Contents

Deep Dive Coding Java + Android Bootcamp Java, SQL, and XML Style Guide

Introduction

Purpose

This document serves as the complete definition of the Deep Dive Coding Java + Android Bootcamp standards for Java and SQL source code, as well as XML and JSON documents. A Java, SQL, XML, or JSON source file may be described as being "in DDC Java + Android Style" **if and only** if it adheres to the rules stated or referenced herein.

Usage

Many of the rules stated or referenced here are applied by the IntelliJ IDEA **Code/Reformat code** command with the IntelliJ scheme included in the Google Style Guide repository is installed. However, not all rules are applied in this fashion: in particular, **Code/Reformat code** does not enforce naming or

general structure/sequence conventions, and it does not remove many instances of extra vertical whitespace. In any given instance where a stated rule is not applied by this command, that *does not* excuse the student from following that rule. Similarly, if some example or starter code provided by an instructor (or anyone else) is not conformant with this style guide, that does not excuse the student from the conformance requirement.

Please note that most of the rules in this style guide are strict but narrow; that is, for any non-trivial processing task, there is a wide range of expressions that implement the given task and still conform to the rules of this style guide.

Note: Words and phrases like "**must**", "**must not**", "**do**", "**do not**", etc., written in bold type (excluding the headings, which are also in bold type), indicate strict rules of this style guide. *Italicized* words and phrases indicate recommendations or qualifications/clarifications of rules.

Java

Overview

This portion of the style guide should be treated as a supplement to the Google Java Style Guide, (GJSG).

The GJSG is not reproduced here; instead, only amendments—additional constraints and relaxations or other changes to stated constraints—are listed below. Thus, a Java source file conforms to this style guide if (and only if) it conforms to the GJSG **and** the amendments stated below—except where the amendments contradict the constraints declared in the GJSG, in which cases the amendments below dictate the rules to be followed.

Numbered links below reference (and link to) the corresponding sections of the GJSG.

Source file basics

GJSG 2 Source file basics is normative, without amendment.

Source file structure

GJSG 3 Source file structure is normative, with the amendments below.

Ordering of class contents

GJSG 3.4.2 Ordering of class contents:

> The order you choose for the members and initializers of your class can have a great effect on learnability. However, there's no single correct recipe for how to do it; different classes may order their contents in different ways.
>
> What is important is that each class uses ***some* logical order**, which its maintainer could explain if asked. For example, new methods are not just habitually added to the end of the class, as that would yield "chronological by date added" ordering, which is not a logical ordering.

In concept, we agree with the above. However, we require more predictable structure in the ordering of class members:

1. The order of class members **must** follow this general sequence:

   - `static final` fields
   - `static` (but non-`final`) fields
   - `final` (but non-`static`) fields
   - other (non-`static`, non-`final`) fields
   - `static` initializers
   - non-`static` initializers
   - constructors
   - methods
   - nested classes

   This still does not fully dictate the order of members within each of the above groups. There's no single correct way of ordering these contents; this may even vary from class to class. However, there are strict rules that must be followed; one is stated in GJSG **3.4.2.1 Overloads: never split**; another follows below.

2. **Never** split accessor/mutator pairs.

   When a class has a pair of methods that follow the JavaBeans naming convention for accessors & mutators (aka *getters* and *setters*) for some inferable property name, these methods **must** appear sequentially, with no other code between them (not even private members).

Formatting

GJSG 4 Formatting is normative, with the amendments below.

Vertical whitespace

GJSG 4.6.1 Vertical whitespace:

> *Multiple* consecutive blank lines are permitted, but never required (or encouraged).

We constrain the use of vertical whitespace further with these rules:

1. Consecutive blank lines are *discouraged but permitted between members of a class,* but the number of blank lines used **must be** consistent or predictable.
2. Consecutive blank lines are **not permitted** *inside the body* of an initializer, constructor, or method.

Grouping parentheses

GJSG 4.7 Grouping parentheses: recommended:

> Optional grouping parentheses are omitted only when author and reviewer agree that there is no reasonable chance the code will be misinterpreted without them, nor would they have made the code easier to read. It is *not* reasonable to assume that every reader has the entire Java operator precedence table memorized.

To the above, we add 2 strict rules:

1. Though the compiler treats parentheses around a *single lambda parameter* as optional, they are **required** in this bootcamp.
2. If the conditional (Boolean) operand of a ternary expression contains 1 or more binary operators, *other than* the `.` member access operator, the entire conditional operand **must be** enclosed in parentheses. For example, take the these 2 statements that include ternary expressions:

   ```
   int a = (c >= 0) ? c : -c;
   ```

   ```
   String s = t.isEmpty() ? null : t;
   ```

   In the first, the conditional operand includes the binary operator `>=`; thus, we enclose the entire conditional operand (`c >= 0`) in parentheses. In the second, while there are parentheses for the `isEmpty` method invocation, the only operator in the conditional operand is the member access operator `.`; thus, parentheses around the conditional operand are optional, but not required.

`switch` statements

GJSG 4.8.4.3 The `default` case is present:

> Each switch statement includes a `default` statement group, even if it contains no code.
>
> **Exception:** A switch statement for an `enum` type *may* omit the `default` statement group, *if* it includes explicit cases covering *all* possible values of that type. This enables IDEs or other static analysis tools to issue a warning if any cases were missed.

The above is further constrained: when a `default` is present (which, as noted, **must be** the case, except when the `case` values are *all* of the enumerated values of an `enum`), it **must be the last** statement group in the `switch`.

Naming

GJSG 5 Naming is normative, with the amendments below.

GJSC 5.2.5 Non-constant field names:

> Non-constant field names (static or otherwise) are written in `lowerCamelCase`.
>
> These names are typically nouns or noun phrases. For example, `computedValues` or `index`.

GJSC 5.2.6 Parameter names:

> Parameter names are written in `lowerCamelCase`.
>
> One-character parameter names in public methods should be avoided.

GJSC 5.2.7 Local variable names:

> Local variable names are written in `lowerCamelCase`.
>
> Even when final and immutable, local variables are not considered to be constants, and should not be styled as constants.

For this bootcamp, the above are further constrained:

1. Parameter names **must** consist of 2 or more characters each, *except for lambda parameters*, which *may have* single-character names. (If single-character names are used for lambda parameter names, the following naming rules don't apply).
2. Field, parameter, and local variable names of *all* scalar types (primitives, wrappers, `String`) *except* `boolean` and `Boolean`, **must be** singular nouns or noun phrases.
3. `boolean` or `Boolean` fields, parameters, and local variables **must be** named using adjectives or adjective phrases.
4. Fields, parameters, and local variables that are references to arrays and collections **must be** named with plural or collective nouns or noun phrases. For example, in a card game application, we may have a field (or several) of type `List<Card>`. Both `cards` and `deck` would be acceptable names for such a field: the former is a plural noun, and the latter is a collective noun.

5. **Do not** name `boolean` or `Boolean` fields with prefixes such as `is`, `are`, or `has`. (The `is` prefix is part of the JavaBeans specification for *accessor* methods, not fields. Also, all of these prefixes make the field name a verb phrase, rather than an adjectival phrase.)

6. **Do not** use Hungarian notation for field, parameter, or local variable names. That is, do not prefix the name with type or role identifiers.

7. *Avoid* unnecessary suffixes on field, parameter, and local variable names. For example, *prefer*

   ```
   private Button update;
   ```

   to

   ```
   private Button updateButton;
   ```

   Similarly, while the `m` and `s` prefixes on non-`public`, non-`static` and `static` fields (respectively) are often seen in Java source code (including the Android library source code), these are also unnecessary, and *should be avoided.*

8. If a non-`static` field is intended to act as a primary key value of a persistent instance (e.g a field annotated with `@PrimaryKey` in an entity class), *prefer* the simpler `id` to `{entity name}Id`.

Note that there are a few long-standing exceptions to the naming & casing rules listed in GJSG and above. For example, `UUID` is a class in the Java standard ibrary, written as indicated in `UPPERCASE`; however, `uuid` (rather than `uUID`) is the correct way to use that initialism as a field or variable name.

Programming practices

GJSG 6 Programming Practices is normative, with the amendments below.

Types

1. If a non-`static` field is intended to act as a non-compound primary key value of a persistent instance (e.g a field annotated with `@PrimaryKey` in an entity class), the type of the field **must be** one of:

   - `UUID`
   - `long` or `LONG`
   - `int` or `Integer`

   Though the type selected *should* be independent of the RDBMS being used, this is not always possible. For example, when using SQLite, the most performant choice (by far) for the type of a field intended for use as a non-compound primary key value is `long` or `Long`.

Access level modifiers

1. Class members are declared at the *lowest practical access level*.
2. The **only fields allowed** to have the `public` access level are those marked `final`. (Note that `interface` fields are implicitly `public static final`, as are the enumerated values of an `enum` class.)
3. Non-`static` fields, even if `final`, **must not** be `public`, *except* in nested `private` classes.
4. *Prefer* `protected` accessors and mutators to `protected` fields.

`if-else if` ladders

1. An `if-else if` statement ladder *should* include a final `else`, *especially* when one or more of the following holds:
   - `else if` occurs multiple times in the ladder.
   - The purpose of the ladder is to assign one of a number of alternative values to a field.
   - The purpose of the ladder is to assign one of a number of alternative values to a local variable, and that variable is not declared immediately before the ladder.

Javadoc

GJSG 7 Javadoc is normative, with the amendments below.

Where Javadoc is used

GJSC 7.3.1 Exception: self-explanatory methods:

> Javadoc is optional for "simple, obvious" methods like `getFoo`, in cases where there *really and truly* is nothing else worthwhile to say but "Returns the foo".
>
> **Important:** it is not appropriate to cite this exception to justify omitting relevant information that a typical reader might need to know. For example, for a method named `getCanonicalName`, don't omit its documentation (with the rationale that it would say only `/** Returns the canonical name. */`) if a typical reader may have no idea what the term "canonical name" means!

While we do not contradicting this exception entirely, we are qualifying it:

1. Omitting the Javadoc comment entirely, even for a "self-explanatory" method, *should be treated as a truly exceptional condition.* In most cases, there *should* be some comment, even if just a summary fragment.

   Remember: Your documentation is likely to be read in an HTML page, without the reader having access to the implementation details of your class. Thus, a comment like

   ```
   /** Returns age. */`
   ```

   may well be meaningless or of little value, especially if `age` is a `private` field.

   However, the comment

   ```
   /** Returns the age (in generations) of this cell. */
   ```

   is potentially much more meaningful, doesn't depend on the reader knowing anything about the existence of an `age` field, and *shouldn't be* omitted.

2. On the other hand, a Javadoc comment *may be* omitted for a `@param` or `@return` tag, when such a comment would simply repeat what is already stated in the method-level Javadoc comment. This is most often the case for an accessor or mutator (getter or setter) method, where a comment fragment for the `@return` or `@param` tag (respectively) would tend to repeat the summary fragment of the method itself.

   Note that the reverse of the above exception is not permitted: The method-level Javadoc comment **must not** be omitted if a comment is provided for a `@param`, `@return`, or `@throws` tag of the same method. In other words, if a method includes any Javadoc comments at all, it **must** include at least the summary fragment comment for the method itself.

SQL

Overview

In SQL programming, there really isn't an equivalent to GJSG, or even to the earlier, more basic Code Conventions for the Java™ Programming Language (last updated by Sun in 1999, and currently made available for archive purposes by Oracle). Most of the rules included in both of those publications (which differ only in a few details, and in elements added to the Java language after the final revisions to the first) are followed by the overwhelming majority of Java developers; however, there isn't a SQL style guide with anything close to the same rate of formal adoption or informal conformance.

In this style guide, the focus is primarily on naming (including casing), and on a few general formatting rules. Most of these are long-established, and widely followed—though with plenty of variation from organization to organization.

One benefit of this flexibility is that many SQL formatters can be configured to apply several of these rules; this includes the **Code/Reformat code** feature of IntelliJ, which—when using the default settings—will apply all of the formatting rules (but not the structural or naming & letter-casing rules) given here.

This style guide is deliberately oriented toward using SQL via an *object-relational mapping* (ORM) library. That is, it's generally assumed that an ORM such as Hibernate or Room will be be managing persistence, and—in most cases—generating the SQL DDL code to implement the data model in a database schema. Thus, there is not much attention placed on formatting details for SQL statements; instead, naming and letter casing of keywords and identifiers are the main focus, with a secondary focus on formatting (primarily for the purpose of producing technical documentation).

Some of the rules here are based on Simon Holywell's SQL Style Guide (SSG). However, where the DDC Java style rules incorporate those in GJSG with only a few amendments, the DDC SQL style guide deviates significantly from SSG. Thus, the rules below should be treated as normative, without reference to similar or contradicting rules in the SSG.

On the other hand, beyond newline and indentation rules, this style guide doesn't address any other specifics of vertical alignment—a topic of great disagreement in SQL code formatting. SSG or other SQL style guides may be consulted for this purpose.

Files

Naming

1. All SQL source code files **must be** named in `spinal-case`, with a `.sql` extension, *unless* another naming convention is a requirement of a tool or library being used.

Structure

1. If multiple SQL statements are included in a single `.sql` file, or in a string literal in a source code file of another type, the code in the file (or literal) **must be** executable as written, in order. Minimally, this implies:
    1. Every statement **must be** terminated by a semicolon (`;`).
    2. The order of statements **must be** such that if all are executed in order, no constraints are violated, and all dependencies are satisfied. For example, an `ALTER TABLE` statement must either follow (immediately or otherwise) the corresponding `CREATE TABLE` statement, or it must be conditioned (using `IF EXISTS`) on the existence of the table in question.

Casing

Keywords & built-in functions

1. SQL keywords (including the names of built-in functions) **must be** written in `UPPERCASE`. (In some SQL dialects, some multi-word keywords have underscores; these would be written in `UPPER_SNAKE_CASE`.)

Identifiers

1. All explicitly named persistent or transient SQL objects (tables, views, columns, aliases, indices, sequences, stored procedures, user-defined functions) **must be** named using `lower_snake_case`.

Naming

General

1. **Do not** use abbreviations (including acronyms and other initialisms), *unless* the abbreviation is widely understood and commonly used in place of the unabbreviated form, *and only if* the unabbreviated form is significantly longer than the abbreviation.

Tables & views

1. If the name of table is generated automatically (e.g. by an ORM from the name of an entity class), the automatically generated name **must be** used, *even if it does not conform* to the casing and naming rules stated here, *unless* one of the following is true:
    1. The automatically generated name is unacceptable—e.g. it conflicts with a SQL keyword—and must be specified explicitly.
    2. The table is a *join table*, automatically defined by an ORM to effect a many-to-many relationship. The name of such a table *may be* left as generated, or a name *may be* specified explicitly.
2. Explicitly named tables and views **must be** named with singular nouns or noun phrases. (This corresponds to the convention for Java class names.)
3. In table and view names, **do not** use unnecessary prefixes, such as `tbl_`, `table_`. Similarly, **do not** use unnecessary suffixes (`_tbl`, `_table`, etc.).

Table & view aliases

Table and view aliases (aka *correlations*) are commonly defined in join expressions, and referenced in column selection lists, `ON` clauses, `WHERE` clauses, `ORDER BY` clauses, etc.

The rules below *do not apply* to SQL statements generated by an ORM. However, in explicitly specified SQL statements using table/view aliases, follow these rule:

1. Aliases *should be* constructed from initialisms for the correlated table or view name. For example, an alias for a table named `user_profile` would be named `up`, while an alias for `building_location_detail` would be `bld`.
2. If a given table is included 2 or more times in a join expression, **all** of its aliases **must be** numbered—e.g. `bld1`, `bld2`. (Underscores *may be* used between the initialism and the number: `bld_1`, `bld_2`; as usual, consistency is *strongly* advised.)

Columns

1. If using Hibernate (or a similar ORM), which automatically maps `lowerCamelCase` field names in an entity class to `lower_snake_case` column names (e.g. `someData` to `some_data`), **do not** specify a column name explicitly, *unless both of the following conditions hold*:
   - The ORM is being used to map a new Java data model to an existing database schema.
   - The ORM's column name mapping scheme does not correctly translate a given field name to the name of the existing column.
2. Explicitly named columns of all types other than `BOOLEAN` **must be** named with singular nouns or noun phrases.
3. `BOOLEAN` columns **must be** named with adjectives or adjectival phrases.
4. With a few exceptions (e.g. columns that are used as primary keys, foreign keys, or unique keys), **do not** use the table name as a column name prefix. For example, in a table named `account`, use the column name `balance` instead of `account_balance`.
5. For `BOOLEAN` columns, **do not** use prefixes like `is_`, `has_`, `contains_`, etc. (All of these make the name a verb phrase, rather than an adjectival phrase.) For example, use `enabled` instead of `is_enabled`.
6. All columns used as non-compound (single-column) primary keys or foreign keys **must be** named with the `_id` suffix. **No other** columns are permitted to use that suffix.
7. All columns constituting non-compound (single-column) primary keys **must be** named with the `{table}_id` pattern (or `{entity}_id`, for a

table corresponding to an entity class). For example, the primary key column for an `appointment` table would be named `appointment_id`.

8. Foreign keys *may be*—but need not be—named to match the corresponding primary keys. An acceptable alternative approach is to name the foreign key according to the role played by the referenced table.

   For example, assume we have the table `contributor`, with the primary key `contributor_id`. We also have another table, `post`, with a foreign key that references `contributor.contributor_id`. This foreign key column might also be named `contributor_id`—or it could be named something like `author_id`, since the referenced contributor record represents the author of any given post.

9. Columns constituting non-compound (single-column) unique keys (other than primary keys, which are implicitly unique) *should be* given names that indicate that values in the column uniquely identify rows. *Suggested* name endings that are commonly (but not exclusively) used for this purpose include `_name`, `_code`, and `_key`.

Column aliases

1. Column aliases **must** follow the same rules as column names.

Subprograms & triggers

1. Stored procedures and triggers **must be** named using verbs or verb phrases.
2. **Do not** use unnecessary prefixes in stored procedure or trigger names, such as `sp_`, `tr_`, or `proc_`. These are similarly unnecessary in suffix form; **don't use them**.
3. Functions may be defined to return scalar values, composite values (this may be thought of as a single row), or rowsets; the naming rules reflect these possibilities:
   1. Scalar-valued functions **must be** named according to the same rules as columns.
   2. Composite- and rowset-valued functions **must be** named according to the same rules as tables and view.
4. **Do not** use unnecessary prefixes, such as `func_`, or `fn_`. These are equally lacking in meaningful information in suffix form; **don't use them**.

Indices

1. In most SQL dialects, index names can be generated automatically, and need not be specified explicitly. If that isn't the case for the underlying RDBMS, but an ORM is used, the ORM virtually always generates index names automatically. *Prefer the automatically generated name* to an explicitly specified name.
2. If an index must be named explicitly, the name **must** follow the applicable pattern from the list below.
    1. For an index backing a primary key comprised of a single column:

        `pk_{table}`

        Placeholders

        `{table}` Name of the given table.

    2. For an index backing a non-primary key (compound or otherwise) or a compound primary key:

        `{prefix}_{table}_{column1}[_{column2}[…]]`

        Placeholders

        `{prefix}` Type prefix: `pk` for primary key index, `uq` for unique key index, `ix` for non-unique index.
        `{table}` Name of the table on which the given index is defined.
        `{column1}` Name of the first column used in the index.
        `{column2}` Name of the second column used in the index (if applicable).
        ... Additional column names (if any) used in the index, separated by underscores (`_`) .

        The square brackets enclose optional parts of the name, and must be replaced accordingly; the brackets themselves are not allowed in the actual index name.

Sequences

Many RDBMSs use sequences to implement auto-numbered (aka auto-incremented) fields.

1. When using an ORM (and in some cases when an ORM is not used), sequences are defined implicitly, with an automatically generated name. When this is the case, the generated name **must be** used.
2. If a sequence name must be specified explicitly, the name **must** follow the applicable pattern from the list below.
    1. For a sequence providing values for a non-compound primary key column:

`{column}_seq`

Placeholders

`{column}` Name of the column that will be assigned values from the given sequence.

2. For a sequence providing values for a column that is not a non-compound primary key:

   `{table}_{column}_seq`

   Placeholders

   `{table}` Name of the table containing the column that will be assigned values from the given sequence.
   `{column}` Name of the column that will be assigned values from the given sequence.

Types

1. If an ORM is being used, and the ORM is capable of mapping a Java type to a SQL type automatically, the SQL type *should be* left as mapped, unless the resulting type violates one of the strict rules that follow.
2. Columns of character types (e.g. `CHAR`, `VARCHAR`, `TEXT`) *may be* used (individual or compounded with other columns) in unique keys, but **must not** be used in primary keys.
3. Any table that is *not* a join table (used to effect a many-to-many relationship) **must have** a non-compound (single-column) primary key, with automatically generated values of one of these types:

   - `UUID`

     For a primary key, a value of this type **must be** stored and indexed as a 16-byte (128-bit) binary value; in some RDBMSs, this requires that we declare the actual type as `BINARY(16)`, `CHAR(16) FOR BIT DATA`, etc.

   - `BIGINT`
   - `INT`

     For a primary key of either of the last 2 types above, the `UNSIGNED` modifier should be used if possible (not all RDBMS platforms support this).

Formatting

Overview

Our style guide does not fully specify the allowed or required formatting of SQL code across multiple lines. In particular, as noted below, single-line statements are permitted in some contexts; in such cases, the conventions for new lines and indentation are moot. Nonetheless, the rules specified do apply, along with the naming and casing rules above, with any conditions as stated. Beyond that, *consistency is critical*: Apply the same formatting rules throughout a project.

Code formatting tools

1. In this bootcamp, the most important guideline to follow for SQL code formatting—apart from following the naming and casing rules, of course—is simple: *Use a formatting tool!* This might be the IntelliJ **Code/Format Code** command, or any of a number of other SQL code formatting tools, including those listed in the SQL section of **Formatting tools** reference section of this document.

   Some of these tools (including the one provided by IntelliJ) are dialect-specific: the appropriate dialect of SQL must be configured in order to format the code properly. In most cases, formatting works much better if the SQL code is syntactically correct for the chosen dialect; for the IntelliJ SQL formatter, this is required.

Single-line statements

In some contexts, even a reasonably complex SQL statement *may be* written in a single line. In the single-line form of a statement, the rules for newlines and indentation (below) don't apply.

1. A SQL statement used as the `value` argument of a Room or Spring Data `@Query` annotation *may be* written in a single line, directly in the annotation. However, if the statement has a multi-table/view `FROM` clause, consider writing it in multiple lines (concatenated as needed) as a `static final String` field, and then referencing that constant in the `value` argument of the `@Query` annotation.
2. A subquery (correlated or not) within another statement *may be* written in a single line, in parentheses. However, you are *encouraged* to split the subquery into multiple lines, indented as appropriate—*especially* if the subquery involves multiple tables, views, or subqueries.

New lines

1. The following keywords (and keyword combinations) **must** start a new line (with the exceptions noted for single-line statements, above):
   - `ADD`
   - `ALTER`

- `CREATE`
- `DECLARE`
- `DELETE`
- `DROP`
- `ELSE`
- `END`, when used to close a block that starts with `BEGIN`.
- `FROM`
- `GROUP BY`
- `HAVING`
- `IF`
- `INSERT`
- *`[JOIN_TYPE]`*`JOIN` , where *`JOIN_TYPE`* (if present) is one of `INNER`, `LEFT`, `RIGHT`, `FULL OUTER`, `CROSS`, `NATURAL`, etc.
- `MODIFY`
- `ORDER BY`
- `SELECT`
- `SET`
- `TRUNCATE`
- `UPDATE`
- `VALUES`
- `WHERE`

2. The second, and every subsequent, column expression in a `SELECT` column list **must** start a new line. The first column expression *should* start a new line.

3. When multiple tables are includes in the `FROM` clause of a `SELECT` statement, and the `JOIN` keyword is not used to declare join conditions, each table name after the first **must** start a new line. (Before the adoption of the `JOIN` keyword, this form was commonly used with a join condition expressed in the `WHERE` clause; *avoid* this practice.)

4. Every column, table constraint, or primary key definition in a `CREATE TABLE` statement **must** start a new line. Column-level constraints, and the `PRIMARY KEY` modifier on a column definition, *should* be on the same line as the column definition.

5. The second, and every subsequent, column assignment following `SET` in an `UPDATE` statement **must** start a new line. The first column assignment *should* start a new line.

6. The closing brace of a `CREATE TABLE` statement **must** start a new line.

7. The opening brace of a `CREATE TABLE` statement *should not* (but is permitted to) start a new line.

8. In the Boolean expression of the `WHERE` or `HAVING` clause, the `AND`, `OR`, and `NOT` conjunction operators *should* start a new line.

9. Between `CASE` and the matching `END`, each `WHEN` and `ELSE` *should* start a new line. If this is done, the `END` **must** start a new line.

Indentation

In most contexts, *indentation* refers to the spacing between the absolute start of a line and the first non-white-space character, with the implicit understanding that items indented to the same level are aligned along the left edge of the text. However, this is not always the case in SQL: Some style guides and formatting tools use indentation to produce a consist alignment of the *right* edge of keywords (or the first words in keyword phrases) Either of these indentation approaches is acceptable; *favor* consistency over convenience.

Regardless of your left vs. right alignment preference, follow these rules:

1. Indentation **must** use space characters instead of tabs.
2. The keywords starting the clauses of a SQL statement (e.g. `FROM`, `WHERE`, etc. in a `SELECT` statement) **must be** indented to the same level as the statement itself.

   As stated above, this does not *necessarily* mean that the clause and statement keywords are vertically aligned at the start (left-most character) of the keyword. Many SQL style guides and formatting tools dictate a right alignment of keywords, creating a *river*—a vertical column of whitespace after the keywords. So the following 2 forms are both acceptable.

   Left-aligned:

   ```
   SELECT
       ...
   FROM
       ...
   WHERE
       ...
   GROUP BY
       ...
   HAVING
       ...
   ORDER BY
       ...
   ```

   Right-aligned

   ```
   SELECT
       ...
     FROM
       ...
    WHERE
   ```

```
    ...
 GROUP BY
    ...
HAVING
    ...
 ORDER BY
    ...
```

Note that in the second form, the *end* of each clause keyword (or the first word in a keyword phrase) is aligned. This is not required, but it is a fairly common practice in SQL formatting, and is allowed.

3. When a new line is added to a clause of a SQL statement—e.g. for the second and subsequent columns in the column list of a `SELECT` statement, or a `JOIN` in the `FROM` clause of a `SELECT` statement—the new line **must be** indented further to the right of the indent level of the clause itself.

   Here's an example `SELECT` statement with a `JOIN`, demonstrating this rule:

   ```
   SELECT
       a.article_id,
       a.title,
       up.name
   FROM
       article AS a
       JOIN user_profile AS up
           ON up.user_id = a.author_id;
   ```

4. The column, table constraint, and primary key definitions of a `CREATE TABLE` statement are contained within **must be** indented within the curly braces of the statement.

5. For `BEGIN…END` blocks, the `END` keyword **must be** indented to the same level as the matching `BEGIN`.

6. For `CASE…END` blocks that extend over multiple lines, the `END` keyword **must be** indented to the same level as the matching `CASE`.

7. All of the `WHEN` and `ELSE` keywords in a multiline `CASE` statement **must be** indented to the same level, and to the right of the enclosing `CASE` and `END`.

8. All statements contained within `BEGIN` and `END` **must be** indented to the same level, and to the right of the `BEGIN` and `END`.

Vertical whitespace

1. When a SQL file contains multiple statements, every semicolon (`;`) statement terminator **must be** followed by *at least* (and preferably *exactly*) 1 blank line.

Programming practices

The rules below *do not apply* when a SQL statement expression is generated automatically by an ORM; however, in explicitly specified SQL statements, they do apply.

Joins

1. **Do not** use the old-style form, where the `JOIN` keyword is not used, and the join condition is specified only in the `WHERE` clause.

   Bad

   ```
   SELECT
       a.article_id,
       a.title,
       up.name
   FROM
       article AS a,
       user_profile AS up
   WHERE
       up.user_id = a.author_id;
   ```

   Good

   ```
   SELECT
       a.article_id,
       a.title,
       up.name
   FROM
       article AS a
       JOIN user_profile AS up
           ON up.user_id = a.author_id;
   ```

2. **Do not** use correlated subqueries in place of `JOIN`. For example, the following is equivalent to the example statements in point 1, but is not acceptable, since there is a `JOIN` equivalent (as seen above):

   Bad

   ```
   SELECT
       a.article_id,
       a.title,
       (
           SELECT
               up.name
           FROM
               user_profile up
           WHERE
   ```

```
            up.user_id = a.author_id
    ) as name
FROM
    article AS a;
```

Note: In some cases, a correlated subquery can't be avoided without complicating the code significantly. However, *almost never* does this happen outside of a `JOIN`. In other words, we're using the correlated subquery to define a table source used in the join. This use of a correlated subquery—as a participant in a `JOIN`, rather than instead of a `JOIN`—is permitted, but *should be avoided when possible.*

Table & view aliases

1. Table/view aliases **must be** used in multi-table `FROM` clauses—that is, in all join expressions.
2. The `AS` keyword **must be** used when declaring a table/view alias.
3. Table/view aliases *should be* constructed from initialisms for the correlated table or view name. For example, an alias for a table named `user_profile` would be named `up`, while an alias for `building_location_detail` would be `bld`.
4. If a given table is included 2 or more times in a join expression, **all** of its aliases **must be** numbered—e.g. `bld1`, `bld2`. (Underscores *may be* used between the initialism and the number: `bld_1`, `bld_2`; as usual, consistency is *strongly* advised.)

Column aliases

1. A column alias **must be** used whenever an item in a query select list is a computed expression, rather than a simple column name.
2. The `AS` keyword **must be** used when declaring a column alias.

XML

Overview

Due to its essential simplicity and flexibility, *eXtensible Markup Language* (XML) plays a variety of roles in software development. Common uses of XML documents include—but are not limited to—these:

- Data interchange (this is the case for many web services—some using SOAP or other standard schemas, others using provider-specified schemas).

- Data transformation rules (XSLT).
- Schemas that can be used to validate the structure and content of other XML documents.
- Domain-specific language code (see the Flowgorithm file format for an example of this).
- Text markup content (e.g. XHTML).
- Project definition files (e.g. Maven `pom.xml` files).
- Build scripts (e.g. Ant files).
- Configuration data (e.g. Java XML properties files, Android package manifests).
- Localizable UI resources (Java XML resource bundles, Android string, color, and drawable resources).
- Style rules (Android style resources).
- UI layouts (Android menu and layout resources, JavaFX layout files).

Given the wide variety of applications, it is difficult to have a comprehensive style guide for XML—especially since at least the element and attribute names in most of the above listed items are not under our control. Thus, the rules and practices below should be interpreted with that understanding.

Files

1. *To the extent that file naming and casing are within our control*, files containing XML documents **must be** named in `spinal-case`, with an appropriate lowercase extension (`.xml`, `.fxml`, etc.).

   (An example of a use where letter casing is not in our control is Android resource files, which must be named in `lower_snake_case`.)

2. In the source code structure of a Java development project (including simple Java libraries & console-mode applications, Android apps & services, Maven projects, etc.), XML files **must not** physically reside within the same classpath element as Java source files. For example, in an Android app project, Java files are located in the `app/src/main/java` subdirectory, while resources (XML and others) are in `app/src/main/res`.

   In an IDE such as IntelliJ IDEA, this will typically be implemented by defining one or more *resource folders* in the project, separate from the *source folders*. The former contains XML files and other non-Java assets, while the latter contains the Java source files.

3. The encoding for an XML file **must be** UTF-8, without a byte order mark (BOM).

Elements

In most cases in this bootcamp, the naming and casing of elements is not in our control; we're usually consuming or authoring XML documents that have a set of supported elements dictated by the tools we're using, or by the schemas recognized by those tools.

When we do have control over naming and casing of elements—e.g. when developing a web service capable of returning XML responses to the client—the rules below apply.

1. Element names **must be** nouns or noun phrases.
2. Elements which act as containers for repeated child elements **must be** given plural or collective names.
3. Element names *should be* in `UpperCamelCase`. Among the permissible exceptions are elements which are specify property values of a parent element (e.g. where a structure is needed, rather than a scalar that could be specified as an attribute value): these *may be* named in `lowerCamelCase`.

Attributes

As with elements, we are usually not in control of the attributes available in a given application. When we are in control, follow these rules;

1. Attribute names *should be* nouns or noun phrases, except for names of Boolean-valued attributes, which *should be* adjectives or adjective phrases.
2. If an attribute is multi-valued (e.g. a comma-separated list), it **must be** given a plural or collective name.
3. Attributes **must be** named in `lowerCamelCase`.

Namespaces

1. Namespace prefixes **must be** in `spinal-case`.
2. While the prefix of a namespace is only significant within its scope (the element where the namespace is declared as an attribute with the `xmlns` prefix, and all child nodes of that element), apply the *principle of least astonishment* (PoLA): if a given namespace has a very commonly used prefix, then *favor* that prefix in your XML as well—unless that conflicts with the above rule.

   In particular, the well-established namespaces below **must** use the prefixes shown:

   - `xmlns:xsd="http://www.w3.org/2001/XMLSchema"` or `xmlns:xs="http://www.w3.org/2001/XML`

- `xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"`
- `xmlns:xsl="http://www.w3.org/1999/XSL/Transform"`
- `xmlns:android="http://schemas.android.com/apk/res/android"`
- `xmlns:app="http://schemas.android.com/apk/res-auto"`
- `xmlns:tools="http://schemas.android.com/tools"`
- `xmlns:fx="http://javafx.com/fxml"`

Formatting

To the extent that XML is used for project definition, configuration, or source code—POM files, XML properties files, Android resources, JavaFX layout files, etc.—and to the extent that we have control over the file formatting, the following rules apply.

1. *Use a formatting tool!* For example, when IntelliJ is configured to use the Google Style Guide schema, the **Code/Format Code** command can be used in XML files, enforcing (at least) the rules below.
2. Indentation of elements and attributes **must** use space characters instead of tabs.
3. Every XML element **must** begin on its own line.
4. All child elements of a given parent **must be** indented by the same amount (at least 2 spaces) to the right of the opening tag of the parent element.
5. If an element contains more than one attribute, then every attribute starting with the second **must** start a new line. The first attribute of such an element *should* start a new line.
6. All attributes starting a new line must be indented the same amount (at least 2 spaces) to the right of the indent position of the element itself.
7. If an element has zero or one attributes, no child elements, and a text node that does not extend beyond a single line, the start and end tags of the element *should be* written on a single line.
8. *Favor empty-element* (self-closing) tags for elements that have no body content (i.e. no child elements or text nodes).
9. When an element has a closing tag, and that closing tag is not on the same line as the opening tag, the closing tag **must be** indented to the same level as the opening tag.
10. **Do not** include whitespace around the equal sign between an attribute name and its value.

11. There *should not* be any whitespace immediately preceding the closing angle bracket of an element's opening or closing tag.
12. In a self-closing tag, there *may* be whitespace preceding the self-closing token (`/>`); any such whitespace *should be* kept to a minimum, and just for the purpose of improving readability.

JSON

Overview

In Java development, the applications of JSON are similar to—though slightly more narrow in scope than—those of XML. It is most often used for data interchange (e.g. sending and receiving data to and from web services), but it can also be used for configuration files (e.g. as an alternative to YAML for an OpenAPI/Swagger service definition) and other purposes.

Structurally, it can be useful to think of JSON as being analogous to XML—but without standard related facilities such as DTDs, schemas, XSL transforms, and XPath; with an anonymous root element; and combining the concepts of attributes and child elements into a single *property* concept. (This may sound very limiting; it actually isn't as limiting as it might appear, and the benefit—a reduction in the verbosity that is a common pain point for XML users—often justifies these constraints.)

As is the case with XML, we have no control over the naming and letter casing of properties in JSON values we receive from an external service or data source. Also, the actual JSON documents we send and receive are often generated by

Files

1. *To the extent that file naming and casing are within our control*, JSON files **must be** named in `spinal-case`, with an appropriate lowercase extension (usually `.json`).
2. If a JSON file is part of the configuration or source code of a Java development project, this file **must not** physically reside within the same classpath element as Java source files; instead, it should be located (directly or indirectly) in a resource folder.

   In an IDE such as IntelliJ IDEA, this will typically be implemented by defining one or more *resource folders* in the project, separate from the *source folders*. The former contains non-Java assets, while the latter contains the Java source files.
3. The encoding for a JSON file **must be** UTF-8, without a byte order mark (BOM).

Properties

In this bootcamp, the naming and casing of JSON properties is rarely in our control; we're usually consuming or authoring JSON objects that have a set of properties dictated by the services or tools we're using.

When we do have control over naming and casing of properties—e.g. when developing a web service capable of returning JSON responses to the client—the rules below apply.

1. Property names **must be** nouns or noun phrases.
2. Array-valued properties **must be** given plural or collective names.
3. Property names **must be** in `lowerCamelCase`.

Formatting

Most of the time, JSON formatting is not that important to us, since we're not authoring JSON files and using them as configuration or other source files in our projects; instead, we're using a library to send and receive them, and they're not seen by human eyes.

There are exceptions to the above, of course. For example, we might incorporate a data dump from an external data source, packaged as JSON, into the startup/post-install step for our web service or Android app. In that case, the file may indeed need to be human-readable. In cases such as those, the following rules apply.

1. *Use a formatting tool!* When IntelliJ is configured to use the Google Style Guide scheme, the **Code/Format Code** command can be used in JSON files, enforcing (at least) the rules below. The tools in the JSON section of **Formatting tools** can also be configured to produce formatted JSON that conforms to the rules here.
2. Indentation of object properties and array elements **must** use space characters instead of tabs.
3. Every property of an object **must** start a new line.
4. All properties of an object **must** be indented by the same amount (at least 2 spaces) to the right of the indentation of the line where the object's opening brace appears.
5. Every element of an array **must** start a new line.
6. All elements of an array **must** be indented by the same amount (at least 2 spaces) to the right of the indentation of the line where the array's opening bracket appears.

7. If a property is scalar-valued (i.e. the value is string, number, Boolean, or `null`), the value **must** follow the property name on the same line.
8. If a property is array- or object-valued, the opening token (bracket or brace, respectively) of the array or object **must** follow the property name on the same line.
9. The closing brace or bracket of a JSON object or array (respectively) **must** start a new line.
10. The closing brace or bracket of a JSON object or array (respectively) **must** be indented to the same level as the line containing the opening brace or bracket.
11. The colon that separates a property name from its value **must have** a trailing space, but *should not* have a leading space. That is, the colon *should* follow immediately after the quoted property name, but it **must** itself be followed by a *single* space, and then the starting token of the property value.
12. The comma following any object property or array element that is *not* the last property listed in the object, or the last element in the array, must appear immediately after the preceding property or element, with no intervening spaces. (Note that if the previous property or element is object- or array-valued, the comma will follow immediately after the closing brace or bracket of the previous property or element.)

Resources

Style guides

- Code Conventions for the Java™ Programming Language (https://www.oracle.com/java/technologies/javase/codeconventions-contents.html)

  A respected (but no longer updated) style guide originally published by Sun, now made available by Oracle for archive purposes only.

- Google Java Style Guides (https://google.github.io/styleguide/)

  This is the foundation for the Java portion of this style guide, as well as elements of the XML and JSON portions.

- SQL Style Guide (https://www.sqlstyle.guide/), S. Holywell

  Comprehensive guide, some portions have which have been incorporated into this style guide.

- Common Programming Case Styles

  Introduction to letter casing styles most commonly used in coding, with notes on general and DDC-specific usage.

Formatting tools

Integrated

- `intellij-java-google-style.xml` (https://raw.githubusercontent.com/google/styleguide/gh-pages/intellij-java-google-style.xml)

  Part of the Google Style Guide repository, this is a style scheme that can be imported into IntelliJ IDEA, providing extensive formatting support for Java source files, as well as less comprehensive support for XML and JSON formatting.

Online

The tools listed here produce (with some configuration) code with formatting that's conformant with the rules in this style guide. However, the recommended tool for this purpose is the IntelliJ IDEA **Code/Format Code** command.

SQL

- Code Beautify: SQL Formatter (https://codebeautify.org/sqlformatter)
- FreeFormatter.com: SQL Formatter (https://www.freeformatter.com/sql-formatter.html)
- Instant SQL Formatter (http://www.dpriver.com/pp/sqlformat.htm)
- SQLFormat (https://sqlformat.org/)

JSON

- Code Beautify: JSON Viewer (https://codebeautify.org/jsonviewer)
- FreeFormatter.com: JSON Formatter (https://www.freeformatter.com/json-formatter.html)
- JSON Formatter & Validator (https://jsonformatter.curiousconcept.com/)