The SeMoA and CODEC Code Conventions

Volker Roth[*], Ulrich Pinsdorf[†], Jan Peters[‡] and Peter Ebinger[§]
Fraunhofer Institut für Graphische Datenverarbeitung
Fraunhoferstr. 5
64283 Darmstadt
Germany

Version 0.2

30th July 2004

[*] vroth@igd.fraunhofer.de
[†] ulrich.pinsdorf@igd.fraunhofer.de
[‡] jan.peters@igd.fraunhofer.de
[§] peter.ebinger@igd.fraunhofer.de

Contents

1 Introduction

The key words "MUST", "MUST NOT", "REQUIRED", "RECOMMENDED", "SHOULD", "SHOULD NOT", "MAY", and "OPTIONAL" in this document are to be interpreted as described in RFC 2119.

1.1 Motivation

This document contains guidelines for writing Java code within the SeMoA[1] and the CODEC project. More precisely, it refers to a number of existing style guides, and amends or modifies them as to reflect the particular "flavour" which shall be promoted in this document.

There are good reasons to have a style guide and to adhere to it. From Sun Inc.'s document "Java Code Conventions"[2] I quote:

> "Code conventions are important for programmers for a number of reasons:
>
> - 80% of the lifetime cost of a piece of software goes to maintenance.
> - Hardly any software is maintained for its whole life by the original author.
> - Code conventions improve the readability of the software, allowing engineers to understand new code more quickly and thoroughly.
> - If you ship your source code as a product, you need to make sure it is as well packaged and clean as any other product you create.
>
> For the conventions to work, every person writing software must conform to the code conventions. Everyone."

1.2 Related Documents

Writing a proper code style from scratch is a huge amount of work. It is prudent to build on well established code conventions as much as possible,

[1] `http://www.semoa.org`
[2] `http://java.sun.com/docs/codeconv/html/CodeConvTOC.doc.html`

and to deviate only in places worth the trouble. In particular, Sun Inc. already defined code conventions for writing Java code which are obligatory for all programmers working with this programming language.

Newcomers to the language MUST first get familiar with the Java language style conventions as given in Section 6.8 of "The Java Language Specification"[3].

The Java Development Kit (JDK) comes with a tool called `javadoc`, which can create *JavaDoc* documentation automatically based on documentation comments in the Java source code. Programmers following this style guide MUST use the documentation comments (appropriately). The syntax of JavaDoc documentation comments is explained in Chapter 18 of "The Java Language Specification". Further information on how to use JavaDoc is given in "How to Write Doc Comments for Javadoc"[4].

Information on how to use `javadoc` to generate JavaDoc documentation from source files comes with the JDK[5]. Programmers following this style guide SHOULD be familiar with this document as well.

The basis of this code style are the "Java Code Conventions"[6] (JCC). Readers of this document MUST be familiar with the conventions described therein. Occasionally I will mark sections which deviate from the JCC by means of margin paragraph.

Advanced users of this code style SHOULD read (and adhere to) the "Requirements for Writing Java API Specifications"[7] document.

Programmers working on security-sensitive code MUST follow the "Security Code Guidelines"[8].

1.3 Goals

The human visual system works according to a small set of psychological rules when grouping and recognizing objects. These rules include but are not limited to *similarity, proximity, continuity* and *orientation.* Code can be read and understood much easier if meaningful blocks of code are arranged

[3] `http://java.sun.com/docs/books/jls/html/index.html`
[4] `http://java.sun.com/products/jdk/javadoc/writingdoccomments/index.html`
[5] `docs/tooldocs/solaris/javadoc.html`
[6] `http://java.sun.com/docs/codeconv/html/CodeConvTOC.doc.html`
[7] `http://java.sun.com/products/jdk/javadoc/writingapispecs/index.html`
[8] `http://java.sun.com/security/seccodeguide.html`

and formatted in agreement with these general rules. A major technique is *indentation*. In contrast to ragged edges which are hard to follow and to read, proper indentation produces recognizable straight edges that are used by the human visual system as a guide (continuity, orientation). The use of blank lines supports grouping of blocks of code that "belongs together", and to set them apart from "other" code (proximity). Consistent programming patterns and style support recognition and thus understanding of code (similarity).

The goal of this codestyle is to deploy these general rules as much as possible, and to achieve a high level of consistency throughout the code.

Remember, always comment and format your code in a clear fashion, such that others can understand and leaf through it like a breeze through a freshly napalmed forest.[9] Now, let's get down to business.

2 Conventions

> *You want you a paradise, the flatline advised, when Task explained his situation.*
>
> — William Gibson, *"Neuromancer"*

2.1 File Management

All Java (`.java`) files and class (`.class`) files MUST be kept in a directory hierarchy that resembles the package structure of the classes, starting from a *base* directory. Java files and their corresponding class files SHOULD be kept in the same directory.

For all practical purposes it is RECOMMENDED that a consistent directory structure is used by all team members. Use of a version control system such as CVS is also RECOMMENDED. Each team member SHOULD have a private *workspace* in addition to the central *repository* where master files are kept. A sample structure of a private workspace is illustrated below:

[9] Quoted from "Snow Crash", Neil Stephenson ;-)

Path	Comment
`java`	Base directory
`java/doc/api`	JavaDoc API documentation
`java/ext`	Customized Java Extensions
`java/cvs`	Workspace for source tree
`java/cvs/classes`	Java and class files
`java/cvs/conf`	Master Makefiles and scripts
`java/cvs/docs`	Dokumentation

Programmers MUST copy/checkout project files they work on from the project's class repository into their workspace and test them there. After successful modification and testing, the classes MUST be passed to the repository manager. The repository manager is the person responsible for the maintainance and integrity of the repository. He SHOULD review the classes for quality, stability, proper documentation, consistency, interoperability with the classes in the repository, and adherence to the code conventions before checking them into the repository. If classes do not meet the criteria, they should be returned to the programmer for correction. The repository manager MUST announce changes in the repository to all team members. Team members other than the repository manager MUST NOT modify or write to the repository unless they want their arms hacked off (they shouldn't be able to do that in the first place).

2.2 File Organization

Differs from JCC, Sec. 3.1

A file consists of sections that should be separated by blank lines and an optional comment identifying each section. Files longer than 1500 lines are cumbersome and should be avoided. Each Java source file contains a single class or interface, inner classes are permitted. Java source files have the following ordering:

- Beginning comments such as copyright notice or disclaimer, formatted as a block comment (see Section 2.4.1).
- Package and import statements.
- Class and interface declarations.

2.2.1 Package and Import Statements

Amends JCC, Sec. 3.1.2

The first non-comment line in a Java file MUST be a `package` statement

```
/* The copyright notice goes here, as well as the
 * obligatory refusal to assume any responsibility
 * on what results from using the code that follows
 * below (given that the license terms permit using
 * the code for any reasonable purpose at all).
 */
package codec.pkcs7;

import codec.InconsistentStateException;

import codec.asn1.ASN1Integer;
import codec.asn1.ASN1ObjectIdentifier;
import codec.asn1.ASN1RegisteredType;
import codec.asn1.ASN1Sequence;

import java.io.IOException;
import java.io.InputStream;
import java.io.OutputStream;

import java.security.AlgorithmParameters;
import java.security.GeneralSecurityException;
import java.security.InvalidAlgorithmParameterException;

import java.util.NoSuchElementException;

import javax.crypto.Cipher;
import javax.crypto.SecretKey;
```

Figure 1: An example that shows how to format package and import statements.

(put only temporary classes used for testing into the default package; such classes are not subject to code conventions). After that, `import` statements can follow.

Sort `import` statements into sections according to the general source from which they stem, and separate these sections by a single blank line. Within sections, sort `import` statements first according to common packages prefixes and second according to alphabetical order. Prefer direct enumeration of classes over asterisk forms. An example is given in Figure 1.

In Figure 1, the second and the third sections contain packages on which the team is working (`codec`), and the following section contain packages distributed with the JDK. References to a commercial library of Java classes used by the team (e.g. `acme.*`) would get a section of their own.

2.2.2 Class and Interface Declarations:

Below, I describe the parts of a class or interface declaration, in the order that they MUST appear.

Class/interface documentation comment: See Section 2.4.2 for a description of what must or should appear in this comment. Do not put a blank line between this part and the next. The comment MUST start in column 0 if the primary class/interface in the file is declared, and MUST be indented if it is an inner class/interface.

`class` or `interface` statement: Wrap lines as appropriate; put the initial curly brace '{' in a new line, aligned to the same column as the start of the class/interface declaration statement. Do not put a blank line between this part and the next.

Class (`static`) variables: As in JCC first the `public` class variables, then the `protected`, then package level (no access modifier), and then the `private`. Separate each variable declaration from the preceeding one by one blank line. Insert a blank line between the last class variable declaration and the first instance variable declaration.

Instance variables: Declare instance variables (non-`static`) in the same order as the class variable declarations, using one blank line as a separator.

Constructors: Separate the first constructor from the last variable declaration by two blank lines. Put two blank lines between constructors and between constructors and methods. Start with simple constructors (few parameters) to more complex ones (more parameters). Put no spaces between a constructor name and the paranthesis '(' starting its parameter list.

Methods: Grouping of methods is as described in JCC, Sec. 3.1.3. Put no spaces between a method name and the paranthesis '(' starting its parameter list.

Inner classes: Inner classes not declared ad-hoc in a method call (e.g. when calling the `AccessController` with a short `PrivilegedAction`) MUST be declared after the primary class of the class file, before the ending curly brace '}'.

Documentation comments (see Section 2.4.2 MUST not be separated from the commented section by a blank line.

2.3 Indentation

Four spaces MUST be used as the unit of indentation. `[Tab]` characters MUST NOT be used in source files.

2.3.1 Line Length

Programmers MUST format their code to a width of at most 72-80 characters.

2.3.2 Wrapping Lines

When expressions do not fit into a single line, break according to the following principles:

- Break after an opening paranthesis '(' if it is not part of a cast.
- Break before an operator.
- Prefer higher-level breaks to lower-level breaks.
- Indent the new line.

If the wrapped line still does not fit into one line then consider a vertical orientation of the sub-expressions (see `reEncryptData()`), as illustrated in Figure 2.

In general, keep the orientation of your code vertically rather than horizontally. Avoid to many nested scopes. If necessary, use `break`, `continue`, `throw`, and `return` to flatten nested scopes.

In particular, this differs from the 8 character rule in the JCC, which squishes code up against the right margin too fast, and causes irregular and sometimes confusing layouts. Differs from JCC, Sec. 4

```
/* PREFERRED
 */
buf = encryptData(
    secretKey, bulkEncryptionAlgorithm, algorithmParameters);

buf = reEncryptData(
    secretKey,
    bulkEncryptionAlgorithm,
    algorithmParameters,
    reEncryptionKey,
    reEncryptionAlgorithm,
    reEncryptionParameters
);

/* PREFERRED (consistent 4-character indentation)
 */
buf = transformDataByFooMethod(
    methodID, principal, dataSource, mode);

/* AVOID (8-spaces rule from JCC)
 */
buf = encryptData(secretKey, bulkEncryptionAlgorithm,
        algorithmParameters)

/* AVOID
 */
buf = transformDataByFooMethod(methodID, principal,
                               dataSource, mode)
```

Figure 2: Do's and dont's of indentation; note the differences to JCC, Sec. 4.

2.4 Comments

Please refer to the JCC, Sec. 5 for an explanantion of differences between implementation and documentation comments. All documentation MUST be done in English. In terms of language, this is the lowest common denominator of our profession.

2.4.1 Implementation Comments

Programs SHOULD have three styles of implementation comments: *block*, *trailing*, and *end-of-line*. Please note that in contrast to the JCC no single-line comments are mentioned. Single-line comments SHOULD be formatted like block comments.

Differs from JCC, Sec. 5.1

Block comments within a code section SHOULD be preceeded by a blank line unless the preceeding non-empty line contains only a single curly brace '{'. In that case the blank line SHOULD be omitted. The single curly brace still leaves enough empty vertical space such that the preceeding code section and the block comment are perceived as separate.

Block comments MUST be formatted as follows:

```
/* Here is a block comment. Maybe it's just a single
 * line, or multiple lines if necessary. Code follows
 * directly without further blank lines.
 */
someVar = obj.getSomeValue();
```

Programmers SHOULD use trailing comments rather than end-of-line comments for commenting lines of code. In general, end-of-line comments SHOULD be used *only* to comment out code. Moreover, trailing comments SHOULD be used sparingly, e.g. for commenting elements in static initializers of arrays as illustrated in Figure 3. For regular code, preceeding block comments SHOULD be used rather than trailing comments.

2.4.2 Documentation Comments

The general conventions set forth in JCC, Sec. 5.2 hold. However, programmers SHOULD use the `@throws` tag to document exceptions rather than `@exception`. If the text after a `@param`, `@return`, or `@throws` tag (give tags in that order) does not fit in one comment line then the line must be

```
/**
 * A hypothetical array of valid type identifiers,
 * defined in a static (class) variable. Similarities
 * to ASN.1 are purely coincidental, of course.
 */
private static String[] typename_ =
{
    null,          /* End of contents marker */
    "Boolean",     /* Tag [1] */
    "Integer",     /* Tag [2] */
    null,          /* BitString not yet supported */
    "OctetString"  /* Tag [4] */
}
```

Figure 3: Prefer trailing comments over end-of-line comments. Prefer blobk-comments over trailing comments.

wrapped and the new lines MUST be indented by 2 characters measured from the column of the tag's `@` character.

All checked exceptions thrown by a method or constructor MUST be documented as well as all unchecked exception thrown *explicitly* by that method or constructor. Unchecked exceptions that are thrown implicitly (e.g. `ArrayIndexOutOfBoundsException`) SHOULD also be documented if they must be expected even in regular operation of a class rather than scarce error conditions. An example is given in Figure 4.

The tagged paragraphs MUST be separated from the general comments by a blank comment line (a line with only an asterisk). Keywords of the java language MUST be typeset in `<code></code>`. References to classes, arguments, and methods MUST either be typeset in `<code></code>` or by means of {`@link [#]ref referee`}. The creation of links is encouraged, yet programmers shouldn't overdo it.

2.5 Declarations

2.5.1 Number Per Line

Differs from JCC, Sec. 6.1

Similar to the JCC, programmers MUST NOT put more then one declaration in a single line. Declarations SHOULD be formatted in triangular form, as illustrated in Figure 5. In other words, declarations SHOULD be sorted first according to the length of the type identifier, and second according to the

```
/**
 * Checks if the bytecode of a class file implements
 * <code>finalize()</code>.
 *
 * @param code The byte array that contains the
 *   bytecode of a class.
 * @exception SecurityException if the class in
 *   <code>code</code> implements <code>finalize()
 *   </code>.
 * @exception ArrayOutOfBoundException if the
 *   bytecode of the class has a bad format, e.g. its
 *   constant pool is corrupted.
 */
public void filterClass(File file) throws ...
```

Figure 4: Illustrates general formatting and indentation of doc comments.

```
AlgorithmParameterSpec spec;
ASN1ObjectIdentifier oid;
ASN1OctetString octets;
Attributes attributes;
Attribute attribute;
Signature sig;
String sigalg;
String mdalg;
```

Figure 5: Formatting of variable declarations in methods and constructors should have a triangular form, sorted first according to the length of the type identifier and second according to the length of the variable name.

length of the variable name. Type identifiers and variable names SHOULD be separated by a single space. The JCC propose a vertical alignment (see JCC, Sec. 6.1). However, this often creates undesireable wide runs of spaces because one type has a very long identifier while the average type identifier is much shorter. The formatting proposed in this style usually creates a clear diagonal line which separates type identifiers from variable names pretty clearly, in particular when used in conjunction with syntax highlighting.

2.5.2 Initialization

In this section I enter a territory which must probably be designated as being "religous". However, programmers MUST NOT initialize variables where they are declared unless it is a class (`static`) or instance variable. Joint

Differs from JCC, Sec. 6.2

declaration and initialization encourages the attitude that "it's ok to create variables in an ad-hoc fashion" – it isn't.

2.5.3 Placement

All declarations MUST be done at the beginning of the block/scope for which they are valid, after the opening curly brace '{' of that scope. **There is no excuse for not doing this**. Avoid declarations that hide declarations at higher levels (compare JCC Sec. 6.3).

2.6 Statements

Each line MUST NOT contain more than one statement. For compound statements (a list of statements enclosed in curly braces "{ *statements* }") the following rules hold:

- The enclosed statements MUST be indented one more level than the compound statement.
- Both the opening and the closing brace MUST begin a new line, and MUST be indented to the level of the compound statement. Additional statements MUST NOT be on the same line as the curly braces. This differs considerably from the JCC, Sec. 7. However, my experience shows that this produces code that is much easier to read.
- Braces MUST be used around all statements, *even single statements,* when they are part of a control structure, such as a `if/else`, or `for` statement.

One exception from the rule is tolerated. A `try` statement is three characters wide, which is one less than the usual indentation. For this reason, programmers can put the brace that follows the `try` statement directly (without a space) behind the `try` statement, such that it reads `try{`. The indented block of compound statements still stands out clearly from the surrounding code, which is the goal after all.

Apart from the differences mentioned above, the conventions described in the JCC, Sec. 7 hold.

Name	Type
`i`	`int` or `Iterator`
`e`	`Exception` in `catch(){}` clauses
`o`	`Object`
`in`	`InputStream`
`out`	`OutputStream`
`bis`	`ByteArrayInputStream`
`bos`	`ByteArrayOutputStream`
`fis`	`FileInputStream`
`fos`	`FileOutputStream`

Table 1: Naming conventions for temporary variables.

2.7 White Space

The JCC, Sec. 8.2 hold, with the following exception: casts MUST NOT be followed by a blank.

2.8 Naming Conventions

This section extends the naming conventions laid down in Section 9 of the "Java Code Conventions". Programmers SHOULD use the "common names" for temporary variables given in Table 2.8 whenever appropriate. The names of instance variables and class variables which are not declared `public` MUST have a trailing underscore. Examples and violations of this rule are given below for illustration.

```
public boolean flag_;      /* WRONG!!  */
protected List members;    /* WRONG!!  */

public boolean flag;       /* CORRECT */
protected List members_;   /* CORRECT */
```

The trailing underscore makes it easy to distinguish local variables from non-local ones and makes constructions such as `this.members` superfluous.

2.9 General Implementation Rules

Programmers MUST build on the Collection Framework rather than those classes in `java.util` which are superseded by the Collection Framework.

In particular, `Enumeration`, `Vector`, and `HashTable` MUST NOT be used unless a particular unavoidable interface requirement mandates use of these classes. If e.g. a Sun API requires that a particular implementation needs to return an instance of `Enumeration` then consider using a `Collection` instead and wrap its `Iterator` in a `DE.FhG.IGD.util.Enumerater` (if you have this class). The `Enumerator` takes an `Iterator` and implements both the `Iterator` interface as well as `Enumeration`.

The conventions described in this document are directed at the devolpment of mobile agents software, which is security critical. A severe threat comes from *Denial of Service* (DoS). One way to deny service to other threads is to grab class or instance locks. This can be avoided if classes and instances synchronize on private lock objects instead. A typical declaration of such a lock object looks like this:

```
/**
 * My private instance lock object. Serves secure
 * synchronization.
 */
private Object lock_ = new Object();
```

Do not hardcode magic numbers. Better declare a `public static final` variable with that value. This allows to change values easily and improves readability and understanding of your code. Do not use arrays or classes whose instances are mutable in this way. Attackers can modify these definitions if they are declared as `public`.

A Example Code

```
/* Copyright 2000 Fraunhofer Gesellschaft
 * Leonrodstr. 54, 80636 Munich, Germany.
 * All rights reserved.
 *
 * You shall use this software only in accordance with
 * the terms of the license agreement you entered into
 * with Fraunhofer Gesellschaft.
 */
package codec;

import java.util.HashSet;
import java.util.Iterator;
import java.util.LinkedList;
import java.util.NoSuchElementException;
```

```
/**
 * This class provides a thread pool with a given capacity (number of
 * threads). The pool can handle as many concurrent jobs as there are
 * threads in it. A job consists of a <code> Runnable</code> that is passed
 * to the pool's <code> run(Runnable)</code> method. A job thread is removed
 * from the pool while it works on a job, and re-enters the pool after
 * completition of the job.
 *
 * <p>
 * The capacity can be changed at runtime. The number of allocated threads
 * will then change dynamically to the new capacity. New threads will be
 * created by the pool as required. If the pool's capacity shrinks then as
 * many surplus threads as there are in the pool will be released while
 * threads that completed a job will enter the pool only if there is room
 * for them. Hence, if lots of jobs are requested then the actual capacity
 * will shrink only after enough threads are idling in the pool such that
 * the surplus returning threads are rejected and terminate.
 * </p>
 *
 * <p>
 * The threads in the pool are unaffected by exceptions thrown by the job
 * objects. However, threads idling in the pool will terminate and reduce
 * the capacity if they are interrupted.
 * </p>
 *
 * @author Volker Roth
 * @version $Id: example.tex,v 1.1 2001/05/21 12:20:48 vroth Exp $
 */
public class ThreadPool extends Object
{
    /**
     * The maximum number of threads spawned by a <code> ThreadPool</code>.
     */
    public static final int MAX_THREADS = 1024;

    /**
     * The <code>ThreadGroup</code> for the threads of the pool.
     */
    protected ThreadGroup group_;

    /**
     * The list that holds the idle threads.
     */
    protected LinkedList idle_;

    /**
     * The list that holds the running (busy) threads.
     */
```

```
protected HashSet busy_;

/**
 * Creates a thread pool with the given capacity. The threads be started
 * and enter an idle state (blocked) until they are given a job.
 *
 * @param capacity The capacity of the thread pool.
 *
 * @throws IllegalArgumentException if <code>capacity </code> is out of
 *   the range [1, maximum number of threads].
 */
public ThreadPool(int capacity)
{
    if ((capacity < 1) || (capacity > MAX_THREADS))
    {
        throw new IllegalArgumentException(
            "Capacity "
            + capacity
            + " out of range [1, "
            + MAX_THREADS
            + "]");
    }
    idle_  = new LinkedList();
    busy_  = new HashSet();
    group_ = new ThreadGroup("Thread pool");

    group_.setDaemon(true);
    setCapacity0(capacity);
}

/**
 * Schedules a job for execution. The job is given as a <a
 * href="#java.lang.Runnable">Runnable</a>. This object is switched to
 * the context of the thread that subsequently <a href="#run()">runs</a>
 * that object. A <code>null</code> is ignored. If no thread is
 * available then the call blocks until a thread becomes available.
 *
 * @param o the <code>Runnable</code> that does the job.
 *
 * @throws InterruptedException if the blocking wait is interrupted.
 * @throws NoSuchElementException if the pool's capacity is zero. See
 *   also method {@link #join join()}.
 */
public void run(Runnable o) throws InterruptedException
{
    ThreadPool.Entry entry;
    int capacity;

    if (o == null)
```

```
    {
        return;
    }

    while (true)
    {
        synchronized (idle_)
        {
            capacity = getCapacity();

            if (capacity == 0)
            {
                throw new NoSuchElementException("Pool is empty!");
            }

            try
            {
                if (idle_.size() > 0)
                {
                    entry = (ThreadPool.Entry)idle_.removeFirst();
                    busy_.add(entry);

                    break;
                }
                else
                {
                    /* Wait for new capacities or completed
                     * jobs that free a thread.
                     */
                    idle_.wait();
                }
            }
            catch (ClassCastException e)
            {
                /* Can't happen. */
            }
        }
    }
    entry.run(o);
}

/**
 * Called by thread pool entries to re-enter the thread pool after
 * completition of a job. This method unblocks a waiting thread (if any)
 * that may claim the now available thread for another job.
 *
 * @return <code>true</code> if <code>entry</code> should prepare to do
 *   another job, and <code>false</code> if it should terminate.
 */
```

```
private boolean addAndContinue()
{
    Thread thread;

    thread = Thread.currentThread();

    synchronized (idle_)
    {
        if (busy_.contains(thread))
        {
            busy_.remove(thread);
            idle_.addFirst(thread);

            /* Notify waiting threads that the current
             * thread is available again.
             */
            idle_.notifyAll();

            return true;
        }
        return false;
    }
}

/**
 * Returns the capacity of this <code>ThreadPool</code>.
 *
 * @return The capacity of this <code>ThreadPool</code>. If the capacity
 *   is 0 then the <code>ThreadPool</code> was shut down.
 */
public int getCapacity()
{
    synchronized (idle_)
    {
        return idle_.size() + busy_.size();
    }
}

/**
 * Sets the capacity of this <code>ThreadPool</code>. The capacity is not
 * instantly reduced, but degrades slowly when threads finish but are
 * not re-inserted into the pool.
 *
 * @param capacity The capacity that is set for this thread pool.
 *
 * @throws IllegalArgumentException if <code>capacity </code> is out of
 *   the range [1, maximum number of threads].
 */
public void setCapacity(int capacity)
```

```
{
    if ((capacity < 1) || (capacity > MAX_THREADS))
    {
        throw new IllegalArgumentException(
            "Capacity " + capacity + " out of range [0, " + MAX_THREADS
            + "]");
    }
    setCapacity0(capacity);
}

/**
 * Returns the current size of the thread pool.
 *
 * @return The current size.
 */
public int size()
{
    return idle_.size();
}

/**
 * Shuts down as many threads of the thread pool as possible, set the
 * capacity to 0, and interrupts all remaining threads. As a result, the
 * <code>ThreadPool</code> should be down pretty quickly unless some
 * running jobs don't cooperate when they receive the interrupt.
 */
public void shutdown()
{
    synchronized (idle_)
    {
        busy_.clear();
        idle_.clear();
        group_.interrupt();

        group_ = null;
    }
}

/**
 * Sets the capacity of this <code>ThreadPool</code> without checking
 * for illegal arguments. The capacity is not instantly reduced, but
 * degrades slowly when threads finish but are not re-inserted into the
 * pool.
 *
 * @param capacity The capacity that is set for this thread pool.
 */
private void setCapacity0(int capacity)
{
    Iterator i;
```

```
Thread thread;
int current;
int delta;
int n;

synchronized (idle_)
{
    current = getCapacity();

    if (capacity == current)
    {
        return;
    }
    else if (capacity > current)
    {
        delta = capacity - current;

        while (delta > 0)
        {
            thread = new ThreadPool.Entry();
            thread.start();

            idle_.addLast(thread);
            delta--;
        }

        /* Notify waiting threads that new capacities
         * are available.
         */
        idle_.notifyAll();
    }
    else
    {
        delta = Math.min(idle_.size(), current - capacity);

        for (n = 0; n < delta; n++)
        {
            thread = (Thread)idle_.removeFirst();
            thread.interrupt();
        }
        delta = Math.min(busy_.size(), current - capacity - delta);

        if (delta > 0)
        {
            i = busy_.iterator();

            /* Threads removed from the busy set will not be
             * able to enter the idle list again.
             */
```

```
                while (delta > 0)
                {
                    i.next();
                    i.remove();
                    delta--;
                }
            }
        }
    }
}

/**
 * Removes the thread of caller from this thread pool.
 */
private void removeEntry()
{
    Thread thread;

    thread = Thread.currentThread();

    synchronized (idle_)
    {
        if (busy_.contains(thread))
        {
            busy_.remove(thread);
        }
        else if (idle_.contains(thread))
        {
            idle_.remove(thread);
        }
    }
}

/**
 * A thread in the ThreadPool. This thread automagically re-enters itself
 * into the thread pool upon completition of the job delegated to it via
 * the thread pool. Jobs are passed to this class using a
 * <code>Runnable</code> object.
 *
 * @author Volker Roth
 */
protected class Entry extends Thread
{
    /**
     * The slave that is actually doing the job.
     */
    private Runnable slave_;

    /**
```

```
 * A private lock object, just to make sure...
 */
private Object lock_ = new Object();

/**
 * Creates a thread pool entry that re-enters itself into the given
 * thread pool.
 */
public Entry()
{
    super(group_, "Thread pool entry");
}

/**
 * Runs the given job in this thread. The given <code>
 * Runnable</code> is switched to this thread's context. Upon
 * completition of the job, this thread is re-entered into the
 * thread pool.
 *
 * <p>
 * This method must only be called once after removing this entry
 * from the pool. After that, discard the reference. This thread
 * will re-enter the thread pool upon completition of the job. Then
 * and only then it is safe to re-use this thread.
 * </p>
 *
 * <p>
 * Do not pass a <code>null</code> argument. The thread will then
 * block forever and will not re-enter the pool.
 * </p>
 *
 * @param o The <code>Runnable</code> that is to be run by this
 *   thread.
 *
 * @throws InterruptedException if the blocking wait is interrupted.
 */
protected void run(Runnable o) throws InterruptedException
{
    synchronized (lock_)
    {
        while (slave_ != null)
        {
            /* Wait until the thread pool thread gets
             * ready to roll (hence goes to sleep on
             * the lock).
             */
            lock_.wait();
        }
        slave_ = o;
```

```
        /* Wake up the waiting thread pool thread.
         */
        lock_.notifyAll();
    }
}

/**
 * The first thing this method (and this thread) does is to enter the
 * owning thread pool.
 *
 * @throws SecurityException if the current thread is not this entry.
 */
public void run()
{
    if (Thread.currentThread() != this)
    {
        throw new SecurityException("Illegal thread.");
    }

    while (true)
    {
        try
        {
            synchronized (lock_)
            {
                /* Wait until there is a job to do.
                 */
                while (slave_ == null)
                {
                    lock_.wait();
                }
            }
        }
        catch (InterruptedException e)
        {
            /* We were interrupted while idling, so we
             * bail out savely, and reduce the capacity
             * of the pool by one in the process.
             */
            removeEntry();

            break;
        }
        setName("Thread pool entry: " + slave_.getClass().getName());

        try
        {
            slave_.run();
```

```
                }
                catch (Throwable t)
                {
                    System.err.println(
                        "[ThreadPool.Entry] Job died with an exception!");

                    t.printStackTrace(System.err);
                }
                setName("Thread pool entry: " + System.currentTimeMillis());

                synchronized (lock_)
                {
                    slave_ = null;
                    lock_.notifyAll();
                }

                if (!addAndContinue())
                {
                    break;
                }
            }
        }
    }

    /**
     * Shuts the pool down, if it isn't already.
     */
    protected void finalize()
    {
        try
        {
            if (getCapacity() > 0)
            {
                shutdown();
            }
        }
        catch (Exception e)
        {
            /* Ignore */
        }
    }
}
```