Cassandra Code Style

The Cassandra project follows Sun's Java coding conventions for anything not expressly outlined in this document.

Note that the project has a variety of styles that have accumulated in different subsystems. Where possible a balance should be struck between these guidelines and the style of the code that is being modified as part of a patch. Patches should also limit their scope to the minimum necessary for safely addressing the concerns of the patch.

Naming and Clear Semantics

Class, Method and Variable Naming

Avoid extraneous words, for example prefer `x()` over `getX()` or `setX()` where it makes semantic sense. At the same time, do not avoid using words that are necessary, for example if a descriptive word provides semantic context such as `liveReplicas` over `replicas`. This is essential when there are many conceptual instantiations for a variable that are not enforced by the type system, but be sure to be consistent in the word choice and order across all instantiations of the variable.

e.g. `allReplicas`, `naturalReplicas`, `pendingReplicas`, `allLiveReplicas`, etc.

Method and Variable Naming Consistency

Ensure consistency of naming within a method, and between methods. It may be that multiple names are appropriate for a concept, but these should not be mixed and matched within the project. If you modify a concept, or improve the naming of a concept, make ***all*** relevant - including existing - code consistent with the new terminology. If possible, correspond with a prior author before modifying their semantics.

Standard word meanings in method or property names

`calculateX, computeX`	Perform some potentially expensive work to produce x
`refreshX`	Recompute a memoized x
`lookupX`	Find x in a map, or other structure, that is efficient but not free
`x`	Return x, relatively cheaply
`toX`	Return a potentially expensive translation to x
`asX`	Return a cheap translation to x
`asXView`	Return a cheap translation to x, that will reflect changes in the source
`isX, hasX, canX`	Boolean property or method indicating a capability or logical state

For boolean variables, fields and methods, choose names that sound like predicates and cannot be confused with nouns.

Semantic Distinctions via the Type System

If possible, enforce semantic distinctions at compile time with the type system.

e.g. `RangesAtEndpoint`, `EndpointsForRange` and `EndpointsForToken` are all semantically different variants on a collection of replicas.

This makes the intent of the code clearer, and helps the compiler indicate where we may have unintentionally conflated concepts. They also provide opportunities to insert stronger runtime checks that our assumptions hold, and these constraints can provide further clarity when reading the code.

In the case of `EndpointsForX`, for instance, we enforce that we have no duplicate endpoints, and that all of the endpoints do fully cover X.

Enums for Boolean Properties

Prefer an `enum` to `boolean` properties and parameters, unless clarity will be harmed (e.g. helper methods that accept a computed boolean predicate result, of the same name as used in the method they assist). Try to balance name clashes that would affect static imports, against clear and simple names that represent the behavioural switch.

Semantic Distinctions via Member Variables

If a separate type for all concepts is too burdensome, a type that aggregates concepts together within member variables might be applicable.

The most obvious counter-example is *not to use `Pair`*, or a similar tuple. Unless it is extremely obvious, prefer a dedicated type with well named member variables.

For example, `FetchReplicas` for source and target replicas, and `ReplicaLayout` for the distinction between natural and pending replicas.

This may help authors notice other semantics they had overlooked, that might have led to subtly incorrect parameter provision to methods. Conversely, methods may choose to accept one of these encapsulating types, so that callers do not need to consider which member they should provide.

e.g. `ConsistencyLevel.assureSufficientLiveReplicas` requires very specific replica collections, that are quite distinct, that might be easily incorrectly provided (though this is still inadequate, as it needs to distinguish between live and non-live semantics, which remains to be improved)

Public APIs

These considerations are especially important for public APIs, including CQL, virtual tables, JMX, yaml, system properties, etc. Any planned additions must be carefully considered in the context of any existing APIs. Where possible the approach of any existing API should be followed. Where the existing API is poorly suited, a strategy should be developed to modify or replace the existing API with one that is more coherent in light of the changes - which should also carefully consider any planned or expected future changes to minimise churn. Any strategy for modifying APIs should be brought to dev@cassandra.apache.org for discussion.

Code Structure

Necessity

If an interface has only one implementation, remove it. If a method isn't used, delete it.

Don't implement `hashCode()`, `equals()`, `toString()` or other methods unless they provide immediate utility.

Specificity

Don't overgeneralise. Implement the most specific method or class that you can, that handles the present use cases.

Methods and classes should have a single clear purpose, and should avoid special-cases where practical.

Class Layout

Consider where your methods and inner classes live with respect to each other. Methods that are of a similar category should be adjacent, as should methods that are primarily dependent on each other. Try to use a consistent pattern, e.g. helper methods may occur either before or after the method that uses them, but not both; method signatures that cover different combinations of parameters should occur in a consistent order visiting the parameter space.

Class declaration order should, approximately, go: inner classes, static properties, instance properties, constructors (incl static factory methods), getters/setters, main functional/API methods, helper (incl static) methods and classes. Clarity should always come first, however.

Method Clarity

A method should be short. There is no hard size limit, but a filled screen is a good warning size. However, be careful not to over-minimise your methods; a page of tiny functions is also hard to read.

The body of a method should be limited to the main conceptual work being done. Substantive ancillary logic, such as computing an intermediate result, evaluating complex predicates, performing auditing, logging, etc, are prime candidates for helper methods.

Compiler Assistance

Always use `@Override` annotations when implementing abstract or interface methods or overriding a parent method.

`@Nullable`, `@ThreadSafe`, `@NotThreadSafe` and `@Immutable` should be used as appropriate to communicate to both the compiler and readers.

Boilerplate

Prefer `public final` fields to `private` fields with getters (but prefer encapsulating behavior in "real" methods to either).

Declare class properties `final` wherever possible, but never declare local variables and parameters `final`. Variables and parameters should still be treated as immutable wherever possible, with explicit code blocks introduced as necessary to minimize the scope of any mutable variables.

Prefer initialization in a constructor to setters, and builders where the constructor is complex with many optional parameters.

Avoid redundant `this` references to member fields or methods, except for consistency with other assignments e.g. in the constructor

Exception handling

Never ever write catch (…) {} or catch (…) { logger.error() } merely to satisfy Java's compile-time exception checking.

Always catch the narrowest exception type possible for achieving your goal. If Throwable must be caught for handling exceptional termination, it must be rethrown. If an exception cannot be safely handled locally, propagate it - but use unchecked exceptions if no caller expects to handle the case. Rethrow as `RuntimeException`, `IOError`, or your own `UncheckedXException`, or `AssertionError` if it "can't happen"

Only if an exception is an explicitly acceptable condition can it be ignored, but this must be explained carefully in a comment detailing how this is handled correctly.

Formatting

`{` and `}` are placed on a new line except when empty or opening a multi-line lambda expression. Braces may be elided to a depth of one if the condition or loop guards a single expression.

Lambda expressions accepting a single parameter should elide the braces that encapsulate the parameter. E.g. `x -> doSomething()` and `(x, y) -> doSomething()`

Multiline statements

Where possible prefer keeping a logical action to a single line. Prefer introducing additional variables, or well-named methods encapsulating actions, to multi-line statements - unless this harms clarity (e.g. in an already short method).

Try to keep lines under 120 characters, but use good judgment. It is better to exceed this limit, than to split a line that has no natural splitting points, particularly when the remainder of the line is boilerplate or easily inferred by the reader.

If a line wraps inside a method call, first extract any long parameter expressions to local variables before trying to group natural parameters together on a single line, aligning the start of parameters on each line, e.g.

```
Type newType = new Type(someValueWithLongName, someOtherRelatedValueWithLongName,
                        someUnrelatedValueWithLongName,
                        someDoublyUnrelatedValueWithLongName)
```

When splitting a ternary, use one line per clause, carry the operator, and where possible align the start of the ternary condition, e.g.

```
var = bar == null
```

```
? doFoo()
: doBar();
```

It is usually preferable to carry the operator for multiline expressions, with the exception of some multiline string literals.

Whitespace

Make sure to use 4 spaces instead of the tab character for all your indentation.
Many lines in the current files have a bunch of trailing whitespace. If you encounter incorrect whitespace, clean up in a separate patch. Current and future reviewers won't want to review whitespace diffs.

Static Imports

Consider using static imports for frequently used utility methods that are unambiguous. E.g. String.format, ByteBufferUtil.bytes, Iterables.filter/any/transform.

When naming static methods, select names that maintain semantic legibility when statically imported, and are unlikely to clash with other method names that may be mixed in the same context.

Imports

Observe the following order for your imports:
java
[blank line]
com.google.common
org.apache.commons
org.junit
org.slf4j
[blank line]
everything else alphabetically

Format files for IDEs

IntelliJ: intellij-codestyle.jar
IntelliJ 13: gist for IntelliJ 13 (this is a work in progress, still working on javadoc, ternary style, line continuations, etc)
Eclipse: (github.com/tjake/cassandra-style-eclipse)

Performance

Considerations primarily for frequently invoked code, but that should always be kept in mind

Streams, Lambdas, Optionals

Streams incur increased costs that are hard to quantify, and should be avoided for any frequently invoked code paths.

Lambdas should be used with care, and only when they greatly improve clarity - particularly those that capture variables or contextual state. If an object will anyway be created, prefer the use of a lambda for clarity.

Optionals should be avoided on common/frequent code paths, and generally unless clarity is greatly improved, i.e. when interacting with supporting libraries.

For Loops

Extract the upper bound of a loop except where you can be certain it will be efficiently hoisted by the compiler, e.g.

```
for (int i = 0, max = list.size(); i < max ; i++)
       doSomething(i);
```

External Dependencies

Dependencies to the project are sticky, and generally poorly audited. They expose the project to security flaws simply by being included, as well as a poorly-managed ongoing maintenance risk. They may also harm the coherency of the project's codebase when introducing alternative mechanisms to existing solutions in the project. It is best to seek wider input before committing the project to this course of action. New dependencies should not be included without community consensus first being obtained via a [DISCUSS] thread on the dev@cassandra.apache.org mailing list.