Requirements for Writing Java API Specifications

Requirements for Writing Java API Specifications [1]

This document describes the requirements for writing API specifications for the Java platform. The specification for each Java ™ platform API library is made up of its Javadoc comments and additional support documentation called out in the doc comments. (See background.)

This document has five sections that correspond to the sections of an API specification; each section (except the first) includes examples.

Note: Examples have been modified to demonstrate completeness. They may not be completely accurate.

Assertions
assertion
Assertions are critical to conformance testing and implementors of the Java Platform. The Java Compatibility Kit includes a test to verify each assertion, to determine what passes as Java Compatible ™ .

Top-Level Specification

The top-level specification is composed of those specifications that apply to the entire set of packages. It can include assumptions that underlie the other specifications, such as all objects are presumed to be thread-safe unless otherwise specified.
In addition to the class specific requirements, there are overall Java platform API documentation requirements with respect to handling unchecked exceptions (exceptions that derive from java.lang.RuntimeException). It would be helpful to develop some blanket statements that describe the general situations when a Java application should be prepared to encounter one or more RuntimeExceptions.

Package Specification

The Package specification includes any specifications that apply to the package as a whole or to groups of classes in the package. It must include:
Executive summary - A precise and concise description of the package. Useful to describe groupings of classes and introduce major terms. See example.
OS/Hardware Dependencies - Specify any reliance on the underlying operating system or hardware. For example, the java.awt package might describe how the general behavior in that package is allowed to vary from one operating system to another (such as Windows, Solaris and Macintosh). See example.
References to any external specifications. These are package-wide specifications beyond those generated by Javadoc by Sun or third-parties. An example is the UNICODE specification for the java.text package. These references can be links to specifications published on the Internet, or titles of specifications available only in print form. The references must be only as narrow or broad in scope as the specification requires. That is, if only a section of a referenced document is considered part of the API spec, then you should link or refer to only that section (and can separately refer to the non-spec of the document as a "related" document). The idea is to clearly delineate what is part of the API spec and what is not. See example.
For details about how to actually structure the package information in the `package.html` file according to Java Software standards, see Package-Level Comments.

Package Example #1

This example demonstrates the **Executive Summary** (first 3 paragraphs) and **OS/Hardware Dependencies** (fourth paragraph).

Package java.awt Description
Contains classes for creating user interfaces and for painting graphics and images. A user interface object such as a button or a scrollbar is called, in AWT terminology, a component. The Component class is the root of all AWT components. See Component for a detailed description of properties that all AWT components share.

Some components fire events when a user interacts with the components. The AWTEvent class and its subclasses are used to represent the events that AWT components can fire. See AWTEvent for a description of the AWT event model.

A container is a component that can contain components and other containers. A container can also have a layout manager that controls the visual placement of components in the container. The AWT package contains several layout manager classes and an interface for building your own layout manager. See Container and LayoutManager for more information.

Behavior of user interface components may be restricted by the underlying operating system. For example, Windows does not allow simultaneously displaying the caret position and selection highlight, while Solaris does. That is, in Windows, applying the `setCaretPosition` method to a text area causes any highlighted text to become unhighlighted, but in Solaris that method does not disturb a highlight.

Package Example #2

This example demonstrates the **Executive Summary** (first section) and **References to External Specifications** (second section). (This second section contains links to the documents published on the Internet.) Notice that the references are specific down to the version number.

Package java.util.zip Description
Provides classes for reading and writing the standard ZIP and GZIP file formats. Also includes classes for compressing and decompressing data using the DEFLATE compression algorithm, which is used by the ZIP and GZIP file formats. Additionally, there are utility classes for computing the CRC-32 and Adler-32 checksums of arbitrary input streams.

Package Specification

The following external documents are part of the specification:

Info-ZIP Application Note 970311 - a detailed description of the Info-ZIP format upon which the java.util.zip classes are based.
ZLIB Compressed Data Format Specification version 3.3 (PostScript) (RFC 1950)
DEFLATE Compressed Data Format Specification version 1.3 (PostScript) (RFC 1951)
GZIP file format specification version 4.3 (PostScript) (RFC 1952)

Class/Interface Specification

This section applies to Java classes and interfaces. Each class and interface specification must include:
Executive summary - A precise and concise description for the object. Useful to describe groupings of methods and introduce major terms. See both examples.
State Information - Specify the state information associated with the object, described in a manner that decouples the states from the operations that may query or change these states. This should also include whether instances of this class are thread safe. (For multi-state objects, a state diagram may be the clearest way to present this information.) If the class allows only single state instances, such as java.lang.Integer, and for interfaces, this section may be skipped. See example.
OS/Hardware Dependencies - Specify any reliance on the underlying operating system or hardware. See example.
Allowed Implementation Variances - Specify how any aspect of this object that may vary by implementation. This description should not include information about current Java Software implementation bugs. See example.
Security Constraints - If the object has any security constraints or restrictions, an overview of those constraints and restrictions must be provided in the class specification. Documentation for individual security constrained methods must provide detailed information about security constraints. See example.
Serialized Form - This spec ensures that a serialized object can successfully be passed between different implementations of the Java Platform. While public classes that implement serializable are part of the serialized form, note that in some cases it is also necessary to include *non-public* classes that implement serializable. For more details, see the specific criteria. The serialized form spec defines the `readObject` and `writeObject` methods, the fields that are serialized, the data types of those fields, and the order those fields are serialized. See example.
References to any External Secifications - These are class-level specifications written by Sun or third parties beyond those generated by Javadoc. References are not necessary here if they have been included in the Package specification. (No example given.)
You may include graphic model diagrams, such as state diagrams, to describe static and dynamic information about objects. Such diagrams may become a requirement in the future. Code examples are useful and illustrative.

Interface Example

This example demonstrates the **Executive Summary** (first paragraph). The following paragraphs provide more detail and special cases for the specification.

> `public interface` **`Set`**
> `extends Collection`
> A collection that contains no duplicate elements. More formally, sets contain no pair of elements `e1` and `e2` such that `e1.equals(e2)`, and at most one null element. As implied by its name, this interface models the mathematical set abstraction.
>
> The `Set` interface places additional stipulations, beyond those inherited from the `Collection` interface, on the contracts of all constructors and on the contracts of the `add`, `equals` and `hashCode` methods. Declarations for other inherited methods are also included here for convenience. (The specifications accompanying these declarations have been tailored to the `Set` interface, but they do not contain any additional stipulations.)
>
> The additional stipulation on constructors is, not surprisingly, that all constructors must create a set that contains no duplicate elements (as defined above).
>
> Note: Great care must be exercised if mutable objects are used as set elements. The behavior of a set is not specified if the value of an object is changed in a manner that affects `equals` comparisons while the object is an element in the set. A special case of this prohibition is that it is not permissible for a set to contain itself as an element.

Class Example #1

This example demonstrates the **Executive Summary** (first paragraph), **State Information** (second paragraph), **OS/Hardware Dependencies** (third paragraph), **Allowed Implementation Variance** (third paragraph) and **Security Constraints** (fourth paragraph).

> `public class` **`FileOutputStream`**
> `extends OutputStream`
> A file output stream is an output stream for writing byte data to a `File` or `FileDescriptor` object. Character data should be written using a `Writer` object attached to a `FileOutputStream` object.
>
> `FileOutputStream` objects have essentially two states: (1) open and available for writing, and (2) closed. Attempting to write to a closed stream will result in a `java.io.IOExeception` being thrown.
>
> While `FileOutputStream` objects have only two states, the behavior while the stream is open and available for writing may differ by platform and by implementation. Since a `FileOutputStream` object may be buffered either by the Java implementation or by the underlying operating system, errors writing to the file or file descriptor may only be exposed when flushing or closing the stream. Those stream write errors include but are not limited to:
>
> Storage medium is full
> Pipe to other process is broken
> File being written to is deleted by another process
> In addition, the ability to create or write to a file or file descriptor may be constrained by a Security Manager. This means that the constructors may throw `java.lang.SecurityException`. See `SecurityManager.checkWrite()` for more information.

Class Example #2

(*Example to come*)

Field Specification

Each field specification must include:
What this field models - Specify what aspect of the object this field models. See example.
Range of valid values - Specify all valid and invalid values for this field. See example.
For each public and protected static final field whose type is a primitive or String, specify its value. (A future version of Javadoc will automatically add this value to the spec, but until then please type the value into the body of the comment.)
Null Value - If this is a reference field, a statement concerning whether this value may be `null`, and how this object will behave in such a case. See example.

Field Example #1

These four examples of fields demonstrate **What this field models** (first sentence) and **Valid range of values** (second sentence). They do not need to specify null value, since they are primitive types.

> Fields from `java.awt.Rectangle`
> **x**
> **`public int x`**
> The x coordinate of one corner of this rectangle. Negative values are allowed for this field.
>
> **y**
> **`public int y`**
> The y coordinate of one corner of this rectangle. Negative values are allowed for this field.
>
> **width**
> **`public int width`**
> The width of this rectangle. When width is less than zero, Rectangle behavior is undefined.
> The behavior of the following methods is undefined when either width or height is less than zero: `add()`, `contains()`, `getBounds()`, `getSize()`, `grow()`, `inside()`, `intersection()`, `intersects()`, `reshape()`, `resize()`, `setBounds()`, `setSize()`, `translate()`, and `union()`.
>
> **height**
> **`public int height`**
> The height of this rectangle. Rectangle behavior is undefined when height is less than zero.

Field Example #2

This example demonstrates **What this field models** (first paragraph) and **Null value** (second paragraph). This field is not constrained to a range of values.

Field from `java.io.FilterInputStream`
in
`protected InputStream in`
The underlying input stream. Calls to `FilterInputStream` methods are usually delegated to the corresponding method of the underlying input stream. That delegation may not happen immediately, since some processing will happen in the `FilterInputStream` object.
This value may be `null`. If `null`, all method calls to the `FilterInputStream` will return `java.lang.NullPointerException`.

Method Specification

This section applies to Java methods and constructors. Each method and constructor specification must include:
Expected Behavior - Specify the expected or desired behavior of this operation. Describe what aspect of the object being modeled this operation fulfills. All examples below include this.
State Transitions - Specify what state transitions this operation may trigger. See example.
Range of Valid Argument Values - Specify all valid and invalid values for each argument, including expected behavior for invalid input value or range of values. See example.
Null Argument Values - For each reference type argument, specify the behavior when `null` is passed in. See two examples. NOTE: If possible, document the general null argument behavior at the package or class level, such as causing a `java.lang.NullPointerException` to be thrown. Deviations from this behavior can then be documented at the method level.
Range of Return Values - Specify the range of possible return values, including where the return value may be `null`. See example.
Algorithms Defined - When required by the specification, specify the algorithms used by this operation. See example.
OS/Hardware Dependencies - Specify any reliance on the underlying operating system or hardware. See example.
Allowed Implementation Variances - Specify what behavior may vary by implementation. This description should not include information about current Java Software implementation bugs. See example.
Cause of Exceptions - Specify the exceptions thrown by the method, include the argument values, state, or context that will cause the specified exception to be thrown. The exceptions thrown from a method need not be mutually exclusive. See example. For more detail about which exceptions to document, see Documenting Exceptions with the @throws Tag.
Security Constraints - If this operation may be security constrained, must specify the security check used to constrain this operation. Mention if the method is implemented using a `AccessController.doPrivileged` construct. Also must include a general description of the context or situations where this method may be security constrained. See example.

Method Example #1

This example demonstrates the **Expected Behavior**, **State Transitions** (first paragraph) and **Cause of Exceptions** (second and following paragraphs).

Method from `java.util.BitSet`
set
`public void set(int index)`
Sets the bit specified by the index to `true`. If the bit is already `true`, it will remain `true`.
If the index is less than zero, an `IndexOutOfBoundsException` will be thrown. If the `BitSet` object does not have `index` number of bits it will attempt to grow to include at least `index` number of bits. This resize operation will fail and throw an `OutOfMemoryError` if the Java system runs out of memory.

Parameters:
`index` - index for the bit to be set to `true`.

Throws:
`java.lang.IndexOutOfBoundsException` - if the specified index is negative.

Throws:
`java.lang.OutOfMemoryError` - if the `BitSet` object cannot grow to accommodate `index` number of bits.

Method Example #2

This example demonstrates the **Expected behavior** and **Range of Valid Argument Values**.

Method from `java.awt.Color`
getBlue
`public int getBlue()`
Returns the blue component in the range 0-255 in the default sRGB space.

Returns:
the blue component.

Method Example #3

This example demonstrates **Null Argument Values**, how `null` can be a valid argument value.

Constructor from `java.lang.Boolean`
Boolean
`public Boolean(String s)`

Allocates a `Boolean` object representing the value `true` if the string argument is not `null` and is equal, ignoring case, to the string `"true"`. Otherwise, allocate a `Boolean` object representing the value `false`. Examples:
`new Boolean("True")` produces a `Boolean` object that represents `true`.
`new Boolean("yes")` produces a `Boolean` object that represents `false`. `new Boolean(null)` produces a `Boolean` object that represents `false`.

Parameters:
`s` - the string to be converted to a `Boolean`.

Method Example #4

This example demonstrates **Null Argument Values** and **Cause of Exceptions**. It shows how a `NullPointerException` can be thrown when `null` is passed in.

Method from `java.lang.Double`
parseDouble
`public static double parseDouble(String s) throws NumberFormatException`

Returns the double value represented by the specified `String`, as performed by the `valueOf` method of class `Double`.

Parameters:
`s` - the string to be parsed.

Returns:
the double value represented by the string argument.

Throws:
`NumberFormatException` - if the string does not contain a parsable double.

`NullPointerException` - if the string is `null`.

Method Example #5

This example demonstrates **Range of Return Values**. In this case, the specification constrains the return value to be non-negative, which adds information to what you could tell by its type alone (`int`). Note that it is not necessary to state the max is `Integer.MAX_VALUE`, since that is understood to be true for all `int` values.

Method from `java.util.BitSet`
length
`public int length()`

Returns the "logical size" of this `BitSet`: the index of the highest set bit in the `BitSet` plus one. Returns zero if the `BitSet` contains no set bits.

Returns:
the non-negative logical size of this `BitSet`.

Method Example #6

This example demonstrates **Algorithms Defined**.

Method from `java.lang.String`
hashCode
`public int hashCode()`

Returns a hashcode for this string. The hashcode for a `String` object is computed as:

```
s[0]*31^(n-1) + s[1]*31^(n-2) + ... + s[n-1]
```

using `int` arithmetic, where `s[i]` is the *i*th character of the string, `n` is the length of the string, and `^` indicates exponentiation. (The hash value of the empty string is zero.)

Returns:
a hash code value for this object.

Method Example #7

This example demonstrates **OS/Hardware Dependencies**, **Allowed Implementation Variances**, and **Cause of Exceptions**.

Method from `java.io.FileOutputStream`
write
`public void write(int b) throws IOException`

Writes the specified byte to this file output stream. Implements the `write` method of `OutputStream`.
The behavior while the stream is open and available for writing may differ by platform and by implementation. Since the byte may be buffered either by the Java implementation or by the underlying operating system, this byte might not be immediately written to disk. Therefore, errors writing to the file or file descriptor might only be exposed when flushing or closing the stream.

Parameters:
`b` - the byte to be written.

Throws:
`IOException` - if an I/O error occurs.

Method Example #8

This example demonstrates **Security Constraints** and **Cause of Exceptions**.

Constructor from `java.io.FileOutputStream`
FileOutputStream
`public FileOutputStream(String name) throws IOException`

Creates an output file stream to write to the file with the specified name.

Parameters:
`name` - the system-dependent filename.

Throws: **`java.io.IOException`**
if the file could not be opened for writing.

Throws: **`java.lang.SecurityException`**
if write access to the named file is not allowed. If a security manager exists, this method calls the security manager `checkWrite` method with the `name` argument. If access to the named file is not allowed, the security exception thrown by the security manager `checkWrite` method will be surfaced here.

[1]This document is based on the Object Class Specification by Edward V. Berard, *Essays on Object-Oriented Software Engineering*, 1993 Simon & Schuster, Englewood Cliffs, NJ; pp. 131-162.

Kevin A. Smith and Doug Kramer
Last Updated: January 2003.