API Knowledge Coding Guide Version 7.2

You will be presented with **documentation blocks** extracted from API reference documentation (Javadocs and the like). For each block, you will be also presented with the name of its corresponding package/namespace, class, method, or field. Your task is to read each block carefully and evaluate whether the block contains knowledge of the different types described below. You will need to evaluate whether each block contains **knowledge of each different type**. Rate the knowledge type as true only if there is clear evidence that knowledge of that type is present in the block. If you **hesitate** about whether or not to rate a knowledge type as true, leave it as false.

Do not evaluate automatically generated information such as the declaration of an element (e.g. extends MyInterface), or generated links in "specified by". Only evaluate human created documentation in the block (see last section in page 5 for more details).

Read the following description very carefully. It explains how to rate each knowledge type for a given block.

Knowledge Types

Functionality and Behavior

Describes **what** the API does (or does not do) in terms of **functionality** or **features**. The block describes **what happens** when the API **is used** (a field value is set, or a method is called). This **also** includes **specified behavior** such as what an element does, given special input values (for example, null) or what may cause an exception to be raised.

Functionality and behavior knowledge can also be found in the **description of parameters** (e.g., what the element does in response to a specific input), **return values** (e.g., what the API element returns), and **thrown exceptions**.

- *Detects stream close and notifies the watcher*
- *Obtains the SSL session of the underlying connection, if any. If this connection is open, and the underlying socket is an SSLSocket, the SSL session of that socket is obtained. This is a potentially blocking operation.*

Only rate this type as true if the block contains information that actually adds to what is obvious given the complete signature of the API element associated with the block. If a description of functionality only repeats the name of the method or field, it does not contain this type of knowledge and you should rate it as false, and instead rate the knowledge type non-information as true. For example, this would be the case if the documentation for a method called getTitle was

- *Returns the title.*

Similarly for constructors, if the documentation simply states "*Constructs a new X*", "*Instantiates a new object*", or something similar the value is false (with non-information coded as true). In some cases non-information will be phrased to look like a description of functionality, for examples with sentences that start with verbs like "gets", "adds", "determines", "initializes". Carefully read the name and signature of the API element and only assign a value of true for this knowledge type if the block adds something to the description of the element.

However, if any other details are provided, rate this type as true. For example:

- *Creates a new MalformedChallengeException with a null detail message.*

Should get a value of true because of the additional information about the value of the message field.

Mentioning that a **value can be obtained** from a field, property, or getter method does not constitute a description of functionality, except the API performs some additional functions when the value is accessed. For example, the block below does not represent a description of functionality. The Non-information type for this block should be rated as true.

- *[LoggerDescription.Verbosity Property] Gets the verbosity level for the logger.*

Note IMPORTANT: Description of functionality is not limited to the functionality of the element associated with the block, but the API as a whole. **However, if the block explains a sequence of method calls or creation of particular objects (e.g. events) code this as Control-flow.** For example, if setting the value of a field results in some perceived behavior by the framework, this knowledge counts as functionality. If the block describes a resulting sequence of method calls or events fired, this is control flow. If the block contains both, then both should be coded as true.

Concepts

Explains the **meaning** of terms used to name or describe an API element, or describes a **design or domain concepts** used or implemented by the API. Code this knowledge type with a value of true if an explanation of concepts is provided, with some useful details. The sentences below show an example of a description of the concept of "secure sockets".

- *Plain sockets may be considered secure, for example if they are connected to a known host in the same network segment. On the other hand, SSL sockets may be considered insecure, for example depending on the chosen cipher suite.*

Note: Basic description of the parameters does not represent concept description, such as

- *header - the challenge header [in a method called processChallenge]*

In this case, the knowledge type "non-information" should be set to true.

Directives

Specifies what users are **allowed / not allowed** to do with the API element. Directives are clear **contracts**. For example how callers must deal with return values, what is permitted by implementers of abstract elements, limitations on the sequences in which API elements may be accessed, or any explicit mention of the input values that are allowed (or not allowed) for an API element (e.g. parameters values for methods or values that can be assigned to fields).

- *If this method returns false, the caller MUST close the connection to correctly comply with the HTTP protocol. If it returns true, the caller SHOULD attempt to keep the connection open for reuse with another request.*
- *An implementation is free to throw an exception, deliver a null result, deliver a misleading value, whatever is convenient.*
- *[Clients] should ensure that the resulting list is asserted back into the graph into the appropriate relationships*
- *An index between 0 and 100.*
- *This parameter can be true, false or "default" (null).*

In contrast to Patterns, directives represent **specific contracts or constraints** on how to use a specific element and the API.

Purpose and Rationale

Explains the purpose of providing an element or the rationale of a certain design decision. Typically, this is information that answers a "**why**" question: **Why is this element provided** by the API? Why is it this designed this way? Why would we want to use this? This includes in which context an API element can (or should) be used, or the **advantages** of using the element.

- *This allows for the header to be sent without another formatting step.*
- *This is important for generating the Cookie header because some cookie specifications require that the Cookie header should include certain attributes only if they were specified in the Set-Cookie header.*
- *[If EOF is detected, the watcher will be notified and this stream is detached from the underlying stream.] This prevents multiple notifications from this stream.*
- *This method can be used for obtaining metainformation about the entity implied by the request without transferring the entity-body itself.*
- *It can be used to quickly parse large amounts of text to find specific character patterns*
- *This enables the programmer to write code in a compact and easy style.*

Note: Descriptions of cases, where an element (such as a field) is used by the API, fall under either functionality or control-flow, as appropriate. Note also that sometimes the difference between **purpose and functionality** can be confused. For example, for a method drawCircle(int r) the description "draws a circle of radius r" could be interpreted as "the purpose is to draw a circle", or "this method should be used to draw a circle". For this study this is **not** the correct interpretation. Only code a value of true for purpose if the block contains a description of the purpose that **is not self-evident from the method's functionality**.

Quality Attributes and Internal Aspects

Describes quality attributes of the API, also known as non-functional requirements, for example, the **performance** or security implications of using the API element (or related elements), such as resources consumed or the execution time of a method. Also code with a value of true if the block provides information about API's **internal implementation** that is only indirectly related to its observable behavior. For example, indicates the main **data structures and algorithms employed**.

- *This is a "graceful" release and may cause IO operations for consuming the remainder of a response entity.*
- *Enumerating through a collection is intrinsically not a thread-safe procedure.*

Control-Flow

Describes how the API (or the framework) manages the flow of control, for example by stating what events cause a certain **callback** to be triggered, or by listing the **order** in which API methods will be **automatically called** by the framework itself. Wherever you find a clear description of the **sequence of methods** that result in calling an API, code control-flow knowledge as true (see Functionality).

- *On the client side, this step [the callback] is performed before the request is sent to the server.*
- *Changing the value of Path for a DirListBox control generates an Change event.*

Note: Descriptions of how the programmer should organize the control-flow of their code when using the API do not apply here. Instead, see the knowledge types Patterns, Functionality, or Example.

Structure

Describes the **internal organization** of a compound element (for example important classes, fields, or methods), information about **type hierarchies**, **how elements are related** to each other, or the **static properties** of an element. Basically, structural knowledge is knowledge about how different API elements related to each other.

- *HTTP messages consist of requests from client to server and responses from server to client.*
- *A Node has five subtypes: Node_Blank, Node_Anon, Node_URI, Node_Variable, and Node_ANY.*
- *Note that the order of arguments here is different from the **similar** public constructor, as required by Java*
- *public interface CookieSpec – Cookie management specification must define :a) rules of parsing "Set-Cookie" header b) rules of validation of parsed cookies; c)formatting of "Cookie" header;* for a given host, port and path of origin *chosen cipher suite.*
- *Instances of this class are unmodifiable*
- *This class is used with the [LINK] GZipStream and [LINK] DeflateStream classes*

Note Pointers to other sections of the reference documentation of the API (element names, hyperlinks, manually-added "see also" references) *can* also represent structural information *if* they indicate how elements relate to each other or what their properties are. (For references to elements in other APIs or to other documents, see the variable References.) For example, a note such as the following contains structural knowledge because it indicates another API element similar to the one documented:

- *Same as {@link #close close()}.*
- *Management interface for client connections. [A link to the class managed by the interface]*

However, simply adding a link to another part of the API does not automatically provide structural knowledge if it is not also mentioned why or how this element is related, or its role in the structure of the API. For example

- *See Also: Constant Field Values*
- See Also: closeExpiredConnections()

The first link simply points to a collection of constant definition: it does not say anything about how API elements are organized; The second link simply indicates the presence of another method that could be of interest to the programmer, it does not indicate how it relates to the current block. Both of these references should not be marked as structural knowledge.

Only evaluate structural information contained in the block: **do not evaluate any automatically generated information**, such as the structural information contained in the declaration of an element (e.g. extends MyInterface), or generated links such as "specified by". Be careful. Do not consider any information that is **directly derived** from the element name, declaring elements, or signature.

Patterns

Describes **how** to accomplish specific outcomes with the API, for example, how to implement a certain scenario, how can the behavior of an element be **customized**, how can the abstract class be **implemented**, and how some **values** can be obtained or some **objects created**.

- *Usually this is accomplished by using the 'Decorator' pattern where a wrapper entity class is used to decorate the original entity.*
- *Nodes are only constructed by the node factory methods*
- *The following parameters can be used to customize the behavior of this class … [followed by a multi-line description of the customization options offered by each parameter].*

Note: This knowledge type is different from Purpose in that it describes **how to do** things, not why elements should be used. It is also different from Directives in that it **provides guidelines and "how tos"**. In contrast to Directives, patterns information provides general hints on how to achieve various outcomes with the API, **not usage rules and constraints**.

Code Examples

The block provides **code examples** of how to use and combine elements to implement certain functionality or design outcomes.

- *The usual execution flow can be demonstrated by the code snippet below: + CODE SNIPPET*
- *keytool -genkey -v -alias "my client key" -validity 365 -keystore my.keystore*

Note: Code example can also be scripts or other machine instructions. They can be found with patterns but sometimes also without. Similarly, patterns can be enhanced with code example, but can also be found without.

Environment

The block contains knowledge about various aspects related to the **environment** in which the API is used, but not the API directly. For example, **compatibility** issues, differences between API **versions**, **licensing** information. For links to other documents use the Reference types instead.

- *Since HTTP/1.1, connections are re-used by default. Up until HTTP/1.0, connections are not re-used by default.*
- *This is a change from the behavior of Jena 1, which took a parameter closed to compute the closure over transitivity and equivalence of sub-classes.*
- *This package includes code adapted from Xerces 2.6.0, which is marked and is Copyright (c) 1999-2002 The Apache Software Foundation. All rights reserved.*

External References

The block includes any pointer to **external** documents, either in the form of hyperlinks, tagged "see also" reference, or **mentions** of other documents (such as standards or manuals). External documents are any documents except pages in the reference documentation of the Framework currently studied (Java or .NET).

- *Digest authentication scheme as defined in RFC2617.*
- *To find the final URI after any redirects have been processed, please see the section entitled HTTP execution context in the HttpClient Tutorial*

Non-information

Rate this knowledge as true if the block contains any complete sentence of self-contained fragment of text that provides only **uninformative boilerplate text**. Common examples include **restating the name** of the API element without adding any detail, or stating the obvious. Another common case is where the information to explain the return value just **restates the information** in the rest of the block. However, if a single sentence contains both uninformative and informative text, we consider that the uninformative part of the sentence is only context for the informative part, and in this case you should not rate it as non-information.

- *uri – The URI*
- *IOException – If something happens.*
- *[PanelArray.ControlAdded] Event Occurs when a new control is added to the PanelArray.*

Note: Non-information **is not mutually exclusive** with Functionality knowledge. If the block only restates the name of a method, rate Functionality as false and Non-information as true. If the block provides a true description of the functionality, but just restates this information for example with the return tag, then rate Functionality as true and Non-information as true. In brief, rate Non-information as true whenever a block **contains complete sentences or self-contained text fragments** that provides no information. **Note:** this is **different from providing no text whatsoever** (empty block), which you should not consider non-information.

The next example shows a combination of directive (the first sentence) and non-information (the second sentence). Although the first sentence provides a directive (do not use this method in your code), the second sentence is just a re-statement of the method's name. This block would receive a value of true for both the directive and the non-information types.

- *[FormatUrl.Execute Method] This API supports the .NET Framework infrastructure and is not intended to be used directly from your code. Executes the FormatUrl task.*

Information to be Coded

Do not evaluate the name of the method. However, you need to read and understand the name to be able to evaluate the documentation.

Please ignore/**do not evaluate** the following automatically generated sections and blocks:

Java

- *@deprecated [just the tag but not the text which might follow]*
- *@immutable [just the tag but not the text which might follow]*
- *Java keyword such as implements, extends, throws, return etc.*
- *since:*
- *specified by [and the text which belongs to it]*
- *overrides [and the text which belongs to it]*

DotNet

- *Inheritance Hierarchy [this might be valuable context information]*
- *Syntax except C#*
- *Version Information*
- *Platforms*
- *This sentence in section Thread Safety: [Any public static (Shared in Visual Basic) members of this type are thread safe. Any instance members are not guaranteed to be thread safe.]*
- *This sentences in section .NET Framework Security: [Full trust for the immediate caller. This member cannot be used by partially trusted code. For more information, see Using Libraries from Partially Trusted Code.]*
- *References in section See Also that only point to an element's own class, package/namespace, or to elements directly visible in the declaration of the element associated with the block, such as parameter or return types for a method, declared type for fields.*
- *References in sect See Also without any explanation that relates the reference to the block.*
- *Community Content*
- *Change History*
- *In the code example section, do not take into account the sentence "The Namespace statement must appear outside of any classes or modules." (that is, ignore this sentence, do not count it as a directive or pattern).*

Tips

- Do not interpret too much. Stick to the guide while evaluating knowledge types!
- Do not code for more that one hour at once!
- Only code as true if the knowledge is clear based on the description of the guide!
- Read the guide from time to time!
- Do not make too long breaks (e.g. several weeks)!
- You might encounter more that one knowledge type in a single sentence.