**Java API Design Checklist**

January 16, 2012[4 Comments](http://theamiableapi.com/2012/01/16/java-api-design-checklist/#respond)

There are many different rules and tradeoffs to consider during Java API design. Like any complex task, it tests the limits of our attention and memory. Similar to the pilots’ pre-flight checklist, this list helps software designers remember obvious and not so obvious rules while designing Java APIs. It is a complement to and intended to be used together with the [API Design Guidelines](http://theamiableapi.com/2011/11/21/api-design-webinar-slides-and-recording/).

This list uses the following conventions:

(Do) **verb...** - Indicates the required design

**Favor...** - Indicates the best of several design alternatives

**Consider...** - Indicates a possible design improvement

**Avoid...** - Indicates a design weakness

**Do not...**  - Indicates a design mistake

**1. Package Design Checklist**

**1.1. General**

* 1.1.1. **Favor** placing API and implementation into separate packages
* 1.1.2. **Favor** placing APIs into high-level packages and implementation into lower-level packages
* 1.1.3. **Consider** breaking up large APIs into several packages
* 1.1.4. **Consider** putting API and implementation packages into separate Java archives
* 1.1.5. **Avoid** (minimize) internal dependencies on implementation classes in APIs
* 1.1.6. **Avoid** unnecessary API fragmentation
* 1.1.7. **Do not** place public implementation classes in the API package
* 1.1.8. **Do not** create dependencies between callers and implementation classes
* 1.1.9. **Do not** place unrelated APIs into the same package
* 1.1.10. **Do not** place API and SPI into the same package
* 1.1.11. **Do not** move or rename the package of an already released public API

**1.2. Naming**

* 1.2.1. **Start** package names with the company’s official root namespace
* 1.2.2. **Use** a stable product or product family name at the second level of the package name
* 1.2.3. **Use** the name of the API as the final part of the package name
* 1.2.4. **Consider** marking implementation-only packages by including “internal” in the package name
* 1.2.5. **Avoid** composite names
* 1.2.6. **Avoid** using the same name for both package and class inside the package
* 1.2.7. **Avoid** using “api” in package names
* 1.2.8. **Do not** use marketing, project, organizational unit or geographic location names
* 1.2.9. **Do not** use uppercase characters in package names

**1.3. Documentation**

* 1.3.1. **Provide** a package overview (package.html) for each package
* 1.3.2. **Follow** standard Javadoc conventions
* 1.3.3. **Begin** with a short, one sentence summary of the API
* 1.3.4. **Provide** enough details to help deciding if and how to use the API
* 1.3.5. **Indicate** the entry points (main classes or methods) of the API
* 1.3.6. **Include** sample code for the main, most fundamental use case
* 1.3.7. **Include** a link to the Developer Guide
* 1.3.8. **Include** a link to the Cookbook
* 1.3.9. **Indicate** related APIs
* 1.3.10. **Include** the API version number
* 1.3.11. **Indicate** deprecated API versions with the @deprecated tag
* 1.3.12. **Consider** including a copyright notice
* 1.3.13. **Avoid** lengthy package overviews
* 1.3.14. **Do not** include implementation packages into published Javadoc

**2. Type Design Checklist**

**2.1. General**

* 2.1.1. **Ensure** each type has a single, well-defined purpose
* 2.1.2. **Ensure** types represent domain concepts, not design abstractions
* 2.1.3. **Limit** the number of types
* 2.1.4. **Limit** the size of types
* 2.1.5. **Follow** consistent design patterns when designing related types
* 2.1.6. **Favor** multiple (private) implementations over multiple public types
* 2.1.7. **Favor** interfaces over class inheritance for expressing simple commonality in behavior
* 2.1.8. **Favor** abstract classes over interfaces for decoupling API from implementation
* 2.1.9. **Favor** enumeration types over constants
* 2.1.10. **Consider** generic types
* 2.1.11. **Consider** placing constraints on the generic type parameter
* 2.1.12. **Consider** using interfaces to achieve similar effect to multiple inheritance
* 2.1.13. **Avoid** designing for client extension
* 2.1.14. **Avoid** deep inheritance hierarchies
* 2.1.15. **Do not** use public nested types
* 2.1.16. **Do not** declare public or protected fields
* 2.1.17. **Do not** expose implementation inheritance to the client

**2.2. Naming**

* 2.2.1. **Use** a noun or a noun phrase
* 2.2.2. **Use** PascalCasing
* 2.2.3. **Capitalize** only the first letter of acronyms
* 2.2.4. **Use** accurate names for purpose of the type
* 2.2.5. **Reserve** the shortest, most memorable name for the most frequently used type
* 2.2.6. **End** the name of all exceptions with the word “Exception”
* 2.2.7. **Use** singular nouns (Color, not Colors) for naming enumerated types
* 2.2.8. **Consider** longer names
* 2.2.9. **Consider** ending the name of derived class with the name of the base class
* 2.2.10. **Consider** starting the name of an abstract class with the word “Abstract”
* 2.2.11. **Avoid** abbreviations
* 2.2.12. **Avoid** generic nouns
* 2.2.13. **Avoid** synonyms
* 2.2.14. **Avoid** type names used in related APIs
* 2.2.15. **Do not** use names which differ in case alone
* 2.2.16. **Do not** use prefixes
* 2.2.17. **Do not** prefix interface names with “I”
* 2.2.18. **Do not** use types names used in Java core packages

**2.3. Classes**

* 2.3.1. **Minimize** implementation dependencies
* 2.3.2. **List** public methods first
* 2.3.3. **Declare** implementation methods private
* 2.3.4. **Define** at least one public concrete class which extends a public abstract class
* 2.3.5. **Provide** adequate defaults for the basic usage scenarios
* 2.3.6. **Design** classes with strong invariants
* 2.3.7. **Group** stateless, accessor and mutator methods together
* 2.3.8. **Keep** the number of mutator methods at a minimum
* 2.3.9. **Consider** providing a default no-parameter constructor
* 2.3.10. **Consider** overriding equals(), hashCode() and toString()
* 2.3.11. **Consider** implementing Comparable
* 2.3.12. **Consider** implementing Serializable
* 2.3.13. **Consider** making classes re-entrant
* 2.3.14. **Consider** declaring the class as final
* 2.3.15. **Consider** preventing class instantiation by not providing a public constructor
* 2.3.16. **Consider** using custom types to enforce strong preconditions as class invariants
* 2.3.17. **Consider** designing immutable classes
* 2.3.18. **Avoid** static classes
* 2.3.19. **Avoid** using Cloneable
* 2.3.20. **Do not** add instance members to static classes
* 2.3.21. **Do not** define public constructors for public abstract classes clients should not extend
* 2.3.22. **Do not** require extensive initialization

**2.4. Interfaces**

* 2.4.1. **Provide** at least one implementing class for every public interface
* 2.4.2. **Provide** at least one consuming method for every public interface
* 2.4.3. **Do not** add methods to a released public Java interface
* 2.4.4. **Do not** use marker interfaces
* 2.4.5. **Do not** use public interfaces as a container for constant fields

**2.5. Enumerations**

* 2.5.1. **Consider** specifying a zero-value (“None” or “Unspecified”, etc) for enumeration types
* 2.5.2. **Avoid** enumeration types with only one value
* 2.5.3. **Do not** use enumeration types for open-ended sets of values
* 2.5.4. **Do not** reserve enumeration values for future use
* 2.5.5. **Do not** add new values to a released enumeration

**2.6. Exceptions**

* 2.6.1. **Ensure** that custom exceptions are serialized correctly
* 2.6.2. **Consider** defining a different exception class for each error type
* 2.6.3. **Consider** providing extra information for programmatic access
* 2.6.4. **Avoid** deep exception hierarchies
* 2.6.5. **Do not** derive custom exceptions from other than Exception and RuntimeException
* 2.6.6. **Do not** derive custom exceptions directly from Throwable
* 2.6.7. **Do not** include sensitive information in error messages

**2.7. Documentation**

* 2.7.1. **Provide** type overview for each type
* 2.7.2. **Follow** standard Javadoc conventions
* 2.7.3. **Begin** with a short, one sentence summary of the type
* 2.7.4. **Provide** enough details to help deciding if and how to use the type
* 2.7.5. **Explain** how to instantiate the type
* 2.7.6. **Provide** code sample to illustrate the main use case for the type
* 2.7.7. **Include** links to relevant sections in the Developer Guide
* 2.7.8. **Include** links to relevant sections in the Cookbook
* 2.7.9. **Indicate** related types
* 2.7.10. **Indicate** deprecated types using the @deprecated tag
* 2.7.11. **Document** class invariants
* 2.7.12. **Avoid** lengthy type overviews
* 2.7.13. **Do not** generate Javadoc for private fields and methods

**3. Method Design Checklist**

**3.1. General**

* 3.1.1. **Make** sure each method does only one thing
* 3.1.2. **Ensure** related methods are at the same level of granularity
* 3.1.3. **Ensure** no boilerplate code is needed to combine method calls
* 3.1.4. **Make** all method calls atomic
* 3.1.5. **Design** protected methods with the same care as public methods
* 3.1.6. **Limit** the number of mutator methods
* 3.1.7. **Design** mutators with strong invariants
* 3.1.8. **Favor** generic methods over a set of overloaded methods
* 3.1.9. **Consider** generic methods
* 3.1.10. **Consider** method pairs, where the effect of one is reversed by the other
* 3.1.11. **Avoid** “helper” methods
* 3.1.12. **Avoid** long-running methods
* 3.1.13. **Avoid** forcing callers to write loops for basic scenarios
* 3.1.14. **Avoid** “option” parameters to modify behavior
* 3.1.15. **Avoid** non-reentrant methods
* 3.1.16. **Do not** remove a released method
* 3.1.17. **Do not** deprecate a released method without providing a replacement
* 3.1.18. **Do not** change the signature of a released method
* 3.1.19. **Do not** change the observable behavior of a released method
* 3.1.20. **Do not** strengthen the precondition of an already released API method
* 3.1.21. **Do not** weaken the postcondition of an already released API method
* 3.1.22. **Do not** add new methods to released interfaces
* 3.1.23. **Do not** add a new overload to a released API

**3.2. Naming**

* 3.2.1. **Begin** names with powerful, expressive verbs
* 3.2.2. **Use** camelCasing
* 3.2.3. **Reserve** “get”, “set” and “is” for JavaBeans methods accessing local fields
* 3.2.4. **Use** words familiar to callers
* 3.2.5. **Stay** close to spoken English
* 3.2.6. **Avoid** abbreviations
* 3.2.7. **Avoid** generic verbs
* 3.2.8. **Avoid** synonyms
* 3.2.9. **Do not** use underscores
* 3.2.10. **Do not** rely on parameter names or types to clarify the meaning of the method

**3.3. Parameters**

* 3.3.1. **Choose** the most precise type for parameters
* 3.3.2. **Keep** the meaning of the null parameter value consistent across related method calls
* 3.3.3. **Use** consistent parameter names, types and ordering in related methods
* 3.3.4. **Place** output parameters after the input parameters in the parameter list
* 3.3.5. **Provide** overloaded methods with shorter parameter lists for frequently used default parameter values
* 3.3.6. **Use** overloaded methods for operations with the same semantics on unrelated types
* 3.3.7. **Favor** interfaces over concrete classes as parameters
* 3.3.8. **Favor** collections over arrays as parameters and return values
* 3.3.9. **Favor** generic collections over raw (untyped) collections
* 3.3.10. **Favor** enumeration types over Boolean or integer types
* 3.3.11. **Favor** putting single object parameters ahead of collection or array parameters
* 3.3.12. **Favor** putting custom type parameters ahead of standard Java type parameters
* 3.3.13. **Favor** putting object parameters ahead of value parameters
* 3.3.14. **Favor** interfaces over concrete classes as return types
* 3.3.15. **Favor** empty collections to null return values
* 3.3.16. **Favor** returning values which are valid input for related methods
* 3.3.17. **Consider** making defensive copies of mutable parameters
* 3.3.18. **Consider** storing weak object references internally
* 3.3.19. **Avoid** variable length parameter lists
* 3.3.20. **Avoid** long parameter lists (more than 3)
* 3.3.21. **Avoid** putting parameters of the same type next to each other
* 3.3.22. **Avoid** out or in-out method parameters
* 3.3.23. **Avoid** method overloading
* 3.3.24. **Avoid** parameter types exposing implementation details
* 3.3.25. **Avoid** Boolean parameters
* 3.3.26. **Avoid** returning null
* 3.3.27. **Avoid** return types defined in unrelated APIs, except core Java APIs
* 3.3.28. **Avoid** returning references to mutable internal objects
* 3.3.29. **Do not** use integer parameters for passing predefined constant values
* 3.3.30. **Do not** reserve parameters for future use
* 3.3.31. **Do not** change the parameter naming or ordering in overloaded methods

**3.4. Error handling**

* 3.4.1. **Throw** exception only for exceptional circumstances
* 3.4.2. **Throw** checked exceptions only for recoverable errors
* 3.4.3. **Throw** runtime exceptions to signal API usage mistakes
* 3.4.4. **Throw** exceptions at the appropriate level of abstraction
* 3.4.5. **Perform** runtime precondition checks
* 3.4.6. **Throw** NullPointerException to indicate a prohibited null parameter value
* 3.4.7. **Throw** IllegalArgumentException to indicate an incorrect parameter value other than null
* 3.4.8. **Throw** IllegalStateException to indicate a method call made in the wrong context
* 3.4.9. **Indicate** in the error message which parameter violated which precondition
* 3.4.10. **Ensure** failed method calls have no side effects
* 3.4.11. **Provide** runtime checks for prohibited API calls made inside callback methods
* 3.4.12. **Favor** standard Java exceptions over custom exceptions
* 3.4.13. **Favor** query methods over exceptions for predictable error conditions

**3.5. Overriding**

* 3.5.1. **Use** the @Override annotation
* 3.5.2. **Preserve** or weaken preconditions
* 3.5.3. **Preserve** or strengthen postconditions
* 3.5.4. **Preserve** or strengthen the invariant
* 3.5.5. **Do not** throw new types of runtime exceptions
* 3.5.6. **Do not** change the type (stateless, accessor or mutator) of the method

**3.6. Constructors**

* 3.6.1. **Minimize** the work done in constructors
* 3.6.2. **Set** the value of all properties to reasonable defaults
* 3.6.3. **Use** constructor parameters only as a shortcut for setting properties
* 3.6.4. **Validate** constructor parameters
* 3.6.5. **Name** constructor parameters the same as corresponding properties
* 3.6.6. **Follow** the guidelines for method overloading when providing multiple constructors
* 3.6.7. **Favor** constructors over static factory methods
* 3.6.8. **Consider** a no parameter default constructor
* 3.6.9. **Consider** a static factory method if you don’t always need a new instance
* 3.6.10. **Consider** a static factory method if you need to decide the precise type of object at runtime
* 3.6.11. **Consider** a static factory method if you need to access external resources
* 3.6.12. **Consider** a builder when faced with many constructor parameters
* 3.6.13. **Consider** private constructors to prevent direct class instantiation
* 3.6.14. **Avoid** creating unnecessary objects
* 3.6.15. **Avoid** finalizers
* 3.6.16. **Do not** throw exceptions from default (no-parameter) constructors
* 3.6.17. **Do not** add a constructor with parameters to a class released without explicit constructors

**3.7. Setters and getters**

* 3.7.1. **Start** the name of methods returning non-Boolean properties with “get”
* 3.7.2. **Start** the name of methods returning Boolean properties with “is”, “can” or similar
* 3.7.3. **Start** the name of methods updating local properties with “set”
* 3.7.4. **Validate** the parameter of setter methods
* 3.7.5. **Minimize** work done in getters and setters
* 3.7.6. **Consider** returning immutable collections from a getter
* 3.7.7. **Consider** implementing a collection interface instead of a public propertie of a collection type
* 3.7.8. **Consider** read-only properties
* 3.7.9. **Consider** making a defensive copy when setting properties of mutable types
* 3.7.10. **Consider** making a defensive copy when returning properties of mutable type
* 3.7.11. **Avoid** returning arrays from getters
* 3.7.12. **Avoid** validations which cannot be done with local knowledge
* 3.7.13. **Do not** throw exceptions from a getter
* 3.7.14. **Do not** design set-only properties (with public setter no public getter)
* 3.7.15. **Do not** rely on the order properties are set

**3.8. Callbacks**

* 3.8.1. **Design** with the strongest possible precondition
* 3.8.2. **Design** with the weakest possible postcondition
* 3.8.3. **Consider** passing a reference to the object initiating the callback as the first parameter of the callback method
* 3.8.4. **Avoid** callbacks with return values

**3.9. Documentation**

* 3.9.1. **Provide** Javadoc comments for each method
* 3.9.2. **Follow** standard Javadoc conventions
* 3.9.3. **Begin** with a short, one sentence summary of the method
* 3.9.4. **Indicate** related methods
* 3.9.5. **Indicate** deprecated methods using the @deprecated tag
* 3.9.6. **Indicate** a replacement for any deprecated methods
* 3.9.7. **Avoid** lengthy comments
* 3.9.8. **Document** common behavioral patterns
* 3.9.9. **Document** the precise meaning of a null parameter value (if permitted)
* 3.9.10. **Document** the type of the method (stateless, accessor or mutator)
* 3.9.11. **Document** method preconditions
* 3.9.12. **Document** the performance characteristics of the algorithm implemented
* 3.9.13. **Document** remote method calls
* 3.9.14. **Document** methods accessing out-of-process resources
* 3.9.15. **Document** which API calls are permitted inside callback methods
* 3.9.16. **Consider** unit tests for illustrating the behavior of the method