



Java Dynamic Management Kit 5.0 Tools Reference

Sun Microsystems, Inc.
4150 Network Circle
Santa Clara, CA 95054
U.S.A.

Part No: 816-4177-10
June 2002

Copyright 2002 Sun Microsystems, Inc. 4150 Network Circle, Santa Clara, CA 95054 U.S.A. All rights reserved.

Sun Microsystems, Inc. has intellectual property rights relating to technology embodied in this product. In particular, and without limitation, these intellectual property rights may include one or more of the U.S. patents listed at <http://www.sun.com/patents> and one or more additional patents or pending patent applications in the U.S. and other countries.

This product or document is protected by copyright and distributed under licenses restricting its use, copying, distribution, and decompilation. No part of this product or document may be reproduced in any form by any means without prior written authorization of Sun and its licensors, if any. Third-party software, including font technology, is copyrighted and licensed from Sun suppliers.

Parts of the product may be derived from Berkeley BSD systems, licensed from the University of California. UNIX is a registered trademark in the U.S. and other countries, exclusively licensed through X/Open Company, Ltd.

Sun, Sun Microsystems, the Sun logo, docs.sun.com, AnswerBook, AnswerBook2, Java, Java Coffee Cup logo, JDK, JavaBeans, JDBC, Java Community Process, JavaScript, J2SE, JMX and Solaris are trademarks, registered trademarks, or service marks of Sun Microsystems, Inc. in the U.S. and other countries. All SPARC trademarks are used under license and are trademarks or registered trademarks of SPARC International, Inc. in the U.S. and other countries. Products bearing SPARC trademarks are based upon an architecture developed by Sun Microsystems, Inc.

DOCUMENTATION IS PROVIDED "AS IS" AND ALL EXPRESS OR IMPLIED CONDITIONS, REPRESENTATIONS AND WARRANTIES, INCLUDING ANY IMPLIED WARRANTY OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE OR NON-INFRINGEMENT, ARE DISCLAIMED, EXCEPT TO THE EXTENT THAT SUCH DISCLAIMERS ARE HELD TO BE LEGALLY INVALID.

Federal Acquisitions: Commercial Software-Government Users Subject to Standard License Terms and Conditions.

Copyright 2002 Sun Microsystems, Inc. 4150 Network Circle, Santa Clara, CA 95054 U.S.A. Tous droits réservés

Sun Microsystems, Inc. a les droits de propriété intellectuels relatants à la technologie incorporée dans ce produit. En particulier, et sans la limitation, ces droits de propriété intellectuels peuvent inclure un ou plus des brevets américains énumérés à <http://www.sun.com/patents> et un ou les brevets plus supplémentaires ou les applications de brevet en attente dans les Etats - Unis et les autres pays.

Ce produit ou document est protégé par un copyright et distribué avec des licences qui en restreignent l'utilisation, la copie, la distribution, et la décompilation. Aucune partie de ce produit ou document ne peut être reproduite sous aucune forme, par quelque moyen que ce soit, sans l'autorisation préalable et écrite de Sun et de ses bailleurs de licence, s'il y en a. Le logiciel détenu par des tiers, et qui comprend la technologie relative aux polices de caractères, est protégé par un copyright et licencié par des fournisseurs de Sun.

Des parties de ce produit pourront être dérivées du système Berkeley BSD licenciés par l'Université de Californie. UNIX est une marque déposée aux Etats-Unis et dans d'autres pays et licenciée exclusivement par X/Open Company, Ltd.

Sun, Sun Microsystems, le logo Sun, docs.sun.com, AnswerBook, AnswerBook2, Java, le logo Java Coffee Cup, JDK, JavaBeans, JDBC, Java Community Process, JavaScript, J2SE, JMX et Solaris sont des marques de fabrique ou des marques déposées, ou marques de service, de Sun Microsystems, Inc. aux Etats-Unis et dans d'autres pays. Toutes les marques SPARC sont utilisées sous licence et sont des marques de fabrique ou des marques déposées de SPARC International, Inc. aux Etats-Unis et dans d'autres pays. Les produits portant les marques SPARC sont basés sur une architecture développée par Sun Microsystems, Inc.

LA DOCUMENTATION EST FOURNIE "EN L'ÉTAT" ET TOUTES AUTRES CONDITIONS, DECLARATIONS ET GARANTIES EXPRESSES OU TACITES SONT FORMELLEMENT EXCLUES, DANS LA MESURE AUTORISÉE PAR LA LOI APPLICABLE, Y COMPRIS NOTAMMENT TOUTE GARANTIE IMPLICITE RELATIVE A LA QUALITE MARCHANDE, A L'APTITUDE A UNE UTILISATION PARTICULIERE OU A L'ABSENCE DE CONTREFAÇON.



020610@4333



Contents

Preface	5
1 Proxy MBean Compiler (proxygen)	9
Starting the proxygen Compiler	10
proxygen Compiler Options	10
proxygen Compiler Example	11
▼ To Generate the Managed Object for the Simple Class	11
Output of the proxygen Compiler	12
Mapping Rules	12
Attributes	12
Operations	13
Methods in the Proxy Interface	13
Using the Generated Code	13
Modifying the Code Generated by the proxygen Compiler	14
2 SNMP MIB Compiler (mibgen)	15
Starting the mibgen Compiler	15
mibgen Options	16
Output From the mibgen Compiler	18
Representation of the Whole MIB	18
Representation of the Whole MIB in an SNMP OidTable	19
Classes Representing SNMP Groups	19
Skeletal MBeans Representing Groups	19
Metadata Files	20
Classes Representing SNMP Tables	20

Class Containing the SNMP View of a Table (Metadata Class)	20
Class Containing the MBean View of the Table	20
Skeletal MBeans Representing SNMP Table Entries	20
Metadata Files	21
Classes Representing SNMP Enumerated Types	22
Information Mapping	22
3 The HTML Protocol Adaptor	25
HTML Connections	26
Limitations of the HTML Protocol Adaptor	27
4 Tracing Mechanism	29
Receiving Trace and Debug Information	29
Specifying the Type of Trace and Debug Information	30
Specifying the Level of Trace and Debug Information	31
Index	33

Preface

The Java™ Dynamic Management Kit (DMK) 5.0 provides a set of Java classes and tools for developing management solutions. This product conforms to the Java Management Extensions (JMX™), v1.1 Maintenance Release, which defines a three-level architecture: resource instrumentation, dynamic agents and remote management applications. The JMX architecture is applicable to network management, remote system maintenance, application provisioning, and the new management needs of the service-based network.

The *Java Dynamic Management Kit 5.0 Tools Reference* document presents the development tools provided with the Java DMK. This book covers `proxygen` for generating manager-side proxy objects, `mibgen` for generating MBeans and relevant classes from SNMP MIBs, the tracing and debugging mechanism and the output of the HTML protocol adaptor. These tools can help you to develop management solutions to suit your requirements.

Who Should Use This Book

This book is aimed at developers who want to use the tools provided with the Java DMK.

You should be familiar with Java programming, the JavaBeans™ component model and the JMX specification.

This book is not intended to be an exhaustive reference. Management tutorials designed to demonstrate each of the management levels and how they interact are covered in the *Java Dynamic Management Kit 5.0 Tutorial*, and the complete Javadoc™ API definitions are provided in the online documentation package.

Before You Read This Book

To use the tool commands described in this book, you must have a complete installation of the Java Dynamic Management Kit on your system. Refer to the *Java Dynamic Management Kit 5.0 Installation Guide* for instructions on how to install the product components and configure your environment.

Related Documentation

Management concepts and product features are covered in the other books of the product documentation set, listed in Table P-1.

TABLE P-1 Related Documentation

Book Title	Part Number
<i>Getting Started with the Java Dynamic Management Kit 5.0</i>	816-4176-10
<i>Java Dynamic Management Kit 5.0 Tutorial</i>	816-4178-10
<i>Java Dynamic Management Kit 5.0 Installation Guide</i>	816-4179-10

These books are available online after you have installed the documentation package of the Java DMK. The online documentation also includes the Javadoc API for the Java packages and classes. To access the online documentation using any web browser, open the homepage corresponding to your operating environment. Table P-2 shows the location of the home page for the Solaris™ operating environment and Windows 2000 operating environment.

TABLE P-2 Accessing Online Documentation

Operating Environment	Home Page Location
Solaris environment	<i>installDir</i> /SUNWjdmk/jdmk5.0/index.html
Windows 2000 environment	<i>installDir</i> \SUNWjdmk\jdmk5.0\index.html

In these file names, *installDir* refers to the base directory (or folder) of your Java DMK installation. In a default installation procedure, *installDir* is:

- /opt on the Solaris platform
- C:\Program Files on the Windows 2000 platform

These conventions are used throughout this book whenever referring to files or directories (folders) that are part of the installation.

The Java DMK relies on the JMX architecture. The specification document, *Java Management Extensions Instrumentation and Agent Specification v1.1* (Maintenance Release, March 2002), is provided in the product documentation package, under the file name `jmx_instr_agent.pdf`.

How This Book Is Organized

This book describes the development tools provided with the Java DMK and explains how to use them. It is divided into the following chapters:

- Chapter 1: “Proxy MBean Compiler (`proxygen`)”
 - Chapter 2: “SNMP MIB Compiler (`mibgen`)”
 - Chapter 3: “HTML Protocol Adaptor”
 - Chapter 4: “Tracing Mechanism”
-

Accessing Sun Documentation

You can view and print a broad selection of Sun™ documentation, including localized versions, at:

<http://www.sun.com/documentation/>

You can also purchase printed copies of select Sun documentation from iUniverse, the Sun documentation provider, at:

<http://corppub.iuniverse.com/marketplace/sun/>

Typographic Conventions

The following table describes the typographic changes used in this book.

Typeface or Symbol	Meaning	Example
AaBbCc123	The names of commands, files, and directories; on-screen computer output	Edit your <code>.login</code> file. Use <code>ls -a</code> to list all files. <code>machine_name%</code> you have mail.
AaBbCc123	What you type, contrasted with on-screen computer output	<code>machine-name% su</code> Password:
<i>AaBbCc123</i>	Command-line placeholder: replace with a real name or value	To delete a file, type <code>rm filename</code> .
<i>AaBbCc123</i>	Book titles, new words, or terms, or words to be emphasized.	Read Chapter 6 in <i>User's Guide</i> . These are called <i>class</i> options. You must be <i>root</i> to do this.

Shell Prompts in Command Examples

The following table shows the default system prompt and superuser prompt for the C shell, Bourne shell, and Korn shell.

Shell	Prompt
C shell prompt	<code>machine-name%</code>
C shell superuser prompt	<code>machine-name#</code>
Bourne shell and Korn shell prompt	<code>\$</code>
Bourne shell and Korn shell superuser prompt	<code>#</code>

Proxy MBean Compiler (proxygen)

You can use the `proxygen` compiler supplied with the Java Dynamic Management Kit (DMK) to generate a proxy MBean from its corresponding MBean. A proxy MBean is an image of an agent-side MBean and exists on the manager side. The `proxygen` compiler allows you to customize your proxy MBeans depending on how you want to use them in your management application. For more information regarding the relationship between MBeans and proxy MBeans, refer to *Getting Started with the Java Dynamic Management Kit 5.0*, which is part of the documentation set.

The `proxygen` compiler takes the compiled Java class of an MBean and generates the Java interface and Java proxies. The Java proxies consist of Java source code that implements the interface. To develop a Java manager with code generated by `proxygen`, you call the methods of the proxy MBean's interface.

Options of the `proxygen` compiler enable you to modify the characteristics of the proxies you generate from an MBean. For example, with some options, you can generate read-only or read-write proxies. By generating from the same MBean a set of proxies with different characteristics, you can develop a Java manager whose behavior is modified at runtime, depending on which proxies are loaded. For example, when the read-only proxies are loaded, the Java manager cannot modify properties in the MBean.

A proxy MBean consists of two components:

- A Java interface that defines which operations of the MBean are accessible to a Java manager
- A Java class that implements the operations defined in the Java interface

For example, if you have an MBean `MyClass`, the `proxygen` compiler gives you a proxy MBean that consists of the following files:

- `MyClassProxyMBean.java` - the Java interface
- `MyClassProxy.java` - the Java class

The `proxygen` compiler generates Java source code, not compiled Java classes. For your proxy MBeans to be accessible to a Java manager, you must compile the files that `proxygen` generates, and make sure that the compiled Java classes are stored at the location specified by the `CLASSPATH` environment variable of the manager, or are accessible through the class loader of the manager.

Note – Classes marked “deprecated” in the Java DMK 4.2 have been removed from version 5.0.

Starting the `proxygen` Compiler

To start `proxygen`, type the command for your operating environment:

- In a Solaris operating environment:

```
prompt% installDir/SUNWjdmk/jdmk5.0/bin/proxygen <options> <classes>
```

- In a Windows 2000 operating environment:

```
C:\> installDir\SUNWjdmk\jdmk5.0\bin\proxygen <options> <classes>
```

Alternatively, invoke the `java com.sun.jdmk.tools.ProxyGen` class by first invoking `java com.sun.jdmk.tools.ProxyGen <options> <classes>`. Provide the class name without the `.class` extension.

`proxygen` Compiler Options

The `proxygen` command takes options, as follows:

`proxygen options classes`

The *options* include:

- | | |
|------------------------------|--|
| <code>-d dir</code> | Specifies a destination directory for the generated code. |
| <code>-ro</code> | Generates read-only proxy MBeans. Calling setter methods of these read-only proxies raises a <code>com.sun.jdmk.RuntimeProxyException</code> . |
| <code>-tp pkgName</code> | Generates code in the target package specified by <i>pkgName</i> . |
| <code>-classpath path</code> | Specifies a class path to use for locating the class to compile. By default, the system class path is used. |

`-help`

Prints a message that briefly describes each `proxygen` option.

proxygen Compiler Example

The following example shows how to generate the managed object for the `Simple` class and `SimpleMBean` interface, which is provided as an example with the Java DMK. You must first call the Java compiler on the source code to obtain its `.class` file before using the `proxygen` compiler to generate the managed object. Finally, you must compile the Java code generated by the `proxygen` compiler.

The source code for the `Simple` class is contained in the `installDir/SUNWjdmk/jdmk5.0/examples/MonitorMBean` directory, where `installDir` is the directory under which the Java DMK was installed.

Note – On the Solaris platform, the `installDir` file hierarchy is not writable by default. In this case you must copy the `Simple.java` and `SimpleMBean.java` files to a directory where you have write permissions.

▼ To Generate the Managed Object for the `Simple` Class

1. Set your `CLASSPATH` as follows:

- In the Solaris operating environment:

```
prompt% installDir/SUNWjdmk/jdmk5.0/lib/jdmkrt.jar and jdmktk.jar
```

- In a Windows 2000 operating environment:

```
installDir\SUNWjdmk\jdmk5.0\1.2\lib\jdmkrt.jar and jdmktk.jar
```

2. Type the following commands:

```
prompt% javac Simple.java SimpleMBean.java
prompt% ../bin/proxygen/proxygen -classpath . Simple
Destination directory set to ../
```

```
Starting compilation of Simple.
```

```
Starting to generate stub SimpleProxy.java for class Simple
Starting to generate MBean interface SimpleProxyMBean.java for class Simple
```

Note – Proxy MBeans generated using `proxygen` in Java DMK 4.2 must be regenerated to run in Java DMK 5.0, because some of the methods used in version 4.2 have been deprecated.

Output of the `proxygen` Compiler

For an MBean defined in the Java class *BeanName*, the `proxygen` compiler generates:

- A Java interface (*BeanNameProxyMBean*), which defines the methods of the MBean that are accessible to a Java manager
- A Java proxy (*BeanNameProxy*), which implements the methods defined in the Java interface *BeanNameProxyMBean*

For example, when an MBean representing a Java class named `Simple` is compiled, the `proxygen` compiler generates the source code of:

- A Java interface named `SimpleProxyMBean`
- A Java class named `SimpleProxy`, which implements the `SimpleProxyMBean` interface

Mapping Rules

The `proxygen` compiler uses the Java Reflection API for analyzing an MBean and generating its associated proxy MBean. It parses an MBean using the JMX-specific design patterns. The mapping rules that `proxygen` uses for generating the proxy MBean are described in the following subsections.

Attributes

The `proxygen` compiler generates code only for exposed operations of the MBean itself. Each attribute of the MBean is present in the proxy MBean with the same accessor getter and setter methods. Therefore, if an attribute is read-only in the MBean, the property is read-only in the generated proxy MBean.

Operations

In addition to the attribute accessors, the `proxygen` compiler generates code only for exposed operations of the MBean itself.

Methods in the `Proxy` Interface

The proxy MBeans that the `proxygen` compiler generates also contain methods that are not present in the MBean. Rather, they are defined in the Java interface `com.sun.jdmk.Proxy`. The proxy MBean that is generated implements this interface. These methods are public methods that do not follow the design patterns defined by the JavaBeans component model.

These methods provide additional functionality for proxy MBeans and the management applications that instantiate them. Their purpose is to twofold:

- To make sure that the information provided by the proxy MBean is up-to-date. For example, methods are defined for binding and unbinding a proxy MBean from a remote MBean server.
- To get the object name and class of the remote MBean represented by the proxy MBean.

Using the Generated Code

The `proxygen` compiler generates Java source code that you use for developing Java managers. To develop a Java manager by using code generated by the `proxygen` compiler, you use the `RemoteMBeanServer` interface. Using this interface enables you to develop Java managers without having to modify the code that the `proxygen` compiler generates. However, you can also modify the code that it generates if you want to define a specific view of an MBean.

The `proxygen` compiler generates Java source code, not compiled Java classes. For your proxy MBeans to be accessible to a Java manager, you must compile the files that the `proxygen` compiler generates, and make sure that the compiled Java classes are stored at a location specified in the `CLASSPATH` environment variable of the manager or are accessible through the class loader of the manager.

Modifying the Code Generated by the proxygen Compiler

If you want to define a specific view of an MBean, you can modify the generated code. To ensure that the modified code remains consistent with the MBean it represents, modify only the proxy and not the interface.

SNMP MIB Compiler (`mibgen`)

The Java Dynamic Management Kit (DMK) provides a toolkit for developing SNMP agents and managers. This toolkit includes the SNMP MIB Compiler, `mibgen`, which is used for compiling SNMP MIBs into Java source code for agents and managers. The `mibgen` tool is a Java technology-based SNMP MIB compiler that takes an SNMP MIB as input and outputs a set of Managed Beans. These MBeans can be customized to implement the MIBs, enabling the Java DMK agent to be managed by an SNMP manager. You can use standard MBeans, model MBeans, dynamic MBeans and open MBeans in conjunction with the `mibgen` compiler.

The `mibgen` compiler is able to process:

- Tables with cross-references indexed across several MIBs
- MIBs that contain SMIV1 and SMI v2 definitions mixed into the same MIB module; a MIB file can contain several MIB modules
- Nested groups
- Default value variables
- Row status (tables controlled by a columnar object obeying the `RowStatus` convention, as defined in RFC 2579)

Starting the `mibgen` Compiler

To start the `mibgen` compiler, type the command for your operating environment:

- In the Solaris operating environment:

```
prompt% installDir/SUNWjdmk/jdmk5.0/bin/mibgen [options] mib1 ... mibN
```

- In a Windows 2000 operating environment:

```
C:\> installDir\SUNWjdmk\jdmk5.0\bin\mibgen [options] mib1 ... mibN
```

mibgen Options

To invoke the `java.com.sun.jdmk.tools.MibGen` class, you need to invoke `java.com.sun.jdmk.tools.MibGen <options> <mib files>`

`mibgen options mib files`

where *options* includes:

<code>-n</code>	Parses the MIB files without generating code.
<code>-d dir</code>	Generates code in the specified target directory.
<code>-tp pkgName (target package)</code>	Generates code within the specified Java package.
<code>-desc</code>	Includes the <code>DESCRIPTION</code> clause of <code>OBJECT-TYPE</code> as comment in generated code.
<code>-mo</code>	(<i>manager-only</i>) Generates code for the SNMP manager only, that is the metadata file for the MIB variables (<code>SnmpOidTable</code> file). By default, the <code>mibgen</code> compiler generates code for both SNMP agents and managers. By selecting the <code>-mo</code> option, you enable the <code>mibgen</code> compiler to generate code for only the manager and not for agents. The <code>-mo</code> option is incompatible with the <code>-n</code> option.
<code>-mc</code>	(<i>MIB-CORE</i>) Does not use the default <code>MIB-CORE</code> definitions file provided with the Java DMK. In this case, the user must specify the <code>MIB-CORE</code> definitions file as one of the MIB files (for example, <code>java.com.sun.jdmk.tools.MibGen -mc mib my_mib_core</code>).
<code>-a</code>	Generates code for all the MIB files. Without this option, the Java code is generated only for the first MIB file. In this case, the following MIB files are simply used to resolve some definitions of the first MIB file.
<code>-p prefix</code>	Uses the specified prefix for naming generated classes.
<code>-g</code>	Generates a <i>generic</i> version of the metadata that will access the MBeans through the MBean server instead of using a direct reference. This enables you

	to plug in dynamic MBeans, instead of the generated standard MBean skeletons.
<code>-gp prefix</code>	Uses the specified prefix to name the <i>generic</i> metadata classes. For example, the metadata class for group system will be named <code>SystemprefixMeta</code> . Default is no prefix.
<code>-sp prefix</code>	Uses the specified prefix to name the <i>standard</i> metadata classes. For example: the metadata class for group system will be named <code>SystemprefixMeta</code> . Default is no prefix.
<code>-help</code>	Prints a usage message explaining how to invoke the compiler, as follows: <ul style="list-style-type: none"> ■ <code><mib files></code>: By default <code>mibgen</code> generates code only for the modules specified in the <i>first</i> file. The other files are only used for closure analysis except when the <code>-a</code> is specified.

The order followed by the `mibgen` compiler to find the `MIB_CORE` definitions file is as follows:

1. The user `MIB_CORE` definitions file specified in the MIB files using the `-mc mibgen` option.
2. The user command line parameter specified using the `-Dmibcore.file` Java property.
3. The default `MIB_CORE` definitions file provided with Java DMK in `installDir/etc/mibgen` (`mib_core.txt`). To succeed, you must be able to derive the installation directory from the `CLASSPATH` environment variable. Otherwise, the `mibgen` compiler will look for the `mib_core.txt` file in `currentDir/etc/mibgen`.
4. When using generic metadata (`-g` option), backward compatibility is not ensured. Using the `-g` option has generic advantages (MBeans are accessed through the MBean server, and any kind of MBeans can be plugged in), but slightly reduces the overall performance.

Note – SNMP MIB implementations generated using the `mibgen` compiler from Java DMK 4.2 can run and recompile on Java DMK 5.0 without modification.

Output From the `mibgen` Compiler

The `mibgen` compiler also generates the Java source code required for representing a whole MIB in an SNMP manager. The `mibgen` compiler parses an SNMP MIB and generates the following:

- For agents and managers:
 - A class mapping symbolic names with object identifiers of MIB variables
- For agents:
 - An MBean representing the whole MIB
 - Classes representing SNMP groups or entities as MBeans (and their corresponding metadata classes)
 - Classes representing SNMP tables
 - Classes representing SNMP enumerated types

MBeans generated by the `mibgen` compiler need to be updated to provide the definitive implementation. For more information, see the corresponding section in the *Java Dynamic Management Kit 5.0 Tutorial*.

Representation of the Whole MIB

The `mibgen` compiler generates a Java file that represents and initializes the whole MIB. This class extends the class `SnmpMib`, which is an abstract Java class in the `com.sun.jdmk.snmp.agent` package and is a logical abstraction of an SNMP MIB. The SNMP adaptor uses the `SnmpMib` class to implement agent behavior. The generated MIB file offers factory methods for group MBeans.

▼ To Implement the Generated MIB File

1. **Subclass the group MBean skeleton you want to implement, completing the getter, checker, and setter methods.**
2. **Subclass the generated MIB file.**
3. **Redefine the factory methods for the group MBeans you have implemented, ensuring that they instantiate the actual implementation class and not the skeleton.**

The `mibgen` compiler uses the module name specified in the MIB definition to name files representing whole MIBs. The compiler removes special characters and replaces them with an underscore character (`_`).

Representation of the Whole MIB in an SNMP OidTable

The `mibgen` compiler generates a Java file that contains the code required for representing a whole MIB in an SNMP manager `OidTable`. This class extends the `com.sun.jdmk.snmp.snmpOidTableSupport` class, which implements the `javax.management.snmp.snmpOidTable` class and maintains a database of MIB variables. A name can be resolved against the database. This file can be used by both the agent and the manager API. It contains metadata definitions for the compiled MIB. The metadata can then be loaded into the SNMP `OidTable`.

The file is always generated when `mibgen` is invoked, and is called `MIBnameOidTable`. The `-mo` option generates *only* the `MIBnameOidTable` file. This file is the only file generated for SNMP managers. All other files are dedicated to the SNMP agents.

Classes Representing SNMP Groups

For each SNMP group defined in the MIB, the `mibgen` compiler generates:

- A skeletal MBean, with its interface
- A metadata file

Skeletal MBeans Representing Groups

The `mibgen` compiler generates an MBean for each group defined in the MIB. These skeletal MBeans need to be completed by adding implementation-specific code (access methods). The generated code is initialized with default values for the various MIB variables. If the MIB specifies a default value for an SNMP variable, this value is used to initialize the corresponding variable in the MBean skeleton. Therefore, if you compile the generated code directly, you obtain a running agent. In this case, values returned by the agent when querying the MIBs will be default values or meaningless values, if no default value has been provided in the MIB file for the variable.

The `mibgen` compiler uses the group names specified in the MIB definition to name MBeans generated from groups.

Metadata Files

In addition to generating skeletal MBeans to represent each group, the `mibgen` compiler generates a metadata file. The metadata file contains Java source code that provides the SNMP view of the MBean. Metadata files do not need to be modified. For metadata files the `Meta` suffix is added.

Classes Representing SNMP Tables

For each SNMP table defined in the MIB, the `mibgen` compiler generates:

- A class containing the view of the table
- A metadata file corresponding to the SNMP table itself
- A skeletal MBean representing a table entry, with its interface
- A metadata file corresponding to the skeletal MBean

Class Containing the SNMP View of a Table (Metadata Class)

The metadata class containing the SNMP view of a table contains all the management of the table index. The class is also prefixed with `Table`, followed by the name of the table.

Class Containing the MBean View of the Table

The class containing the MBean view of a table enables you to dynamically add or remove entries from the table. It contains callbacks and factory methods that enable you to instantiate or delete entries upon receiving requests from a remote SNMP manager (see `RowStatus` example). This class also is prefixed with `Table`, followed by the name of the table.

Skeletal MBeans Representing SNMP Table Entries

For each table in a MIB, the `mibgen` compiler generates an MBean representing a table entry. These skeletal MBeans must be completed by adding implementation specific code (access methods). The generated code is initialized with default values for table-entry fields. Therefore, if you compile the generated code directly, you obtain a

running agent. In this case, values returned by the agent when querying the MIBs are not meaningful. The `mibgen` compiler uses the entry names specified in the MIB definition to name MBeans generated from table entries.

Note – Remote creation of table entries is disabled by default, for security reasons. You can dynamically enable and disable remote creation of table entries by calling the `setCreationEnabled` operation on the generated MBean-like object.

The `RowStatus` convention, defined in RFC 2579, is fully supported by the code generator. When a table is defined using `v2`, and if it contains a control variable with row status syntax, the `mibgen` compiler generates a set of methods allowing this table to be remotely controlled by this variable. However, the remote creation and deletion of rows remains disabled by default.

Table objects are divided into two categories:

- A metadata class
- An MBean-like object

When remote table-entry creation is enabled, and the creation of a new table is requested, a factory method is called on the MBean-like object to instantiate the new table entry. By default, an instance of the skeleton class for that table entry is instantiated.

▼ To Instantiate Your Own Implementation Class

1. **Subclass the MBean-like object to redefine the factory method for remote entry creation.**
2. **Redefine this factory method so that it returns an instance of your implementation class, instead of the default skeleton.**
3. **Subclass the group MBean, to which this table belongs, to instantiate your new MBean-like object, instead of the generated default object.**

This is demonstrated in the `RowStatus` example.

Metadata Files

In addition to generating skeletal MBeans to represent each table entry, the `mibgen` compiler generates a Java file containing the SNMP view of the MBean. Metadata files do not need to be modified. For metadata files, the `Meta` suffix is added.

Classes Representing SNMP Enumerated Types

The `mibgen` compiler generates a specific class for each enumerated type defined in the MIB. This class contains all the possible values defined in the enumerated type. The generated class extends the generic class `Enumerated`, defined in the `com.sun.jdmk` package. The HTML adaptor can use the `Enumerated` class to display all the labels contained in an enumeration. The `mibgen` compiler can handle enumerated types defined as part of a type definition or in-line definition.

Generated code representing SNMP enumerated types is prefixed with `Enum` followed by the type name or the variable name for inline definition.

Note – The `mibgen` compiler has an option `-p prefix` that you can use to prefix the names of all generated files with a specific string.

For example, in MIB II, TCP connection states are represented by an enumeration containing all the possible states for a TCP connection. The `mibgen` compiler generates a Java class named `EnumTcpConnState` to represent the enumeration.

Information Mapping

For each group defined in your MIB, the `mibgen` compiler generates an MBean. Each variable in the group is represented as a property of the MBean. If the MIB allows read access to a variable, the `mibgen` compiler generates a getter method for the corresponding property. If the MIB allows write access to a variable, the `mibgen` compiler generates a setter method for the property. Tables are seen as indexed properties whose type corresponds to the table entry type. The SNMP view of the table is maintained by a specific table object contained in the generated MBean. The `mibgen` compiler maps the MIB variable syntax to a well-defined Java type.

The MBeans that the `mibgen` compiler generates do not have any dependencies on specific SNMP objects. Therefore, they can be easily browsed or integrated into the various Java DMK components. The translation between the SNMP syntax and the MBean syntax is performed by the metadata.

Note – To change the Java type of a specific MIB variable within a generated MBean, edit the metadata file associated with the group that contains the variable.

MBeans generated by the `mibgen` compiler must be updated to provide the definitive implementation. The generated code is an operational agent. Thus, it can be compiled, run, and tested without any modification. If MBeans are generated too often, use subclasses to reduce major modifications in your code.

Example 2-1 shows how to implement a skeletal MBean.

EXAMPLE 2-1 Implementing a Skeletal MBean

```
public class g1 implements g1MBean, Serializable {
    protected Integer myVar = new Integer (1);
    public g1(SnmpMib myMib) {
        {
        public Integer getMyVar() throws SnmpStatusException {
            return myVar;
        }

        public void setMyVar(Integer x) throws SnmpStatusException {
            myVar = x;
        }
    }
}
```

You must modify the skeletal MBean to implement your MIB behavior. For information about developing an SNMP agent, SNMP Manager, SNMP API, and SNMP Proxy, see the corresponding sections in the *Java Dynamic Management Kit 5.0 Tutorial*.

The HTML Protocol Adaptor

A protocol adaptor provides access to MBeans through a communications protocol. It enables management applications to perform management operations on a Java Dynamic Management Kit (DMK) agent. For a Java DMK agent to be manageable, it must contain at least one adaptor. However, an agent can contain *many* adaptors, allowing it to be managed remotely through various protocols.

The HTML protocol adaptor acts as an HTML server. It enables web browsers to access agents through the HTTP communications protocol, to manage all MBeans in the agent. It can be used as a tool for debugging and speeding the development of agents. The HTML protocol adaptor is implemented as a dynamic MBean.

The HTML protocol adaptor provides the following main HTML pages for managing MBeans in an agent:

- *Agent View*: Provides a list of object names of all the MBeans registered in the agent.
- *Agent Administration*: Registers and unregisters MBeans in the agent.
- *MBean View*: Reads and writes MBean attributes and performs operations on MBeans in the agent.

The HTML page displayed is generated by the HTML adaptor and enables you to perform the following operations on MBeans in the agent:

- Reading or writing the attributes of an MBean instance
- Performing an operation on an MBean instance
- Instantiating an MBean
- Deleting an MBean

HTML Connections

The HTML adaptor is an instance of the `com.sun.jdmk.comm.HtmlAdaptorServer` MBean. Your agent application must instantiate this class, register the MBean, and explicitly start it by invoking its `start` method to allow HTML connections. When the HTML protocol adaptor is started, it creates a TCP/IP socket, listens for manager connections, and waits for incoming requests. By default, the HTML adaptor listens for incoming requests on port 8082. You can change this default value by specifying a port number:

- In the object constructor
- By using the `setPort` method before starting the adaptor

If a manager tries to connect, the `HtmlAdaptorServer` creates a thread which receives and processes all subsequent requests from this manager. The number of managers is limited by the `maxActiveClientCount` property. The default value of the `maxActiveClientCount` is 10.

When an `HtmlAdaptorServer` is stopped, all current HTTP connections are interrupted (some requests might be terminated abruptly), and the TCP/IP socket is closed. The `HtmlAdaptorServer` can perform user authentication. The `addUserAuthenticationInfo` method and the `removeUserAuthenticationInfo` method can be used to manage users and their corresponding authentication information. The HTML server uses the Basic Authentication Scheme, as defined in RFC 1945, section 11.1, to authenticate clients connecting to the server.

Before connecting a web browser to an agent, you must make sure that:

- The agent is running on a system that you can access by using the HTTP protocol
- The agent contains an instance of an HTML adaptor
- The compiled MBean classes are stored at a location specified in the `CLASSPATH` environment variable of the agent

To connect a browser to an agent, open the page given by the following URL in a web browser:

```
http://host:port
```

where:

- *host* is the host name of the machine on which the agent is running.
- *port* is the port number used by the HTML adaptor in the agent. The default port number is 8082.

Limitations of the HTML Protocol Adaptor

The HTML protocol adaptor has the following limitations:

- The minimum value for the reload period is 5 seconds (0 defaults to no reloading).
- Arrays of class are always displayed in read-only mode.
- Arrays of dimension 2 and higher are not fully expanded.
- Supported attribute types (for reading and writing) are as follows:
 - `boolean boolean[] Boolean Boolean[]`
 - `byte Byte Byte[]`
 - `char char[] Character Character[]`
 - `Date Date[]` (for example, July 21st, 2002 8:49:04 PM CEST)
 - `double double[] Double Double[]`
 - `float float[] Float Float[]`
 - `int int[] Integer Integer[]`
 - `long Long Long[]`
 - `Number`
 - `javax.management.ObjectName javax.management.ObjectName[]`
 - `short Short Short[]`
 - `String String[]`
 - `com.sun.jdkm.Enumerated`: supported for readable attributes. Because `com.sun.jdkm.Enumerated` is an abstract class, only write-only attributes whose actual subclass is declared in the signature of its setter can be set through the HTML adaptor.

Note – For unsupported read-only attribute types, if not null, the `toString()` method is called. If the getter of a read-only or a read-write attribute throws an exception, the thrown exception name and message are displayed, and this attribute cannot be set through the HTML adaptor even if it is a read-write attribute.

- Supported operation and constructor parameter types are as follows:
 - `boolean Boolean`
 - `byte Byte`
 - `char Character`

- Date (for example, July 21st, 2002 8:49:04 PM CEST)
- double Double
- float Float
- int Integer
- long Long
- Number
- javax.management.ObjectName
- short Short
- String

Note – When reading a value of type `Number` the server tries to convert it first to an `Integer`, then a `Long`, then a `Float`, and finally a `Double`, stopping at the first that succeeds.

Use the "Reload" button displayed in the HTML page of an MBean view rather than the reload button of the web browser. Otherwise you might reinvoke the setters of all attributes, if this was your last action.

Tracing Mechanism

This chapter explains how to use the tracing mechanism to help you trace or debug the Java Dynamic Management Kit (DMK) API. The tracing mechanism uses the notification mechanism to distribute the trace and debug information, giving you internal runtime information. You can specify the information type and level of trace and debug information you want to receive. To receive trace and debug information you must add a notification listener to the class `com.sun.jdmk.TraceManager`.

You control the tracing by defining the trace properties specific to the Java DMK. Three factors affect tracing:

- Various components that send trace messages
- Level of detail
- Output destination

The `com.sun.jdmk.trace.Trace` class is used to emit trace messages. All the classes of the Java DMK use this `Trace` class for sending traces. You can use the `Trace` class in your own code for producing debug traces for your own classes.

The `com.sun.jdmk.TraceManager` class provides methods for receiving trace and debug messages. Options provided by the `TraceManager` class are described in the following sections:

- “Receiving Trace and Debug Information” on page 29
- “Specifying the Type of Trace and Debug Information” on page 30
- “Specifying the Level of Trace and Debug Information” on page 31

Receiving Trace and Debug Information

The `com.sun.jdmk.TraceManager` class uses the notification mechanism to distribute the information. You must add a notification listener to receive information (see example Example 4-1). There are two ways to receive trace information:

- Adding a notification listener with a filter in the code. It is possible to have more than one notification listener but with different filters. With `TraceFilter`, you can specify the type and level of information you want to receive. See Example 4-3 and Example 4-2.
- Specifying system properties in the command to start the Java interpreter when you run a class. In this case, the code of the class must include a call to the `TraceManager` method. When the `TraceManager` method is called, all the previously enabled trace and debug information are disabled. Only the properties currently defined when the method is called are enabled.

EXAMPLE 4-1 Creating a Notification Listener

```
// Create a listener and save all info to the file /tmp/trace
TraceListener listener = new TraceListener("/tmp/trace");
```

EXAMPLE 4-2 Creating a Trace Filter

```
// create a trace filter with LEVEL_DEBUG and INFO_ALL/
TraceFilter filter = new TraceFilter(Trace.LEVEL_DEBUG, Trace.INFO_ALL);
```

EXAMPLE 4-3 Adding the Notification Listener to the class

```
// add the listener to the class Trace/
TraceManager.addNotificationListener(listener, filter, null);
```

Specifying the Type of Trace and Debug Information

It is possible to specify the type of trace and debug information you want to receive. The following types are specified:

- `INFO_MBEANSERVER`: information about the MBean server
- `INFO_MLET`: information from an m-let service
- `INFO_MONITOR`: information from a monitor
- `INFO_TIMER`: information from a timer
- `INFO_ADAPTOR_CONNECTOR`: information concerning all adaptors and connectors
- `INFO_ADAPTOR_HTML`: information from an HTML adaptor
- `INFO_CONNECTOR_RMI`: information from a RMI connector
- `INFO_CONNECTOR_HTTP`: information from HTTP connectors
- `INFO_CONNECTOR_HTTPS`: information from HTTPS connectors
- `INFO_ADAPTOR_SNMP`: information from an SNMP adaptor
- `INFO_DISCOVERY`: information from a discovery service
- `INFO_SNMP`: information from an SNMP manager service
- `INFO_NOTIFICATION`: information from notification mechanism

- `INFO_HEARTBEAT`: information from heartbeat mechanism
- `INFO_RELATION`: information from relation service
- `INFO_MODELMBEAN`: information from the model MBean components
- `INFO_MISC`: information sent from any other classes
- `INFO_ALL`: information from all classes

The preceding information is held by the `TraceTags` class

Specifying the Level of Trace and Debug Information

The level of detail controls the number of messages you receive. The *trace* level is the default that gives information about the actions of the MBean server and other components. The *debug* level includes all the trace information providing information to help diagnose Java DMK implementation. If this level is specified, the information of `LEVEL_TRACE` is sent too. It is possible to specify the level of trace or debug information you want to receive. Two levels of information are specified in the `TraceTags` class:

- | | |
|--------------------------|---|
| <code>LEVEL_TRACE</code> | Provides information to help a developer when programming |
| <code>LEVEL_DEBUG</code> | Provides information to help diagnose Java DMK implementation |

If you choose the second option, you will automatically receive all trace information as well as debug information. By default, the level is set to `LEVEL_TRACE`.

Index

A

accessing agents, using a web browser, *See*
HTML protocol adaptor, 25
adaptor, HTML protocol, *See* HTML protocol
adaptor, 25

D

debug information
 default level, 31
 receiving, 29
 specifying the level, 31
 specifying the type, 30
debugging, Java DMK API, 29

H

HTML protocol adaptor, 25
 limitations, 27
 usage, 26

M

mapping
 mibgen information mapping, 22
 proxygen mapping rules, 12
mibgen compiler
 introduction, 15
 options, 16
 output, 18

mibgen compiler (Continued)

 starting, 15

MIBs

 representation in a Java file, 18
 representation in an SNMP `OidTable`, 19

N

notification listener
 adding to a class, 30
 creating, 30

P

proxy MBeans, generating, *See* proxygen
 compiler, 9
proxygen compiler
 example, 11
 introduction, 9
 mapping attributes, 12
 mapping operations, 13
 mapping rules, 12
 modifying generated code, 14
 options, 10
 output, 12
 proxy MBean methods, 13
 starting, 10
 using generated code, 13

S

SNMP groups

- representation in a metadata file, 20
- representation in an MBean, 19

SNMP MIBs

compiling

See `mibgen` compiler

- representation in a Java file, 18
- representation in an SNMP `OidTable`, 19

SNMP tables, representation by classes, 20

T

trace filter, creating, 30

trace information

- default level, 31
- receiving, 29
- specifying the level, 31
- specifying the type, 30

tracing, Java DMK API, 29

tracing mechanism, introduction, 29

W

web browser, connecting to an agent, 26