BSSC 2005(2) Issue 1.0

Java

Coding Standards

Prepared by:
ESA Board for Software
Standardisation and Control
(BSSC)

european space agency / agence spatiale européenne
8-10, rue Mario-Nikis, 75738 PARIS CEDEX, France

Document Status Sheet

DOCUMENT STATUS SHEET			
1. DOCUMENT TITLE: BSSC 2005(2) Issue 1.0			
2. ISSUE	3. REVISION	4. DATE	5. REASON FOR CHANGE
0	1	10/23/2003	New document
0	2	02/02/2004	Distinction rules and recommendations, security information added to the security chapter, various corrections, elaboration on real-time Java™ 2 chapter, clean-up of acronyms and terms, introductory text added to each chapter, merger *casting* and *template* chapters.
0	3	19/03/2004	New sections 6.5, 13.4
0	4	28/05/2004	Reworked and streamlined draft.
0	5	30/06/2004	First draft (restructured) for open review.
0	6	06/12/2004	Issues from review round incorporated. Extensive restructuring. Material added on real-time Java, security, and portability.
0	7	13/01/2005	Comments of Philippe Chevalley incorporated, text-paragraphs rephrased, examples streamlined, new rules and recommendations added.
0	8	02/09/2005	Incorporated C++ chapter from F. Siebert. Added “Rationale” sections to many rules. Formatting of examples corrected. New rules and recommendations added.
1	0	03/03/2005	First issue

March 2005 - Board for Software Standardisation and Control
M. Spada and J-L Terraillon, BSSC Co-chair

Table of Contents

List of Rules

List of Recommendations

Preface

This Coding Standard is based upon the experience of developing custom space system software using the Java programming language. Both published experience and best practice rules obtained by Industry or by means of in-house developments are included.

The BSSC wishes to thank the European Space Research and Technology Centre (ESTEC), Noordwijk, The Netherlands, and in particular Peter Claes, for preparing the standard. The BSSC also thank all those who contributed ideas for this standard, in particular Dr Kelvin Nilsen (R/T chapter) and Dr James Hunt, Dr Fridtjof Siebert (R/T chapter and C/C++ integration chapter). The BSSC members that have reviewed the standard: Mariella Spada, Michael Jones, Jean-Loup Terraillon, Jean Pierre Guignard, Jerome Dumas, Daniel Ponz, Daniel de Pablo and Lothar Winzer. The BSSC also wishes to thank the following ESA reviewers of this standard: Jon Brumfitt, A. Bonfiel, Fernando Aldea Montero, Hans Ranebo, Jean-Pascal Lejault, Jose Hernandez, Jose Pizarro, Philippe Chevalley, Vicente Navarro, and the expert reviewer, editor, Martin Soto, from *Fraunhofer Institute for Experimental software Engineering* (IESE).

Requests for clarifications, change proposals or any other comments concerning this standard should be addressed to:

BSSC/ESOC Secretariat
Attention of Ms M. Spada
ESOC
Robert Bosch Strasse 5
D-64293 Darmstadt
Germany

BSSC/ESTEC Secretariat
Attention of Mr J.-L. Terraillon
ESTEC
Postbus 299
NL-2200 AG Noordwijk
The Netherlands

Chapter 1
Introduction

1.1 Scope and Applicability

These standards present rules and recommendations about the use of the language constructs of Java. Many books and documents describe how these features can be used. These texts usually describe what is possible and not necessarily what is desirable or acceptable, especially for large software engineering projects intended for mission- or safety-critical systems. This document is valid for the Java 2 v5.0 standard specification (J2SE 5.0.x) as well as for the Java Real-Time Specification [RTSJ] as published by the Java Real-Time Experts Group.

This document provides a set of guidelines for programming in Java which are intended to improve the overall quality and maintainability of software developed by, or under contract to, the European Space Agency. The use of this standard should improve consistency across different software systems developed by different programming teams in different companies or developed in-house in the Agency.

The guidelines in this standard should be met for Java source code to fully comply with this standard. The standard has no contractual implication. Contractual obligations are given in individual project documents.

This document is in principle a reference document. As with other BSSC coding standards, project managers may decide to make it applicable, particularly in the case that a supplier does not have a suitable in-house standard.

The readers are expected to be Java programmers (related to, or doing work for ESA) that understand very well the workings of the language.

Disclaimer: Programs and code snippets presented in this document are by no means guaranteed to be usable as runnable code. They are only included to demonstrate concepts outlined in the rules and guidelines.

1.2 Position of this document with respect to the ECSS-E40 and ECSS-Q80 Standards

The ECSS bundle of standards is organized around families. In particular, the Engineering family has a dedicated number for software (40), and the Quality family has a dedicated number for software product assurance (80).

The *ECSS-E-40 Part 1B, Space Engineering - Software - Part 1: Principles and requirements* (approved 28 Nov 2003) document recalls the various software engineering processes and list requirements for these processes in terms of activities that have to be performed, as well as pieces of information that have to be produced. *ECSS-E40 1B* does not address directly the coding standards, but requires that the coding standards are defined and agreed, at various levels of the development, be-

tween the customer and the supplier.

In particular, the selection of this Java standard could be the answer to the following requirements in the *ECSS-E40 1B* standard:

5.2.2.1 System requirements specification, Expected Output d): Identification of lower level software engineering standards [RB; SRR] (see ECSS- Q- 80B subclauses 6.3.2 and 6.3.3);

Also of relevance, the selection of this Java standard could be the answer to the following requirements of the ECSS-Q-80B standard (Software Product Assurance, approved 10 Oct 2003):

6.3.3.1 Coding standards (including consistent naming conventions, and adequate commentary rules) shall be specified and observed.

EXPECTED OUTPUT: Coding standards [PAF; PDR].

6.3.3.4 Coding standards shall be reviewed with the customer to ensure that they reflect product quality requirements.

EXPECTED OUTPUT: Coding standards and description of tools [PAF; PDR].

1.3 Document Overview

This document is intended to build on the output of the *Software Top-Level Architectural Design*, *Design of Software Items* and *Coding and Testing* phases (*ECSS-E-40* terminology) and follows a "top-down" approach so that guidelines can begin to be applied as soon as detailed design of software items starts and before any code is produced. Subsequent chapters describe the specific rules and recommendations to be applied to the production of Java code.

All rules (mandatory) and recommendations (optional) are numbered for reference purposes.

All rules and recommendations have a short title and an explanation. Many rules and recommendations are also followed by a rationale section justifying the application of the rule. Rules and recommendations may also contain examples showing how to apply them, or illustrating the consequences of not applying them.

All rules and recommendations are enclosed in boxes. Recommendations are printed in italic type.

1.4 Glossary

Abstract class – A class that exists only as a superclass of another class and can never be directly instantiated. In Java, an abstract class contains or inherits one or more abstract methods or includes the `abstract` keyword in its definition.

Abstraction – The process and result of extracting the common or general characteristics from a set of similar entities.

Accessor – A method that sets or gets the value of an object property or attribute.

Algorithm – A finite set of well-defined rules that gives a sequence of operations for performing a specific task.

Architecture – A description of the organization and structure of a software system.

Argument – Data item specified as a parameter in a method call.

Assertion – A statement about the truth of a logical expression. In Java, a special instructions that checks the validity of such a statement during run-time.

Attribute – A feature within a class that describes a range of values instances of the class may hold. A named characteristic or property of a type, class, or object.

Behavior – The activities and effects produced by an object in response to an event.

Block statement – The Java language construct that combines one or more statement expressions into a single compound statement, by enclosing them in curly braces "`{...}`".

Boolean – An enumerated type whose values are true and false.

Built-in type – A data type defined as part of the language. The built-in or native types defined by Java include the primitive types `boolean`, `byte`, `char`, `double`, `float`, `int`, `long`, `short`, and `void`, and the various classes and interfaces defined in the standard Java API, such as `Object`, `String`, `Thread`, and so forth.

Checked exception – Any exception that is not derived from `java.lang.RuntimeException` or `java.lang.Error`, or that appears in the `throws` clause of a method. A method that throws, or is a recipient of, a checked exception must handle the exception internally or otherwise declare the exception in its own `throws` clause.

Child – In a generalization relationship, the specialization of another element, the parent.

Class – A set of objects that share the same attributes and behavior.

Client – An entity that requests a service from another entity.

Code – The implementation of particular data or a particular computer program in a symbolic form, such as source code, object code or machine code.

Code sharing – The sharing of code by more than one class or component, e.g. by means of implementation inheritance or delegation. See: implementation inheritance, delegation.

Compiler – Program that translates source code statements of a high level language, such as Java, into byte code or object code.

Component – (1) A self-contained part, combination of parts, sub-assemblies or units, which performs a distinct function of a system. (2) A physical, replaceable part of a system that packages implementation and provides the realization of a set of interfaces. (3) A physical and discrete software entity that conforms to a set of interfaces.

Composition – A form of aggregation where an object is composed of other objects.

Concrete class – A class that can be directly instantiated. A concrete class has no abstract operations. Contrast: abstract class.

Concurrency – The degree by which two or more activities occur or make progress at the same time.

Global constant – A class variable defined as `public static final`.

Constraint – A semantic condition or restriction. Constraints include preconditions, postconditions, and invariants. They may apply to a single class of objects, to relationships between classes of objects, to states, or to use cases.

Constructor – A special method that initializes a new instance of a class.

Container – An object whose purpose is to contain and manipulate other objects.

Contract – A clear description of the responsibilities and constraints that apply between a client and a type, class, or method.

CORBA – An industry wide standard for communication between distributed objects, independent of their location and target language. The CORBA standard is defined by the Object Management Group (OMG). CORBA itself is an acronym for Common Object Request Broker Architecture.

Coupling – The degree to which two or more entities are dependent on each other.

Critical software – Software supporting a safety or dependability critical function that if incorrect or inadvertently executed can result in catastrophic or critical consequences.

Data abstraction – An abstraction denotes the essential characteristics of an object that distinguish it from all other kinds of objects, suppressing all non-essential details. In data abstraction the non-essential details deal with the underling data representation.

Database – A set of data, part or the whole of another set of data, consisting of at least one file that is sufficient for a given purpose or for a given data processing system.

Data type, – (1) A class of data characterized by the members of the class and the operations that can be applied to them. Examples are character types and enumeration types. (2) A descriptor of a set of values that lack identity and whose operations do not have side effects.

Dependency – A relationship where the semantic characteristics of one entity rely upon and constrain the semantic characteristics of another entity.

Design pattern – A documented solution to a commonly encountered design problem. In general, a design pattern presents a problem, followed by a description of its solution in a given context and programming language.

Destructor – A method that is executed when the object is garbage collected (automatically or *manually*)

Documentation comment – A Java comment that begins with a "/**" and ends with "*/", and contains a description and special tags that are parsed by the Javadoc utility to produce documentation.

Domain – An area of expertise, knowledge, or activity.

Dynamic loading (of classes) – The loading of classes dynamically (at run time) when they are first referenced by an application. The desktop Java environment, for example, provides a class loader capable of finding and loading a named class appearing in any of a prescribed list of locations, which may be either local or remote. In real-time systems, dynamic class loading is generally not supported or permitted.

Encapsulation – The degree to which an appropriate mechanism is used to hide the internal data, structure, and implementation of an object or other entity.

Enumeration – A type that defines a list of named values that make up the allowable range for values of that type.

Error – Discrepancy between a computed, observed or measured value or condition and the true, specified or theoretically correct value or condition.

Factor – The act of reorganizing one or more types or classes by extracting respon-

sibilities from existing classes and synthesizing new classes to handle these responsibilities.

Field – An instance variable or data member of an object.

Fundamental data type – A type that typically requires only one implementation and is commonly used to construct other, more useful types. Dates, complex numbers, linked-lists, and vectors are examples of common fundamental data types.

Hard real-time system – A system that guarantees that time-critical actions will always be performed at the specified time. Hard real time systems rely on timing constraints being proved using theoretical static analysis techniques prior to deployment.

Implementation – The concrete realization of a contract defined by a type, abstract class, or interface. The actual code.

Implementation class – A concrete class that provides an implementation for a type, abstract class, or interface.

Implementation inheritance – The action or mechanism by which a subclass inherits the implementation and interface from one or more parent classes.

Inheritance – A mechanism by which more specific elements incorporate (inherit) the structure and behavior of more general elements. Inheritance can be used to support generalization, or misused to support only code sharing, without attempting to follow behavioral subtyping rules.

Instance – The concrete representation of an object.

Instantiation – The act of allocating and initializing an object from a class.

Interface – A definition of the features accessible to clients of a class. Interfaces are distinct from classes, which may also contain methods, associations and modifiable attributes.

Note: The UML definition of interface differs slightly from that defined by Java in that Java interfaces may contain constant fields, while UML interfaces may contain only operations.

Interface inheritance – The inheritance of the interface of a more specific element. Does not include inheritance of the implementation.

Interrupt – A suspension of a task, such as the execution of a computer program, caused by an event external to that task, and performed in such a way that the task can be resumed.

Invariant – An expression that describes the well-defined, legal states of an object.

Keyword – A word used to mark language constructs in the syntax definition of a programming language.

Lazy initialization – The act of delaying the initialization of a data value until the first use or access of the data value.

Local variable – A variable whose scope is restricted to a single compound statement.

Method – The implementation of an operation. A method specifies the algorithm or procedure associated with an operation.

Monitoring – Functionality within a system which is designed to detect anomalous behavior of that system.

Object – An entity with a well-defined boundary and identity that encapsulates state

and behavior. State is represented by attributes and relationships; behavior is represented by operations and methods, and state machines.

Operation – A service that can be requested of an object. An operation corresponds to an abstract method declaration in Java. It does not define an associated implementation.

Overriding – The redefinition of an operation or method in a subclass.

Package – A mechanism for organizing and naming a collection of related classes.

Package access – The default access control characteristic applied in Java to interfaces, classes, and class members.

Parameter – A variable that is bound to an argument value passed into a method.

Parent – In an inheritance relationship, the generalization of another element, producing the child.

Pattern – A documented solution to a commonly encountered analysis or design problem. Each pattern documents a single solution to the problem in a given context.

Polymorphism – The concept or mechanism by which objects of different types inherit the responsibility for implementing the same operation, but respond differently to the invocation of that operation.

Polymorphic – A trait or characteristic of an object whereby that object can appear as several different types at the same time.

Postcondition – A constraint or assertion that must hold true following the completion of an operation.

Precondition – A constraint or assertion that must hold true at the start of an operation.

Primitive type – A basic language type that represents a pure value and has no distinct identity as an object. The primitives provided by Java include `boolean`, `byte`, `char`, `double`, `float`, `int`, `long`, and `short`.

Private access – An access control characteristic applied to class members. Class members declared with the `private` access modifier are only accessible to code in the same class and are not inherited by subclasses.

Property – A named characteristic or attribute of a type, class, or object.

Protected access – An access control characteristic applied to class members. Class members declared with the `protected` access modifier are accessible to code in the same class and package, and from code in subclasses, and they are inherited by subclasses.

Public access – An access control characteristic applied to interfaces, classes, and class members. Class members declared with the `public` access modifier are accessible anywhere the class is accessible and are inherited by subclasses. Classes and interfaces declared with the `public` access modifier are visible, accessible and heritable outside of a package.

Relationship – A semantic connection among model elements. Examples of relationships include associations and generalizations.

Responsibility – A purpose or obligation assigned to a type.

Robustness – The extent to which software can continue to operate correctly despite of invalid inputs.

Service – One or more operations provided by a type, class, or object to accomplish useful work on behalf of one or more clients.

Signature – The name, parameter types, return type, and possible exceptions associated with an operation.

Soft real-time system – A system in which an action performed at the wrong time (either too early or too late) is considered acceptable but not desirable. Soft real-time systems rely on empirical (statistical) measurements and heuristic enforcement of resource budgets to improve the likelihood of complying with timing constraints.

Software – A set of computer programs, procedures, documentation and their associated data.

Source code – Code written in a source languages, such as assembly language and/or high level language, in a machine-readable form for input to an assembler or a compiler.

State – A condition or situation during the life of an object during which it satisfies some condition, performs some activity, or waits for some event.

Static analyzer – A software tool that helps to reveal certain properties of a program without executing the program.

Subclass – In a generalization relationship, the specialization of another class; the so-called superclass or parent class.

Subclass – A class that inherits attributes and methods from another class.

Subinterface – The specialization of another interface.

Subtype – The more specific type in a specialization-generalization relationship.

Superclass – In a generalization relationship, the generalization of another class; the subclass.

Synchronization – The process or mechanism used to preserve the invariant states of a program or object in the presence of multiple threads.

Synchronized – A characteristic of a Java method or a block of code. A synchronized method or block allows only one thread at a time to execute within the critical section defined by that method or block.

System – A collection of hardware and software components organized to accomplish a specific function or set of functions.

Testing – The process of exercising a system or system component to verify that it satisfies specified requirements and to detect errors.

Traceability – The evidence of an association between items, such as between process outputs, between an output and its originating process, or between a requirement and its implementation.

Thread – A single flow of control flow within a process that executes a sequence of instructions in an independent execution context.

Type – Defines the common responsibilities , behavior, and operations associated with a set of similar objects. A type does not define an implementation.

Unchecked exception – Any exception that is derived from `java.lang.RunTimeException` or `java.lang.Error`. A method that throws, or is a recipient of, an unchecked exception is not required to handle the exception or declare the exception in its `throws` clause.

Variable – A typed, named container for holding object references or data values.

Visibility – The degree to which an entity may be accessed from outside of a particular scope.

1.5 Acronyms

API	Application Programming Interface
ASCII	American Standard Code for Information Interchange
AWT	Abstract Window Toolkit
CORBA	Common Object Request Broker Architecture
CPU	Central Processing Unit
ECSS	European Cooperation for Space Standardisation
EJB	Enterprise JavaBeans
GC	Garbage Collection
GUI	Graphical User Interface
HRT	Hard Real-Time
HS	Hot Spot (compiling)
HTML	Hypertext Mark-up Language
IDE	Integrated Development Environment
I/O	Input/Output
JAR	Java Archive
JCE	Java Cryptography Extension
JCP	Java Community Process
JDBC	Java Data Base Connectivity
JDK	Java Development Kit
J2EE	Java Enterprise Edition
JNI	Java Native Interface
J2ME	Java 2 Micro Edition
J2SE	Java Standard Edition
JSSE	Java Secure Sockets Extension
JVM	Java Virtual Machine
OO	Object-oriented
ORB	Object Request Broker
PCP	Priority Ceiling Protocol
POSIX	Portable Operating System Interface - UNIX
RMI	Remote Method Invocation
R/T	Real Time

RTOS	Real Time Operating System
RTSJ	Real Time Specification for Java
SSL	Secure Sockets Layer
SWT	Standard Widget Library
UML	Unified Modeling Language

Chapter 2
Installation, Build and Updates

2.1 Introduction

It is important for every software development project to provide common and uniform build and update procedures, and to allow for convenient and reliable deployment and installation of its products. The Java platform offers a number of facilities that directly address these essential project needs. This chapter is concerned with how to make better use of them.

2.2 Apache Ant

Apache Ant [ANT] (or simply *Ant*) is a Java based build tool intended to automate the compilation of software systems, as well as many other related tasks. *Ant* based compilation systems are guaranteed to run on any Java compliant platform.

Rule 1: Use the *Apache Ant* tool to automatically build your project.

Rationale

Ant allows to create portable compilation systems, thus making it possible for the development activities of a project to take place in heterogeneous software and hardware platforms.

Recommendation 1: Use the Ant tool to automate as many additional project tasks as possible.

Rationale

Apart from supporting the compilation process, *Ant* makes it possible to automate a variety of software quality assurance, installation, distribution, and maintenance related tasks, like running automated test suites, creating packaged files for distribution, and generating documentation, among many others.

Use *Ant* to automate as many such tasks as possible.

2.3 Preferences

Recommendation 2: Use the Java Preferences API to store and retrieve all run-time configuration data.

Run-time configuration data is all data necessary to adapt a software application to the needs of different users and environments. Use the *Java Preferences API* [SUNPref] to store all configuration related data.

Rationale

The Preferences API manages configuration data in a simple, completely platform independent way. In particular, it offers a number of advantages with respect to using property files, which is the traditional approach to handle this problem.

2.4 Software Distribution

Rule 2: When distributing a project, package all necessary class and resource files in a *jar* file.

Jar files are a standard way to distribute Java programs. In order to distribute a program, package all relevant class and resource files (icons, internal read-only data files) in a *jar* file.

Rationale

A number of problems can arise when copying a program's run-time files to a different hardware or software platform. Such problems can be already visible when attempting to copy the files, or may only become evident when running the software later on:

- *File name length:* File name length restrictions in some systems may cause file names to be incorrectly copied, or may generate naming conflicts because of name truncation.
- *Upper- and lower-case distinctions:* The way a system manages upper- and lower-case in file names could cause conflicts when files are transferred.
- *Special file names:* Some platforms assign special meaning to certain file names, such as “`LPT`” or “`con`”.
- *Special characters:* Each platform limits the set of characters accepted in file names to a different set.

Using a *jar* file avoids all of these problems, because *jar* files have an internal file naming system that is completely independent from that of the underlying operating system.

2.5 Implementation Files

Rule 3: Define only one class or interface per `.java` file.

Rationale

Doing so simplifies maintenance.

Chapter 3
Source Code Structure

Uniform source code structure and formatting are fundamental for an adequate collaboration between programmers. The rules and recommendations in this chapter define a consistent code formatting style to be applied in all ESA related software development projects.

Code formatting issues are often contentious, probably because they are mainly a matter of taste. Although this document cannot fit everyone's personal preferences, it specifies a reasonable formatting style, intended to be generally readable and relatively simple to follow while writing code. Additionally, since most of the rules and recommendations in this chapter are based on existing, widely known standards (like those from Sun Microsystems [SUNCode]) many members of the Java development community are likely to already be familiar with them.

The sections in this chapter address a variety of code elements, starting with the bigger and more general ones, and going down to the smaller, more detailed ones.

3.1 Implementation Files

Rule 4: Use the following structure for all implementation files:

- Start with the beginning comments.
- Follow with package and import statements.
- Put class and interface definitions at the end of the file.

Rationale

Formatting conventions are necessary to achieve code uniformity.

Recommendation 3: Use a standard template or utility program to provide a starting point for implementation files.

Rationale

Introducing the elements specified by this standard or by any additional project specific standards in every implementation file can be a burden for the programmer. Copying these elements from a standard, project-wide template file usually simplifies the task of creating new files and makes it more consistent.

An additional possibility is to provide a utility program or script, which can generate an initial implementation file, based on project-wide information and possibly information provided by the programmer or obtained from the programming environ-

ment. Such a program could save the programmer a number of additional customization steps that would be needed when using a simple template file.

Rule 5: Do not use tab characters in implementation files, use plain spaces instead.

Rationale

Interpretation of tab characters varies across operating systems and programming environments. Code that appears to be correctly formatted when displayed in the original editing environment can be virtually impossible to read when moved to an environment that interprets tab characters in a different way.

To avoid this problem, use spaces instead of tabs to indent and align source code. Most modern programming environments offer options to do this automatically. Otherwise it can be done manually.

Rule 6: Use the following order to declare members of a class:

- Class variables (declared `static`).
- Instance variables.
- Constructors (at least one).
- `finalize` method (destructor) if necessary.
- Class methods (declared `static`).
- Methods:
  - `set`/`get`
  - Other methods

Additionally, inside each one of these groups, declare members with a higher visibility first. That is, start with `public` members, continue with package visible members (not qualified), move to `protected` members, and end with `private` members.

Rationale

Formatting conventions are necessary to achieve code uniformity.

3.2 General Code Indentation Rules

This section contains some rules and recommendations that apply to all code elements. They are intended to be consistent with Sun Microsystems's Java Coding Conventions [SUNCode].

Rule 7: Use four spaces of indentation.

When indenting the code inside declaration and control structures, always use four additional spaces with respect to the previous level. More specific rules and recommendations in this document are always consistent with this general convention.

Rationale

Formatting conventions are necessary to achieve code uniformity.

Recommendation 4: Avoid lines longer than 80 characters.

Rationale

Shorter lines are easier to read. Additionally, utility programs dealing with source code, like text editors and code pretty printers, may have trouble dealing with excessively long lines.

Recommendation 5: When breaking long lines, follow these guidelines:

- Break after commas.
- Break before operators.
- Prefer breaking between large high-level subexpressions to breaking between smaller lower-level subexpressions.
- Align the text in a new line, with the beginning of the expression at the same syntactical level on the previous line.
- If the above guidelines lead to confusing or cumbersome formatting, indent the second line 8 spaces instead.

Rationale

These guidelines correspond to commonly used typographical conventions, which are generally accepted to improve readability.

Example

- Break after commas. Prefer

```
system.out.printline(final_time,
                     (final_time - initial_time) * 4);
```

to

```
system.out.printline(final_time, (final_time -
                     initial_time) * 4);
```

- Break before operators. Prefer

```
final_price = basic_price * tax_rate
              + initial_price;
```

(break before “+”) to

```
final_price = basic_price * tax_rate +
              initial_price;
```

(break after “+”.)

- Prefer breaking between large high-level subexpressions to breaking between smaller lower-level subexpressions:

```
pos = pos0 + speed * time +
      (acceleration * time * time) / 2;
```

should be preferred to

```
pos = pos0 + speed * time + (acceleration * time
                             * time) / 2;
```

- Align the text in a new line, with the beginning of the expression at the same syntactical level on the previous line. Prefer

```
self.getClientList().retrieve(baseClient - processCount
                              + offset);
```

to

```
self.getClientList().retrieve(baseClient - processCount
                                          + offset);
```

- If the above guidelines lead to confusing or cumbersome formatting, indent the second line 8 spaces instead. Prefer:

```
members.getMemberByName(memberName).store(
        self.database, self.getPriority() + priorityIncrease);
```

to

```
members.getMemberByName(memberName).store(self.database,
                                          self.getPriority() +
                                          priorityIncrease);
```

3.3 Definitions

Rule 8: Format class and interface definitions according to the following model:

```
class Sample extends Object
{
    int ivar1;
    int ivar2;

    Sample(int i, int j)
    {
        ivar1 = i;
        ivar2 = j;
    } // end method

    int emptyMethod() {}

    ...
} // end class
```

Some remarks about the previous model:

- Leave no space between a method name and the following parenthesis "(".
- The open brace "{" is put on its own line (just below the declaration statement) and is alligned with the end brace [GNUJAVA] Note this differs from the SUN guideline [SUNCode].
- The closing brace "}" starts a line by itself indented to match its corresponding opening statement, except when it is a null statement in which case, the "}" should appear immediately after the "{".
- Methods are separated by a blank line.

Rationale

Formatting conventions are necessary to achieve code uniformity.

Following the GNU convention for bracing style makes it more easier and quicker to recognize code structure. This is particularly true in the case of deep nesting. Consistency in brace placement remains even in the face of line breaks in long declarations. This is particularly good for code inspection.

Rule 9: Put single variable definitions in separate lines.

Rationale

Defining many variables in a single line, makes it more difficult to change the type of one of the declarations without changing the remaining ones. This could lead to inadvertently introducing programming errors.

Example

Instead of writing

```
int counter, total; // Wrong!
```

write

```
int counter;
int total;
```

3.4 Statements

Rule 10: Put single statements in separate lines.

Rationale

Many statements in a single line are harder to read.

Example

Instead of writing

```
counter = initial; counter++; // Wrong!
```

write

```
counter = initial;
counter++;
```

Rule 11: Format compound statements according to the following guidelines:

- Put the opening brace "{" at the end of the opening line for the compound statement.
- Indent enclosed statements one level (four spaces) with respect to the compound statements.
- Put the closing brace "}" in a line of its own, with the same level of indentation of the opening line.

Rationale

Formatting conventions are necessary to achieve code uniformity.

Rule 12: Always put braces around statements contained in control structures.

Statements hanging from control structures like `if`, `while`, `for`, and others shall always have braces surrounding them, even if the block contains a single statement.

While the language enables you to use simple, non-block statements as the body of these constructs, always use a block statement in such situations.

Rationale

Block statements reduce the ambiguity that often arises when control constructs are nested, and provide a mechanism for organizing the code for improved readability.

This practice helps to avoid programming errors resulting from interpreting the program as the indentation suggests, and not as the compiler parses it. For example, a programmer wanting to modify the incorrect example above could change it into

```
if (position < size)
    position++;
    remaining--;
```

to mean that both (increment and decrement) statements should be executed when the condition is true. Since the indentation is consistent with his intent, it could be difficult for him to spot the lack of the braces. This problem would not have arisen, had the program been written as in the correct example above.

Example

Instead of writing

```
if (position < size) // Wrong, add braces!
    position++;
```

write

```
if (position < size)
{
    position++;
} // end if
```

Rule 13: Format `if-else` statements according to the following models:

Simple branch `if` statement:

```
if (condition)
{
    statements;
} // end if
```

`if` statement with `else`:

```
if (condition)
{
    statements;
```

```
} // end if
else
{
    statements;
} // end else
```

Multi-branch `if` statement:

```
if (condition)
{
    statements;
} // end if
else if (condition)
{
    statements;
} // end if
else
{
    statements;
} // end else
```

The indentation of `if` statements with an `else` clause and of multi-branch `if` statements differs slightly from that recommended by the SUN standard [SUNCode]. Putting `else` clauses in a separate line increases the readability of the whole structure.

Rationale

Formatting conventions are necessary to achieve code uniformity.

Rule 14: Format `for` statements according to the following model:

```
for (initialization; condition; update)
{
    statements;
} // end for
```

`for` statements without a body should be formatted as follows:

```
for (initialization; condition; update);
```

(note the semicolon "`;`" at the end.)

Rationale

Formatting conventions are necessary to achieve code uniformity.

Rule 15: Format `while` statements according to the following model:

```
while (condition)
{
    statements;
} // end while
```

Rationale

Formatting conventions are necessary to achieve code uniformity.

Rule 16: Format `do-while` statements according to the following model:

```
do
{
    statements;
} while (condition);
```

Rationale

Formatting conventions are necessary to achieve code uniformity.

Rule 17: Format `switch` statements according to the following model:

```
switch (condition)
{
case CASE1:
    statements;
    /* falls through */
case CASE2:
    statements;
    break;
case CASE3:
    statements;
    break;
default:
    statements;
    break;
} // end switch
```

- Cases without a `break` statement should include a `/* falls through */` commentary to indicate explicitly that they *fall through* to the next case.
- All `switch` statements should include a `default` case.
- The last case in the `switch` statement should also end with a `break` statement.

Rationale

Formatting conventions are necessary to achieve code uniformity.

Rule 18: Format `try-catch` statements according to the following model:

Simple `try` statement:

```
try
{
    statements;
} // end try
catch (ExceptionClass e)
{
    statements;
} // end catch
```

`try` statement with `finally` clause:

```
try
```

```
{
    statements;
} // end try
catch (ExceptionClass e)
{
    statements;
} // end catch
finally
{
    statements;
} // end method
```

Rationale

Formatting conventions are necessary to achieve code uniformity.

Recommendation 6: Avoid parentheses around the return values of `return` *statements.*

Example

Instead of writing

```
return(total);
```

or

```
return (total);
```

simply write

```
return total;
```

Rationale

Formatting conventions are necessary to achieve code uniformity.

3.5 Blank Lines and Spaces

Rule 19: Leave *two* blank lines:

- Between sections of a source file.
- Between class and interface definitions.

Rationale

Blank lines in the specified places help identify the general structure of an implementation file.

Rule 20: Leave *one* blank line:

- Between methods.
- Between the local variable definitions in a method or compound statement and its first statement.

Rationale

Blank lines in the specified places help identify the structure of class and method definitions.

Recommendation 7: Separate groups of statements in a method using single blank lines.

Statements in a method can usually be broken into groups that perform conceptually separate tasks (i.e. basic steps in an algorithm). It is a good idea to separate such groups with single blank lines to make them more obvious.

Rationale

This practice helps to better communicate the intent of the program.

Example

```
void printAttribList(Attributes atts)
{
    int length = atts.getLength();
    int i;

    for (i = 0; i < length; i++)
    {
        if (SAXHelpers.isXMLAttrib(atts, i, "space"))
        {
            // Omit xml:space declarations in HTML.
            break;
        } // end if

        // Find a suitable attribute name.
        String name = atts.getQName(i);
        if (name == null || name.equals(""))
        {
            name = atts.getLocalName(i);
        } // end if

        ti.printMarkup(" " + name + "=\"");

        // Print the attribute value, but don't compress spaces.
        ti.escapeText(atts.getValue(i));

        ti.printMarkup("\"");
    } // end for
} // end method
```

Rule 21: Always use a space character:

- After commas in argument lists.
- Before and after all binary operators, except for the "`.`" operator.
- After the semicolons ("`;`") separating the expressions in a `for` statement.
- After casts in expressions.

Rationale

Space characters in the specified places help readability:

- They are located at the end or around certain syntactical constructs, and thus help to identify them visually.
- They are consistent with common typographical conventions that are well known to improve readability.

Chapter 4
Naming

4.1 Introduction

Projects often undergo periods of fast development, where developers concentrate on implementing functionality, and leave other important, but not so urgent tasks like documentation for a later time. A key implication of this fact is that often, the only part of an overall software product that is up to date is its source code. For this reason, clear and self-documenting code is always very valuable. Choosing descriptive program identifiers is one important step to achieve such code.

This chapter is concerned with issues of naming within Java source code. The Java programming language allows long and meaningful names, consistent with what modern software engineering practices encourage. Since names are used in a variety of contexts and scopes, this chapter covers the various naming situations starting with the more general and wider ones and going down to more specific and local ones.

4.2 General Naming Conventions

The rules and recommendations in this section apply to all types of identifiers possible in a Java program.

Rule 22: Use American English for identifiers.

Identifiers shall correspond to English words or sentences, using the American spelling (i.e. “`Color`”, “`initialize`”, “`Serializable`”, instead of “`Colour`”, “`initialise`” or “`Serialisable`”.)

Rationale

Mixed American and British spelling can result in identifiers being mistyped. Additionally, the Java standard library uses American spelling for its identifiers.

Rule 23: Restrict identifiers to the ASCII character set.

Rationale

Files containing non-ASCII identifiers may not display properly in some platforms and may be hard to edit properly.

Recommendation 8: Pick identifiers that accurately describe the corresponding program entity.

While coding, spend some time looking for identifiers that accurately and con-

cisely describe the program entity (class, package, instance, local variable, etc.) in question.

Names should be short, yet meaningful. The choice of a name should be mnemonic, i.e., designed to indicate to the casual observer the intent of its use. One-character variable names should be avoided, except for temporary “throwaway" variables. Common names for temporary variables are `i`, `j`, `k`, `m`, and `n` for integers; `c`, `d`, and `e` for characters.

Rationale

Naming conventions are necessary to achieve code uniformity.

Recommendation 9: Use terminology applicable to the domain.

While choosing identifiers, rely as much as possible on accepted, domain specific terminology.

Rationale

Doing so can facilitate communication between programmers and other professionals involved in a project. It also helps to produce more concise and accurate identifiers.

Recommendation 10: Avoid long (e.g. more than 20 characters) identifiers.

While trying to keep names descriptive, avoid using very long identifiers.

Rationale

Extremely long identifiers may be hard to remember, are difficult to type, and may make code harder to format properly.

Recommendation 11: Use abbreviations sparingly and consistently.

Although abbreviating words in identifiers should be generally avoided, it is sometimes necessary to do it in order to prevent identifiers from becoming excessively long. In such a case, ensure that the same abbreviation is always used for the same term.

Rationale

Naming conventions are necessary to achieve code uniformity.

Rule 24: Avoid names that differ only in case.

Never use in the same namespace identifiers that have the same letters, but that are capitalized in different ways. As an additional recommendation, generally avoid putting similar identifiers in the same namespace.

Rationale

Similar names in a single namespace easily lead to coding errors.

Rule 25: Capitalize the first letter of standard acronyms.

When using standard acronyms in identifiers, capitalize only the first letter, not the whole acronym, even if such acronym is usually written in full upper-case.

Example

Use `XmlFile`, `auxiliaryRmiServer` and `mainOdbcConnection` instead of `XMLFile`, `auxiliaryRMIServer` or `mainODBCConnection`.

Rationale

Doing this allows for clearer separation of words within the name, making identifiers easier to read. When only the first letter is capitalized, words are more easily distinguished without any one word being dominant.

Rule 26: Do not hide declarations.

Do not declare names in one scope that hide names declared in a wider scope. Pick different names as necessary.

Rationale

Errors resulting from hiding a declaration, but still trying to directly refer to the declared element, may be very difficult to spot.

4.3 Package Names

Rule 27: Use the reversed, lower-case form of your organization's Internet domain name as the root qualifier for your package names.

Whenever possible, any package name should include the lower-case domain name of the originating organization, in reverse order.

In the case of ESA, however, names beginning with the components "`int.esa`" are not possible, since "`int`" is a reserved word in the Java programming language and will be rejected by compilers and other tools compliant with the language definition. The recommended package naming schema for packages developed by ESA projects is

`esa.projectname.applicationname.componentname`

Additionally, the "`java`" and "`javax`" package name components must not be used because they are reserved for the standard libraries and for extension packages provided directly by Sun Microsystems.

Rationale

Naming conventions are necessary to achieve code uniformity.

Example

```
esa.galileo.ipf.ifcalculator
esa.herschel.mps.scheduler
```

Rule 28: Use a single, lower-case word as the root name of each package.

The qualified portion of a package name should consist of a single, lower-case word that clearly captures the purpose and utility of the package. A package name may consist of a meaningful abbreviation.

Rationale

Naming conventions are necessary to achieve code uniformity.

4.4 Type, Class and Interface Names

Rule 29: Capitalize the first letter of each word that appears in a class or interface name.

Rationale

The capitalization provides a visual cue for separating the individual words within each name. The leading capital letter allows for differentiating between class or interface names and variable names.

Example

```
public class PrintStream extends FilterOutputStream
{
...
} // end class
public interface ActionListener extends EventListener
{
...
} // end interface
```

Rule 30: Use nouns or adjectives when naming interfaces.

Use nouns to name interfaces that act as service declarations. Use adjectives to name interfaces that act as descriptions of capabilities. The latter are frequently named with adjectives formed by tacking an “able” or “ible” suffix onto the end of a verb.

Rationale

An interface constitutes a declaration of the services provided by an object in which case a noun is an appropriate name, or it constitutes a description of the capabilities of an object, in which case an adjective is an appropriate name.

Example

An interface declaring a service:

```
public interface ActionListener
{
    public void actionPerformed(ActionEvent e);
} // end interface
```

Interfaces declaring object capabilities:

```
public interface Runnable
{
    public void run();
} // end interface
public interface Accessible
{
    public Context getContext();
} // end interface
```

Rule 31: Use nouns when naming classes.

Rationale

Classes represent categories of objects of the real world. Such objects are normally referred to using nouns.

Example

Some examples from the Java standard class library:

```
SocketFactory
KerberosTicket
MediaName
Modifier
StringContent
Time
```

Rule 32: Pluralize the names of classes that group related attributes, static services or constants.

Give classes whose instances group related attributes, static services, or constants a name that corresponds to the plural form of the attribute, service, or constant type defined by the class.

Rationale

The plural form makes it clear that the instances of the class are collections.

Example

Some examples from the Java standard class library:

```
BasicAttributes
LifespanPolicyOperations
PageRanges
RenderingHints
```

4.5 Method Names

Rule 33: Use lower-case for the first word and capitalize only the first letter of each subsequent word that appears in a method name.

Rationale

The capitalization provides a visual cue for separating the individual words within each name. The leading lower-case letter allows for differentiating between a method and a constructor invocation.

Example

```
insertElement
computeTime
save
extractData
```

Rule 34: Use verbs in imperative form to name methods that:

- Modify the corresponding object (modifier methods).
- Have border effects (i.e., displaying information, writing to or reading from a storage device, changing the state of a peripheral, etc.)

Rationale

Modifiers and methods with border effects perform actions, which are better described by verbs.

Example

Modifier methods:

```
insert
projectOrthogonal
deleteRepeated
```

Methods with border effects:

```
save
drawLine
sendMessage
```

Rule 35: Use verbs in present third person to name analyzer methods returning a boolean value.

Particularly, for classes implementing JavaBeans using the verb "`is`" may be necessary.

Rationale

Methods returning a boolean value test some characteristic of the corresponding object.

Example

```
isValid
hasChild
canPrint
```

Rule 36: Use nouns to name analyzer methods returning a non-boolean value, or, alternatively, name them using the verb "`get`".

Use the "`get`" style if:

- The class implements a Java Bean.
- There is a corresponding "`set`" method.
- The result of the method corresponds directly to the value of an instance variable.

Rationale

The "`get`" convention is widely accepted by the Java community.

Example

```
totalTime
getTotalTime
getLength
validSuccesors
```

Rule 37: Name methods setting properties of an object (set methods) using the verb "`set`".

Use this convention when:

- The class implements a Java Bean.
- There is a corresponding "`get`" method.
- The method's purpose is to set the value of an instance variable.

Rationale

The "`set`" convention is widely accepted by the Java community.

4.6 Variable Names

Rule 38: Use nouns to name variables and attributes.

A variable name must correspond to an English noun, potentially accompanied by additional words that further describe or clarify it.

Rationale

Variables represent either entities from the real world or digital entities used inside a program. In both cases, those entities are referred to in natural language using nouns.

Example

```
shippingAddress
counter
currentPosition
maximalPower
```

4.6.1 Parameter Names

Rule 39: When a constructor or “set” method assigns a parameter to a field, give that parameter the same name as the field.

Rationale

While hiding the names of instance variables with local variables is generally poor style, this particular case brings some benefits. Using the same name relieves the programmer of the responsibility of coming up with a different name. It also provides a clue to the reader that the parameter value is destined for assignment to the field of the same name.

Example

```
class TestFacility
{
    private String name;

    public TestFacility(String name)
    {
        this.name = name;
    } // end method

    public setName (String name)
    {
        this.name = name;
    } // end method
} // end class
```

4.6.2 Instance Variable Names

Rule 40: Qualify instance variable references with `this` to distinguish them from local variables.

Rationale

To facilitate distinguishing between local and instance variables, always qualify field variables using the `this` keyword.

Example

```
public class AtomicAdder
{
    private int count;
```

```
    public AtomicAdder(int count)
    {
        this.count=count;
    } // end method

    public synchronized int fetchAndAdd(int value)
    {
        int temp = this.count;
        this.count += value;
        return temp;
    } // end method

    ...
} // end class
```

4.7 Constant Names

Rule 41: Use upper-case letters for each word and separate each pair of words with an underscore when naming Java constants.

Rationale

The capitalization of constant names distinguishes them from other non-final variables:

Example

```
class Byte
{
    public static final byte MAX_VALUE = +127;
    public static final byte MIN_VALUE = 0;
    ...
} // end class
```

Chapter 5
Documentation and Commenting Conventions

5.1 Introduction

As discussed elsewhere in this document, the final and most reliable source of information about a project is its own source code. However, source code is often difficult to interpret on its own, thus making it necessary to write additional explanatory documentation. A very practical way to keep such documentation are code comments. Documentation in comments is generally easier to maintain, since it is located closer to the documented code. It also makes it more practical to maintain the code itself, because it reduces the need to refer to separate documents when understanding it.

Java developers have an additional reason to document their code using comments: the Javadoc tool [SUNDoc]. Javadoc takes as input a set of Java source files, extracts information from especially formatted comments in them, and produces well structured, cross-referenced documentation as a result. The Javadoc approach combines the practicality of keeping documentation in code comments with the convenience of having separate, high level reference documentation for the code interfaces in a system.

The rules and recommendations in this chapter are concerned with how to write appropriate source level documentation for Java. Particularly, many of the rules and recommendations have to do with how to write comments in such a way that the Javadoc tool can produce high quality reference documentation from them.

5.2 Comment Types

The Java programming language supports three comment types:

- A one-line or end-line comment that begins with "//" and continues through to the end of the line.
- A standard, or C-style, comment, which starts with "/*" and ends with "*/".
- A documentation comment that starts with "/**" and ends with "*/". The Javadoc tool processes only comments of this type.

5.3 Documenting the Detailed Design

Recommendation 12: Use documentation comments to describe programming interfaces before implementing them.

Rationale

The detailed design of a class interface (as opposite to the detailed design of its

implementation) consists of the method signatures, together with their specifications. An excellent way to document such a design is to write a skeleton class definition, consisting of the method declarations (with no code in their bodies) and corresponding documentation comments specifying them. The best time to write this documentation is early in the development process, while the purpose and rationale for introducing the new classes or interfaces is still fresh in your mind.

Following this practice makes it possible to run Javadoc in the earliest stages of implementation, to produce documents that can be used for reviewing the design as well as for guiding the developers implementing it. Additionally, this documentation constitutes a solid basis for any final API reference documentation that may need to be produced.

5.4 Javadoc General Descriptions

Rule 42: Provide a summary description and overview for each application or group of packages.

The Javadoc utility provides a mechanism for including a package-independent overview description in the documentation it generates. Use this capability to provide an overview description for each application or group of related packages you create.

The Javadoc documentation [SUNDoc] explains how to make use of this feature.

Rationale

Adequate overview documentation is fundamental for program comprehension.

Rule 43: Provide a summary description and overview for each package.

The Javadoc utility provides a mechanism for including package descriptions in the documentation it generates. Use this capability to provide a summary description and overview for each package you create.

The Javadoc documentation [SUNDoc] explains how to make use of this feature.

Rationale

Adequate overview documentation is fundamental for program comprehension.

5.5 Javadoc Comments

Rule 44: Use documentation comments to describe the programming interface.

Place documentation comments in front of any class, interface, method, constructor, or field declaration that appears in your code.

Rationale

These comments provide information that the Javadoc utility uses to generate hypertext-based, reference, Application Programming Interface (API) documentation.

Rule 45: Document public, protected, package, and private members.

Supply documentation comments for all members, including those with package, protected, and private access.

Rationale

The developer who must understand your code before implementing an enhancement or a defect fix will appreciate your foresight in providing quality documentation for all class members, not just for the public ones.

Rule 46: Use a single consistent format and organization for all documentation comments.

A properly formatted documentation comment contains a description followed by one or more Javadoc tags. Format each documentation comment as follows:

- Indent the first line of the comment to align the slash character of the start comment symbol "`/**`" with the first character in the line containing the associated definition.
- Begin each subsequent line within an asterisk "`*`". Align this asterisk with the first asterisk in the start comment symbol.
- Use a single space to separate each asterisk from any descriptive text or tags that appear on the same line.
- Insert a blank comment line between the descriptive text and any Javadoc tags that appear in the comment block.
- End each documentation comment block with the asterisk in the end comment symbol "`*/`" aligned with the other asterisks in the comment block.

Rationale

Formatting conventions are necessary to achieve code uniformity.

Example

```
/**
* Descriptive text for this entity.
*
* @tag Descriptive text for this tag.
*/
```

Rule 47: Wrap keywords, identifiers, and constants mentioned in documentation comments with `<code>...</code>` tags.

Nest keywords, package names, class names, interface names, method names, field names, parameter names, constant names, and constant values that appear in a documentation comment within HTML `<code> ...</code>` mark-up tags.

Rationale

The `<code>` ... `</code>` tags tell HTML browsers to render the content in a style different from that of normal text, so that these elements will stand out.

Example

```
/**
* Allocates a <code>Flag</code> object
* representing the <code>value</code> argument.
* ...
*/
public Flag(boolean value)
{
    ...
} // end method
```

Rule 48: Wrap full code examples appearing in documentation comments with `<pre>` ... `</pre>` tags.

Rationale

The `<pre>` ...`</pre>` tags are used to tell HTML browsers to retain the original formatting, including indentation and line ends, of the “preformatted” element.

Recommendation 13: Consider marking the first occurrence of an identifier with a `{@link}` tag.

Rationale

Each package, class, interface, method, and field name that appears within a documentation comment may be converted into a hypertext link by replacing its name with an appropriately coded `{@link}` tag. Some classes and methods are so frequently used and well known that it is not necessary to link to their documentation every time they are mentioned. Create links only when the documentation associated with the referenced element would truly be of interest or value to the reader. This makes documentation generally easier to read and maintain.

Rule 49: Include Javadoc tags in a comment in the following order:

`@author`

`@param`

`@return`

`@throws`

`@see`

`@since`

`@serial`

`@deprecated`

Rationale

Formatting conventions are necessary to achieve code uniformity.

Rule 50: Include an `@author` and a `@version` tag in every class or interface description.

List multiple `@author` tags in chronological order, with the class or interface

creator listed first.

Rationale

This way, the history of the file is easier to follow.

Rule 51: Fully describe the signature of each method.

The documentation for each method shall always include a description for each parameter, each checked exception, any relevant unchecked exceptions, and any return value.

Include a `@param` tag for every parameter in a method. List multiple `@param` tags in parameter declaration order. Include a `@return` tag if the method returns any type other than void. Include an `@exception` tag for every checked exception listed in a throws clause. Include an `@exception` tag for every unchecked exception that a user may reasonably expect to catch. List multiple `@exception` tags in alphabetical order of the exception class names..

Sort multiple `@see` tags according to their distance from the current location, in terms of document navigation and name qualification. Order each group of overloaded methods according to the number of parameters each accepts, starting with the method that has the least number of parameters:

```
/**
* ...
* @see #field
* @see #Constructor()
* @see #Constructor(Type...)
* @see Class
* @see Class#field
* @see Class#Constructor()
* @see Class#Constructor(Type ...)
```

Rationale

Formatting conventions are necessary to achieve code uniformity.

5.6 Comment Contents and Style

Recommendation 14: Document preconditions, post conditions, and invariant conditions.

The primary purpose for documentation comments is to define a programming contract between a client and a supplier of a service. The documentation associated with a method should describe all aspects of behavior on which a caller of that method can rely and should not attempt to describe implementation details.

Rationale

As preconditions, post conditions, and invariants are the assumptions under which you use and interact with a class, documenting them is important, especially if these conditions are too costly to verify using run-time assertions.

Recommendation 15: Include examples.

Rationale

One of the easiest ways to explain and understand how to use software is by giving specific examples. Use the HTML `<pre> ...</pre>` tags to guarantee that the formatting of the examples is preserved in the final documentation.

Rule 52: Document synchronization semantics.

Rationale

Methods declared as `synchronized` will be automatically marked as such in Javadoc generated documentation. Methods not declared as `synchronized` may, however, still be thread-safe, since they could explicitly implement any needed thread synchronization. In such a situation, you must indicate that the method is internally synchronized in the corresponding method documentation.

Recommendation 16: Use "this" rather than "the" when referring to instances of the current class.

When describing the purpose or behavior of a method, use "*this*" instead of "*the*" to refer to an object that is an instance of the class defining the method.

Rationale

Comments written this way are more accurate and easier to formulate.

5.7 Internal Comments

Recommendation 17: Document local variables with an end-line comment.

Document all but trivial local variables with end-line comments.

Rationale

Variable documentation is one of the most helpful aids to understand a program.

Example

```
int i;
float currentMax;     // Maximal value seen until now.
```

Rule 53: Add a "fall-through" comment between two case labels, if no break statement separates those labels.

When the code following a switch statement's case label does not include a break but, instead, "falls through" into the code associated with the next label, add a comment to indicate this was your intent. Note that two adjacent labels do not require an intervening comment.

Rationale

Other developers may either incorrectly assume a break occurs, or wonder

whether you simply forgot to code one.

Example

```
switch (command)
{
case FAST_FORWARD:
    isFastForward = true;
    // Fall through!
case PLAY:
case FORWARD:
    isForward = true;
    break;
case FAST_REWIND:
    isFastRewind =true;
    // Fall through!
case REWIND:
    isRewind = true;
    break ;
} // end switch
```

Rule 54: Label empty statements.

When a control structure, such as a `while` or `for` loop, has an empty block by design, add a comment to indicate that this was your intent.

Rationale

Empty blocks may be confusing for programmers trying to understand the code. Making them explicit helps to prevent such confusion.

Example

```
// Strip leading space
while ((c = Reader.read()) == SPACE)
{
    // Empty!
} // end while
```

Rule 55: Use end-line comments to explicitly mark the logical ends of conditionals loops, exceptions, enumerations, methods or classes.

The end-line comments must have the format

```
// end <keyword>
```

where *<keyword>* is one of `class`, `method`, `if`, `for`, `while`, `switch`, `constructor`, `interface`, `enum`, `try`, or `catch`.

Rationale

Largely improves the readability of code by making control structures more visible.

Example

See examples throughout this document.

Chapter 6
Java Design and Programming Guidelines

6.1 Introduction

The experience of years of software development with the Java programming language as well as with other object oriented languages has left a wide variety of valuable learned lessons that can be applied in new projects to improve almost every aspect of the development work.

The present chapter summarizes a number of such lessons, explaining, where necessary, their particular relevance to the Java language and programming environment.

6.2 Packages

A package is a conceptual unit consisting of a set of files which together implement a collection of interfaces and classes.

Recommendation 18: Use separate packages for each of the software components defined during the design phase.

The design phase shall break the overall software system into logical components, as a means of managing the complexity of the overall system. Each one of these components should correspond to a Java package, properly organized in a separate file system directory.

Rationale

Following this recommendation makes the file structure of the system reflect its conceptual structure. Putting separate components into packages also helps encapsulation, since the Java programming language offers mechanisms to protect the implementation of a package from being accessed by other, external packages.

Recommendation 19: Place into the same package types that are commonly used, changed, and released together, or mutually dependent on each other.

If a set of classes and/or interfaces are so closely coupled that you cannot use one without using the other, put them in the same package. Some examples of closely related types include:

- Containers and iterators.
- Database tables, rows, and columns.
- Calendars, dates, and times.
- Points, lines, and polygons.

Combine classes that are likely to change at the same time for the same reasons into a single package. If two classes are so closely related that changing one of them generally involves changing the other, place them in the same package.

Rationale

Packages are effective units of reuse and release. Effective reuse requires tracking of releases from a change control system. A package release captures the latest version of each class and interface.

Recommendation 20: Avoid cyclic package dependencies.

Take steps to eliminate cyclic dependencies between packages, either by combining mutually dependent packages or by introducing a new package of abstractions that many packages can depend on.

Rationale

Cyclic dependencies make systems more fragile and can make parallel development (i.e., development of many packages by many developers or teams working simultaneously) much more difficult.

Recommendation 21: Isolate volatile classes and interfaces in separate packages.

Separate volatile classes from stable classes to reduce the code footprint affected by new releases, thereby reducing the impact on users of said code. Avoid placing volatile classes or interfaces in the same package with stable classes or interfaces.

Rationale

Otherwise, when using packages as the unit of release, each time a package is released, the users must absorb the cost of reintegrating and retesting against all the classes in the package, although many may not have changed.

Recommendation 22: Avoid making packages that are difficult to change dependent on packages that are easy to change.

Do not make a package depend on less stable packages. If necessary, create new abstractions that can be used to invert the relationship between the stable code and the unstable code.

Rationale

Dependencies between packages should be oriented in the direction of increasing stability. A package should only depend on packages that are as stable, or more stable, than itself.

Recommendation 23: Maximize abstraction to maximize stability.

Separate abstract classes and interfaces from their concrete counterparts to form stable and unstable packages.

Rationale

The more abstract a package is, the more stable it is.

Recommendation 24: Capture high-level design and architecture as stable abstractions organized into stable packages.

Define packages that capture the high-level abstractions of the design. Place the detailed implementation of those abstractions into separate packages that depend on the high-level abstract packages.

Rationale

To plan and manage a software development effort successfully, the top-level design must stabilize quickly and remain that way.

Rule 56: Do not use the *wildcard* ("`*`") notation in `import` statements.

Java `import` statements have two possible forms. The first one imports a single class into the local module's name space, i.e.:

```
import java.util.Vector;
```

The second one imports all classes from a given package into the local module's name space, i.e.:

```
import java.util.*;
```

This second form must be avoided.

Rationale

Explicitly importing classes makes it much easier to know which package a class comes from. Additionally, since two separate packages could have classes or interfaces of the same name (i.e. `java.util.List` and `java.awt.List`) `import` statements using the wildcard notation can lead to name conflicts.

Rule 57: Put all shared classes and interfaces that are internal to a project in a separate package called "`internal`".

Classes and interfaces that are *internal* to a project, that is, that are not part of the project's public interface, and that are shared among other packages in the project, must be put in a separate package called "`internal`". Classes in the internal package could be freely structured in other packages.

Rationale

Internal classes that are shared among two or more packages must me declared `public`, in order to make the sharing possible at all. This has the side-effect of making them accessible also to external users of the project. Putting them in a separate, `internal` package, makes it clear that they are not part of the official interface of the system and should not be used outside it.

Rule 58: Make classes that do not belong to a package's public API `private`.

Rationale

This guarantees that these classes will not be accessed by external clients, and excludes them from the public documentation.

Recommendation 25: Consider using Java interfaces instead of classes for the public API of a package.

Rationale

Making interfaces `public` instead of their corresponding classes makes it easier to change the implementation later on, while keeping the internal API stable.

6.3 General Class Guidelines

Rule 59: Make all class attributes `private`.

Rationale

Making attributes `private` ensures consistency of member data, since only the owning class may change it. If necessary, a class can provide access to selected member data by defining appropriate public accessor methods.

This practice actually hides the underlying class implementation, making it possible to transparently change it without affecting the external class interface. The result is that code depending on the class does not need to change when internal details of a class are modified. If attributes were public, direct uses of them in other parts of the system would have to be adapted, with the consequent additional effort and increased risk of introducing defects.

Example

The classic example is a class representing complex numbers, which offers accessor routines for the real and imaginary parts of the complex number, as well as accessor routines for the modulus and argument of the complex number. Class users do not need to know which representation is used internally.

If the attributes are hidden, any interaction between the attributes can be strictly controlled within the class itself so that they are guaranteed to be in a consistent state, thereby reducing the amount of checking code that is needed before their values are used elsewhere.

Example

```
class CRange

{
    // Invariant: lowerLimit <= upperLimit

    private int lowerLimit;
    private int upperLimit;

    public void setLimits(int lowerLimit, int upperLimit)

{
        if (lowerLimit <= upperLimit)

        {
            this.lowerLimit = lowerLimit;
            this.upperLimit = upperLimit;
        } // end if
        else
```

```
            {
                this.lowerLimit = upperLimit;
                this.upperLimit = lowerLimit;
            } // end else
        } // end else
    } // end class
```

In the example above the programmer who uses this class does not have access to the individual attributes and is forced to use the `setLimits()` method, which guarantees that the limits are consistent.

Recommendation 26: Consider declaring classes representing fundamental data types as final.

Rationale

Declaring a class as `final` allows its methods to be invoked more efficiently. Simple classes representing fundamental data types such as, for example, a `ComplexNumber` class in an engineering package, often find widespread use within their target domain. In such a case, efficiency can become an issue of importance.

Of course, declaring your class as final will prohibit its use as a superclass. Nevertheless, there is seldom any reason to extend a class that implements a fundamental data type. In most such cases, object composition is a more appropriate mechanism for reuse.

Recommendation 27: .Reduce the size of classes and methods by refactoring.

Rationale

Smaller classes and methods are easier to design, code, test, document, read, understand, and use. Because smaller classes generally have fewer methods and represent simpler concepts, their interfaces tend to exhibit better cohesion. If a class or method seems too big, consider refactoring that class or method into additional classes or methods.

Recommendation 28: Avoid inheritance across packages; rely on interface implementation instead.

Rationale

Inheritance (as achieved by using the `extends` keyword) causes a strong coupling between a base class and its subclasses. Any change to the base class may generate unwanted behavior in the inheritance tree of subclasses. Although this sort of coupling is often tolerable inside a single package, it can cause problems in a larger system.

Inheritance should be avoided not only to reduce coupling, but to prevent the so-called fragile base class problem. Base classes are considered fragile when they can be modified in a seemingly safe way, but their new behavior, if inherited by derived classes, might cause them to malfunction. So, derived classes as well as base classes must be tested for the new behavior.

Flexibility is lost because explicit use of concrete classes names locks you into specific implementations, making down-the-line changes unnecessarily difficult. Programming to interfaces is at the core of flexible structure.

All these disadvantages can be avoided by relying on interface implementation, as the majority of design patterns [GAM] do. The following example demonstrates this:

Example

```
interface Stack

{
    void push(Object o);
    Object pop();
    void pushMany(Object[] source);
} // end interface

class SimpleStack implements Stack

{
    private int stackPointer;
    private Object[] stack;

    public Stack()

    {
        stackPointer = -1;
        stack = new Object[1000];
    } // end constructor

    public void push(Object o)

    {
        assert stackPointer < stack.length-1;

        ++stackPointer;
        stack[stackPointer] = o;
    } // end method

    public Object pop()

    {
        assert stackPointer >= 0;

        stackPointer--;
        return stack[stackPointer+1];
    } // end method

    public void pushMany(Object[] source)

    {
        assert (stackPointer + source.length) < stack.length;

        System.arraycopy(source, 0, stack, stackPointer + 1,
                         source.length);
        stackPointer += source.length;
    } // end method
} // end class

class MonitorableStack implements Stack

{
    private int highSizeMark;
    private int currentSize;
    SimpleStack stack;

    public MonitorableStack()
```

```
        {
            highSizeMark = 0;
            stack = new SimpleStack();
        } // end constructor

        public void push(Object o)

        {
            ++currentSize;
            if (currentSize > highSizeMark)

            {
                highSizeMark = currentSize;
            } // end if
            stack.push(o);
        } // end method

        public Object pop()

        {
            --currentSize;
            return stack.pop();
        } // end method

        public void pushMany(Object[] source)

        {
            currentSize += source.length;
            if (currentSize + source.length > highSizeMark)

            {
                highSizeMark = currentSize + source.length;
            } // end if

            stack.pushMany(source);
        } // end method

        public int maximumSize()

        {
            return highSizeMark;
        } // end method
    } // end class
```

Since the two implementations must provide versions of everything in the public interface, it is much more difficult to get things wrong.

6.4 Nested Classes, Inner Classes, and Anonymous Classes

An inner class is a nested class whose instance exists within an instance of its enclosing class and has direct access to the instance members of its enclosing instance. An inner class is a non-static nested class.

Inner classes are used primarily to implement adapter classes. You can also declare an inner class without naming it. This is a so-called *anonymous* class.

Recommendation 29: Limit the use of anonymous classes.

Limit the use of anonymous classes to classes that are very small (no more than a method or two) and whose use is well understood (i.e. AWT event handling adapter classes or `Enumerator` classes).

Rationale

Anonymous classes can make code difficult to read and, for this reason, its use should be reduced to a minimum.

Example

```
public class Stack
    {
    private Vector items;

    // Stack's methods and constructors
    ...

    public Enumeration enumerator()
    {
        return new Enumeration()
        {
            int currentItem = items.size() - 1;

            public boolean hasMoreElements()
            {
                return (currentItem >= 0);
            } // end method

            public Object nextElement()
            {
                if (!hasMoreElements())
                {
                    throw new NoSuchElementException();
                } // end if
                else
                {
                    return items.elementAt(currentItem--);
                } // end else
            } // end method
        } // end method
    } // end method
} // end class
```

6.5 Constructors and Object Lifecycle

Constructors are needed to create new objects and initialize them to a valid state. Constructors have a large impact on performance. Good coding rules, especially to minimize their use, are important for well performing applications.

Rule 60: A class shall define at least one constructor.

Rationale

The default constructor, i.e., the one with no arguments, is automatically provided by the compiler if no other constructors are explicitly declared. The default constructor supplied by the compiler will initialize data members to `null` or equivalent values. For many classes, this may not be an acceptable behavior.

Rule 61: Hide any constructors that do not create valid instances of the corresponding class, by declaring them as `protected` or `private`.

If a constructor produces class invalid instances (i.e., class instances that do not comply with the class invariant), hide it by declaring it as `protected` or `private`.

Rationale

Public constructors are always expected to return valid instances. Constructors that do not return valid instances are only conceivable as a means for implementing higher level functionality in class, and for that reason should always be hidden.

Rule 62: Do not call non-final methods from within a constructor.

Rationale

Subclasses may override non-final methods. Java's runtime system dispatches calls to such methods according to the actual type of the constructed object, before executing the derived class constructor. This means that when the constructor invokes the derived method, the instance variables belonging to the derived class may still be in an invalid state. To prevent this situation from happening, call only final methods from the constructor.

Recommendation 30: Avoid creating unnecessary objects.

Rationale

Because of the complex memory management operations it involves, object creation is an expensive process. Creating an object not only implies allocating its memory, but also taking care of releasing it later. Since the Java programming language provides an automatic garbage collector that operates transparently, Java programmers often forget that garbage collection cycles are expensive, and that they can seriously increase the overall load of a system.

It is often possible to rationalize the data structures in a program to use less objects. Also, situations requiring large numbers of objects to be created only to be discarded soon (i.e., nodes of a dynamically allocated queue in a network application) can be handled by collecting discarded objects in an additional data structure and reusing them when needed.

Recommendation 31: Avoid using the `new` keyword directly.

Rationale

From an object-oriented design point of view, the `new` keyword has two serious disadvantages:

- It is not polymorphic. Using `new` implies referencing a concrete class explicitly by name. `New` cannot deal with situations where objects of different classes must be created depending on dynamic conditions.
- It fails to encapsulate object creation. Situations where objects could be created only optionally cannot be properly handled with `new`.

Properly handling these drawbacks is, however, possible. A number of design patterns, like static factory methods, abstract factories, and prototypes [GAM] can be used to encapsulate object creation to make it more flexible and scalable.

Critical systems have special rules for memory allocation, that may supersede this recommendation.

Recommendation 32: Consider the use of static factory methods instead of constructors.

Rationale

Static factory methods are static class methods that return new objects of the class. Among their advantages, with respect to constructors, are:

- They can have descriptive names. This is particularly useful when you need to have a variety of different constructors that differ only in the parameter list.
- They do not need to create a new object every time they are invoked (that is, they encapsulate object creation).
- They can return an object of any subtype of their return type (that is, they are polymorphic).

Recommendation 33: Use nested constructors to eliminate redundant code.

Rationale

To avoid writing redundant constructor code, call lower-level constructors from higher-level constructors.

Example

This code implements the same low-level initialization in two different places:

```
class Equipment
{
    private String name;
    private double balance;

    private final static double DEFAULT_BALANCE = 0.0d;

    Equipment(String name, double balance)
    {
        this.name = name;
        this.balance = balance;
    } // end constructor

    Equipment(String name)
    {
        this.name = name;
        this.balance = DEFAULT_BALANCE;
    } // end constructor
} // end class
```

This code implements the same low-level initialization in one place only:

```
class Equipment
```

```
{
    private final static double DEFAULT_BALANCE = 0.0d;

    private String name;
    private double balance;



    Equipment(final String name, final double balance)

    {
        this.name = name ;
        this.balance = balance;
    } // end constructor

    Equipment(final String name)

    {
        this(name, DEFAULT_BALANCE) ;
    } // end constructor
} // end class
```

This approach is also helpful while validating parameters, as it typically reduces the number of places a given constructor argument appears.

Recommendation 34: Use lazy initialization.

Do not build something until you need it. If an object may not be needed during the normal course of the program execution, then do not build the object until it is required. Use an accessor method to gain access to the object. All users of that object, including within the same class, must use the accessor to get a reference to the object.

Rationale

Lazy initialization makes memory use more efficient.

Example

```
class Satellite

{
    private AocsSubsystem aocsSubsystem ;

    Satellite()

    {
        this.aocsSubsystem == null ;
    } // end constructor

    AocsSubsystem getAocsSubsystem()

    {
        if (this.aocsSubsystem == null)

        {
            this.aocsSubsystem = new AocsSubsystem();
        } // end if
        return this.aocsSubsystem;
    } // end constructor
} // end class
```

6.6 Methods

The Java programming language has very specific recommendations and rules for methods, the muscles of the code where the real processing is defined. Some of these rules and recommendations are related to performance, others are related to good object oriented programming practice, language constructs/keywords, or the use of threads.

Recommendation 35: Refrain from using the `instanceof` operator. Rely on polymorphism instead.

Do not use `instanceof` to choose behavior depending upon the object's type. Implement, instead, object-specific behavior in methods derived from an appropriate base class or interface.

Rationale

Choices based on `instanceof` must be modified every time the set of choice object types changes, leading to brittle code. Implementations based on polymorphism, on the other hand, enable clients to interact with the base abstraction without requiring any knowledge of the derived classes. This makes it possible to introduce new classes without modifying the client.

Rule 63: Methods that do not have to access instance variables shall be declared `static`.

Rationale

If a method does not require access to the state of an instance, i.e., it neither reads nor writes any instance variables, it must be declared `static`.

Example

```
double static middle(final double x1, final double x2)
{
    return (x1 + x2) / 2;
} // end method
```

Rule 64: A parameter that is not changed by the method shall be declared `final`.

Rationale

The compiler should be used to trap as many potential problems as possible. Declaring parameters that are meant to remain unchanged as `final` will make the compiler issue a warning if they are modified inadvertently.

The `final` declaration also provides the compiler with information that potentially allows it to optimize the code in ways that would not be possible otherwise.

6.7 Local Variables and Expressions

Expressions are the workhorse of an application. Coding rules and recommendations for expressions are needed in every coding language. The coding rules for the Java programming language related to expressions are very similar to those for other programming languages.

Recommendation 36: Use local variables for one purpose only.

Rationale

Programmers often “recycle” local variables, by using them for different purposes in different parts of a complex method (i.e., a single `int` variable is used as counter for two totally independent loops). Unfortunately, this practice can cause confusion when modifying the program, and even lead to errors difficult to spot.

Defining a new local variable for every separate task not only makes code clearer and easier to maintain, but can be as efficient as using a single variable, since most optimizing compilers will transparently allocate memory only for the local variables that are needed at a particular code spot.

Recommendation 37: Replace repeated non-trivial expressions with equivalent methods.

Factor out common functionality and repackage it as a method or a class.

Rationale

Doing this makes code potentially easier to learn and understand. Changes are localized, thus reducing maintenance and testing effort.

Recommendation 38: Consider using the `StringBuffer` class when concatenating strings.

Rationale

Concatenating Java strings is a relatively expensive operation, because it always involves creating a new object to store the result. Algorithms that rely on performing large numbers of concatenation operations may incur large performance penalties if they are not implemented carefully.

The standard `StringBuffer` class was designed with this problem in mind. `StringBuffer` objects are dynamically growing text buffers that can handle an arbitrary number of concatenations without creating any new objects. Using them properly can lead to large performance improvements in text processing algorithms.

Rule 65: Use parentheses to explicitly indicate the order of execution of numerical operators .

Rationale

The default rules for order of execution of numerical operators as defined by the Java Programming Language differ sometimes from those traditionally used in mathematics. Usage of parenthesis is thus obligatory, in order to make explicit the order intended by the developer.

6.8 Generics and Casting

Generics are a feature introduced to the Java programming language during the JDK 5.0 development cycle, similar in spirit to C++ templates. They are particularly

useful for implementing generic collection classes (containers) such as trees, vectors, linked lists, and hash maps. In doing so, they make it possible to avoid the process of casting, which is highly error-prone and potentially inefficient, since casts must always be checked dynamically for type compatibility. The use of generics is highly recommended and is always preferable over the casting mechanism.

Generic classes are, in reality, not as straightforward as they might at first appear, especially when trying to provide true genericity. The design of a generic class needs some forethought, as that of any class which is used as a template parameter.

Recommendation 39: Use the enhanced `for` control structure and generics wherever possible/applicable.

Rationale

These features of the Java programming language not only make code simpler and easier to read, but safer and potentially more efficient. They constitute an excellent way of producing highly reusable code.

Example

First an example that iterates over a collection, without using any of the features:

```
void cancelAll(collection c)
{
    for (Iterator i = c.iterator(); i.hasNext();)
    {
        TimerTask tt = (TimerTask) i.next();
        tt.cancel();
    } // end for
} // end method
```

The new way as illustrated below is shorter, more readable and easier to understand:

```
void cancelAll(Collection<TimerTask> c)
{
    for (TimerTask task : c)
    {
        task.cancel();
    } // end for
} // end method
```

Rule 66: Use generics instead of casting when navigating through collections.

The original Java method of casting simple variables from one type to another is still available in newer versions, but its use is strongly discouraged.

Rationale

Casts can fail at run time and often make code unreadable.

Example

A method implemented using the traditional cast mechanism:

```
static void expurgate(Collection c)
{
    for (Iterator i = c.iterator(); i.hasNext();)
    {
        String s = (String) i.next();
        if (s.length() == 4)
        {
            i.remove();
        } // end if
    } // end for
} // end method
```

Casts instruct the compiler to override any type of information about an expression. Generics should be used instead as outlined below:

```
static void expurgate(Collection<String> c)
{
    for (Iterator<String> i = c.iterator(); i.hasNext(); )
    {
        if (i.next().length() == 4)
        {
            i.remove();
        } // end if
    } // end for
} // end method
```

It is now clear from the method signature that the input collection is only allowed to contain strings. A client program trying to pass in, for example, a collection of string buffers, would not even compile. This would not be the case with the cast-based implementation, which would only fail at run-time.

6.9 Constants and Enumerated Types

Recommendation 40: Be careful when using the `import static` feature to define global constants.

Rationale

The `import static` facility (available from J2SE SDK 5.0 onwards) eliminates the need to prefix static members with class names. It also eliminates the need to resort to problematic patterns, like defining constants inside an interface. The use of this feature, however, must be restricted to truly class independent constants. Constants related to a particular class, like those defining input or output values for class methods, should remain associated to the class they belong to.

In large projects, the `import static` feature can lead to code traceability problems. Because of that, it is important to use it carefully, if at all.

Example

A common (but not recommended) pattern is to put frequently used constants in an interface, to avoid explicitly referencing the class they belong to:

```
/* "Constant interface" antipattern - do not use */
public interface SpacePhysicsConstants
```

```
{
    public static final double LIGHT_SPEED = 3000000000d ;
    public static final double ELECTRON_MASS = 9.10938188e-31;
} // end interface

public class Satellite implements SpacePhysicsConstants
{
    public static void main(String[] args)
    {
        double mass = ...;
        double energy = (LIGHT_SPEED ^ 2) * mass;
        ...
    } // end main
} // end class
```

This confuses the clients of the class and creates a long-term commitment. If a set of constants is really so generic and global that it cannot be associated to any particular class, using `import static` is a better solution.

Example

The `import static` facility lets the programmer avoid qualifying static member names without subtyping. It is analogous in syntax to the package `import` facility, except that it imports static members from a class, rather than classes from a package:

```
import static esa.SpacePhysicsConstants.*;

class Satellite
{
    public static void main(String[] args)
    {
        double mass = ...;
        double energy = LIGHT_SPEED ^ 2 * mass;
        ...
    } // end main
} // end class
```

Recommendation 41: Use type-safe enumerations as defined using the `enum` keyword.

Rationale

The J2SE development cycle introduced a new feature to Java: typesafe enumerations. Typesafe enumerations offer a number of advantages with respect to the previous approach of using `static final` class variables (so-called `int` enumerations):

- They provide compile-time type safety. Values of an `int` enumeration could be assigned to variables intended to contain values of a different enumeration. Type-safe enumerations prevent that.
- They provide a proper name space for the enumerated type. With `int` enumerations, constants must be prefixed to avoid name clashes.
- They are robust. `int` enumerations are compiled into clients. Clients must be recompiled to add, remove, or reorder constants.

- They are informative when printed. Printing `int` enumeration constants displays only the numeric value.
- Because they are objects, you can put them in collections.
- Because they are essentially classes, you can add arbitrary fields and methods.

Example

```
public enum Planet

{
    MERCURY(3.303e+23, 2.4397e6),
    VENUS(4.869e+24, 6.0518e6),
    EARTH(5.976e+24, 6.37814e6),
    MARS(6.421e+23, 3.3972e6),
    JUPITER(1.9e+27,   7.1492e7),
    SATURN(5.688e+26, 6.0268e7),
    URANUS(8.686e+25, 2.5559e7),
    NEPTUNE(1.024e+26, 2.4746e7),
    PLUTO(1.27e+22,  1.137e6);

    private final double mass;   // in kilograms
    private final double radius; // in meters

    Planet(final double mass, final double radius)

    {
        this.mass = mass;
        this.radius = radius;
    } // end constructor

    private double setMass()

    {
        return mass;
    } // end method

    private double setRadius()

    {
        return radius;
    } // end method
} // end enum
```

6.10 Thread Synchronization Issues

Recommendation 42: Use threads only where appropriate.

Rationale

Threads are not a “silver bullet” for improving application performance. Depending on a variety of factors, the overhead required to switch between threads may indeed make an application slower.

Before introducing threads into an application, try to determine whether it can really benefit from their use. Consider using threads if your application needs:

- To react to many events simultaneously (i.e., an Internet server).

- To provide a high level of responsiveness (i.e., an interactive application that continues to respond to user actions even when performing other computations).
- To take advantage of machines with multiple processors.

Recommendation 43: Reduce synchronization to the minimum possible.

Rationale

Synchronization is expensive. Acquiring and releasing the special objects necessary to synchronize a section of code is often a costly operation. Moreover, synchronization serializes access to an object, reducing concurrency.

Do not arbitrarily synchronize every public method. Before synchronizing a method, consider whether it accesses shared and non-synchronized states. If it does not, if the method only operates on its local variables, parameters, or synchronized objects, synchronization is probably not required. Additionally, do not synchronize classes that provide fundamental data types or structures.

Recommendation 44: Do not synchronize an entire method if the method contains significant operations that do not need synchronization.

Rationale

A method annotated with the `synchronized` keyword acquires a lock on the associated object at the beginning of the method and holds that lock until the end of the method. As is often the case, however, only a few operations within a method may require synchronization. In such a situation, method level synchronization can be much too coarse.

The alternative to method level synchronization is to use the synchronized block statement:

```
protected void processRequest ()
{
    Request request = getNextRequest();
    RequestId id = request.getId();

    synchronized (this)
    {
        RequestHandler handler = this.handlerMap.get(id);
    } // end method
    handler.handle(request);
} // end method
```

Though this does not pertain to safety critical Java (see chapter 9), it does seem to contradict the no synchronized block rule in safety critical Java. As an alternative, one could provide a synchronized private method called by the public or protected method to reduce the extent of the synchronization. Such a method could also be used by other public methods as well.

```
protected void processRequest()
{
      Request request = getNextRequest();
```

```
        RequestId id = request.getId();
        RequestHandler handler = getHandler(id);
        handler.handle(request);
    } // end method

    private synchronized RequestHandler getHandler(RequestId id)
    {
    return this.handlerMap.get(id);
    } // end method
```

Note that the example as given will not work, since the handler variable is defined within the synchronization block but is referenced after the end of the block.

Recommendation 45: Avoid unnecessary synchronization when reading or writing instance variables.

Rationale

The Java programming language guarantees that read and write operations are atomic for object references as well as for all primitive types, with the exception of `long` and `double`. Therefore, it is possible to avoid the use of synchronization when reading or writing atomic data. On the other hand, if the value of an atomic variable depends on, or is related to, those of other variables, synchronization may still be necessary.

Example

In the following example, the assignments of `x` and `y` must be synchronized together because they are interdependent values:

```
public void synchronized setCenter(int x, int y)
{
    this.x = x;
    this.y = y;
} // end method
```

The following example does not require synchronization because it uses an atomic assignment of an object reference:

```
public void setCenter (Point p)
{
    this.point = (Point) p.clone() ;
} // end method
```

Please note, that the example given trades off less synchronization for more garbage collection overhead. However, if Point is immutable, then *clone* is not necessary.

Recommendation 46: Use synchronized wrappers to provide synchronized interfaces.

Use synchronized wrappers to provide synchronized versions of classes in situations where they are needed, i.e. to protect the integrity of shared data and to communicate program state changes efficiently between cooperating threads. Synchronized wrappers provide the same interface as the original class, but their methods are synchronized. A static method of the wrapped class provides access to the synchronized wrapper.

Rationale

Synchronized wrappers allow for the same code to be used in an unsynchronized and in a synchronized fashion in the same process.

Example

The following example (not meant to be run or be a full class definition but only to show the principle) demonstrates a stack, which has a default, non-synchronized interface and a synchronized interface provided by the wrapper class.

```
public class Stack
{
    public void push (Object o)
    {
        ...
    } // end method

    public Object pop()
    {
        ...
    }// end method

    public static Stack createSynchronizedStack()
    {
        return new SynchronizedStack();
    } // end method
} // end class

class SynchronizedStack extends Stack
{
    public synchronized void push (Object o)
    {
        super.push(o);
    } // end method

    public synchronized Object pop()
    {
        return super.pop();
    } // end method
} // end class
```

Please note that having a factory method for a subclass creates a circular dependency. This will often not be possible without modifying code from

other sources (libraries). If one does have control over the original class and one wants a factory method, it may be better to use an inner class for the synchronized version.

Recommendation 47: Consider using `notify()` instead of `notifyAll()`.

The `notify()` and `notifyAll()` methods are used when a thread must block waiting for other threads to perform a particular operation. Whenever possible, use the more efficient `notify()` method instead of its `notifyAll()` counterpart.

Use `notify()` when threads are waiting on a single condition and when only a single waiting thread may proceed at a time.

Use `notifyAll()` when threads may wait on more than one condition or if it is possible for more than one thread to proceed in response to a signal.

Rationale

The `notify()` method is more efficient.

Recommendation 48: Use the double-check pattern for synchronized initialization.

Use the double-check pattern in situations where synchronization is required during initialization, but not after it.

Rationale

The double-check pattern makes it possible to avoid expensive synchronization operations in many common situations.

Example

This code also protects against simultaneous initialization but it uses the double-check pattern to avoid synchronization except during initialization:

```
Log getLog()
{
    if (this.log == null)
    {
        synchronized (this)
        {
            if (this.log == null)
            {
                this.log = new Log();
            } // end if
        } // end method
    } // end if
    return this.log;
} // end method
```

Chapter 7
Robustness

7.1 Introduction

The concept of robustness applies to software development at different levels. A running software system is considered robust if it reacts properly to unexpected situations. A piece of code is considered robust when it is instrumented in such a way that it helps detect and correct errors (both in programming and during execution).

This chapter is concerned with robustness in Java systems. Some of the rules and recommendations are directed to achieving more robust code (concretely, the sections on design by contract and assertions). Others, particularly those related to error handling, are more concerned with run-time robustness. The overall application of the rules and recommendations in this chapter should lead to more reliable systems.

7.2 Design by Contract

Recommendation 49: Define method contracts and enforce them.

Define a “contract” for each method you write, consisting of its pre- and post-conditions:

- The preconditions are the set of logical conditions on the object state and parameter values that must hold in order for the method to be able to perform its task.
- The postconditions are the set of logical conditions on the object state and method return value that must hold after the method has performed its task.

Preconditions are the part of the contract a method caller must comply with. Postconditions are the part of the contract a method implementation must fulfill.

Whenever possible, a program should use assertions or other adequate means to explicitly check for method contracts being respected. Unfortunately, though, some logical pre- or postconditions of a method could be technically very difficult or onerous to check. Do your best, however, to always check as much as technically feasible.

Rationale

The usage of contracts is a well known way to make modules more robust and reliable.

Recommendation 50: Whenever possible, a method should either return the result specified by its contract, or throw an exception when that is not possible.

Rationale

The ideal behavior for a method is to return normally only when its contract was completely fulfilled. If this is not the case, the method should terminate abnormally with an exception. This can be achieved by properly checking the pre- and postconditions of a method using an adequate combination of normal conditions and assertions.

Rule 67: Preserve method contracts in derived classes.

Methods that override methods in a base class must preserve the pre- and postconditions specified in the base class. More concretely:

- A subclass method is not allowed to strengthen the preconditions of its counterpart in a superclass.
- A subclass method is not allowed to weaken the postconditions of its counterpart in a superclass.

Rationale

Only by respecting this rule is it possible to guarantee that instances of a superclass can be safely and transparently substituted by instances of one of its subclasses. Such substitutability is a key principle of object-oriented design.

7.3 Assertions

Recommendation 51: Rely on Java's `assert` statement to explicitly check for programming errors in your code.

Use Java's `assert` statement liberally to ensure that the basic premises upon which a piece of code was designed and built actually hold during runtime.

Rationale

Assertions are a simple, yet very valuable mechanism that can greatly help to diagnose problems during testing and even after deployment. Additionally, since assertions can be easily disabled, their impact on performance in a production environment is usually negligible.

Rule 68: Explicitly check method parameters for validity, and throw an adequate exception in case they are not valid. Do not use the `assert` statement for this purpose.

Rationale

It is a common programming error to invoke a method with invalid parameter values, i.e., parameter values that do not lie within the value ranges expected by the method. The accepted practice in the Java programming community is to check for parameter validity using standard code (as opposite to assertions) in order to keep such checks permanently enabled during program execution, regardless of whether

assertions are enabled or not.

Checking for parameter validity and, generally, for method preconditions, helps to guarantee that established interfaces in a system are being used as specified. This, in turn, helps to reduce the coupling in a system, by making it possible to replace module or component implementations when necessary.

Rule 69: Add diagnostic code to all areas that, according to the expectations of the programmer, should never be reached.

Areas of a program that are not expected to ever be reached by the flow of control (for example, `default` cases for `switch` statements, when it is known that the `case` sections cover all possible situations) should still contain code to produce a diagnostic message if they are reached due to a programming error.

Rationale

Adding such diagnostic code makes it easier to detect and correct a number of potential software defects. This is an effective form of defensive programming.

Rule 70: Do not use expressions with side effects as arguments to the `assert` statement.

Rationale

Besides producing a result value, expressions with side effects potentially modify variable values (think of an expression invoking a method that not only returns a value but modifies the object). If an expression with a side effect is used as argument for an `assert` statement, the behavior of the program will probably change when assertions are disabled. Since it should always be possible to disable assertions for a program or subsystem, they are not allowed to have side effects.

7.4 Debugging

Rule 71: Use the Java logging mechanism for all debugging statements instead of resorting to the `System.out.println` function.

Rationale

The system streams (e.g. `in`, `out`, `err`) should only be used by command line applications. Additionaly, using the logging mechanism makes it possible to keep all debugging statements in the code, and to use them at the finest granularity during trouble shooting. During normal operation of the software, on the other hand, logging can be set to "coarse" and, as such, will not process any debug messages. It also saves work in trying to find the source of stray `System.out.println` statements left in the code after debugging.

7.5 Exceptions and Error Handling

Exceptions are the main technique the Java programming language offers to handle errors and abnormal conditions. They have a number of advantages over more traditional error management techniques:

- They cleanly separate error handling code from other code.

- They automatically propagate errors up the call stack in a fashion that is compatible with the general program control structure.
- They allow grouping of error types, allowing for more flexible and structured error handling.

The rules and recommendations in this section are related to the appropriate use of exceptions in Java programs.

Rule 72: Use unchecked, run-time exceptions to handle serious unexpected abnormal situations, including those that may indicate errors in the program's logic.

Use unchecked exceptions (i.e., exception objects derived directly or indirectly from either the `java.lang.Error` or the `java.lang.RuntimeException` class) to handle situations typically arising from programming errors, or from abnormal situations of such a severe nature that program termination is imminent.

Rationale

The situations described by this rule are such that there is not much an application can do to handle them properly. Solving such situations usually requires external intervention (for example, to fix a code defect). Consequently, it is not worthwhile to make the program logic more complex because of them, which would be the case if checked exceptions were used.

Unchecked exceptions can still be trapped if the program has a reasonable way of handling them.

Example

Unchecked exceptions could be used to handle situations such as:

- A failed assertion (the assert `statement` automatically throws an exception).
- An out-of-bounds index.
- A division by zero.
- An attempt to dereference a null reference.
- A serious input/output error.
- An operating system failure.

Rule 73: Use checked exceptions to report errors that may occur, even if rarely, under normal program operation.

Use checked exceptions to handle problematic situations that may occur during normal program operation. In many cases, such problems can be handled appropriately by the caller of the method throwing the exception.

Rationale

Errors that may occur under during normal program operation must be reported through checked exception in order to make the caller aware of the fact that they should be handled in some way or another.

Example

Checked exception could be used to handle situations such as:

- The user typed invalid information.
- Incorrectly formatted information was read from an open network connection.
- A client does not have the required security privileges.

Rule 74: Do not silently absorb a run-time or error exception.

Rationale

Breaking this rule makes code hard to debug because valuable information gets lost. Even if you have coded a `catch` block simply to catch an exception you do not expect to occur, print at least a stack trace. You never know when something "impossible" might occur within your software.

Example

```
try
{
    for (int i = v.size() - 1; i >= 0; i--)
    {
        ostream.println(v.elementAt(i));
    } // end for
} // end try
catch (ArrayIndexOutOfBoundsException e)
{
    // Should never get here but if we do, nobody will ever know.
    // Print a stack trace just in case.
    e.printStackTrace();
} // end catch
```

Recommendation 52: Whenever possible, use `finally` blocks to release resources.

Rationale

Once a `try` block is entered, the corresponding `finally` block is guaranteed to be executed, regardless of whether an exception is thrown or not. This makes the `finally` block an ideal place to release any resources acquired prior to entering or within the `try` block.

Recommendation 53: Only convert exceptions to add information.

It is often necessary to trap an exception only to re-throw it right away (for example, because another type of exception is now required). While doing so, do not discard any information, but add any new information you may have to the existing exception.

Rationale

Discarding information may make some problems extremely difficult to diagnose. In certain cases, it may lead to problems going undetected that would have

otherwise been noticed.

Rule 75: Never ignore error values reported by methods.

Although the typical Java practice is to rely on exceptions for handling error conditions, some libraries still use special return values to indicate abnormal conditions. In such cases, always check for error values and code an appropriate response. A possible course of action is to wrap such methods into methods that check for error return values and throw appropriate exceptions when they appear.

Rationale

Ignoring error messages may difficult the detection of certain problems.

7.6 Type Safety

Recommendation 54: Encapsulate enumerations as classes.

Encapsulate enumerations as classes to provide type-safe comparisons of enumerator values. The `enum` construct, available in J2SE 5.0 and later, is a very convenient way of doing so.

Rationale

A proper encapsulated enumeration is type-safe, which means that the compiler is able to detect improper uses of the enumeration constants.

Chapter 8
Portability

8.1 Introduction

The Java platform is one of the few software technologies allowing both source and byte code to be made 100% platform-independent with relatively little effort. It means writing and compiling code once and running it everywhere without modification (under the condition, of course, that a suitable Java Virtual Machine is installed on each target platform). The result is complete independence of operating system and hardware type. The rules and recommendations in this chapter are intended to support programmers in producing 100% portable Java code.

It is important to point out, that some particular applications cannot be (and probably do not need to be) 100% portable. Code written for real-time Java virtual machines is one example. Applications that directly access hardware are another. Even in such cases, however, the rules and recommendations presented here could be helpful, since achieving a maximum of portability is always valuable.

8.2 Rules

Recommendation 55: Whenever possible, prefer the Swing API to the old AWT API for developing graphical user interfaces.

Rationale

The *Swing* classes are designed for 100% portability, and support a tunable look-and-feel (Windows-like, Unix-like, Motif-like, Linux-like, etc). They should be used in preference to the older AWT API. The AWT API will still run on various platforms, but will react or look differently depending on the underlying operating system.

Rule 76: Do not rely on thread scheduling particularities to define the behavior of your program, use synchronization instead.

Rationale

The way an operating system or its underlying hardware distribute processor cycles among threads, can vary widely from platform to platform. A program that expects the thread system to, for example, handle control from one thread to the next at a particular point in the execution, may work on some platforms but not on others. One implication of this fact is that no amount of testing is enough to guarantee that a thread-based algorithm is actually correct in a platform independent fashion. Careful inspection, or even formal verification are always necessary.

Unfortunately, this is a problem that is neither easy to detect nor to correct. De-

signing correct thread-safe algorithms requires applying proper principles and validating them thoroughly.

Example

Because of the lack of synchronization between threads, the program below could write "1" or "2", or even produce an error depending on the characteristics of the underlying thread implementation. This could not be noticed by casual testing. If the thread implementation, for example, guarantees that the assignment in method "run" is atomic, it would consistently print "2".

```
class Counter implements Runnable
{
    static int counterValue = 0;

    public void run()
    {
        counterValue += 1;
    } // end method

    public static void main(String[] args)
    {
        try
        {
            Thread thread1 = new Thread(new Counter());
            thread1.setPriority(1);
            Thread thread2 = new Thread(new Counter());
            thread2.setPriority(2);

            thread1.start();
            thread2.start();
            thread1.join();
            thread2.join();

            System.out.println(counterValue);
        } // end try
      catch (Exception e)
        {
            e.printStackTrace();
        } // end catch
    } // end main
} // end class
```

Rule 77: Avoid native methods.

Rationale

Native methods (i.e., methods written in C++ or other programming languages and linked to the Java virtual machine through the *Java Native Method Interface*) are usually tied to the platform they where developed in. Porting them to other platforms may be a big challenge, involving multi-platform compilation systems, and conditional code.

Recommendation 56: Do not use the `java.lang.Runtime.exec` method.

Rationale

The `java.lang.Runtime.exec` method starts arbitrary programs in the same computer system where the caller program is running, referencing them by name. Since the naming rules for programs differ widely among computing platforms, use of `java.lang.Runtime.exec` can never be made totally portable.

Note: This is quite a hard restriction. It is sufficient to avoid hard coding program names and to use a configuration file instead.

Recommendation 57: Do not hard-code display attributes, like position and size for graphical element, text font types and sizes, colors, layout management details, etc.

Rationale

It can be very difficult, if not impossible, to find a fixed set of display values that makes an application run properly in every single system. A judicious use of the system properties, together with proper application of relevant graphical toolkit facilities (for example, layout managers) should avoid most portability problems related to display attributes.

Recommendation 58: Check all uses of the Java reflection features for indirect invocation of methods that may cause portability problems.

Rationale

The reflection features available in the Java programming language allow indirectly instantiating arbitrary classes and invoking arbitrary methods. While doing this, a program may invoke methods that are not portable, or invoke portable methods in non-portable ways. Code using reflection must be carefully inspected for such problems if portability is a priority.

Rule 78: Restrict the use of the `System.exit` method to the cases described below.

The `System.exit` method terminates a program instantly. Restrict its use to:

- Fatal errors that absolutely require terminating the application immediately.
- Utility programs intended to be invoked from the command line or from script interpreters, where the program return value may be important.

Rationale

Sudden termination of a program may lead to undesired interactions with the user or with the operating system (i.e., a program suddenly closing all of its windows or terminating without releasing all resources). Such behaviors may be acceptable in some platforms and unacceptable in others.

Rule 79: Do not hard-code file names and paths in your program.

Rationale

File naming rules and rules to form file paths vary widely depending on the operating system. Hard coded names can be interpreted quite differently when a program is moved to a different platform. With careful programming, it is possible for a program to parse or build file paths portably by using the fields and methods in the `java.io.File` class. However, doing this should be left as a last resort. Programs should rely, as much as possible, on other facilities offered by the Java platform, like property files or file selection dialog boxes.

Example

Some file naming convention differences between the UNIX and DOS/Windows platforms:

- In UNIX the character separator is the slash ("`/`"), whereas in DOS/Windows it is the back-slash ("`\`").
- DOS/Windows uses drive letters (i.e. "`c:\windows`") while UNIX does not.
- Spaces are commonly used as part of Windows file names. They are unusual in UNIX and may cause problems in some systems.
- The period ("`.`") is a special character in DOS/Windows, separating the file name from its three-letter extension. In UNIX, periods are not special and can be (and often will be) used twice in a single name.
- DOS/Windows ignores case differences, whereas UNIX does not. Invoking the `java.io.FileInputStream` with the string "`README.TXT`" will work under Windows to open a file called "`Readme.txt`". In UNIX it will fail.

Rule 80: Always make JDBC driver names configurable, do not hard code them.

Rationale

JDBC drivers, particularly those including native code, can be less portable than the Java program using them. Making the specific driver name configurable through the standard `jdbc.drivers` system property, a property file, or some other, custom mechanism, guarantees that a program can be reconfigured to use a different driver if that happens to be necessary when moving it to a new platform.

Rule 81: Do not rely on a particular convention for line termination.

Rationale

The way text is represented in files is different for every single platform. Even on platforms using the ASCII character set, line ends are represented using a variety of characters and character combinations. Writing code that relies on a particular character or character sequence to terminate lines will lead to portability problems

Use the `readLine` method from the `java.io.BufferedReader` class to fetch complete lines of text, and the `writeLine` or `println` functions to output lines of text. These methods work reliably with any local conventions the platform may have. Depending on the problem at hand, other classes and methods in the

Java library may be able to handle text files in a portable way.

Rule 82: Restrict the use of `System.in`, `System.out` or `System.err` to programs explicitly intended for the command line.

Rationale

Not all system platforms supported by the Java programming language have native standard input, output, and error streams, and even those that do, may run programs under conditions where some or all of those streams are not accessible (i.e. server programs under UNIX). Using a GUI or, alternatively, writing to and reading from normal files may solve the problem in many cases.

Recommendation 59: Rely on the widely known POSIX conventions to define the syntax of your command line options.

Use the well known POSIX syntax (option names preceded with a dash "-" character) for options. Whenever possible, provide alternatives to the command line, like a graphical interface or configuration files.

Rationale

Command line option parsing is done completely by Java applications. Unfortunately, the usual syntax for command line options can vary widely from platform to platform.

Rule 83: When necessary, use the internationalization and localization features of the Java platform.

If your program contains user visible texts, or any other form of data display or entry that may vary depending on local conventions, use the internationalization and localization features of the Java platform to make your work easily adaptable to the local conventions of any potential users.

Rationale

The Java internationalization and localization features are robust, well documented and platform-independent.

Recommendation 60: Restrict the use of non ASCII characters in your messages to the minimum possible.

Rationale

Some platforms cannot display arbitrary *Unicode* characters. On the other hand, using non ASCII characters in internationalization resources is still appropriate, since internationalization can always be turned off. Restricting your original messages to ASCII guarantees that your program works (even if with somewhat restricted functionality) in any platform.

Rule 84: Do not hard code position and sizes of graphical elements.

Rationale

Appropriate sizes for for graphical elements can only be calculated depending on the current screen size and resolution, selected font size, and other factors that depend on the particular hardware and software platform, as well as on the user-selected options. Using a layout manager not only abstracts this details in a portable way, but often makes programing easier.

Rule 85: Do not hard code text sizes or font names.

Rationale

The availability of font styles and sizes depends on the underlying hardware and software platform, as well as on the current settings of the Java runtime environment. Use the Java library to get metrics information from any fonts you use, and rely on that information for laying out the corresponding texts. When selecting a non-standard font (one not guaranteed to be available on every platform) make sure that a reasonable standard replacement will be used if the font is not available.

Rule 86: Do not hard code colors or other GUI appearance elements.

Rationale

The number of available colors and the actual color palette available changes depending of the platform. A color selection that works adequately in a certain platform, may render text or graphical elements unreadable or unrecognizable in other platform.

Rule 87: Do not retain `Graphics` objects passed to update methods of graphical components.

Rationale

The standard AWT `Component.paint` and `Component.update` class methods (all standard *Swing* components are derived from class `java.awt.Component`) receive an object implementing the `Graphics` interface as parameter. This object is expected the be valid only for the duration of the corresponding paint or update operation. Retaining it for using in subsequent operations may work on some platforms and fail on others.

Example

The following pattern should be avoided:

```
Graphics retainedGraphics = null; // retained, do not do this!

void paint(Graphics g)
{
    if (retainedGraphics == null)
    {
        retainedGraphics = g.create();
    } // end if
```

```
        // painting code goes here
        ...
    } // end
```

Rule 88: Do not use methods marked as deprecated in the Java API.

Rationale

Methods marked as deprecated are scheduled for removal in future versions of the API.

Rule 89: Do not rely on the format of the result of the `java.net.InetAddress.getHostName` method.

Rationale

The actual format of the `getHostName` method depends on the underlying software and hardware platform. Sometimes it is just the simple host name, whereas some other times it includes a fully qualified domain name.

Rule 90: Always check for local availability of Pluggable Look and Feel (PLAF) classes, and provide a safe fall back in case they are not available.

When setting a particular PLAF class, check that it is both available and supported. If not, fall back to one of the standard PLAFs.

Rationale

Pluggable Look and Feel (PLAF) classes allow for graphical elements to take a distinctive appearance and behavior (for example those of the underlying operating system). Of course, not all PLAF classes are installed in all platforms. Even if you include PLAF classes with your program's distribution, they may not work on particular platforms.

Rule 91: Do not mix classes compiled against different versions of the Java platform.

Rationale

In some rare situations, bug fixes or subtle changes between versions of the Java platform may cause problems when classes compiled against different versions of the Java platform are used together.

Chapter 9
Real-Time Java

9.1 Introduction

As a very high-level programming language, Java offers programmer and software maintenance productivity benefits that range from two to ten-fold over uses of C and C++. By carefully applying Java technologies to embedded real-time systems, software engineers are able to deliver higher software quality, increased functionality, and greater architectural flexibility in software systems. The rules and recommendations in this chapter are oriented towards making the use of Java for real-time systems implementation as effective and reliable as possible.

9.2 A Note on Automatic Garbage Collection

One of the key reasons why Java developers are more productive than C and C++ developers is because of automatic garbage collection. According to a study performed by Xerox Palo Alto Research Center in the early 1980s, automatic garbage collection reduces programming effort associated with large, complex software systems by approximately 40%. These benefits are amplified significantly in the Java environment because automatic garbage collection is the foundation upon which millions of lines of commercial off-the-shelf software, including all of the standard Java libraries, are based. If you remove garbage collection from the Java environment, not only do you make it more difficult to develop new software, but you also preclude the use of all existing Java library code.

The power of garbage collection comes with a cost. Traditional Java implementations occasionally pause execution of Java threads to scan all of memory in search of objects that are no longer being used. These pauses can last tens of seconds with large memory heaps. Memory heaps ranging from 100 Mbytes to a full Gigabyte are being used in certain mission-critical systems. The 30-second garbage collection pause times experienced with traditional Java virtual machines are incompatible with the real-time execution requirements of most mission-critical systems.

Special real-time virtual machines have been implemented to support preemptible and incremental operation of the garbage collector. With these virtual machines, the interference by garbage collection on application code can be statistically bounded, making this approach suitable for soft real-time systems with timing constraints measured in the hundreds of microseconds.

One of the costs of automatic garbage collection is the overhead of implementing sharing protocols between application threads. Application threads are continually modifying the way objects relate to each other within memory, while garbage collection threads are continually trying to identify objects that are no longer reached from any threads in the system. This coordination overhead is one of the main reasons that compiled Java programs run at one third to one half of the speed of optimized C code.

The complexity of the garbage collection process and of any software that depends on garbage collection for reliable execution is beyond the reach of cost-effective static analysis to guarantee compliance with all hard real-time constraints. Thus, the use of automatic garbage collection for software that has hard real-time constraints is not recommended.

9.3 Soft Real-Time Development Guidelines

The following guidelines apply to soft real-time software development.

Rule 92: Use the Java 2 Standard Edition (J2SE) platform.

Rationale

The benefits that Java brings to soft real-time mission-critical systems are most relevant to large, complex, dynamic applications. Since the J2ME platform represents an incompatible subset of full J2SE, it does not provide access to J2SE-standard COTS library components. If applications require J2EE capabilities, obtain the specific J2EE libraries that are required and run them on a soft real-time J2SE platform. Alternatively, run the required J2EE functionality on traditional (non real-time) JVM platforms which communicate with the soft real-time JVM machines using RMI or other networking protocols.

Rule 93: Baseline a particular version of the J2SE libraries.

Rationale

For any given development project, it is necessary to standardize on a particular version of the J2SE libraries (1.2, 1.3, 1.4, 5.0?). Document this decision to all developers and managers.

Recommendation 61: Consider using JFace and SWT for Graphical User Interfaces.

Rationale

Most mission-critical software does not require graphical user interfaces. If soft real-time systems do require graphical user interfaces, consider using the open-source SWT and Jface [SWT] libraries instead of the proprietary AWT and *Swing* components. SWT and JFace may run in less memory and faster than AWT and *Swing*.

Rule 94: Use cooperating hard real-time components to interface with native code.

Rationale

The JNI protocol introduces significant data marshalling overhead when objects are shared between the Java and native environments. Furthermore, the sharing protocols may expose Java objects and “private” virtual machine data structures to undisciplined C components, introducing the risk that misbehaving C code will compromise the integrity of the virtual machine environment. Experience of existing customers in several real projects involving hundreds of man years of development document that these risks are real, having cost development teams significant effort and

calendar time to correct errors introduced into the Java environment by C developers writing JNI components.

Better performance and stronger separation of concerns is realized by implementing all interfaces to native code as cooperating hard real-time components.

Rule 95: Use cooperating hard real-time components to implement performance-critical code.

If the throughput of certain soft real-time components is not sufficient to meet performance requirements, implement the required functionality as cooperating hard real-time components.

Rationale

Because the code generation model for hard real-time components does not need to coordinate with garbage collection, these components generally run two to three times faster than soft real-time Java components.

Rule 96: Use cooperating hard real-time components to interact directly with hardware devices.

If the soft real-time component needs to communicate directly with hardware devices which are not represented by operating system device drivers, implement the device driver as a cooperating hard real-time component. If the operating system provides a device driver that represents this device as a file, use the standard `java.io` or `java.nio` libraries to access the device. If the operating system provides a device driver with a different API than the file system, use a cooperating hard real-time component to implement the interface to the device driver.

Recommendation 62: Restrict the use of advanced *libraries.*

Certain standard Java libraries are not available in certain embedded environments because the underlying operating system or hardware is missing desired capabilities. Among the libraries that may not be available on all platforms, listed in decreasing order of portability concern, are:

- JFace and SWT libraries: These graphical libraries are only available on systems that have graphical hardware and the `SWT` integration software required to drive the graphical hardware.
- `java.nio` libraries: Many embedded operating systems do not support asynchronous I/O.
- `java.io` libraries: Some embedded targets have no notion of `stdin`, `stdout`, or `stderr`. Some embedded targets have no notion of non-volatile file storage.
- `java.net` libraries: Some embedded targets have no network connectivity.

Recognize that the use of these libraries may limit the portability of code and may contribute to the future maintenance burden.

Rule 97: Isolate JVM dependencies.

Existing soft real-time virtual machines differ in how they support certain important mission-critical capabilities. Wrap all JVM dependencies in special classes that

can be given extra attention if the code must be ported to a different JVM. Specific services that require this handling include:

- High-precision timing services: obtaining real-time with greater precision than 1 ms; drift-free `sleep()`, `wait()`, and `join()` services.
- CPU-time accounting: How much CPU time consumed by each thread? How much CPU time consumed at each priority level?
- Garbage collection pacing: How to monitor the memory allocation behavior of the application software and the effectiveness of GC? How to schedule GC to maintain pace with allocation rates?
- Scheduling: If a virtual machine offers high-level scheduling support, such as earliest-deadline first or maximum accrued utility scheduling, the scheduling and synchronization services should be isolated within a centralized API.

Recommendation 63: Carefully select an appropriate soft real-time virtual machine.

One of the most important decisions in determining the success of a soft real-time Java development effort is the selection of a suitable JVM. Each development project has unique requirements and constraints, so it may be necessary to independently evaluate the relevance of various available virtual machine products for each development effort. In selecting a virtual machine, consider at minimum each of the following issues:

- Real-Time garbage collection should have a maximum preemption latency and should be incremental so that when the garbage collector is preempted by higher priority application threads, it can resume with the next increment of work when the application thread relinquishes the CPU. The garbage collector should defragment the heap in order to assure reliable long-running operation. And it must accurately reclaim all dead memory rather than reclaiming only a conservative approximation of the dead memory. Finally, it must be paced to assure that memory is reclaimed at rates consistent with the application's steady-state demand for new memory allocation.

  Please note that virtual machines exist that do not need to be paced.

  The realtime garbage collector guarantees for such virtual machines that

  enough memory will be reclaimed by tying GC to allocation.

  If code is compiled when it is loaded, this is called a

  load time compiler. Load time compilation is an older technique used in

  list implementations.

- All synchronization locks must implement priority inheritance. All wait queues must be ordered according to thread priorities.
- The virtual machine needs to provide monitoring facilities to allow supervisory threads to observe and measure the real-time resource requirements of individual components. Among required capabilities are the ability to determine how much CPU time is consumed by particular threads, how much CPU time is consumed by the garbage collection thread(s), the rates at which particular threads are allocating memory, and the total amount of memory being retained as live.

- Determine which release level of the J2SE libraries are required for a particular project (1.2, 1.3, 1.4, 5.0?) and assure that the vendor is able to support the desired library version throughout the duration of your development project.
- Assure that the virtual machine provides libraries for high-precision time measurements, and for drift-free `wait()`, `join()`, and `sleep()` services.
- If the system is statically compiled and loaded, assure that the virtual machine is supported by appropriate Ahead-of-Time compilation and linking tools.
- If the system must dynamically load components, assure that the dynamic class loader can be configured to run at lower priority than the ongoing real-time application workload. If the dynamic class loader must perform JIT compilation, assure that the JIT compiler can be configured to support eager linking and translation, meaning that all components are fully resolved and translated when the first of the interdependent modules is loaded, rather than deferring JIT translation until the moment each code module is first executed. Some systems need to dynamically load components which have themselves been ahead-of-time compiled. Verify this capability is supported if relevant to your project requirements.
- Assure that the virtual machine includes necessary development tools, including symbolic debugging of both interpreted and compiled code and run-time performance and memory usage profiling.
- If the planned development project may require integration with cooperating hard real-time components, assure that the virtual machine includes support for cooperating hard real-time Java components.

9.4 Hard Real-Time Development Guidelines

The following guidelines apply to soft real-time software development. The recommendations of this section are based on standards for safety-critical and mission-critical Java which are being developed within the Open Group.

Rule 98: Use a hard real-time subset of the standard Java libraries.

Rationale

There is no automatic garbage collection in the hard real-time domain so many of the standard Java libraries will not function reliably. Other motivations to restrict usage of the standard libraries are (1) to reduce the standard memory footprint and (2) to reduce the amount of code that must be certified in case safety certification requirements must be satisfied.

Rule 99: Use a hard real-time subset of the real-time specification for Java.

Rationale

The full RTSJ includes many capabilities that are not portable between different compliant implementations. Furthermore, supporting the full generality of the RTSJ imposes certain performance-limiting restrictions on the implementation.

Rule 100: Use enhanced replacements for certain RTSJ libraries.

Certain RTSJ libraries lack the features desired for hard real-time and safety-critical development. Analogous replacement libraries are available in the `javax.realtime.util.sc` package. Use these replacement libraries instead of the traditional RTSJ libraries. The specific replacement libraries, which differ only slightly from their RTSJ counterparts, are listed below:

- `AbsoluteTime`
- `AperiodicParameters`
- `AsyncEvent`
- `BoundAsyncEventHandler`
- `Clock`
- `HighResolutionTime`
- `NoHeapRealtimeThread`
- `OneShotTimer`
- `PeriodicParameters`
- `PeriodicTimer`
- `RelativeTime`
- `ReleaseParameters`
- `SizeEstimator`
- `SporadicParameters`
- `Timer`

Rule 101: Assure availability of supplemental libraries.

If particular applications require additional libraries beyond this minimal set, assure that the libraries are available for all intended target platforms.

Rule 102: Use an intelligent linker and annotations to guide initialization of static variables.

In traditional Java, class variables are to be initialized “immediately before first use”. This requires run-time checks, introduces non-determinism into the worst-case execution-time analysis, and hinders efficient translation of programs for native execution. Further, it introduces certain race conditions in which the initial values of particular class variables (even the values of certain `final` variables) depend on the sequence in which classes are accessed (and initialized). Use an intelligent static linker guided by `@StaticDependency` and `@InitializeAtStartup` annotations to perform initialization of all static variables.

Note: Still an open issue for safety-critical applications in the Open Group RT Java Forum process.

Rule 103: Use only 128 priority levels for `NoHeapRealtimeThread`.

The official RTSJ specification states that a compliant implementation must provide at least 28 priorities, but may support many more. For hard real-time mission-critical development, application software should limit its use of priorities to the range from 1 though 128. Vendors can readily support this priority range as a standard hard real-time mission-critical platform.

Rule 104: Do not instantiate `java.lang.Thread` or `javax.realtime.RealtimeThread`.

The only threads allowed to run in a hard real-time program are instances of `NoHeapRealtimeThread`.

Rule 105: Preallocate `Throwable` instances.

Rationale

The traditional Java convention of allocating a new `Throwable` each time an exceptional condition is encountered is not compatible with limited-memory hard real-time development practices. Preallocate necessary `Throwable` objects in scopes that are sufficiently visible that they can be seen by the intended `catch` statement. Throw the preallocated `ImmortalMemory Throwable` instances available in `javax.realtime.util.sc.PreallocatedExceptions` when appropriate.

Rule 106: Restrict access to `Throwable` attributes.

Rationale

In a traditional Java environment, memory is allocated to represent private information associated with each thrown `Throwable`. Because a hard real-time environment is assumed to have limited memory resources and no automatic garbage collection, the typical hard real-time programming style avoids allocation of memory for each thrown exception. One fixed-size buffer holding up to 20 `StackTraceElement` objects is maintained for each thread. Each time an exception is thrown, this buffer is overwritten with no more than 20 of the inner-most nested method activation frames. The buffer's contents can be copied by invoking `Throwable.getStackTrace()` before any other `throw` statements are executed by the thread. Any code that attempts to access more than 20 stack frames, or delays invocation of `Throwable.getStackTrace()` until after a second `Throwable` has been thrown, will not run reliably in the hard real-time environment.

Rule 107: Annotate all program components to Indicate scoped memory behaviors.

In order to enable static analysis to prove referential integrity without the need for run-time fetch and store checks, programmers must annotate their software to identify variables that might hold references to objects allocated in temporary memory scopes.

Rule 108: Carefully restrict use of methods declared with `@AllowCheckedScopedLinks` annotation.

Rationale

Methods with this annotation may terminate with a run-time exception resulting from inappropriate assignment operations. Automated static analysis tools are not able to guarantee the absence of these run-time exceptions, and reference assignment operations contained within these methods will run slower than other code because each assignment must be accompanied by a run-time check. For each method that is declared with the `@AllowCheckedScopedLinks` annotation, programmers should provide commentary explaining why they believe the code will not violate scoped-memory referential integrity rules.

Note: Still an open issue for safety-critical applications in the Open Group RT Java Forum process.

Rule 109: Carefully restrict use of methods declared with `@ImmortalAllocation` annotation.

As a rule of thumb, `ImmortalMemory` should only be allocated during application startup. Any other allocation of `ImmortalMemory` introduces the risk that the supply of `ImmortalMemory` will become exhausted in a long-running application.

Rule 110: Use `@StaticAnalyzable` annotation to identify methods with bounded resource needs.

The `@StaticAnalyzable` annotation identifies methods that have bounded CPU time and memory needs. For any program component declared with the `@StaticAnalyzable` annotation, programmers should provide `StaticLimit` assertions to identify iteration limits on loops and other resource constraints.

Note: The definition of @StaticAnalyzable is still insufficiently defined. What is statically analyzable is highly dependent on the tools available. The use of libraries dictates a notation about the order of execution of a method and the corresponding dependencies so that the actual execution time can be determined. For example, there may be some structure with n elements, and we know the elements can be examined. That alone is not sufficient to give a worst case execution time, but if n is know, then a maximum time can be determined. A combination of functional verification, data flow analysis, and worst case execution analysis can provide such an analysis. The important point is that the computational effort and dependencies are know. One can say that all methods should have well defined resource usage bounds.

Rule 111: Use hierarchical organization of memory to support software modules.

Organize software modules to support modular composition of components so that all memory allocation for individual components, including the memory for all of the threads that comprise the software module, is hierarchically organized. The memory for the complete module is incrementally divided into memory for sub-modules. Each sub-module may further divide its memory for smaller sub-modules. All memory reclamation is handled in last-in-first-out order with respect to allocation sequence.

Rule 112: Use the `@TraditionalJavaShared` conventions to share objects with traditional Java.

When it is necessary or desirable to share hard real-time data and/or control abstractions with the traditional Java domain, use the `@TraditionalJavaShared` and `@TraditionalJavaMethod` annotations to arrange the sharing of selected objects.

Note: Still an open issue for safety-critical applications in the Open Group RT Java Forum process.

Rule 113: Avoid `synchronized` statements.

When synchronization is required, use `synchronized` methods instead of individual statements.

Rule 114: Inherit from `PCP` in any class that uses `PriorityCeilingEmulation MonitorControl` policy.

Developers should decide when the synchronization code is written whether it will use `PriorityCeilingEmulation` or `PriorityInheritance MonitorControl` policy. Code that intends to use `PriorityCeilingEmulation` should inherit from the `PCP` interface.

Rule 115: Inherit from `Atomic` in any class that synchronizes with interrupt handlers.

The `Atomic` interface extends the `PCP` interface. The byte-code verifier enforces that all synchronized methods belonging to classes that implement Atomic are `@StaticAnalyzable` in all execution modes.

Note: Still an open issue in the Open Group RT Java Forum process.

Rule 116: Annotate the `ceilingPriority()` method of `Atomic` and `PCP` classes with `@Ceiling`.

The `@Ceiling` annotation expects its `value` attribute to be set to the ceiling priority at which this object expects to synchronize. Availability of an object's intended ceiling priority as a source code attribute makes it possible to prove compatibility between components using static analysis tools. In particular, the static analyzer can demonstrate that nested locks have strictly increasing ceiling priority values.

Rule 117: Do not override `Object.finalize()`.

In traditional Java code, an object's `finalize()` method is invoked by the garbage collector before the object's memory is reclaimed. In the hard real-time domain, we do not have a garbage collector. Memory is reclaimed as particular control contexts are left. If finalization code is required, place it in in the `finally` clause of a `try-finally` statement.

Recommendation 64: Use development tools to enforce consistency with hard real-time guidelines.

To enforce that programmers make proper use of the hard real-time API subsets and that all code is consistent with the intent of the hard real-time programming annotations described in this section, use special byte-code verification tools that help assure reliable and efficient implementation of programmer intent.

9.5 Safety-Critical Development Guidelines

Safety-critical developers use a subset of the full hard real-time mission-critical capabilities.

Rule 118: Except where indicated to the contrary, use hard real-time programming guidelines.

In general, all of the hard real-time guidelines are appropriate for safety-critical development, except that certain practices acceptable for hard real-time mission-critical development should be avoided with safety-critical software.

Rule 119: Use only 28 priority levels for `NoHeapRealtimeThread`.

The official RTSJ specification states that a compliant implementation must provide at least 28 priorities, but may support many more. For safety-critical development, application software should limit its use of priorities to the range from 1 though 28. Vendors can readily support this priority range as a standard safety-critical platform.

Rule 120: Prohibit use of `@OmitSubscriptChecking` annotation.

In safety-critical code, turning off subscript checking is strongly discouraged, even though static analysis of the program presumably has proven that the program will not attempt to access invalid array elements. In safety-critical systems, the key benefit of subscript checking is to prevent an error in one component from propagating to other components.

Note: Checking should only be turned of automatically by a compiler that can prove a given check is unnecessary!

Rule 121: Prohibit invocation of methods declared with `@AllowCheckedScopedLinks` annotation.

This annotation is designed to allow programmers to use practices that cannot be certified safe by automatic static theorem provers. Thus, there is a risk that any

software making use of this annotation will abort with a run-time exception. Allow this practice only in safety-critical systems for which developers are able to provide absolute proof that run-time exceptions will not be thrown.

Rule 122: Require all code to be `@StaticAnalyzable`.

In hard real-time mission-critical code, the use of the `@StaticAnalyzable` annotation is entirely optional. In safety-critical code, we require all components to have this annotation, and for all relevant modes of analysis to have a `true` value for the `enforce_analysis` attribute.

Rule 123: Require all classes with `Synchronized` methods to inherit `PCP` or `Atomic`.

The safety-critical profile does not allow the use of priority inheritance locking.

Note: Still an open issue for safety-critical applications in the Open Group RT Java Forum process.

Rule 124: Prohibit dynamic class loading.

While dynamic class loading may be supported in the hard real-time mission-critical domain, it should be strictly avoided in safety-critical software.

Rule 125: Prohibit use of blocking libraries.

Because of difficulties analyzing blocking interaction times when software components contend for shared resources, all services that might block are forbidden in safety-critical code. Specifically, the following APIs should *not* be invoked from safety-critical application software:

- `java.lang.Object.wait()`
- `java.lang.Object.wait(long)`
- `java.lang.Object.wait(long, int)`
- `java.lang.Thread.join()`
- `java.lang.Thread.join(long)`
- `java.lang.Thread.join(long, int)`
- `java.lang.Thread.sleep(long)`
- `java.lang.Thread.sleep(long, int)`
- `javax.realtime.util.sc.ThreadStack.join();`

Rule 126: Prohibit use of `PriorityInheritance MonitorControl` policy.

Priority inheritance is more difficult to certify, more complicated to implement, and less efficient than priority ceiling emulation. Thus, we prohibit its use in safety-critical software systems.

Rule 127: Do not share safety-critical objects with a traditional Java virtual machine.

Combining safety-critical code with traditional Java code using the `@TraditionalJavaMethod` and `@TraditionalJavaShared` conventions compromises the integrity of the safety-certification artifacts. This practice is therefore strictly forbidden.

Recommendation 65: Use development tools to enforce consistency with safety-critical guidelines.

To enforce that programmers make proper use of the safety-critical subset and that all code is consistent with the intent of the hard real-time programming annotations described in this section, use special byte-code verification tools that help assure reliable and efficient implementation of programmer intent.

Chapter 10
Embedding C++ or C in Java

10.1 Introduction

There are many legacy systems written in C and C++. Sometimes the use of these languages is preferred for numerical calculations (many libraries exist) or for easy access to hardware. Sometimes it is also company or project policy to write code in these languages.

It can be a solution to embed such C/C++ code in Java using the JNI API, in order to make such applications interoperate with Java applications. Other solutions exist however and are normally preferable to embedding C/C++ code in Java, as such embedding is often complex and error-prone.

However, in some cases, the direct embedding of native code becomes necessary for performance reasons or to overcome the strict access restrictions that the Java environment sometimes imposes. If embedding of C++ or C code is required, these guidelines should help to increase portability, safety and performance on different Java platforms.

10.2 Alternatives to JNI

Recommendation 66: Avoid embedding C++ or C code in Java as much as possible. Use other coupling solutions instead if C++ or C code needs to be integrated to the software product.

Rationale

Embedding C++ and C code in Java can be done with the Java Native Interface (JNI)-API. However this API is very complex and error-prone. It is better to interface with C/C++ via CORBA or via XML-files. This keeps the C/C++ environment sufficiently separated from the Java-platform, and has the following advantages:

- Better interoperability
- Less errors
- Java-component retains all the power of its object-oriented features
- Less complex than when using JNI
- Well-defined interfaces
- Interoperability, security and communication services when CORBA is used.

10.3 Safety

C++ or C code is not covered by the advanced safety features (access control, index, type and null pointer checks, exception handling, etc.) Java developers are used to. For code safety, these checks need to be performed explicitly by the developer.

Rule 128: Use the established coding standards for C++ or C for the development of C++ or C code that is embedded into the Java code.

Rationale

Respecting the established conventions helps to avoid errors, eases understanding of the code by developers with a C++ or C background in the area, and enables interoperability with existing C++ or C code.

Rule 129: Check for `ExceptionOccurred()` after each call of a function in the JNI interface if that may cause an exception.

Rationale

Functions available in the JNI interface (as defined in C/C++ include file "`jni.h`") often signal an error condition to the caller by storing an exception in the current JNI environment. Unlike in Java code, this exception is not thrown and propagated automatically to the next surrounding exception handler that accepts an exception of this kind. Instead, execution of C++ or C code continues normally unless there is an explicit check that no exception occurred.

Checking for exceptions is therefore required after each call that may cause an exception to indicate an error situation. Checking the return value of such a call is generally not sufficient.

Rule 130: Mark native methods as `private`.

Rationale

Native functions may perform operations that are not safe with respect to the checks performed by Java code. If public access to a native function is required, a wrapper function written in Java can ensure that the call is performed from an environment that guarantees that the call is safe..

Example

```
private static native int nativeReadRegister(int register);

public static int readRegister(int register)
{
    int result;

    if (isLegalRegister(register) && isReady(register))
    {
        result = nativeReadRegister(register);
    } // end if
    else
```

```
        {
            throw new IllegalArgumentException(
                    "register: "+register);
        } // end else
        return result;
    } // end method
```

Rule 131: Select method names for C++ or C methods that state clearly that such a method is a native method.

Common means to mark a native method are a method name that ends with the character '0' or that starts with the string "native", e.g., `nativeReadRegister()` or `readRegister0()`.

Rationale

Marking native methods avoids confusion.

Rule 132: Avoid name overloading for native methods.

Rationale

The JNI name mangling rules for overloaded methods (methods that have equal name but different argument lists) results in very cumbersome names. Adding an overloaded native method to an existing class causes renaming the C++ or C function of existing native methods that were not overloaded before, causing difficult linking problems of the application.

Rule 133: Do not use weak global references.

Rationale

A weak global references obtain by `NewWeakGlobalRef()` may change its value to `null` at any point in time. This makes error detection and handling extremely difficult.

Note: In general, this is a good rule; however, there may be cases where weak references are necessary to insure that back pointers from native code do not prevent garbage collection. In that case, the code must insure that whenever the native object is available, the back pointer (weak pointer) is valid. Still explicit deallocation of native objects is preferred.

10.4 Performance

Performance gains can be one important reason to embed C++ or C code into Java code. However, there is additional overhead involved in the use of JNI that may work against the performance gained through the use of C++ or C code.

The additional overhead from JNI can be explained in many different ways::

- References passed to C++ or C code need to be protected from the garbage collector activity. Particularly, the garbage collector must not reclaim memory of an object that is referenced from JNI code and it cannot update a reference value so it may not move objects in order to defragment memory.
- Since JNI code may run for an arbitrary amount of time that is not known the to VM, detaching the thread that performs the JNI call from the rest of the VM may be necessary in order to avoid blocking other Java threads.
- The calling conventions of JNI are different from the calling conventions that are used internally by compiled or interpreted code running within the VM. Special wrapper functions are required to interface between the VM and native methods causing a higher call overhead compared to normal Java method calls.

Recommendation 67: Avoid the use of C++ or C code embedded using the JNI to increase performance.

Rationale

The additional overhead of JNI and the loss in safety and portability are likely to counter the performance gain.

Recommendation 68: Avoid passing reference values to native code.

Rationale

Passing reference values to native code causes extra overhead to protect these references from garbage collection. Furthermore, referenced values cannot be used directly within JNI code, they can only be accessed through calls to functions in the JNI (as defined in "`jni.h`") causing additional overhead.

Example

Do not pass a reference to an instance of class `Point`, but pass two integers for the X any Y coordinate values instead. These values can then be used directly in the native code.

Rule 134: Use `DeleteLocalRef()` to free references in native code that were obtained in a loop.

Rationale

The protection of reference values passed to native code requires memory that needs to be allocated by the VM. If references are obtained in a loop, even if these references are all equal as in repeated calls to `GetFieldId()` for the same field, the memory that is required to protect these references will grow unless the references are released by calls to `DeleteLocalRef()`.

Rule 135: Use `NewGlobalRef()`/`DeleteGlobalRef()` only for references that are stored outside of reachable memory that survives from one JNI call to the next.

Rationale

The overhead for creation and deletion of a global reference is typically signifi-

cantly higher than that for local references. Furthermore, a forgotten `DeleteGlobalRef()` will make it impossible for the VM to ever release the memory that was protected globally, causing a memory leak.

Recommendation 69: Avoid calling back into Java code from C/C++ code.

Rationale

The JNI to Java calling conventions are different to the internal calling conventions used by the VM and require special treatment. A call of a Java method from within JNI code is typically much more expensive than the corresponding call from within Java code.

Recommendation 70: Put as much functionality as possible into the Java code and as little as possible in the JNI code.

Rationale

Larger C++/C functionality usually implies complexer interactions between C/C++ and Java code, which in turn may lead to more errors, and reduced reliability and efficiency.

Example

Throwing an exception from within JNI code is a complex procedure, it is better defer this activity to Java code that is simpler.

Recommendation 71: Avoid `Get*ArrayElements()` and `Get*ArrayElementsCritical()` functions.

Rationale

These functions may require allocation of temporary arrays and copying of the array contents into a temporary array. The reasons for the need to copy the data are that the internal representation of arrays that is used by the virtual machine may be different than a simple C++ or C array. Furthermore, the garbage collector may move arrays when it is defragmenting the heap, which would be impossible if C++ or C code holds a direct reference to the array data.

Apart from the allocation and copying overhead, these functions may fail due to memory fragmentation or low memory. They are unsafe for long running applications.

Recommendation 72: Avoid frequent calls to the reflective functions `FindClass()`, `GetMethodID()`, `GetFieldID()`, and `GetStaticFieldID()`.

Rationale

These functions perform an expensive string search. If frequent accesses to a class, method or field are required, obtain these values by a native initialization function, protect them via `NewGlobalRef` and store the results in a data structure accessible from the code that needs to access these values.

10.5 Low Level Hardware Access

Rule 136: Avoid using JNI for native HW access if alternative means are available.

Rationale

The `RawMemoryAccess` classes from *The Real Time Specification for Java* [RTSJ] provide safe means for direct hardware access that avoids the danger and overhead involved with JNI code.

10.6 Non-Standard Native Interfaces

Some Java implementations provide proprietary native interfaces in addition to the standard JNI. These interfaces may provide higher performance since they may use the same interface that is used by compiled Java code, they can avoid the detaching and attaching overhead from the VM and they can avoid the pointer registering and unregistering overhead.

However, these interfaces are proprietary, i.e., they are not portable between different VMs. Also, code written for them is typically more complex since many aspects of the internals of the underlying VM and garbage collector may be exposed.

Rule 137: Do not use non-standard native interfaces unless there are very good reasons to do so.

Rationale

Giving up on the portability and compatibility with other Java environments is a major disadvantage for all future reuse of the code. Only if the achievable performance using pure Java or JNI is absolutely insufficient to solve the problem at hand, the use of proprietary interfaces may be justified.

Rule 138: Restrict the use of non-standard native interface uses to as few functions as possible.

Rationale

Even if the use of a non-standard native interface is required at some point, this point should clearly be isolated from the rest of the application. Such methods should be defined in a separate class that is clearly documented to be dependent on the corresponding virtual machine.

All less critical native code should use the standard JNI to interact with the Java code in a portable way.

Chapter 11
Security

11.1 Introduction

Security is a complex area, which nowadays represents a concern for organizations of all sizes and in all fields of endeavor. ESA is no exception. On the one hand, its international nature poses security challenges, and, on the other hand, security plays an important role in ESA's Galileo and GMES projects.)

Although making a software system secure goes way beyond coding, adequate coding is still fundamental to achieve proper security. The following rules and recommendations, are intended to facilitate producing more secure and reliable Java code.

11.2 The Java Security Framework

The standard Java platform (compiler, bytecode verifier, runtime system) is designed to enforce the following rules:

- Class member access (as defined by the `private`, `protected`, and `public` keywords) is strictly adhered to.
- Programs cannot access arbitrary memory locations (pointers do not exist in standard Java).
- Entities that are declared as `final` cannot be changed.
- Variables cannot be used before they are initialized.
- Array bounds are checked during all array accesses.
- Objects of one type cannot be arbitrarily cast into objects of other types.

Additionally, the platform implements an elaborated access control mechanism, making it possible to restrict access to system resources in a fine-grained way.

Real-time Java implementations may not enforce some or all of these restrictions. Particularly, it is typical for real-time Java applications to directly access memory locations in order to control hardware devices or read information from them.

11.3 Privileged Code

Java's access control system is responsible for protecting system resources from unauthorized access. The basic mechanism used for this purpose is to guarantee that *all* code traversed by a thread, up to the current execution point, has appropriate permissions. It will often be the case, though, that a program needs to access resources that it normally could not use, given its permissions. The privileged blocks API [SUNPriv] was designed to handle such situations in a secure, controlled way.

Code in a privileged block runs with the permissions granted to the block, without taking into account the security restrictions of any caller code. Privileged blocks make the Java security model a lot more flexible, by making it possible to wrap potentially insecure operations into code that, in some way or another, restricts or controls access to them. This additional flexibility comes, however, at some cost, because every privileged block poses a certain security risk that must be properly assessed and managed.

Recommendation 73: Keep privileged code as short as possible.

Rationale

Errors in privileged code can potentially lead to security exploits. For this reason, privileged code should always be carefully audited (as opposite to just tested) before deploying it. Making privileged code as short as possible, not only simplifies the process of auditing it, but reduces the risk of dangerous errors being overlooked.

Recommendation 74: Check all uses of tainted variables in privileged code.

A variable used in a privileged block is considered tainted, if it contains a value that was directly passed as parameter by the method caller.

Rationale

All uses of tainted variables should be carefully checked to make sure they cannot lead to inappropriate privilege escalation.

Example

Consider the following method:

```
public static String getProp(final String name)
{
    return (String) AccessController.doPrivileged(new
            PrivilegedAction()
    {
        public Object run()
        {
            // 'name' is tainted, beware!
            return System.getProperty(name);
        } // end method
    } // end constructor
} // end method
```

The value of parameter `name` will be used directly to retrieve a property, without imposing any restrictions or making any checks on it. This means that any class being able to call this `public` method will be able to retrieve arbitrary properties.

Uses of tainted variables in privileged code must always be carefully checked. Helper methods allowing unrestricted access to a resource should always be declared `private`.

11.4 Secure Coding

Rule 139: Refrain from using non-`final` `public` `static` variables.

Rationale

It is impossible to check that code changing such variables has adequate permissions.

Recommendation 75: Reduce the scope of methods as much as possible.

Make as few methods `public` as strictly necessary. A proper design of class interfaces done before implementing the class should help reduce the number of `public` methods to a minimum.

Rationale

Every additional public method increases the risk of unauthorized access to privileged data, and makes code auditing harder.

Rule 140: Never return references to internal mutable objects containing sensitive data.

Rationale

Internal objects are part of the state of the object containing them. Returning references to mutable internal objects makes it possible for a caller to directly alter the object's state, potentially in dangerous ways. Particularly, containers like arrays, vectors or hash tables are always mutable, even if the values stored in them are not. An attacker having access to a container of immutable values cannot change the values directly, but can alter the set of values stored in the container.

Rule 141: Never store user provided mutable objects directly.

Rationale

User provided mutable objects could be intentionally or unintentionally modified after being stored, thus indirectly and unexpectedly affecting the internal state of an object. Instead of storing a reference to the user provided object, a copy (clone) should be made an stored in its place.

11.5 Serialization

Serialized objects stored in a regular file or traveling along a network connexion are outside of the control of the Java runtime environment, and therefore not subject to any of the security measures provided by the Java platform. By altering serialized object data, an attacker could possibly defeat security measures that would otherwise be adequate.

Rule 142: Use the `transient` keyword for fields that contain direct handles to system resources, or that contain information relative to an address space.

Rationale

A serialized handle to a file or other system resource, could be altered in order to gain unauthorized access to system resources once the object is deserialized.

Rule 143: Define class specific serializing/deserializing methods.

Rationale

The only way to guarantee that internal class invariants are still valid after deserializing and object, is to write a custom deserializing method that uses the `ObjectInputValidation` interface to check invariants.

Recommendation 76: Consider encrypting serialized byte streams.

Rationale

Encrypting a byte stream is an effective way to protect it from accidental or malicious alteration happening during the time it is outside the Java runtime environment. Unfortunately, encryption also requires the application to take care of encryption key handling, including generating keys, storing them, and passing them to any applications needing to read the data.

Rule 144: While deserializing an object of a particular class, use the same set of restrictions used while creating objects of the class.

If you impose restrictions on untrusted code to create objects of a certain class, enforce the same restrictions while deserializing such objects.

Rationale

Deserializing is just another form of object creation. Having laxer restrictions while deserializing makes any additional restrictions imposed during object creation superfluous.

Example

If an applet creates a frame, it will always have a warning label. If such a frame is serialized, the application should make sure that it still has the label after being deserialized.

11.6 Native Methods and Security

Recommendation 77: Check native methods before relaying on them for privileged code.

Native methods can break security in a variety of ways. They should be checked for:

- Their return value.

- Their parameters.
- Whether they bypass security checks.
- Whether they are declared `public`, `protected`, `private`, ...
- Whether they contain method calls which bypass package-boundaries, thus bypassing package protection

11.7 Handling Sensitive Information

Sensitive information like user passwords or private encryption keys must be handled with particular care.

Rule 145: Explicitly clear sensitive information from main memory.

Rationale

Information stored in main memory could possibly be accessed by attackers gaining access to memory pages after they were used by the application. For this reason it is always indicated to explicitly overwrite sensitive information as soon as it is not necessary anymore.

Rule 146: Always store sensitive information in mutable data structures.

Rationale

Immutable data structures cannot be overwritten. Programs can only delete all references to them, and wait for the garbage collector to release (but not overwrite) their memory at some unspecified time afterwards. The length of this garbage collection cycle may open a window of opportunity for attackers to get hold of the sensitive data.

Example

Use `StringBuffer` objects instead of `String` objects to store sensitive passwords.

Bibliography

AMB Ambler, S. *Writing Robust Java Code (v17.01d).* AmbySoft Inc. 2000. `http://www.ambysoft.com/JavaCodingStandards.pdf`

ANT The Apache Ant Project. *Home Page.* `http://ant.apache.org/`

ARN Arnold K. et al. *The Java™ Programming Language (3rd Edition).* Addison-Wesley, 2000.

AVSI Aerospace Vehicle Systems Institute (AVSI). *Guide to the Certification of Systems with Embedded Object-Oriented Software (Version 1.5).*

BLO Bloch, J. *Effective Java Programming Language Guide*. Addison-Wesley, 2001.

BOL Bollella G. et al. *The Real-Time Specification for Java.* Addison-Wesley, 2000.

BOO Booch, G. *Object-Oriented Analysis and Design with Applications (2nd Edition).* Addison-Wesley, 1993.

BUR Burke E. et al. *Java Extreme Programming*, O'Reilly & Associates, 2003.

COO Cooper, J. W. *Java Design Patterns*. Addison-Wesley, 2000.

ECSS-E40-1B
European Space Agency (ESA). *Space engineering - Software - Part 1: Principles and requirements (ECSS-E-40 Part 1B).* November 2003.

ECSS-Q-80B
European Space Agency (ESA). *Space Product Assurance (ECSS-Q-80B)*. October 2003.

GAM Gamma, E. et al. *Design Patterns.* Addison-Wesley, 1995.

GNUJAVA
The GNU Java Programming Standard: http://www.gnu.org/software/classpath/docs/hacking.html#SEC6

GOS Gosling, J et al. *The Java Language Specification (2nd Edition).* Addison-Wesley, 2000. `http://java.sun.com/docs/books/jls/`

GRA Grand, M. *Patterns in Java - Volume 1 (2nd Edition)*. Wiley, 2002.

LAP Laplante P. A. editor. *Dictionary of Computer Science, Engineering, and Technology.* CRC Press, 2001.

MCL McLaughlin, B. *Java & XML.* O'Reilly & Associates, 2001.

MEY Meyer, B. *Object-Oriented Software Construction (2nd Edition).* Prentice-Hall, 1997.

OAK Oaks, S. *Java Security*. O'Reilly & Associates, 2001.

OOTiABook
Federal Aviation Administration (FAA). *Handbook for Object-Oriented Technology in Aviation (OOTiA).* 2004.

OOTiAPage
Federal Aviation Administration (FAA). *Object-Oriented Technology in Aviation Web Site.* `http://shemesh.larc.nasa.gov/foot/`

RTSJ Real-Time for Java™ Expert Group. *RTSJ: The Real Time Specification for Java*. `https://rtsj.dev.java.net/`

SUNDoc
Sun Microsystems. *How to Write Doc Comments for the Javadoc Tool.* `http://java.sun.com/j2se/javadoc/writingdoccomments/`

SUNCode
Sun Microsystems. *Code Conventions for the Java Programming Language*. `http://java.sun.com/docs/codeconv/`

SUNJava
Gosling, J. et al. *The Java Language Specification (2nd Edition).* Addison-Wesley, 2000. `http://java.sun.com/docs/books/jls/`

SUNLook
Sun Microsystems. *Java Look and Feel Design Guidelines.* `http://java.sun.com/products/jlf/`

SUNPref
Sun Microsystems. *Java Preferences API.* `http://java.sun.com/j2se/1.4.2/docs/guide/lang/preferences.html`

SUNPriv
Sun Microsystems. *Java API for Privileged Blocks.* `http://java.sun.com/j2se/1.4.2/docs/guide/security/doprivileged.html`

SUNTech
Sun Microsystems. *Java Technology Page*. `http://java.sun.com/`

SWT Standard Widget Toolkit (SWT) Project. *Home Page.* `http://www.eclipse.org/swt/`

VER Vermeulen, A. et al. *The Elements of Java Style.* Cambridge University Press, 2000.