Genome Analysis Toolkit Coding Standards for Java

Genome Analysis Software Engineering

July 1, 2009

Contents

Chapter 1

Overview

1.1 Credit

The majority of text in this document is verbatim from the production informatics coding standard document, produced by Ted Sharpe and the production informatics team here at the Broad Institute. They deserve all the credit for the insights in this document, and if you feel so inclined please share statements of gratitude with them.

1.2 Summary

This document is an attempt to describe a brief and minimal set of standards for Java code for use by production informatics projects at the Broad. The goal of the standards is to allow programmers to be innovative and expressive, while allowing their peers a vague hope of maintaining and extending their output. This document describes a set of arbitrary standards for naming and documentation based on industry-standard practices, and a set of good practices guidelines intended to improve the maintainability, correctness, and reusability of code. Good citizenship demands that you suggest improvements if you feel that the document is not well crafted, rather than merely going your own way.

1.3 Cheat Sheet

Table 1.1: default

Type	Style	Example
Function names	uppercase words, with the first word lowercase	`convertToParser`
Package names	lowercase, with specific functional encapsulation for each class	`org.broadinstitute.sting.gatk`
Variable names	uppercase words, with the first word lowercase	`resetCounter`
Class names	uppercase words representing a noun	`TraversalEngine`
Interface names	the same as class names, no 'i' before the name	`GenomeEngine`
Tab length	4 spaces	

Chapter 2

Naming

Choosing names is among the more arduous tasks in programming. There is constant contention between creativityfinding the mot justeand following fashion so as to more easily allow names to be guessed; and a tug between being conciseif only to save typingand being verbose so as to provide greater explicitness. Expending the time necessary to create a really good name conflicts with the programmers appropriate desire to get on with the job and get something done. Unfortunately, naming has a great impact on the maintainability of the code, and despite its being a somewhat fussy and fusty topic, a few guidelines are appropriate. If the standards document is well crafted it will free the programmer from some of the arbitrary decision-making that, when inconsistent, detracts from the intelligibility of the code, while allowing the programmer to focus on the semantics of the name.

2.1 Package Naming

Package names should be all lower case, and should begin with:

`org.broadinstitute.sting`

To this add the product name (e.g., basecaller), and subdivide packages according to functionality below that. Put all classes in a package (to allow for more easily understood dependency trees, and to allow reuse via a CLASSPATH of reasonable length).

Listing 2.1: Good Package Names

```
package org.broadinstitute.sting.basecalller.basecallerEngine; // Good
package org.broadinstitute.sting.baseCallingStuff;              // Bad
package myBaseCaller;                                          // Unacceptable
```

2.1.1 Good practices

Try to organize packages so that the dependency tree isnt total spaghetti. Ideally, there should be a hierarchy among the packages so that, for example, the web support classes are in one package, and dont know anything about database access, and the database access classes are in another package, and dont know anything about web support. Something that knows about both (a package that supports servlet development, for example) should be in a separate package higher in the hierarchy. Think model-view-controller. Think Law of Demeter.

2.2 Interface Naming

Interfaces require no special naming convention to distinguish them from classes. (See the Class naming conventions below.) The presumption is that most of the types that you are passing around are, in fact, abstract types that dont lock you into a particular implementation: elaborate decoration of interface names is therefore unnecessary and undesirable. Distinguish the implementation class names instead (since they should be repeated far less often throughout the code than the names of the interfaces)

Listing 2.2: Good Interface Names

```
interface DatabaseBinder          // OK
interface ITagFactory          // not as good
```

Some people like to reserve able or ible names for interfaces (e.g., Cloneable, Comparable, or Runnable). This is especially useful for mix-in interfaces, where the interface describes a capability that can be shared among otherwise disparate types of objects. Not all interfaces fall into this pattern, however, and you neednt be concerned if your interface seems to want to have a noun as its name, rather than forcing it into the verb-able mold.

2.2.1 Good practices

Expending the time necessary to abstract common behavior from a set of related classes is probably the single most important thing you can do to improve the extensibility and adaptability of your code. Use interfaces to describe the common behavior so that you dont force your clients to use particular implementations. In all cases, the methods of the interface describe the complete repertoire of behaviors necessary to be an object of the type named by the interface. So check your name to see if it describes something possessing the interfaces set of behaviors, and check your methods to see if thats what something by that name should be able to do. Ruthlessly eliminate any inconsistencies.

2.3 Class Naming

Naming: Use mixed-case names, capitalizing the first character of each word. Avoid overly short, and overly common names, and overly long, verbose names. If there is doubt about whether the components of a name are separate words, use less capitalization rather than more. (E.g, Timezone, not TimeZone. Barcode, not BarCode.) You are aiming to capitalize each concept, not each morpheme. A primary interface name is a good prefix for a class name. A package name is not a good prefix. (For one thing, its redundant, and for another, it inhibits easy refactoring of packages that have come to have tangled dependencies.) Should you need to distinguish a common, or base-level implementation from its primary interface, you may add a suffix of your choosing. A suffix that describes the implementation in some way is preferable to -Impl or -Base or some generic suffix. (Note that the JDK classes dont respect this, or any other, consistent naming scheme. So it goes. Youll have to find another source of inspiration.) Dont abbreviate any of the words in a class name. Its just too hard to remember, and too likely to introduce inconsistency. You may, however, use very common acronyms as if they were a word. For example, StructuredQueryLanguageHelper would be quite ridiculousSQL is a common acronym, and the class should be called SqlHelper (note the lower case ql). Class names must be nouns.

Listing 2.3: Good Class Names

```
class FormModule          // OK
class SOExecutor          // not very good   obscure abbreviations
class State               // not good   too vague, ambiguous, and common
```

2.3.1 Good practices

A class should model a clear concept. If you cant explain the concept behind a class in a sentence or two, there is probably something wrong. (And if your sentence or two seems to require extensive use of the word or you can be certain that you have a problem.)

2.4 Field and Variable Naming

Use mixed case names, capitalizing the first character of each word except the first. Constants may be all upper case, using underscores to separate words. Idiosyncratic abbreviations are acceptable for variable and field names, because their scope is very limited. (This is because you wont be using any public fields: see below.) Public constants should follow the semantic rules already specified for classes: not too short, not too long, no cryptic abbreviations. Remember that your clients will qualify the name of the constant with the class name, so overly common words are not so much of a problem as for class names.

2.4.1 Good practices

No non-final public fields. Ever. Period. You know why. (But just to be tediously explicit, well go on a bit about it.) Just when you think youve got a totally passive data-bearing object that might just as well be a C-language struct as a Java class, and you reckon youll just make its fields public, youll figure out that it needs behavior. It needs to be persistent, and has to keep track of whether its been dirtied or not. Or it needs a new field that is dependent in some way on the other fields. Something will inevitably crop up that will be completely impossible to implement if anyone from anywhere can modify the properties at any time. Writing accessors for the properties of passive objects is enormously time-consuming. But trying to track down all clients usage of the public fields of an object you need to modify is also enormously time-consuming (and terrifying). Maybe you could write a macro to save some of the typing. Sacks of data are so boringobjects long to have behavior. Note that public members of private, nested classes arent really public at all, and so theyre perfectly fine. (This allows you to create struct-like objects for internal use within a class.) Be careful about static objects (whether public or private, final or not): they get created when the class is loaded and there is no easy way to handle complex dependencies among them that might require some particular loading order. Its best to keep them very simplefor example, creating tiny immutable static objects to simulate C++-style enumerations is completely appropriate. Experience has shown that distinguishing the names of instance fields, class (i.e., static) fields, and constants from each other and from temporaries (i.e., stack frame variables, whether declared in a local block or passed as arguments) is enormously helpful in quickly comprehending a method: Its good to know at a glance how the code is affecting the state of the object (instance fields) or of all the objects in the class (class fields) without having to study the entire class to learn the names of its member fields. The easiest way to do this is to use a single letter prefix to designate the scope of the field. You may choose any consistent scheme. One common standard is to use the prefix g for static members, m for instance members, all capital letters for constants, and nothing for temporaries. If this has too much of a Microsoft taint for your taste, you could use g for static members, f for instance members, k for constants, and nothing for temporaries. This comes from the folks at Netscape. (Therefore, one or the other must be politically acceptable.) If you use such a scope prefix, this counts as the first, lower-case word. Note also that the use of such a prefix for members pretties-up initialization: youll never be forced to use this in your constructor to disambiguate an argument from a member, since arguments and members cannot have the same name. Overly short variable names, especially one-character names, and names that are common words, or parts of common words, are a problem for programmers who use editors with a limited understanding of Javas syntax. Using iii as a for-loop index instead of i doesnt really take much extra time, and it makes it a great deal easier to find all the places where the variable is used.

Listing 2.4: Good Variable Names

```
private static Section gHeader; // class member
```

```
private URL mMyURL;                 // instance member
String wfQName;                     // local    ugly, but who cares?
for ( int i = 0; i < k; ++i )       // Please, dont.
int iii;                            // Thank you.
```

2.5 Method Naming

Use mixed case names, capitalizing the first character of each word except for the first. The first, uncapitalized word should be a verb. Uncomplicated, local fetchers and modifiers of independent state ought to use get and set as their first word. Getters for boolean values can use is, has, or can instead of get. (Conversely, a method that causes the lights to dim over a several square block area should not be called getFoo. Merely to name it fetchFoo instead appropriately hints that something more complicated might be going on. And, of course, the methods documentation will make all this explicit!) Its especially important that you dont use obscure abbreviations or leave the vowels out of words in method names: theyre too hard to remember. Common acronyms, treated as if they were words, are fine. Its quite helpful if the name of the method is pronounceable, and spelled as its pronounced.

2.5.1 Good practices

Methods that have more lines than can fit on the screen all at once are much harder to understand than those that dont. Ditto for methods that have numerous lines that must be wrapped due to their length. Some suggest that each method can have at most one looping control structure. That doesnt always work in practice, but you get the gist. Following strict structured programming rules is usually an aid to comprehension. (This might not be true if doing so requires you to use a complicated set of flags to control the flow.) If you feel that a continue, labelled break, or early return is less confusing than the structured alternatives, provide a comment to call attention to the easily overlooked, one- or two-token statement that breaks the rules of structured programming.

Listing 2.5: Good Method Names

```
while ( itr.hasNext() )
{
    if ( "foo".equals((String)itr.next()) )
    {
        return; // HEY!  Theres an early return, here.
    }
}
```

A method should do one job. And, as stated earlier for classes, you ought to be able to describe that job in a crisp sentence or two. After youve done so, type it in as javadoc. Take switches and long series of if / else if blocks as a warning sign that polymorphism might have served you better. Methods with more than a very few parameters are difficult to use. If you have a method with more than three or four parameters you might consider passing an object instead. If you have more than seven, you definitely need to do something else. There is a delicate balance to strike between sanity-checking every argument passed to every method and never testing your inputs at all. Too much checking chews into performance, too little means the code will not be robust. Here are two criteria for achieving a happy medium: Arguments that are not used directly by the method, but simply passed through to lower layers need not be checked. Arguments that are used directly by the method ought to be checked, especially if the consequence of not checking them will be something completely uninformative like a NullPointerException. You cannot assume that it will be easy to produce a stack trace to home in on what code is producing the exception, and so the exceptions message must serve that purpose. The second criterion is to ask whether the consequences of passing bad arguments will be felt immediately or whether doing so will deploy a delayed-action time bomb. For example, passing

a null object that gets saved as a part of the objects state (even if its not used directly within the method that received it) may put a nave object into an unstable state that may not reveal itself until much later in the programs execution. This can be the very devil to debug. Your goal is to make certain that each method that affects an objects state causes the object to make a transition from one valid state to another valid state. You must do enough checking of arguments to insure that this is so. Consider a validate() method that checks your state against your design criteria for valid object state. (Or an assertion of a validate() method.) Youll typically need to put it in a finally clause to make sure it gets called, which makes it expensive. So you cant blindly use this technique everywhere. (The programming-by-contract idea formalizes thinking about what constitutes valid object stateits a useful concept.)

Chapter 3

Layout

3.1 File Template

Begin each file with the following bit of rote material. Just cut and paste it into your IDE so that it becomes the first few lines of every source file:

Listing 3.1: A Good Code Header

```
/*
 * $Id$
 * BROAD INSTITUTE SOFTWARE COPYRIGHT NOTICE AND AGREEMENT
 * Software and documentation are copyright 2005 by the Broad Institute.
 * All rights are reserved.
 *
 * Users acknowledge that this software is supplied without any warranty or support.
 * The Broad Institute is not responsible for its use, misuse, or
 * functionality.
 */
```

3.2 Documentation

All classes must have the following, minimum level of javadoc:

Listing 3.2: Javadoc Class Style

```
/**
 * One crisp, informative sentence or noun phrase that explains
 * the concept modeled by the class.
 *
 * This class is [<em>not</em>] thread safe [because it is immutable].
 *
 * @author  I. M. Coder
 * @version $Revision$
 */
class CrispConcept
{
    public static final String ID = "$Id$";
```

You are encouraged to provide as much explanatory material as you feel is helpful following that first, summary sentence. Information on algorithms and other information that will help a client make appropriate use of the class is particularly welcome. (But see Stateful Interfaces, and Rules for Use, below.) Tell us

whether your class is thread-safe or not. Thread safety due to immutability is particularly well worth mentioning. (It may warn a maintenance programmer off adding the set methods that you apparently forgot to provide.) All methods must have the following, minimum level of javadoc:

Listing 3.3: Good Method Javadoc

```
/**
 * One crisp, informative sentence or noun phrase that explains
 * what the method does.
 *
 * @param   parm1   Parm1 selects the widget to be frobnicated.  Cannot be null.
 * @param   parm2   Parm2 specifies the type of frobnication to apply.
 * @return  The frobnicated widget.
 * @throws  FrobnicationException Thrown if widget isn t frobnicable.
 */
```

For each parameter of reference type, tell us whether the reference may be null.

3.3 Imports

There is a balance between trying to maintain lengthy lists of classes imported one-by-one on the one hand, and importing many packages wholesale using an asterisk on the other. It shouldnt often be a big problem: there are exceptions, but a class that makes use of dozens of other classes may be trying to tell you that it needs some redesign. The suggestion is to import classes explicitly from packages we have writtenespecially those under active development. Also import explicitly when using just a few classes from a given package. Try to restrict whole-package imports to well tested, slowly changing third party packages. (Packages in the JDK, for example, are reasonable to import as a whole.)

3.4 Order of Class Members

Classes should be laid out consistently. You may put the member fields at either the bottom or the top, but you must not sprinkle them throughout. There is an argument that a class ought to be ordered with its public constants and constructors at the very top, its public methods next, and its internal stuff last, since that concentrates at the very top what a programmer needs to know to make use of the class. This isnt obligatory, but you may wish to give it some consideration. Nested classes go at the very bottom, after everything else.

3.5 Make em Pretty

Use four space tabs. Dont omit braces around single statements. Line things up so that its clear what things are on the same level. Give us enough white space to make it pretty.

Chapter 4

Design Considerations

If youre doing things right (and the DoD doesnt) design isnt a distinct phase that ends when coding begins. Ideally, youll develop a comprehensive, top-down design before you begin coding. This may take more than one napkin. Even so, youll face many decisions about implementation details that are not completely specified by the overall design. In other words, its inevitable that youll be doing design while you code. What follows are some coding standards for you to consider when doing this implementation-phase design.

4.1 Encapsulation

With very rare exception, all fieldsboth instance members and static membersshould be private. Protected and default (i.e., package-scoped) access is like public access, only less so. (Making a change in such a field still requires you to locate and analyze use of the field in indefinitely many files: for public fields you need to look everywhere, for protected fields you need only scan everything that extends you, and for default-access fields, you need only scan everything in the same package. This is arduous for those poor souls who must try to maintain your code.) Always use the most restrictive permission consistent with the design of your class. Dont make all of your internal methods protected in the vain hope that someday, some extending class might need to tweak your internal state, and youll make it easy. Similarly, dont provide a getter and setter for every piece of your internal state: the goal is to meet the contract of the interfaces you implement, and to hide the details of how you do it. Even the most legitimately passive of objectsthe model objects youve just hauled out of the relational databasewill likely have private, internal state that should not be exposed directly to clients. (Check out all the hidden state in EJB entity beans, for example.)

4.2 Is-A versus Has-A

A subclass and its superclass have an Is-A (or specialization-generalization) relationship. A class and a component of that class have a Has-A (or containment) relationship. If you have a crisp, clear idea of what kind of things two classes represent, then simply saying to yourself Thing A is a (special kind of a) Thing B, and Thing A has a Thing B (as one of its parts), will often make it clear what the relationship should be: one of the two sentences may sound very odd. A Dog is an Animal. A Car has a Steering Wheel. So Dog extends (or implements) Animal. (And not vice versa.) A Car has a Steering Wheel as one of its members. Sometimes people try to save programming (or computer) time by adding properties to a Steering Wheel to try to turn it into a Car. This is a very bad idea: think of the trouble youll have changing the Cars steering wheel for a nice padded-leather model if the Car is the Steering Wheel. Think of the trouble that youll have comparing Steering Wheels if some of them are Cars.

4.3 Redundant data

If you have only one copy of a given datum, it will be either right or wrong, but it wont be inconsistent with other data. Guaranteeing consistency is a great deal more complex than guaranteeing accuracy. Checking for and maintaining consistency among multiple copies of a datum often robs you of the efficiency you hoped to gain by the denormalization; not checking for and not maintaining consistency is a very frequent source of hard-to-fix bugs, and weird, unreliable program behavior.

4.4 Exceptions

Exceptions handle, well, exceptional conditions. Properly used, they provide a last-gasp attempt to allow a robust program to clean-up and recover from catastrophic situations. Exceptions should not be thrown frequently, certainly not as a part of the normal, expected flow of a program. Do not use them as a nifty hack for implementing non-local transfer of control. (Exceptions are far more expensive than normal returns, so the performance wizards wont be tempted to do this, anyway. For those who care more about well-designed, maintainable code than about saving clicks, youll realize that code that relies on exceptions for normal flow is just too difficult to comprehend and debug.) RuntimeExceptions are for even more rare, more catastrophic situations from which recovery is unlikely, at best. In code which will be called by general clients outside your package, catch and re-throw exceptions from the lower layers of code on which you depend to give a more package-oriented explanation of the bad thing that happened. However, preserve information when you do this: Wrap the original exception in a new exception that supplements rather than replaces the original message. And respond to all flavors of printStackTrace with the nested exceptions stack trace (i.e., delegate these methods to the nested exception). Never create exceptions with null messages.

4.5 Recovering External Resources

Java frees you from having to worry about memory as a resource. (Well, it reduces the worry, anyway.) Therefore you should have oodles of time left over to make certain that you free other resources when youre done with them. Two key external resources that you must make certain to release are open streams (which chew up a precious operating system file handle), and database connections (which chew up precious DBMS memory). The only really reliable way to make certain that these resources get freed is to create and use them within a single try block, and to release them in the finally clause. Try to avoid designs that require a class to maintain an open stream. One technique is to use an event-driven model to turn the file processing upside down: you can still have a nicely modular and reusable class while processing the file within the scope of a single block by using Listeners.

4.6 Stateful Interfaces and Rules for Use

Good interfaces are concise, comprehensive, and orthogonal. Concise means that there are no superfluous operations that dont seem to fit the underlying abstraction, and that there is one good way of accomplishing a given end, not a variety of ways from which you must choose. Comprehensive means that everything you might need to do in manipulating the object has been provided for. And orthogonal means that each method does something independent, and that any method can be called at any time. This is very difficult to achieve, but is an ideal toward which we must strive. Its hard work. Poor interfaces are cluttered with Rules for Use. If you are lucky, these rules are made explicit in documentation: Be sure to call this method before calling that one, but never call this method if youve ever called that one, and do, please, remember to call this one when youre all done. Needless to say, these interfaces are very difficult for clients to use correctly. One particularly common form of this blight is the Stateful Interface: the object has a lifecycle, and certain methods are appropriate only when the object is in some particular phase of the lifecycle. For a simple example, read the javadoc for the java.sql.CallableStatement class. It describes how you must

call registerOutputParameter before calling execute, how, for maximum portability, you should not call getMoreResults if you have called any of the getOutputParameter methods, and how you should remember to call close when youre all done. (And you thought we were exaggerating!) If you cant seem to work out how to avoid Rules for Use on your interfaces, you must at the very least make sure that each method call detects misuse, and throws an appropriate exception or does something other than trash your internal state. Every public method, if it affects object state at all, must transform the object from one valid, consistent state to another valid, consistent state. If you need some ego-incentive to motivate you, consider this: Hard-to-use, hard-to-understand, hard-to-maintain code is quickly replaced after you cease to maintain it. What kind of legacy is that? Sometimes you can see several distinct patterns of use among the clients of an interface: they might be telling you that you need to re-factor the interface into two separate interfaces.

4.7 Multithreading

Making your implementations thread-safe is a enormously complex issue. Unfortunately, almost all of us are doing some work in multi-threaded environments (writing servlets, for example), and it's an issue that we are forced to confront. If you havent examined the issue for one of the classes youve implemented (either because you dont anticipate its being used in a multi-threaded environment, or because the whole thing makes your brain ache), please provide a javadoc comment for the class indicating that it is not thread-safe. If youre not sure, its not safe! Dont just synchronize every method. Synchronization is far too expensive to use carelessly. (Less so than it used to be, but still expensive. And it doesnt resolve all multi-threading issues, anyway.) One way of beginning to address the issue of thread-safety is to understand what doesnt need any special thread-safety code, and try to produce as much of that as you can. Here are a couple of quick tips. Objects that can be seen only by a single thread are immune from the issue: If the only references to an object are from local variablesthat is, if a reference to the object is never stored in an instance or static fieldit will be visible only from the thread that creates it. Immutable objects are automatically thread-safe. If you cant change it, you cant see it in an inconsistent, intermediate state.

4.8 Canonical methods

Most simple classes either do or are. In the EJB environment session beans are the do classes, and entity beans are the are classes. In the model-view-controller paradigm, model objects are the are objects, controllers are the do objects, and views are mostly are, but typically also have a little bit of do flavor. So, you see, it does depend on what your definition of is is. The point of the distinction is that most of the are classes, those often immutable little bags of independent state, usually need to override equals and hashCode to behave properly. You must implement these basic object operations in each of your passive, data-bearing classes; you may wish (or need) to implement them in the others. The ares are typically more useful when Comparableimplementing that interface allows you to put them into sorted Collectionsso you ought to consider that next. Being Cloneable and Serializable usually come for free (no code to write), so throw those into the mix, too, unless theres a compelling reason not to. An example of a reason not to might be that you need to maintain uniqueness at a level of abstraction higher than object identityyou dont want to allow clients to make copies.

4.9 Performance

Your overall designyour selection of algorithms and data structures, for examplehas a far greater impact on performance than any little hacks you can apply while implementing the code. So design for performance, and implement for clarity. Nonetheless, the java compiler that most of us use can use a little help in doing optimization. Dont expect order-of-magnitude performance gainsyoull get those by designing away I/O, replacing searches with hashes, etc.these are percentage point tweaks. Move invariant code out of loops. For

example, many for loops can calculate their terminating condition once, before the loop starts, rather than at each iteration through the loop.

Listing 4.1: Improvements to loops

```
for ( int iii = 0; iii < str.length(); ++iii )  // bad

int nnn = str.length();
for ( int iii = 0; iii < nnn; ++iii )        // good
```

Strength reduction: do a simple calculation to update the value of some variable using the value it has from a previous trip through a loop, rather than from scratch each time.

Listing 4.2: Good looping practices

```
for ( int iii = 0; iii < nnn; ++iii )        // bad
{
    double val = pow( 2., iii );
    . . .
}

double val = 1.;
for ( int iii = 0; iii < nnn; ++iii )        // good
{
    . . .
    val *= 2.;
}
```

Avoid some performance dogs in the SDK: Use the underlying Stream classes rather than Readers when appropriate. Use the newer, non-synchronized collection classes rather than Vector and Hashtable. Penalties Code that fails to follow these guidelines will be posted around the MIT campus, along with the authors email address and an urgent request for comments.