# Introduction

The most important thing to achieve through the creation and use of a style guide is the ability to help various members of a team perform more consistently. It is NOT a goal of this guide to create an ideal style representation, but a mutually acceptable and sufficiently expressive set of guidelines. Both new and existing developers can be benefited by knowing, referencing, and potentially maintaining this guide.

# Some Helpful Reading

1. [Google Java Style](https://google-styleguide.googlecode.com/svn/trunk/javaguide.html)
2. [Effective Java](http://www.amazon.com/Effective-Java-Edition-Joshua-Bloch/dp/0321356683)
3. [Double-Checked Locking](http://www.cs.umd.edu/~pugh/java/memoryModel/DoubleCheckedLocking.html)
4. [Java Concurrency](http://www.amazon.com/Java-Concurrency-Practice-Brian-Goetz/dp/0321349601)

# General Coding Guidelines

## Documentation

1. All attributes and methods, whose names are not **completely** self-explanatory, should be commented using Javadoc formatted comments.
2. Do not leave automatically-created or half written documentation comments in code.
3. Comments must be helpful to readers of the code and should **not**, with very few exceptions, reference project names or defect identifiers.
4. Referencing external URLs is ok as supplemental documentation for very complex logic, but comments should also stand alone, while being succinct.
5. When submitting code reviews for partially-completed features, use "todo" comments to indicate work to be done.

// TODO (james.gosling): Refractor

// TODO (joshua.bloch): Implement

* 1. Should be descriptive enough to code reviewers so they know what can be ignored – and hasn't been unintentionally left out.

## Repetition

1. Do **not** copy and paste code!
2. If you need to reuse a piece of code that isn't directly usable, first consider refactoring.
3. If refactoring isn't immediately possible due to risk or time constraint issues, raise a task radar and then duplicate the code as necessary – talk to the component owner about this first.
4. Do **not** call the same methods or use the same accessors repeatedly.

**Avoid**

subTotal = viewPnrResponse.getPnr().getTotals().getSubTotal();

discount = viewPnrResponse.getPnr().getTotals().getDiscount();

fee = viewPnrResponse.getPnr().getTotals().getFee();

tax = viewPnrResponse.getPnr().getTotals().getTax();

total = viewPnrResponse.getPnr().getTotals().getTotal();

**Preferred**

totals = viewPnrResponse.getPnr().getTotals();

subTotal = totals.getSubTotal();

fee = totals.getFee();

tax = totals.getTax();

total = totals.getTotal();

## Methods

1. Methods should be kept short (max 15 lines is a good rough guideline).
2. Methods should always do what they say they do, nothing more or less.
3. Long methods should be refactored.

## Nested Method Calls

Very nested method calls are often difficult to read and to figure out what they're doing. They're also hard to debug because you don't get convenient access to the results of subexpressions.

* Use properly formatted and wrapped code.
* Breakup complicated nested calls by assigning subexpressions to local variables.
* Refactor into separate methods when appropriate to enhance readability.

**Avoid**

BigDecimal discountAmount = pnr

.getTotals()

.getSubTotal()

.multiply(

(pnr.getTotals().getDiscountPercentage()

.divide(BigDecimal.valueOf(100.00))))

.setScale(2, BigDecimal.ROUND\_HALF\_UP);

**Preferred**

BigDecimal subTotal = pnr.getTotals().getSubTotal();

BigDecimal discountPercentage = pnr.getTotals().getDiscountPercentage();

BigDecimal discountMultiplier = discountPercentage.divide(BigDecimal.valueOf(100.00));

BigDecimal discountedSubTotal = subTotal.multiply(discountMultiplier);

discountedSubTotal.setScale(2, BigDecimal.ROUND\_HALF\_UP);

## Encapsulation and Abstraction

### Literals vs. Constants

A literal is a value that, when typed, literally describes its own value. For example: 5 (a literal int), 3.14 (a literal double), "hello" (a literal string).

* Literal strings should almost never be used outside of defining constants.

**Preferred**

serviceName.equals(TAXATION)

**Avoid**

serviceName.equals("tax")

* Literal numeric values, other than 0 and 1, should never be used outside of defining constants.

**Preferred**

paxArray.length > 0

paxArray.length >= MIN\_GROUP\_STRENGTH

**Avoid**

paxArray.length > 10

### Exceptions and Exception Handling

* In a multi-level architecture, only wrap or log exceptions when it makes sense – directly before returning to an external caller, when fatally erroring out, or when an important exception case is recoverable.
* Programming errors, in almost all cases, should not throw Exceptions! They should error out and cause Internal Server Errors – using "finally" blocks as appropriate to cleanup any critical tasks.
* Use built-in exception types when possible.
  + IllegalArgumentException
* Thrown exception types are part of the interface, treat them as such.
  + For example, a method that does **not** obviously involve File I/O, for example, should not declare that it throws an IOException.
  + Conversely, a method that **does** obviously involve File I/O, for example, probably should declare that it throws an IOException and should **not** wrap such an exception or log it.
  + Throw Errors for things that should be unrecoverable by the core logic. Error types should **not** be declared in method prototypes.
* Exceptions should be thrown for **exceptional** circumstances.
  + Anti-example: in "indexOf", it is not an exception if the specified substring isn't found. That's a normal, expected scenario.
  + Example: if null is passed to indexOf, it is an exception, because passing null as a substring value is non-sensical).
    - It is however, **not** an error, because it is impossible to tell, at the point of calling indexOf, whether the specified substring value is an input value or a programmatic value.

## Logging

* Avoid descriptive prose texts in log – it severs no purpose except bloating the log file

**Preferred**

logger.info(“Flight Delay: {}”, fltNum)

**Avoid**

logger.info("A flight delay event from FCM has been captured within FCMAnalyzer bolt");

* Exception trace should be logged utilizing the log method that accept Throwable. This dumps the stack trace as well into log file
* Production systems should run at info level. This means that in the code:
  + any information worth notifying in production should be logged at info level
    - E.g. logger.info(“{} reloaded”, airlineConfigCache);
  + debugging messages should not be at logged at info level
  + stack trace for business exceptions excepted in normal operations (e.g. PNRNotFound) should be logged at debug/trace/fine levels (instead of info/warn/error levels) – so that the log file is not bloated with unnecessary stack traces
* When calling a third-party/external system, always log response time (at a level visible in production).
* Use MDC or [ThreadContext](https://logging.apache.org/log4j/2.x/manual/thread-context.html) to log the context info identifying log messages pertaining to one transaction/flow. Context info can be
  + Request ID
  + Client IP
  + User ID
* Splunk is used for aggregation, searching and analysis of all log files. Follow [best practices](http://dev.splunk.com/view/logging/SP-CAAAFCK) for logs targeted on splunk
  + Use clear key-value pairs
  + Use timestamps for every event
  + Use unique identifiers (IDs)
  + Use structured logging - e.g. in JSON format
  + Keep multi-line events to a minimum

# Guideline for Microservices

## Directory Structure

(replace smartrez-pnl with your component's name)

* smartrez-pnl
  + smartrez-pnl/deploy/log4j.xml *– should not be included with the build*
  + smartrez-pnl/smartrez-pnl-client *– service proxy – communicating with server*
    - smartrez-pnl/smartrez-pnl-client/release-notes.txt
  + smartreoodz-pnl/smartrez-pnl-common *– code used by both client and server*
    - smartrez-pnl/smartrez-pnl-common/release-notes.txt
  + smartrez-pnl/smartrez-pnl-server *– services*
    - smartrez-pnl/smartrez-pnl-server/release-notes.txt
    - smartrez-pnl/smartrez-pnl-server/resources/config/database *– schemas, scripts, documents related to Oracle db and migration*

## Versioning

* All version numbers should be in the style: MAJOR . FEATURE . BUG\_FIX
* [CHANGELOG.md](http://keepachangelog.com/en/1.0.0/) (or Release Notes) file should be maintained listing all the changes, defect fixes and enhancements in each release.
* Releases with the same major version should have public APIs that are backwards compatible with releases of the same major version.
* Any time a non-backwards compatible change is introduced, the MAJOR revision number must be incremented.
* The FEATURE number should be incremented for new APIs, major refactoring, or other types of more substantial changes.
* The BUG\_FIX number should only be incremented when fixing existing functionality.

## Component URIs

* URIs, when broken into parts and read from left to right, should be least to most specific and should follow REST guidelines.
* See [How to Create REST URLs Without Verbs](http://stackoverflow.com/questions/1619152/how-to-create-rest-urls-without-verbs/1619677#1619677)

**Avoid**

http://1.2.3.4:8080/configurator/getConfiguration?configGroup=pnl

**Preferred**

http://1.2.3.4:8080/configurator/pnl

# Additional Style Guidelines

## Naming, Spelling, and Correctness

* Package name should be in lower case and begin with com.flydubai.<project>

**Preferred**

com.flydubai.smartrez

**Avoid**

com.fz.smartrez

com.flydubai.SmartRez

* Do **not** abbreviate words, except for: "max", "min", and "num".
  + Don't make up acronyms. For example, if a class is called NumberFormatter, **nf** is not a good instance variable name. Better examples are: numberFormatter (ok), currencyFormatter (very good), percentFormatter (very good).
* Use descriptive, accurate names for classes, variables, and methods.
  + If a method does more or less than is indicated by the method name, reconsider it.
* Always use correct, American-English spelling for class, variable, and method names.
* Use Java-standard camel casing for class, variable, and method names.
* Names should be precise and technically correct, using the **most common** naming for items that can be named multiple ways.
* Don't use leading underscores in variable or method names.

## Method Order

* The order of sections should be as following, skipping irrelevant ones:
  + Public Constants
  + Public Fields
  + Protected Constants
  + Protected Fields
  + Private Constants
  + Private Fields
  + Initialization
  + Subclass Overrides (potentially multiple sections, organized by class name from least to most specific)
    - Section names should be: <NameOfSuperclass> Overrides, e.g. "Object Overrides"
  + Interface Method Implementations (potentially multiple sections, organized by interface name in alphabetical order)
    - Section names should be: <NameOfInterface> Methods, e.g. "DragSourceListener Methods"
  + Public Methods
    - Optionally create more explicit organization by naming distinct public methods sections, e.g. "Public Printing Methods", "Public Status Methods" (in alphabetical order, allowing the generic "Public Methods" to be used at the tops of these sections as a catch-all).
  + Protected Methods
    - See "Public Methods" for creating more explicit sections.
  + Private Methods
    - See "Public Methods" for creating more explicit sections.
* All attributes and methods within sections should be alphabetized.
  + When overloading methods, use the names of the types (rather than the parameter names) to alphabetize.

## Spacing, Wrapping, and Alignment

* Limit code to 100 columns. The only exceptions are line comments describing code on the same line.
* When wrapping long lines, try to keep more-related things together.
* When wrapping, try to align parameters in a manner indicating things at the same level.

thisIsAMethodName(thisIsTheFirstParameter,

thisIsTheSecondParameterAndANestedMethodCall(parameterOne,

parameterTwo),

thisIsTheThirdParameter);

* If a wrapped line will be visually confusing (e.g. aligned with a block of code), over-indent by 1 extra tab.

**Avoid**

if (x == 5 || y == thisIsAReallyLongVariableName

|| z == 10) {

System.out.println("x is 5 or y is 10");

}

**Preferred**

if (x == 5 || y == thisIsAReallyLongVariableName

|| z == 10) {

System.out.println("x is 5 or y is 10");

}

* Use 2 newlines between method definitions and other major sections (between package and import, between import and class definition, and between class definitions) outside of methods (never use 2 newlines as separators within a method).
* Use 1 newline between code "paragraphs", but keep in mind that methods with many paragraphs should usually be broken down.
* Do **not** use extraneous newlines, e.g. at the beginning or end of a method or block.
* All binary operators should have spaces around the operators.

y == x + 5

* For function call, there should **not** be a space between the method name and open parenthesis.

thisIsAMethodName(x, y, z)

* For keywords that use arguments (e.g. for loops, if statements, synchronized), there **should** be a space between the keyword and open parenthesis.

if (x == 5) {

* Do **not** try to align common structures across statements.

**Preferred**

String x = 5;

String anotherVariable = 5;

LinkedList myLinkedList = new LinkedList();

**Avoid**

String x = 5;

String anotherVariable = 5;

LinkedList myLinkedList = new LinkedList();

## Imports

* Use the Eclipse feature "Organize Imports".

## Loops, Branches, and Curly Braces

* An open curly brace should be preceded by a single space and be placed on the same line as the conditional or keyword it belongs to.

while (x == 5) {

* There should not be extraneous newlines at the start or end of a block.

**Avoid**

void thisIsAMethod(String appendString) {

System.out.println("Hello World " + appendString);

}

**Preferred**

void thisIsAMethod(String appendString) {

System.out.println("Hello World " + appendString);

}

* All branching statements and loops, even those with single contained statements, must use curly braces.

**Avoid**

if (x == 5) System.out.println("Hello World");

**Preferred**

if (x == 5) {

System.out.println("Hello World");

}

* "else if" and "else" branches should be continued on the same lines as closing curly braces, like "} else if (...) {".
* Use "else" branches sparingly.
  + If there are, for example, 3 known possibilities, don't test for 2 and assume the 3rd is the "else" case. Test for all 3 and use else to indicate an unsupported state – e.g. return an error code or throw an exception or error.
* Comments related to an entire branching structure should go before the if statement.
* Comments related to a single branch should be placed inside the branch, indented with the contained code, and followed by an extra newline.

if (isAdult) {

paxType = ADULT

} else if (isChild) {

paxType = CHILD

} else if (isInfant) {

paxType = INFANT

} else {

// Unsupported use case

throw new Error("Expected ADT, CHD or INF");

}

# Programming Best Practices

## Use BigDecimal

* When doing financial calculations, always use BigDecimal.
* When converting a double to a BigDecimal, **always use the BigDecimal(String s)** constructor. Never use the BigDecimal(double d) constructor as it has unpredictable behavior. More details: <http://download.oracle.com/javase/6/docs/api/java/math/BigDecimal.html#BigDecimal(double)>.

**Avoid**

double myDoubleValue = 0.1;

BigDecimal incorrect = new BigDecimal(myDoubleValue);

System.out.println("Incorrect BigDecimal: " + incorrect);

**Preferred**

double myDoubleValue = 0.1;

BigDecimal correct =

new BigDecimal(String.valueOf(myDoubleValue));

System.out.println("Correct BigDecimal: " + correct);

* Use movePointLeft and movePointRight instead of dividing and multiplying multiples of 10.

**Avoid**

BigDecimal discountMultiplier

= discountPercentage.divide(BigDecimal.valueOf(100.00));

**Preferred**

BigDecimal discountMultiplier = discountPercentage.movePointLeft(2);