritten to be generic classes

// Create a Vector of String type
Vector<String> vs = new Vector<String>();
vs.add(new Integer(5));  // Compile error!
vs.add(new String(“hello”));
String s = vs.get(0);  // No casting needed
Using Generic Classes: Example 2

- Generic class can have multiple type parameters
- Type argument can be a custom type

```java
// Create HashMap with two type parameters
HashMap<String, Mammal> map = new HashMap<String, Mammal>();
map.put("wombat", new Mammal("wombat"));
Mammal w = map.get("wombat");
```
Generics: Sub-typing
Sub-typing (Inheritance)

- Sub-typing between type arguments
- Sub-typing between generic classes
- Sub-typing between collection elements
Sub-typing between Type Arguments

• You can do this (using pre-J2SE 5.0 Java)
  > Object o = new Integer(5);
• You can even do this (using pre-J2SE 5.0 Java)
  > Object[] oa = new Integer[5];
• So you would expect to be able to do this in J2SE 5.0 (Well, you can't do this!!! You will get compile error!!!)
  > ArrayList<Object> ao = new ArrayList<Integer>();
  > Not being able to do above is counter-intuitive at the first glance
Sub-typing between Type Arguments

• Why this compile error? It is because if it is allowed, ClassCastException can occur during runtime – this is not type-safe

  > ArrayList<Integer> ai = new ArrayList<Integer>();
  > ArrayList<Object> ao = ai; // If it is allowed at compile time,
  > ao.add(new Object());    // This should be allowed
  > Integer i = ai.get(0);   // This would result in
    // runtime ClassCastException

• So there is no inheritance relationship between type arguments of a generic class
Sub-typing between Generic Classes

- The following code works
  > ArrayList<Integer> ai = new ArrayList<Integer>();
  > List<Integer> li2 = new ArrayList<Integer>();
  > Collection<Integer> ci = new ArrayList<Integer>();
  > Collection<String> cs = new Vector<String>(4);

- Inheritance relationship between generic classes themselves still exists (same as pre-J2SE 5.0)
  > Polymorphism still works
Sub-typing between Collection Elements

• The following code work

  > ArrayList<Number> an = new ArrayList<Number>();
  > an.add(new Integer(5));       // OK
  > an.add(new Long(1000L));     // OK
  > an.add(new String("hello")); // compile error

• Elements in a collection maintain inheritance relationship (same as pre-J2SE 5.0)
Generics:
Wild card
Why Wildcards? Problem

• Consider writing a utility method that prints out the elements of a collection
  > Different type of collection object is passed as an argument
• Here's how you write it in a pre-J2SE 5.0

```java
static void printCollection(Collection c) {
    Iterator i = c.iterator();
    for (k = 0; k < c.size(); k++) {
        System.out.println(i.next());
    }
}
```
Why Wildcards? Problem

• And here is a naive attempt at writing it using generics - Well.. You can't do this! Remember there is no inheritance relationship between type arguments!

```java
static void printCollection(Collection<Object> c) {
    for (Object o : c)
        System.out.println(o);
}

public static void main(String[] args) {
    Collection<String> cs = new Vector<String>();
    printCollection(cs); // Compile error
    List<Integer> li = new ArrayList<Integer>(10);
    printCollection(li); // Compile error
}
```
Why Wildcards? Solution

• Use Wildcard type argument <?>
• Collection<?> means Collection of unknown type
• Accessing entries of Collection of unknown type with Object type is safe

```java
static void printCollection(Collection<?> c) {
    for (Object o : c)
        System.out.println(o);
}

public static void main(String[] args) {
    Collection<String> cs = new Vector<String>();
    printCollection(cs); // No Compile error
    List<Integer> li = new ArrayList<Integer>(10);
    printCollection(li); // No Compile error
}
More on Wildcards

You cannot access entries of Collection of unknown type other than Object type

```java
static void printCollection(Collection<? super Object> c) {
    for (String o : c) // Compile error
        System.out.println(o);
}

global static void main(String[] args) {
    Collection<String> cs = new Vector<String>();
    printCollection(cs); // No Compile error
    List<Integer> li = new ArrayList<Integer>(10);
    printCollection(li); // No Compile error
}
```
More on Wildcards

- It isn't safe to add arbitrary objects to it however, since we don't know what the element type of c stands for, we cannot add objects to it.

```java
static void printCollection(Collection<?> c) {
    c.add(new Object()); // Compile time error
    c.add(new String());  // Compile time error
}

public static void main(String[] args) {
    Collection<String> cs = new Vector<String>();
    printCollection(cs); // No Compile error
    List<Integer> li = new ArrayList<Integer>(10);
    printCollection(li); // No Compile error
}
```
Bounded Wildcard

• If you want to bound the unknown type to be a subtype of another type, use Bounded Wildcard

```java
static void printCollection(
    Collection<? extends Number> c) {
    for (Object o : c)
        System.out.println(o);
}

public static void main(String[] args) {
    Collection<String> cs = new Vector<String>();
    printCollection(cs); // Compile error
    List<Integer> li = new ArrayList<Integer>(10);
    printCollection(li); // No Compile error
}
```
Generics:
Raw Type & Type Erasure
Raw Type

- Generic type instantiated with no type arguments
- Pre-J2SE 5.0 classes continue to function over J2SE 5.0 JVM as raw type

```java
// Generic type instantiated with type argument
List<String> ls = new LinkedList<String>();

// Generic type instantiated with no type
// argument - This is Raw type
List lraw = new LinkedList<>();
```
Type Erasure

• All generic type information is removed in the resulting byte-code after compilation
• So generic type information does not exist during runtime
• After compilation, they all share same class
  > The class that represents ArrayList<String>, ArrayList<Integer> is the same class that represents ArrayList
Type Erasure Example Code: True or False?

ArrayList<Integer> ai = new ArrayList<Integer>();
ArrayList<String> as = new ArrayList<String>();
Boolean b1 = (ai.getClass() == as.getClass());
System.out.println("Do ArrayList<Integer> and ArrayList<String> share same class? " + b1);
Generics:
Type Safe Code Again
What is “Type-safe Code” Again?

- If your code compiles without any compile errors and without warnings (or with warnings on safe operations), then you will never get a ClassCastException during runtime.

- Generics lets you build “Type-safe code” because the compiler guarantees that either:
  - the code it generates will be type-correct at run time, or
  - it will output a warning (if you are using Raw type) at compile time – in this case, you are responsible to make sure the warning is a benign one.
Generics: Mixing Generics and Non-Generics Code
What Happens to the following Code?

```java
import java.util.LinkedList;
import java.util.List;

public class GenericsInteroperability {

    public static void main(String[] args) {

        List<String> ls = new LinkedList<String>();
        List lraw = ls;
        lraw.add(new Integer(4));
        String s = ls.iterator().next();
    }

}
```
Compilation and Running

• Compilation results in a warning message
  > GenericsInteroperability.java uses unchecked or unsafe operations.

• If you don't take care of this warning and run the code, the ClassCastException will result during runtime
Generics:
Creating Your Own Generic Class
Defining Your Own Generic Class

```java
public class Pair<F, S> {
    F first;  S second;

    public Pair(F f, S s) {
        first = f;  second = s;
    }

    public void setFirst(F f) {
        first = f;
    }

    public F getFirst() {
        return first;
    }

    public void setSecond(S s) {
        second = s;
    }

    public S getSecond() {
        return second;
    }
}
```
Using Your Own Generic Class

```java
public class MyOwnGenericClass {

    public static void main(String[] args) {

        // Create an instance of Pair <F, S> class. Let's call it p1.
        Number n1 = new Integer(5);
        String s1 = new String("Sun");
        Pair<Number,String> p1 = new Pair<Number,String>(n1, s1);
        System.out.println("first of p1 (right after creation) = " + p1.getFirst());
        System.out.println("second of p2 (right after creation) = " + p1.getSecond());

        // Set internal variables of p1.
        p1.setFirst(new Long(6L));
        p1.setSecond(new String("rises"));
        System.out.println("first of p1 (after setting values) = " + p1.getFirst());
        System.out.println("second of p1 (after setting values) = " + p1.getSecond());
    }
}
```
Demo and Hands-on Labs

• Generics
  > www.javapassion.com/handsonlabs/javase5generics (online document)
  > www.javapassion.com/handsonlabs/1111_javase5generics.zip (hands-on lab zip file)
Sub-topics of Annotations

- What is and Why annotation?
- How to define and use Annotations?
- 3 different kinds of Annotations
- Meta-Annotations
How Annotation Are Used?

• Annotations are used to affect the way programs are treated mostly by tools

• Annotations are used by tools to produce derived files
  > Tools: Compiler, IDE, Runtime tools
  > Derived files: New Java code, deployment descriptor, class files
Ad-hoc Annotation-like Examples in pre-J2SE 5.0 Platform

- Ad-hoc Annotation-like examples in pre-J2SE 5.0 platform
  - Transient
  - Serializable interface
  - @deprecated
  - javadoc comments
  - Xdoclet

- J2SE 5.0 Annotation provides a standard, general purpose, more powerful annotation scheme
Why Annotation?

• Enables “declarative programming” style
  > Less coding since tool will generate the boiler plate code from annotations in the source code
  > Easier to change

• Eliminates the need for maintaining "side files" that must be kept up to date with changes in source files
  > Information is kept in the source file
  > Example: Eliminate the need of deployment descriptor
Annotation:
How do you define & use annotations?
How to “Define” Annotation Type?

• Annotation type definitions look similar to normal Java interface definitions
  > An at-sign (@) precedes the interface keyword

• With some differences
  > Each method declaration defines an element of the annotation type
  > Method declarations must not have any parameters or a throws clause
  > Return types are restricted to primitives, String, Class, enums, annotations, and arrays of the preceding types
  > Methods can have default values
Example: Annotation Type Definition

/**
 * Describes the Request-For-Enhancement (RFE) that led
 * to the presence of the annotated API element.
 */

public @interface RequestForEnhancement {
    int id();
    String synopsis();
    String engineer() default "[unassigned]";
    String date() default "[unimplemented]";
}
How To “Use” Annotation

• Once an annotation type is defined, you can use it to annotate declarations
  > class, method, field declarations

• An annotation is a special kind of modifier, and can be used anywhere that other modifiers (such as public, static, or final) can be used
  > By convention, annotations precede other modifiers
  > Annotations consist of an at-sign (@) followed by an annotation type and a parenthesized list of element-value pairs
Example: Usage of Annotation

```java
@RequestForEnhancement(
    id       = 2868724,
    synopsis = "Enable time-travel",
    engineer = "Mr. Peabody",
    date     = "4/1/3007"
)
public static void travelThroughTime(Date destination) {
    ... }
```

It is annotating `travelThroughTime` method
Annotation:
3 Types of Annotations
(in terms of Sophistication)
3 Different Kinds of Annotations

- Marker annotation
- Single value annotation
- Normal annotation
Marker Annotation

• An annotation type with no elements
  > Simplest annotation

• Definition

```java
/**
 * Indicates that the specification of the annotated API element
 * is preliminary and subject to change.
 */
public @interface Preliminary { }
```

• Usage – No need to have ()

```java
@Preliminary
public class TimeTravel { ... }
```
Single Value Annotation

• An annotation type with a single element whose name is “value”

• Definition

```java
/**
 * Associates a copyright notice with the annotated API element.
 */
public @interface Copyright {
    String value();
}
```

• Usage – can omit the element name and equals sign (=)

```java
@Copyright("2002 Yoyodyne Propulsion Systems")
public class SomeClass { ... }
```
Normal Annotation

• We already have seen an example

• Definition

```java
public @interface RequestForEnhancement {
    int id();
    String synopsis();
    String engineer() default "[unassigned]";
    String date(); default "[unimplemented]";
}
```

• Usage

```java
@RequestForEnhancement(
    id       = 2868724,
    synopsis = "Enable time-travel",
    engineer = "Mr. Peabody",
    date     = "4/1/3007"
)
public static void travelThroughTime(Date destination) { ... }
```
@Retention Meta-Annotation

• How long annotation information is kept
• Enum RetentionPolicy
  > SOURCE - SOURCE indicates information will be placed in the source file but will not be available from the class files
  > CLASS (Default)- CLASS indicates that information will be placed in the class file, but will not be available at runtime through reflection
  > RUNTIME - RUNTIME indicates that information will be stored in the class file and made available at runtime through reflective APIs
@Target Meta-Annotation

• Restrictions on use of this annotation
• Enum ElementType
  > TYPE, FIELD, METHOD, PARAMETER, CONSTRUCTOR, LOCAL_VARIABLE, ANNOTATION_TYPE, PACKAGE
Example: Definition and Usage of an Annotation with Meta Annotation

**Definition of Accessor annotation**

```java
@Target(ElementType.FIELD)
@Retention(RetentionPolicy.CLASS)
public @interface Accessor {
    String variableName();
    String variableType() default "String";
}
```

**Usage Example of the Accessor annotation**

```java
@Accessor(variableName = "name")
public String myVariable;
```
Reflection

• Check if MyClass is annotated with @Name annotation

```java
boolean isName =
    MyClass.class.isAnnotationPresent(Name.class);
```
Reflection

• Get annotation value of the @Copyright annotation

String copyright = MyClass.class.getAnnotation(Copyright.class).value();

• Get annotation values of @Author annotation

Name author = 
    MyClass.class.getAnnotation(Author.class).value();
String first = author.first();
String last = author.last();
Demo and Hands-on Labs

• Generics
  > www.javapassion.com/handsonlabs/javase5annotation (online document)
  > www.javapassion.com/handsonlabs/1107_javase5annotation.zip (hands-on lab zip file)
Concurrency
Concurrency Utilities: JSR-166

• Enables development of simple yet powerful multi-threaded applications
  > Like Collection provides rich data structure handling capability

• Beat C performance in high-end server applications
  > Fine-grained locking, multi-read single write lock

• Provide richer set of concurrency building blocks
  > `wait()`, `notify()` and `synchronized` are too primitive

• Enhance scalability, performance, readability and thread safety of Java applications
Why Use Concurrency Utilities?

- Reduced programming effort
- Increased performance
- Increased reliability
  - Eliminate threading hazards such as deadlock, starvation, race conditions, or excessive context switching are eliminated
- Improved maintainability
- Increased productivity
Concurrency Utilities

- Task Scheduling Framework
- Callable's and Future's
- Synchronizers
- Concurrent Collections
- Atomic Variables
- Locks
- Nanosecond-granularity timing
Concurrency: Task Scheduling Framework
Task Scheduling Framework

• Task scheduling framework supports
  > Submission of tasks to task handler
  > Scheduling of tasks
  > Execution of tasks
  > Control of asynchronous tasks according to a set of execution policies

• Task scheduling Java APIs
  > `Executor` is an interface
  > `ExecutorService` extends `Executor`
  > `Executors` is factory class for creating various kinds of `ExecutorService` implementations
Executor Interface

- **Executor** interface provides a way of de-coupling task submission from the mechanics of how each task will be run, including details of thread use, scheduling.

  > Each **Executor** implementation imposes some sort of limitation on how and when tasks are scheduled.

  > You can use a different Executor implementation for your task.

Executor executor = getSomeKindofExecutor();
executor.execute(new RunnableTask1());
executor.execute(new RunnableTask2());
Executor and ExecutorService

ExecutorService adds lifecycle management

```java
class Executor {
    void execute(Runnable command);
}

class ExecutorService extends Executor {
    void shutdown();
    List<Runnable> shutdownNow();
    boolean isShutdown();
    boolean isTerminated();
    boolean awaitTermination(long timeout, TimeUnit unit);

    // other convenience methods for submitting tasks
}
```
Creating ExecutorService Instance From Executors Utility Class

```java
public class Executors {
    static ExecutorService
        newSingleThreadedExecutor();

    static ExecutorService
        newFixedThreadPool(int n);

    static ExecutorService
        newCachedThreadPool(int n);

    static ScheduledExecutorService
        newScheduledThreadPool(int n);

    // additional versions specifying ThreadFactory
    // additional utility methods
}
```
Example: pre-J2SE 5.0 Code
Web Server—poor resource management

class WebServer {

    public static void main(String[] args) {
        ServerSocket socket = new ServerSocket(80);
        while (true) {
            final Socket connection = socket.accept();
            Runnable r = new Runnable() {
                public void run() {
                    handleRequest(connection);
                }
            };
            // This creates a new thread for each
            // request, which is not efficient.
            new Thread(r).start();
        }
    }
}
Example: Use Executors
Web Server—better resource management

class WebServer {
    Executor pool =
        Executors.newFixedThreadPool(7);

    public static void main(String[] args) {
        ServerSocket socket = new ServerSocket(80);

        while (true) {
            final Socket connection = socket.accept();
            Runnable r = new Runnable() {
                public void run() {
                    handleRequest(connection);
                }
            };
            pool.execute(r);
        }
    }
}
Concurrency:
Callables and Futures
Callable's and Future's: Problem (pre-J2SE 5.0)

If a new thread (callee thread) is started by a calling thread, there is currently no way to return a result from the callee thread to the calling thread without the use of a shared variable and appropriate synchronization.

This is complex and makes code harder to understand and maintain.
Callables and Futures

• Callable thread (Callee) implements Callable interface
  > Implement call() method rather than run()
• Calling thread (Caller) submits Callable object to Executor and then moves on
  > Through submit() not execute()
  > The submit() returns a Future object
• Calling thread (Caller) then retrieves the result using get() method of Future object
  > If result is ready, it is returned
  > If result is not ready, calling thread will block
class CallableExample
    implements Callable<String> {

    public String call() {
        String result = "The work is ended";

        /* Do some work and create a result */

        return result;
    }

}
Future Example (Caller)

```java
ExecutorService es =
    Executors.newSingleThreadExecutor();

Future<String> f =
    es.submit(new CallableExample());

/* Do some work in parallel */
try {
    String callableResult = f.get();
} catch (InterruptedException ie) {
    /* Handle */
} catch (ExecutionException ee) {
    /* Handle */
}
```
Concurrency:
Synchronizers: Semaphore
Semaphores

• Typically used to restrict access to fixed size pool of resources

• New Semaphore object is created with same count as number of resources

• Thread trying to access resource calls `acquire()`
  > Returns immediately if semaphore count > 0
  > Blocks if count is zero until `release()` is called by one of the users of the resource
  > `acquire()` and `release()` are thread safe atomic operations
Semaphore Example

```java
private Semaphore available;
private Resource[] resources;
private boolean[] used;

public Resource(int poolSize) {
    /* Initialise resource pool */
    available = new Semaphore(poolSize);
}

public Resource getResource() {
    /* Acquire a resource */
    try { available.acquire() } catch (IE) {}
}

public void returnResource(Resource r) {
    /* Return resource to pool */
    available.release();
}
```
Concurrency: Concurrent Collections
BlockingQueue Interface

• Provides thread safe way for multiple threads to manipulate collection

• `ArrayBlockingQueue` is simplest concrete implementation

• Full set of methods
  > `put()`
  > `offer()` [non-blocking]
  > `peek()` [non-blocking]
  > `take()`
  > `poll()` [non-blocking and fixed time blocking]
Blocking Queue Example: Logger placing log messages

private ArrayBlockingQueue messageQueue =
    new ArrayBlockingQueue<String>(10);

Logger logger = new Logger(messageQueue);

public void run() {
    String someMessage;
    try {
        while (true) {
            /* Do some processing */
            /* Blocks if no space available */
            messageQueue.put(someMessage);
        }
    } catch (InterruptedException ie) { }
Blocking Queue Example: Log Reader reading log messages

private BlockingQueue<String> msgQueue;

public LogReader(BlockingQueue<String> mq) {
    msgQueue = mq;
}

public void run() {
    try {
        while (true) {
            String message = msgQueue.take();
            /* Log message */
        }
    } catch (InterruptedException ie) {
        /* Handle */
    }
}
Concurrency: Atomic Variables
Atomics

- `java.util.concurrent.atomic`
  - Small toolkit of classes that support lock-free thread-safe programming on single variables

```java
AtomicInteger balance = new AtomicInteger(0);

public int deposit(integer amount) {
    return balance.addAndGet(amount);
}
```
Concurrency: Locks
Lock and ReentrantLock

• Lock interface
  > More extensive locking operations than synchronized block
  > Caution: No automatic unlocking like synchronized block – use try/finally to unlock
  > Advantage: Non-blocking access is possible using tryLock()

• ReentrantLock
  > Concrete implementation of Lock
  > Holding thread can call lock() multiple times and not block
  > Useful for recursive code
ReadWriteLock

• Has two locks controlling read and write access
  > Multiple threads can acquire the read lock if no threads have a write lock
  > If a thread has a read lock, others can acquire read lock but nobody can acquire write lock
  > If a thread has a write lock, nobody can have read/write lock
  > Methods to access locks
    
    ```
    rwl.readLock().lock();
    rwl.writeLock().lock();
    ```
class ReadWriteMap {
    final Map<String, Data> m = new TreeMap<String, Data>();
    final ReentrantReadWriteLock rwl =
        new ReentrantReadWriteLock();
    final Lock r = rwl.readLock();
    final Lock w = rwl.writeLock();
    public Data get(String key) {
        r.lock();
        try { return m.get(key) }
        finally { r.unlock(); }
    }
    public Data put(String key, Data value) {
        w.lock();
        try { return m.put(key, value); }
        finally { w.unlock(); }
    }
    public void clear() {
        w.lock();
        try { m.clear(); }
        finally { w.unlock(); }
    }
}
Demo and Hands-on Labs

• Generics
  > www.javapassion.com/handsonlabs/javase5concurrency (online document)
  > www.javapassion.com/handsonlabs/1108/javase5concurrency.zip (hands-on lab zip file)
JMX (Java Management Extension)
JMX Introduction

- Overview of JMX
- Instrument your Application
- Accessing your instrumentation remotely
What is JMX?

- Standard API for developing observable applications – JSR 3 and JSR 160
- Provides access to information such as:
  - Number of classes loaded
  - Virtual machine uptime
  - Operating system information
- Applications can use JMX for:
  - Management – changing configuration settings
  - Monitoring – getting statistics and notifications
JMX: Architecture
JMX Architecture

• Instrumentation Level
  > MBeans instrument resources, exposing attributes and operations

• Agent Level
  > MBean Server
  > Predefined services

• Remote Management
  > Protocol Adaptors and Standard Connectors enables remote Manager Applications
JMX Architecture

Clients: Management Consoles/Scripting

Management Level

Agent Level

Instrumentation Level

Connectors and Protocol Adapters

MBeanServer + Services

MBean

Business Objects

MBean

MBean Interface

Business Object

JVM

JVM

Courtsey: Borislav Iordanov
JMX Architecture

Remote Manager

Application

JMX Agent

Manages
JMX:
MBean
Managed Beans (MBeans)

- A MBean is a named *managed object* representing a *resource*
  - An application configuration setting
  - Device
  - Etc.

- A MBean can have
  - Attributes that can be read and/or written
  - Operations that can be invoked
  - Notifications that the MBean can broadcast
### A MBean Example

**CacheControlMBean**

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Type</th>
<th>Access</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>Used</td>
<td>int</td>
<td>R</td>
<td></td>
</tr>
<tr>
<td>Size</td>
<td>int</td>
<td>RW</td>
<td></td>
</tr>
</tbody>
</table>

**Operations**

- `save()`: void
- `dropOldest(int n)`: int

**Notifications**

- “com.example.config.change”
- “com.example.cache.full”
Standard MBean

• Standard MBean is the simplest model to use
  > Quickest and Easiest way to instrument static manageable resources

• Steps to create a standard MBean
  > Create an Java interface call FredMBean
  > Follows JavaBeans naming convention
  > Implement the interface in a class call Fred

• An instance of Fred is the MBean
Dynamic MBean

• Expose attributes and operations at Runtime
• Provides more flexible instrumentations
• Step to create Dynamic MBeans
  > Implements DynamicMBeans interface
  > Method returns all Attributes & Operations
• The same capability as Standard MBeans from Agent’s perspective
DynamicMBean Interface

getMBeanInfo():MBeanInfo
getAttribute(attribute:String):Object
getAttributes(attributes:String[]):AttributeList
setAttribute(attribute:Attribute):void
setAttributes(attributes:AttributeList):AttributeList
invoke(actionName:String, params:Object[], signature:String[]):Object
JMX Notification

- JMX notifications consists of the following
  - `NotificationEmitter` – event generator, typically your MBean
  - `NotificationListener` – event listener
  - `Notification` – the event
  - `NotificationBroadcasterSupport` – helper class

- Register with MBean server to receive events
JMX: MBean Server
MBean Server

com.example:type=CacheControl
MBean Server

- To be useful, an MBean must be registered in an MBean Server
- Usually, the only way to access MBeans is through the MBean Server
- You can have more than one MBean Server per Java™ Virtual Machine (JVM™ machine)
- But usually, as of Java SE 5, everyone uses the Platform MBean Server
  ```java
  java.lang.management.ManagementFactory.
  getPlatformMBeanServer()
  ```
JMX: Client Types
MBean Server: Local Clients

```java
MBeanServer mbs;
mbs.createMBean(...);
mbs.invoke(...);
mbs.queryMBeans(...);
```
MBean Server: Connector Client

```java
MBeanServerConnection mbs;
mbs.createMBean(...);
mbs.invoke(...);
mbs.queryMBeans(...);
```
MBean Server: Connector

• Connectors defined by the JMX Remote API (JSR 160)
  > Unrelated to the J2EE™ Connector Architecture
• Java SE architecture includes RMI and RMI/IIOP connectors
• JSR 160 also defines a purpose-built protocol, JMXMP
• Future work: a SOAP-based connector for the Web Services world (JSR 262)
MBean Server: Adaptor Client

Adaptor client

\[ \text{SNMP, HTML, ...} \]

Adaptor

MBean Server
Mapping SNMP or CIM to JMX API

- Generation not currently standard
  - proprietary solutions exist (Sun's is JDMK)
- Implementing semantics may mean mapping to another, “native” JMX API model
- Automated reverse mapping from JMX API to SNMP or CIM gives poor results
JMX:
JMX API Services
JMX API Services

- JMX API includes a number of pre-defined services
  > Services are themselves MBeans
- Monitoring service (thresholding)
  > javax.management.monitor
- Relation service (relations between MBeans)
  > javax.management.relation
- Timer service
  > javax.management.timer
- M-let service
  > javax.management.loading
JMX:
Steps of instrumenting Your Application
Steps for Instrumenting Your App

- **Create MBean's**
  - Define an MBean interface
  - Add attributes and operations
  - Add notifications
  - Implement MBean interface

- **Create JMX agent**
  - Provides a method to create and register your MBeans.
  - Provides access to the MBean server

- **Run the application with JConsole**
JMX: Demo – Running Anagram application with JMX support
Demo Scenario

• Anagram game is managed via JMX
  > Manage and monitor number of seconds it takes a user to provide a right answer
  > Monitor number of times a user has provided solutions
  > Subscribe event notification

• You can try this yourself
  > http://www.netbeans.org/kb/articles/jmx-tutorial.html
Java SE 5 Features

Sang Shin
sang.shin@sun.com
www.javapassion.com
Sun Microsystems Inc.