

**RDA JAVA LANGUAGE CODING STANDARDS (JCS)
WPI0053**

**Prepared by:
WSR-88D Radar Operations Center
1313 Halley Circle,
Norman OK 73069**

Table of Contents

1 Introduction.....	2
2 Source Files.....	3
3 Package Declaration.....	3
4 Import Declarations.....	3
5 Class/Interface Declarations	4
6 Naming Conventions	4
6.1 Package Naming.....	4
6.2 Class/Interface Naming.....	4
6.3 Field Naming.....	4
6.4 Method Naming.....	5
6.5 Local Variable Naming.....	6
7 White Space	6
8 Comments	7
8.1 Block Comments	8
8.2 Single-line Comments	8
8.3 JavaDoc Comments for Automatic Documentation.....	8
9 Classes.....	10
10 Class (static) and Instance Variable Field Declarations.....	11
11 Static_INITIALIZER.....	12
12 Static Member Inner Class Declarations.....	12
13 Static Method Declarations.....	12
14 Instance_INITIALIZER.....	12
15 Constructor Declarations.....	12
16 Instance Method Declarations.....	13
17 Method Declarations.....	13
18 Local Inner Classes	14
19 Anonymous Inner Classes.....	14
20 Anonymous Array Expressions and Array Initializers	15
21 Interfaces.....	15
22 Simple Statement Construction.....	16
23 try, catch, and finally Statements.....	16
24 synchronized Statement	16
25 Labeled Statements	17
26 Existing Code.....	17

1 Introduction

The RDA Java Coding Standards (JCS) defines a set of coding standards and common practices related to developing software in the Radar Data Acquisition (RDA) project using the Java programming language. This JCS is an adaptation of the ORDA Java Coding Standards .

This CCS is to be followed in all C code developed for the RDA project, except with special permission from the RDA programming group leader, or as described in Section 26 ‘Existing Code’.

2 Source Files

On file-based host implementations of Java, the compilation unit is a Java source file. A Java source file should contain only one public class or interface definition; although it may also contain any number of non-public support classes or interfaces.

Java source file names are in the form `ClassOrInterfaceName.java` where `ClassOrInterfaceName` is exactly the name of the public class or interface defined in the source file. The file name suffix is always `.java` except on systems that support only three-character extensions; on such systems, the suffix is `.jav`. JAR (Java Archive) file names are of the form: `ArchiveName.jar` or `ArchiveName.zip`

A Java source file should contain the following elements, in the following order:

1. Heading comments
2. package declaration
3. import declarations
4. one or more class/interface declarations

At least one blank line should separate all of these elements.

Every Java file should contain a heading section identical to the heading section for 'C' code files. It should include the filename, a short description and a longer statement of the purpose of the file and its relation to the overall program. The prologue comments should be tagged for automatic documentation generation in 'Javadoc' style. The Javadoc tags used to record this information in the heading of a Java file are identical to the tags used in the prologue of a 'C' source file.

Example:

```
/**
 * @file filename.c
 * @brief A short, one sentence description of this file.
 * @see other files related to this one. (optional)
 *
 * A longer, detailed description of the contents of the file.
 */
```

3 Package Declaration

Every source file should contain a package declaration. Omitting the package declaration causes the types to be part of an unnamed package, with implementation-defined semantics. The package statement should start in column 1, and a single space should separate the keyword `package` from the package name. Example: `package java.lang;`

4 Import Declarations

Import statements should start in column 1, and a single space should separate the keyword `import` from the type name. Import statements should be grouped together by package name. A single blank line may be used to separate groups of import statements. Within groups, import statements should be sorted lexically.

5 Class/Interface Declarations

Following the import sections are one or more class declarations and/or interface declarations, collectively referred to simply as type declarations. The number of type declarations per file should be kept small. There should be at most one public type declaration per file. The public type, if any, should be the first type declaration in the file. Every public type declaration should be immediately preceded by a documentation comment describing its function and parameters (using the @param tag). The description should be concise. Non-public type declarations should also be preceded by a comment, but it need not be a documentation comment.

6 Naming Conventions

The naming conventions specified here apply only to Java code written in the basic ASCII character set. Terms such as “upper case” are obviously meaningless for some Unicode character sets.

6.1 Package Naming

Generally, package names should use only lower case letters and digits, and no underscore. Examples:

```
java.lang  
java.awt.image  
dinosaur.theropod.velociraptor
```

6.2 Class/Interface Naming

All type names (classes and interfaces) should use the InfixCaps style as follows. Start with an upper case letter, and capitalize the first letter of any subsequent word in the name, as well as any letters that are part of an acronym. All other characters in the name are lower-case. Do not use underscores to separate words. Class names should be nouns or noun phrases. Interface names depend on the salient purpose of the interface. If the purpose is primarily to endow an object with a particular capability, then the name should be an adjective (ending in -able or -ible if possible) that describes the capability; e.g., Searchable, Sortable, NetworkAccessible. Otherwise use nouns or noun phrases.

Good type names:

```
LayoutManager, AWTException, ArrayIndexOutOfBoundsException
```

Bad type names:

```
ManageLayout /* verb phrase */  
awtException /* first letter lower-case */  
array_index_out_of_bounds_exception /* underscores */
```

6.3 Field Naming

Names of non-constant fields (reference types, or non-final primitive types) should use a camelCase

style as follows. Start with a lower case letter, and capitalize the first letter of any subsequent word in the name, as well as any letters that are part of an acronym. All other characters in the name are lower-case. Do not use underscores to separate words. The names should be nouns or noun phrases.

Examples:

```
boolean resizable;  
char recordDelimiter;
```

Names of fields being used as constants should be all upper-case, with underscores separating words.

The following are considered to be constants:

1. All static final primitive types (Remember that all interface fields are inherently static final).
2. All static final object reference types that are never followed by "." (dot).
3. All static final arrays that are never followed by "[" (dot).

Examples:

```
MIN_VALUE, MAX_BUFFER_SIZE, OPTIONS_FILE_NAME
```

One-character field names should be avoided except for temporary and looping variables. In these cases, use:

- b for a byte
- c for a char
- d for a double
- e for an Exception object
- f for a float
- g for a Graphics object
- i, j, k, m, n for integers
- p, q, r, s for String, StringBuffer, or char[] objects

An exception is where a strong convention for the one-character name exists, such as x and y for screen coordinates. Avoid variable l ("el") because it is hard to distinguish it from 1 ("one") on some printers and displays.

6.4 Method Naming

Method names should use the camelCase style as follows. Start with a lower-case letter, and capitalize the first letter of any subsequent word in the name, as well as any letters that are part of an acronym.

All other characters in the name are lower-case. Do not use underscores to separate words.

Method names should be imperative verbs or verb phrases. Examples:

Good method names:

```
showStatus(), drawCircle(), addLayoutComponent()
```

Bad method names:

```
mouseButton() /* noun phrase; doesn't describe function */  
DrawCircle() /* starts with upper-case letter */  
add_layout_component() /* underscores */
```

The function of the following method is unclear.

```
serverRunning() /* verb phrase, but not imperative */
```

Does it start the server running (better: startServer()), or test whether or not it is running (better: isServerRunning())?

A method to get or set some property of the class should be called getProperty() or setProperty() respectively, where Property is the name of the property. Examples:

```
getHeight(), setHeight()
```

A method to test some boolean property of the class should be called isProperty(), where Property is the name of the property. Examples:

```
isResizable(), isVisible()
```

6.5 Local Variable Naming

Local variables follow the same naming rules as field names.

7 White Space

Blank lines can improve readability by grouping sections of the code that are logically related. A blank line should also be used in the following places:

1. After the version block comment, package declaration, and import section.
2. Between class declarations.
3. Between method declarations.
4. Between the last field declaration and the first method declaration in a class.
5. Before a block or single-line comment, unless it is the first line in a block.

A single blank space (not tab) should be used:

1. Between a keyword and its opening parenthesis. This applies to the following keywords: catch, for, if, switch, synchronized, while. It does not apply to the keywords super and this; these should never be followed by white space.
2. After any keyword that takes an argument. Example: return true;
3. Between two adjacent keywords.
4. Between a keyword or closing parenthesis, and an opening brace “{”.
5. Before and after binary operators except .(dot). Note that instanceof is a binary operator:

```
if (obj instanceof Button) { /* RIGHT */  
if (obj instanceof(Button)) { /* WRONG */
```

6. After a comma in a list.
7. After the semi-colons in a for statement, e.g.:

```
for (expr1; expr2; expr3) {
```

Blanks should not be used:

1. Between a method name and its opening parenthesis. Some judgment is called for in the case of

complex expressions, which may be clearer if the “inner” operators are not surrounded by spaces and the “outer” ones are.

2. Before or after a `.`(dot) operator.
3. Between a unary operator and its operand.
4. Between a cast and the expression being casted.
5. After an opening parenthesis or before a closing parenthesis.
6. After an opening square bracket [or before a closing square bracket].

Examples:

```
a += c[i + j] + (int)d + foo(bar(i + j), e);
a = (a + b) / (c * d);
if (((x + y) > (z + w)) || (a != (b + 3))) {
    return foo.distance(x, y);
}
```

Do not use special characters like form-feeds or backspaces.

Line indentation is always 4 spaces, for all indentation levels.

Lines should be limited to 80 columns (but not necessarily 80 bytes, for non-ASCII encodings). Lines longer than 80 columns should be broken into one or more continuation lines, as needed. All the continuation lines should be aligned, and indented from the first line of the statement. The amount of the indentation depends on the type of statement. If the statement must be broken in the middle of a parenthesized expression, such as for compound statements, or for the parameter list in a method invocation or declaration, the next line should be aligned with the first character to the right of the first unmatched left parenthesis in the previous line. In all other cases, the continuation lines should be indented by a full standard indentation (4 spaces). If the next statement after a continuation line is indented by the same amount as the continuation line, then a single blank line should immediately follow the opening brace to avoid confusing it with the continuation line. It is acceptable to break a long line sooner than absolutely necessary, especially if it improves readability.

Examples:

```
foo(long_expression1, long_expression2, long_expression3,
    long_expression4);

foo(long_expression1,
    long_expression2,
    long_expression3,
    ong_expression4);
```

A continuation line should never start with a binary operator. Never break a line where normally no white space appears, such as between a method name and its opening parenthesis, or between an array name and its opening square bracket.

8 Comments

Java supports three kinds of comments: block, single-line and JavaDoc comments. These are described separately in the subsequent sections below. Here are some general guidelines for comment usage:

- Comments should help a reader understand the purpose of the code. They should guide the reader through the flow of the program, focusing especially on areas, which might be confusing or obscure.
- Avoid comments that are obvious from the code, as in this famously bad comment example:

```
i = i + 1; /* Add one to I */
```

- Remember that misleading comments are worse than no comments at all.
- Avoid putting any information into comments that is likely to become out-of-date.
- Temporary comments that are expected to be changed or removed later should be marked with the special tag “XXX:” so that they can easily be found afterwards. Example:

```
/* XXX: Change this to call sort() when the bugs in it are fixed */  
list->mySort();
```

8.1 Block Comments

A regular block comment is a traditional “C-style” comment. It starts with the characters `/*` and ends with the characters `*/`. It is used to “comment out” several lines of code. Since block comments do not nest, their use in other parts of the source code would make it difficult to comment out code. Hence, the use of block comments other than for commenting out code is strongly discouraged.

8.2 Single-line Comments

A single-line comment consists of the characters `//` followed by comment text. There is always a single space between the `//` and the comment text. A single line comment must be at the same indentation level as the code that follows it. More than one single-line comment can be grouped together to make a larger comment. A single-line comment or comment group should always be preceded by a blank line, unless it is the first line in a block. If the comment applies to a group of several following statements, then the comment or comment group should also be followed by a blank line. If it applies only to the next statement (which may be a compound statement), then do not follow it with a blank line.

Example:

```
// Traverse the linked list, searching for a match  
for (Node node = head; node.next != null; node = node.next)
```

Single-line comments can also be used as trailing comments. Trailing comments are similar to single-line comments except they appear on the same line as the code they describe. At least one space should separate that last non-white space character in the statement, and the trailing comment. If more than one trailing comment appears in a block of code, they should all be aligned to the same column.

Example:

```
if (!isVisible())  
    return; // nothing to do  
length++; // reserve space for null terminator
```

Avoid the assembly language style of commenting every line of executable code with a trailing comment.

8.3 JavaDoc Comments for Automatic Documentation

Java has support for special comments documenting types (classes and interfaces), fields (variables), constructors, and methods, hereafter referred to collectively as declared entities. The javaDoc program can then be used to automatically extract these comments and generate formatted HTML pages.

A documentation comment should immediately precede the declared entity, with no blank lines in between. The first line of the comment should be simply the characters `/**` with no other text on the line, and should be aligned with the following declared entity. Subsequent lines consist of an asterisk, followed by a single space, followed by comment text, and aligned with the first asterisk of the first line. The first sentence of the comment text is special, and should be a self-contained summary sentence. A sentence is defined as text up to the first period that is followed by a space, tab, or new-line. Subsequent sentences further describe the declared entity.

The comment text can include embedded HTML tags for better formatting, with the exceptions of the following tags: `<H1>`, `<H2>`, `<H3>`, `<H4>`, `<H5>`, `<H6>`, `<HR>`.

Following the comment text are the documentation tag lines. A documentation comment should include all the tags that are appropriate for the declared entity.

Class and interface comments can use the `@version`, `@author`, and `@see` tags, in that order. If there are multiple authors, use a separate `@author` tag for each one.

Required tags: none.

Constructor comments can use the `@param`, `@exception`, and `@see` tags, in that order.

Required tags:

one `@param` tag for each parameter, and

one `@exception` tag for each exception thrown.

Method comments can use the `@param`, `@return`, `@exception`, and `@see` tags, in that order. Required tags:

one `@param` tag for each parameter,

one `@return` tag if the return type is not void, and

one `@exception` tag for each exception thrown.

Variable comments can use only the `@see` tag.

Required tags: none.

All of the above can also use the `@deprecated` tag to indicate the item might be removed in a future release, and to discourage its continued use.

A documentation comment ends with the characters `*/`. This is an example of a documentation comment for a method.:

```
/**
 * Checks an object for "coolness". Performs a comprehensive
 * coolness analysis on the object. An object is cool if it
 * inherited coolness from its parent; however, an object can
 * also establish coolness in its own right.
 *
 * @param obj the object to check for coolness
 * @param name the name of the object
 * @return true if the object is cool; false otherwise.
 * @exception OutOfMemoryError If there is not enough memory to
 * determine coolness.
 * @exception SecurityException If the security manager cannot be
 * created
 * @see isUncool
 * @see isHip
```

```

    */
public boolean isCool(Object obj, String name) throws OutOfMemoryError,
                                                    SecurityException
{
    /* ... */
}

```

9 Classes

A class declaration looks like the following. Elements in square brackets [] are optional.

```

[[ClassModifiers]] class ClassName [[Inheritances]]
{
    // ClassBody
}

```

ClassModifiers are any combination of the following keywords, in this order:

```
public abstract final
```

Inheritances are any combination of the following phrases, in this order:

```
extends SuperClass
implements Interfaces
```

SuperClass is the name of a superclass. Interfaces is the name of an interface, or a comma-separated list of interfaces. If more than one interface is given, then they should be sorted in lexical order.

A class declaration always starts in column 1. All of the above elements of the class declaration should appear on a single line (unless it is necessary to break it up into continuation lines if it exceeds the allowable line length).

The ClassBody is indented by the standard indentation of four spaces. The closing brace “}” appears on its own line in column 1. There should not be a semi-colon following the closing brace. If the class declaration has one or more continuation lines, then a single blank line should immediately follow the opening brace.

Example:

```

/* Long class declaration that requires 2 continuation lines. */
/* Notice the opening brace is immediately followed by a blank line. */
public abstract class VeryLongNameOfTheClassBeingDefined extends
    VeryLongNameOfTheSuperClassBeingExtended
    implements Interface1, Interface2,
    Interface3, Interface4
{
    static private String buf[256];
}

```

The body of a class declaration should be organized in the following order:

1. Static variable field declarations
2. Instance variable field declarations
3. Static initializer
4. Static member inner class declarations
5. Static method declarations

6. Instance initializer
7. Instance constructor declarations
8. Instance member inner class declarations
9. Instance method declarations

These elements, fields, constructors, and methods, are collectively referred to as “members”. Within each numbered group above, sort in lexical order.

Note that there are four access levels for class members in Java: public, protected, default, and private, in order of decreasing accessibility. In general, a member should be given the lowest access level, which is appropriate for the member. For example, a member who is only accessed by classes in the same package should be set to default access. Also, declaring a lower access level will often give the compiler-increased opportunities for optimization. On the other hand, use of private makes it difficult to extend the class by sub-classing. If there is reason to believe the class might be sub-classed in the future, then members that might be needed by sub-classes should be declared protected instead of private.

All public members must be preceded by a documentation comment. Protected and default access members may have a documentation comment as well, at the programmer’s discretion. Private fields should not have a documentation comment. However, all fields that do not have documentation comments should have single-line comments describing them, if their function is not obvious from the name.

10 Class (static) and Instance Variable Field Declarations

Class variable field declarations, if any, come first. Class variables are those fields, which have the keyword static in their declarations. Instance variable field declarations, if any, come next. Instance variables are those, which do not have the keyword static in their declarations. It is tempting to want to group these declarations together by access level; i.e., group all the public members together, then all the default access member, then all the protected members, etc. However, static/non-static is a more important conceptual distinction than access level. Also, there are so many different access levels in Java that it becomes too confusing, and does not work well in practice. The private protected access level is obsolete and should not be used.

```
[[FieldModifiers]] Type fieldName [= Initializer];
```

FieldModifiers are any legal combination of the following keywords, in this order:

```
public protected private static final transient volatile
```

Always put field declarations on separate line; do not group them together on a single line:

```
static private int useCount, index; /* WRONG */
static private int useCount; /* RIGHT */
static private long index; /* RIGHT */
```

A field that is never changed after initialization should be declared final. This not only serves as useful documentation to the reader, but also allows the compiler to generate more efficient code. It is also a good idea to align the field names so that they all start in the same column.

11 Static_INITIALIZER

A static initializer, if any, comes next. It is called when the class is first referenced, before any constructors are called. It is useful for initializing blank static final fields (static final fields not initialized at point of declaration). There should be at most one static initializer per class. It has the following form:

```
static
{
    statements;
}
```

12 Static Member Inner Class Declarations

Static inner (nested) classes, which pertain to a class as a whole rather than any particular instance, if any, come next:

```
public class Outer
{
    static class Inner
    {
        /* static inner class */
    }
}
```

13 Static Method Declarations

Any static methods come next. A static method follows the same rules as instance methods. Note that `main()` is a static method.

14 Instance_INITIALIZER

An instance (non-static) initializer, if any, comes next. If present, it is called from every constructor after any calls to super-class constructors. It is useful for initializing blank final fields (final fields not initialized at point of declaration), and for initializing anonymous inner classes since they cannot declare constructors. There should be at most one instance initializer per class:

```
/* Instance initializer */
{
statements;
}
```

15 Constructor Declarations

Constructor declarations, if any, come next. All of the elements of the constructor declaration should

appear on a single line (unless it is necessary to break it up into continuation lines if it exceeds the allowable line length). Example:

```
/**
 * Constructs a new empty FooBar.
 */
public FooBar()
{
    value = new char[0];
}
```

If there is more than one constructor, sort them lexically by formal parameter list, with constructors having more parameters always coming after those with fewer parameters. This implies that a constructor with no arguments (if it exists) is always the first one.

16 Instance Method Declarations

Instance method declarations, if any, come next. Instance methods are those, which do not have the keyword `static` in their declarations.

17 Method Declarations

All of the elements of a method declaration should appear on a single line (unless it is necessary to break it up into continuation lines if it exceeds the allowable line length). A method declaration looks like the following. Elements in square brackets are optional.

```
[[MethodModifiers]] Type MethodName(Parameters) [[throws Exceptions]]
```

MethodModifiers are any combination of the following phrases, in this order:

```
public protected private abstract static final synchronized native
```

Exceptions are the name of an exception, or a comma-separated list of exceptions. If more than one exception is given, then they should be sorted in lexical order.

Parameters are a list of formal parameter declarations. Parameters may be declared `final` in order to make the compiler enforce that the parameter is not changed in the body of the method, as well as to provide useful documentation to the reader. Parameters must be declared `final` in order to make them available to local inner classes. A method that will never be overridden by a sub-class should be declared `final`. This allows the compiler to generate more efficient code. Methods that are `private`, or declared in a class that is `final`, are implicitly `final`; however, in these cases the method should still be explicitly declared `final` for clarity. Methods are sorted in lexical order, with one exception: if there is a `finalize()` method, it should be the very last method declaration in the class. This makes it easy to quickly see whether a class has a `finalize()` method or not. If possible, a `finalize()` method should call `super.finalize()` as the last action it performs. Examples:

```
/* Long method declaration */
public static final synchronized long methodName() throws
    ArithmeticException,
```

```

{
    static int count;
}
/* Line broken in the middle of a parameter list */
/* Align just after left parenthesis */
public boolean imageUpdate(Image img, int infoflags,
                           int x, int y, int w, int h)
{
    int i;
}

```

18 Local Inner Classes

Inner (nested) classes may be declared local to a method. This makes the inner class unavailable to any other method in the enclosing class. They follow the same format rules as top-level classes:

```

Enumeration enumerate()
{
    class Enum implements Enumeration
    {
    }
    return new Enum();
}

```

19 Anonymous Inner Classes

Anonymous classes can be used when then following conditions are met:

1. The class is referred to directly in only one place.
2. The class definition is simple, and contains only a few lines.

In all other cases, use named classes (inner or not) instead.

AWT Listeners are a common case where anonymous classes are appropriate. In many such cases, the only purpose of the class is simply to call another method to do most of the work of handling an event. Anonymous inner classes follow similar rules as named classes; however there are a few rules specific to anonymous classes:

- When possible, the whole new expression, consisting of the new operator and the type name, should appear on the same line as the expression of which it is a part. If it does not fit on the line, then the whole new expression should moved to the next line as a unit.
- The body of the anonymous class should be indented by the normal indentation from the beginning of the line that contains the new expression.
- The closing brace should not be on a line by itself, but should be followed whatever tokens are required by the rest of the expression. Usually, this means the closing brace is followed by at least a semi-colon, closing parenthesis, or comma. The closing brace is indented to the same level as the line containing the new expression. There is no space immediately following the closing brace.

Examples:

```

/* Anonymous class inside a return expression */
Enumeration myEnumerate(final Object array[])
{

```

```

return new Enumeration()
{
    int count = 0;
    public boolean hasMoreElements()
    {
        return count < array.length;
    }
    public Object nextElement()
    {
        return array[count++];
    }
}; /* } followed by ; */
}

/* Anonymous class inside a parenthesized expression */
helpButton.addActionListener(new ActionListener()
{
    public void actionPerformed(ActionEvent e)
    {
        showHelp();
    }
}); /* } followed by ); */

```

20 Anonymous Array Expressions and Array Initializers

Anonymous arrays can be used wherever an array value is needed. If the entire anonymous array expression fits on one line, then it is acceptable to place it on a single line. Otherwise, there should be one initializer per line, with the same rules as for anonymous inner classes. The same rules also apply to array initializers in array declarations.

```

/* Example where entire array expression fits on one line */
Polygon p = new Polygon(new int[] {0, 1, 2}, new int[] {10, 11, 12}, 3);
/* Example with one array initializer per line */
String errorMessages[] = {"No such file or directory",
                          "Unable to open file",
                          "Unmatched parentheses in expression"};
/* Example of embedded anonymous array expression */
createMenuItems(new menuItemLabels[] {"Open",
                                       "Save",
                                       "Save As...",
                                       "Quit"} );

```

21 Interfaces

Interfaces follow a similar style to classes. An interface declaration looks like the following. Elements in square brackets “[]” are optional.

```

[[public]] interface InterfaceName [[extends SuperInterfaces]]
{
    // InterfaceBody
}

```

“SuperInterfaces” is the name of an interface, or a comma-separated list of interfaces. If more than one interface is given, then they should be sorted in lexical order.

An interface declaration always starts in column 1. All of the above elements of the interface declaration should appear on a single line (unless it is necessary to break it up into continuation lines if it exceeds the allowable line length). The InterfaceBody is indented by the standard indentation of four spaces. The opening brace "{" and closing brace "}" both appear on their own line in column 1. There should not be a semi-colon following the closing brace. All interfaces are inherently abstract; do not explicitly include this keyword in the declaration of an interface.

All interface fields are inherently public, static, and final; do not explicitly include these keywords in the declaration of an interface field. All interface methods are inherently public and abstract; do not explicitly include these keywords in the declaration of an interface method.

Except as otherwise noted, interface declarations follow the same style guidelines as classes.

The body of an interface declaration should be organized in the following order:

1. Interface constant field declarations.
2. Interface method declarations

The declaration styles of interface fields and methods are identical to the styles for class fields and methods.

22 Simple Statement Construction

See the C coding standards section for statement standards not listed here.

23 try, catch, and finally Statements

```
try
{
    statements;
}
catch (exception-declaration)
{
    statements;
}
finally
{
    statements;
}
```

24 synchronized Statement

```
synchronized (expression)
{
    statements;
}
```

25 Labeled Statements

Labeled statements should always be enclosed in braces “{}”. The label itself should be indented to the normal indentation level, followed by a colon and single space. The closing brace should have a trailing comment on the same line with the label repeated:

```
statement-label:
{
} /* statement-label */
```

26 Existing Code

Some code used in the RDA was originally written using other coding styles. When writing code for a module that does not conform to these guidelines, adopt the style of the existing code.