Admitto Unum

Overview

Description

Route66 is a set of Java libraries for general computing.

TODO: Add description of projects here.

1.0 Conventions

`/1.0 Conventions`

1.1 Development Environment

`/1.0 Conventions/1.1 Development Environment`

1.1.1 Documentation Tools

`/1.0 Conventions/1.1 Development Environment/1.1.1 Documentation Tools`

Description

TODO: give description here.

Design Documenation

Design documentation will be created using C4builder. This documenation tool supports Markdown documents and C4 Models using C4-PlantUML. This tool is integrated with VSCode with *PlantUML v2.17.3* plugn.

The C4Builder tool generates a Web site locally for view using at (http://localhost:3000) while developing documentation.

When editing design documentation the C4Builder can be run to automatically regenerate the site content with

```
    c4builder site --watch
```

To run the Web site locally

```
    c4builder site
```

Markdown

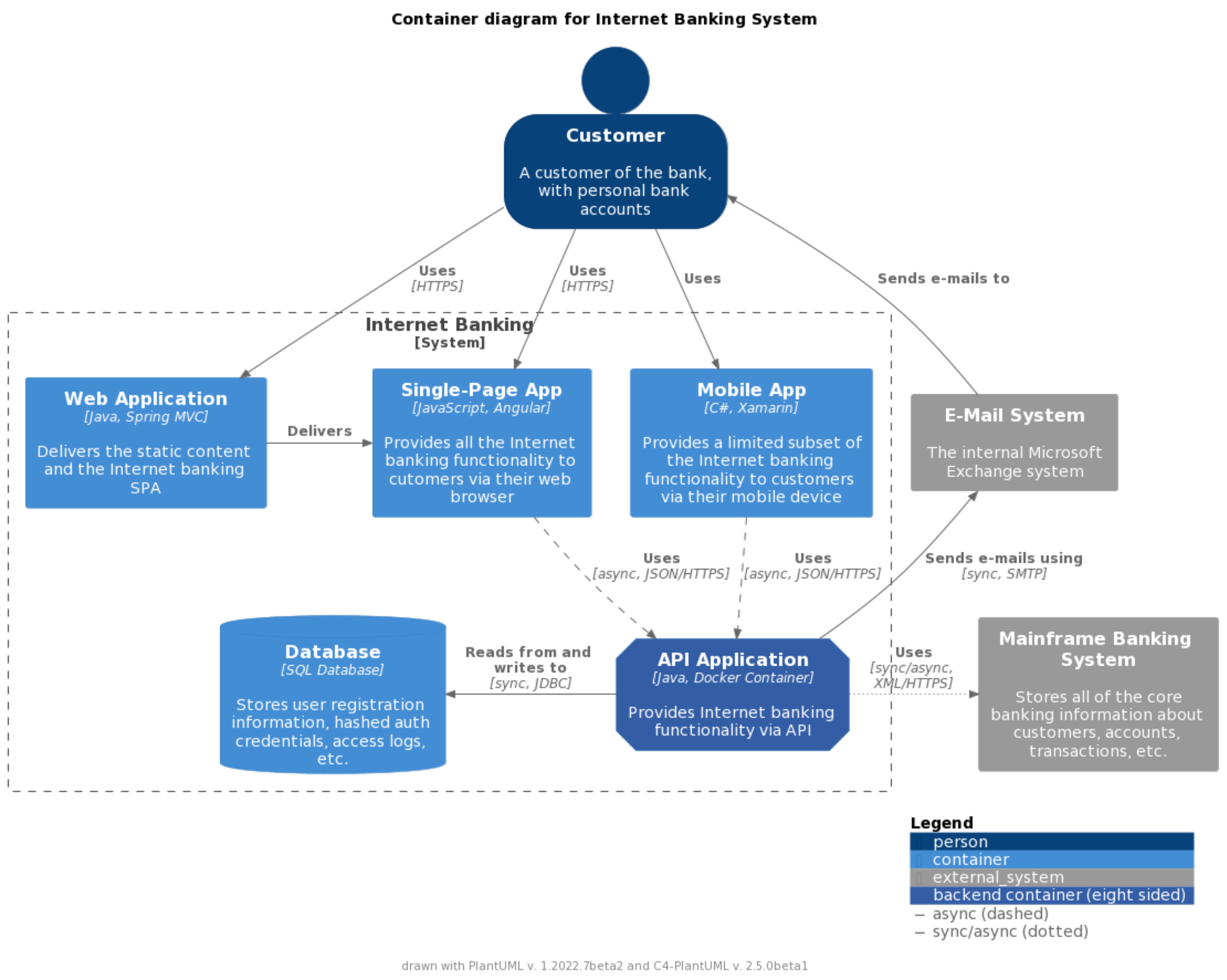
Markdown for GitHub used to create all textual documentation.

C4 Model

C4 model is a lean graphical notation technique for modelling the architecture of software systems. It is based on a structural decomposition of a system into containers and components and relies on existing modelling techniques such as the Unified Modelling Language (UML) or Entity Relation Diagrams (ERD) for the more detailed decomposition of the architectural building blocks. (C4 model, Wikipedia)

C4-PlantUML

C4-PlantUML includes macros and stereotypes for creating C4 diagrams with PlantUML.


Container diagram for Internet Banking System
Customer
A customer of the bank, with personal bank accounts
Uses [HTTPS]
Uses [HTTPS]
Uses
Sends e-mails to
Internet Banking [System]
Web Application [Java, Spring MVC]
Delivers the static content and the Internet banking SPA
Delivers
Single-Page App [JavaScript, Angular]
Provides all the Internet banking functionality to cutomers via their web browser
Mobile App [C#, Xamarin]
Provides a limited subset of the Internet banking functionality to customers via their mobile device
E-Mail System
The internal Microsoft Exchange system
Uses [async, JSON/HTTPS]
Uses [async, JSON/HTTPS]
Sends e-mails using [sync, SMTP]
Database [SQL Database]
Stores user registration information, hashed auth credentials, access logs, etc.
Reads from and writes to [sync, JDBC]
API Application [Java, Docker Container]
Provides Internet banking functionality via API
Uses [sync/async, XML/HTTPS]
Mainframe Banking System
Stores all of the core banking information about customers, accounts, transactions, etc.
Legend
person
container
external_system
backend container (eight sided)
async (dashed)
sync/async (dotted)
drawn with PlantUML v. 1.2022.7beta2 and C4-PlantUML v. 2.5.0beta1

1.1.2 Build Tools

/1.0 Conventions/1.1 Development Environment/1.1.2 Build Tools

Required Build Tools

1. Maven
2. Eclipse
3. Bitbucket
4. Jenkins CI
5. AdmitOne Maven Repository

Static Code Analysis

This project will utitlize the following automated analysis tools:

1. Code Formatting - formatting java source code using the Eclipse code formatter
2. CheckStyle - automates the process of checking Java code
3. SonarLint and SonarQube - identifies and helps you fix quality and security issues
4. OWASP Dependency Checking - attempts to detect publicly disclosed vulnerabilities contained within a project’s dependencies

Maven Plugins

All Java artifacts will implement the following Maven build plugins and reporting.

1. Code Formatting.

Use the maven build plugin formatter-maven-plugin configured to use **JavaLambdaFormat** style. This style configuration is provied by the **admitone:code-quality-config** resource plugin.

Standard setup for formatting code is

```
    <!-- Automatically reformat source code to Java Lambda Format -->
    <plugin>
        <groupId>net.revelc.code.formatter</groupId>
        <artifactId>formatter-maven-plugin</artifactId>
        <version>2.18.0</version>
        <executions>
            <execution>
                <goals>
                    <goal>format</goal>
                </goals>
                <configuration>
                    <configFile>eclipse/JavaLambdaFormat.xml</configFile>
                </configuration>
            </execution>
        </executions>
        <configuration>
            <encoding>UTF-8</encoding>
            <excludes>
                <exclude>**/*Test.java</exclude>
                <exclude>**/*_Spec.java</exclude>
            </excludes>
        </configuration>
        <dependencies>
            <dependency>
                <groupId>admitone</groupId>
                <artifactId>code-quality-config</artifactId>
                <version>2022.03.01</version>
            </dependency>
        </dependencies>
    </plugin>
```

2. Setting up Checkstyle

Checkstyle will help educate and enforce our coding standards. You can set up your IDE to use Checkstyle to examine code for conformance to the standards. Learn more about the checks or Google the error message to find out why it complains about certain things.

```
    <!-- Verify code matches the AdmitOne Rout66 Java Lambda Function
Requirements-->
    <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-checkstyle-plugin</artifactId>
        <version>3.1.2</version>
        <configuration>
            <configLocation>checkstyle.xml</configLocation>
        </configuration>
        <dependencies>
            <dependency>
                <groupId>com.puppycrawl.tools</groupId>
                <artifactId>checkstyle</artifactId>
                <version>10.0</version>
            </dependency>

            <dependency>
                <groupId>admitone</groupId>
                <artifactId>code-quality-config</artifactId>
                <version>2022.03.01</version>
            </dependency>
        </dependencies>
    </plugin>
```

3. SonarLint and SonarQube with SonarSannar

Execute the SonarLint and SonarQube analysis via a regular Maven SonarScanner.

TODO: add setup here

4. OWASP Dependency Checking

*TODO*_: add setup here

OWASP Dependency-Check

Eclipse

TODO

Jenkins

URL - http://admitone.ci:50080/jenkins/

AdmitOne Repository

TODO:

URL - http://admitone.repo:52180/

1.2 Coding Conventions

Java Code Conventions

Introduction

Why Have Code Conventions?

Code conventions are important to programmers for a number of reasons:

- 80% of the lifetime cost of a piece of software goes to maintenance.
- Hardly any software is maintained for its whole life by the original author.
- Code conventions improve the readability of the software, allowing engineers to understand new code more quickly and thoroughly.
- If you ship your source code as a product, you need to make sure it is as well packaged and clean as any other product you create.

Coding Standards

All projects and artifacts with in the AdmitOne Project must adhere to Clean Code principles, built on SOLID OOP principles and Design Patterns. Design and build tools will be used to ensure designs and code adhere to standards (See Development Environment).

File Organization

A file consists of sections that should be separated by blank lines and an optional comment identifying each section.

Source files are encoded in UTF-8

Files longer than 2000 lines are cumbersome and should be avoided.

Java Source Files

Each Java source file contains a single public class or interface. When private classes and interfaces are associated with a public class, you can put them in the same source file as the public class. The public class should be the first class or interface in the file.

Java source files have the following ordering:

- Beginning comments

All source files should begin with a c-style comment that lists the programmer(s), the date, a copyright notice, and also a brief description of the purpose of the program.

For example:

```
/*
 * Classname
 *
 *
 * Copyright notice
 */
```

- Package and Import statements; for example:

The first non-comment line of most Java source files is a package statement. After that, import statements can follow.

Import statements must be grouped with associated packages together and one blank line between groups

Imported classes should always be listed explicitly

For example:

```
package java.awt;
import java.awt.peer.CanvasPeer;
import java.applet.Applet;
```

- Class or interface declarations

	Part of Class/Interface Declaration	Notes
1	Class/interface documentation comment (`/*...*/`)	
2	class or interface statement	
3	Class/interface implementation comment (`/*...*/`), if necessary	This comment should contain any class-wide or interface-wide information that wasn't appropriate for the class/interface documentation comment.
4	Class (static) variables	First the public class variables, then the protected, and then the private.
5	Instance variables	First public, then protected, and then private.
6	Constructors	
7	Methods	These methods should be grouped by functionality rather than by scope or accessibility. For example, a private class method can be in between two public instance methods. The goal is to make reading and understanding the code easier.

Indentation

Four spaces should be used as the unit of indentation. The exact construction of the indentation is 4 spaces.

Special characters like TAB and page break should be avoided

1. Line Length

Avoid lines longer than 120 characters, since they're not handled well by many terminals and tools.

2. Wrapping Lines

When an expression will not fit on a single line, break it according to these general principles:

• Break after a comma.

• Break before an operator.

• For arithmetic expressions, avoid breaking within a set of parentheses.

• Align the new line with the beginning of the expression at the same level on the previous line.

• If the above rules lead to confusing code or to code that's squished up against the right margin, just indent 8 spaces instead.

Here are some examples of breaking method calls:

```
function (longExpression1, longExpression2, longExpression3,
longExpression4, longExpression5);

aVar = function1(longExpression1,
          function2 (longExpression2, longExpression3));
```

Following are two examples of breaking an arithmetic expression. The first is preferred, since the break occurs outside the parenthesized expression, which is at a higher level.

```
Name1 = Name2 * (Name3 + Name4 - Name5)
             + (4 * name6); // PREFER

Name1 = Name2 * (Name3 + Name4
            - Name5) + 4 * name6; // AVOID
```

Following are two examples of indenting method declarations. The first is the conventional case. The second would shift the second and third lines to the far right if it used conventional indentation, so instead it indents only 8 spaces.

```
//CONVENTIONAL INDENTATION

someMethod(int anArg, Object anotherArg, String yetAnotherArg,
              Object andStillAnother) {
    ...

}



//INDENT 8 SPACES TO AVOID VERY DEEP INDENTS

private static synchronized horkingLongMethodName(int anArg,
        Object anotherArg, String yetAnotherArg,

Object andStillAnother) {
    ...

}
```

Here are three acceptable ways to format ternary expressions:

```
alpha = (aLongBooleanExpression) ? beta : gamma;

alpha = (aLongBooleanExpression) ? beta
    : gamma;

alpha = (aLongBooleanExpression)
    ? beta
    : gamma;
```

Comments

Java programs can have two kinds of comments: implementation comments and documentation comments. Implementation comments are those found in C++, which are delimited by /.../, and //. Documentation comments (known as "doc comments") are

Java-only, and are delimited by `/**...*/`. Doc comments can be extracted to HTML files using the javadoc tool.

Implementation comments are meant for commenting out code or for comments about the particular implementation. Doc comments are meant to describe the specification of the code, from an implementation-free perspective to be read by developers who might not necessarily have the source code at hand.

Comments should be used to give overviews of code and provide additional information that is not readily available in the code itself. Comments should contain only information that is relevant to reading and understanding the program. For example, information about how the corresponding package is built or in what directory it resides should not be included as a comment.

Avoid duplicating information that is present in (and clear from) the code. It is too easy for redundant comments to get out of date. In general, avoid any comments that are likely to get out of date as the code evolves.

Note: The frequency of comments sometimes reflects poor quality of code. When you feel compelled to add a comment, consider rewriting the code to make it clearer.

Comments should not be enclosed in large boxes drawn with asterisks or other characters.

Comments should never include special characters such as form-feed and backspace.

Implementation Comment Formats

Programs can have four styles of implementation comments: block, single-line, trailing and end-of-line.

Block Comments

Block comments are used to provide descriptions of files, methods, data structures and algorithms.

Block comments should be used at the beginning of each file and before each method. They can also be used in other places, such as within methods.

Block comments inside a function or method should be indented to the same level as the code they describe.

A block comment should be preceded by a blank line to set it apart from the rest of the code.

Block comments have an asterisk "`*`" at the beginning of each line except the first.

```
/*
 * Here is a block comment.
 */
```

Single-Line Comments

Short comments can appear on a single line indented to the level of the code that follows. If a comment can't be written in a single line, it should follow the block comment format. A single-line comment should be preceded by a blank line. Here's an example of a single-line comment in Java code

```
if (condition) {
    //Handle the condition.
    // we can use single line comment if it is a one line comment
    ...
}
```

Documentation Comments

Doc comments describe Java classes, interfaces, constructors, methods, and fields. Each doc comment is set inside the comment delimiters `/**...*/`, with one comment per class, interface, or member. This comment should appear just before the declaration:

```
/**
 * The Example class provides ...
 */
```

```
public class Example { ...
```

Notice that top-level classes and interfaces are not indented, while their members are. The first line of doc comment (`/**`) for classes and interfaces is not indented; subsequent doc comment lines each have 1 space of indentation (to vertically align the asterisks). Members, including constructors, have 4 spaces for the first doc comment line and 5 spaces thereafter.

If you need to give information about a class, interface, variable, or method that isn't appropriate for documentation, use an implementation block comment or single-line comment immediately after the declaration. For example, details about the implementation of a class should go in in such an implementation block comment following the class statement, not in the class doc comment.

Doc comments should not be positioned inside a method or constructor definition block, because Java associates documentation comments with the first declaration after the comment.

- All classes are to include a comment block that describing the basic purpose of the class. (This is also a good place to put any overarching TODO statements).
- All public methods need to include Java doc comments except for accessors.
    - Parameters are to be included, but do not require documentation unless it is something meaningful i.e. avoid * @param name The name
    - Return statements are to be included and documented.
    - Thrown exceptions may be included, but do not need to be documented.
- Protected / Private methods should include java doc comments when they are not easily understood. This is up to developer / reviewer discretion.

Declarations

Number Per Line

One declaration per line. In other words,

```
int level = 0;
int size = 0;
int level, size; // Avoid
int foo,  fooarray[]; //WRONG!
```

Initialization

Try to initialize local variables where they're declared. The only reason not to initialize a variable where it's declared is if the initial value depends on some computation occurring first.

Placement

Local variables are not habitually declared at the start of their containing block or block-like construct. Instead, local variables are declared close to the point they are first used (within reason), to minimize their scope. Local variable declarations typically have initializers, or are initialized immediately after declaration.

Avoid local declarations that hide declarations at higher levels. For example, do not declare the same variable name in an inner block:

```
int count;
...
myMethod() {
    if (condition) {
        int count = 0;     // AVOID!
        ...
    }
    ...
}
```

Class and Interface Declarations

When coding Java classes and interfaces, the following formatting rules should be followed:

No space between a method name and the parenthesis "(" starting its parameter list

Open brace "{" appears at the end of the same line as the declaration statement

Closing brace "}" starts a line by itself indented to match its corresponding opening statement, except when it is a null statement the "}" should appear immediately after the "{"

```
class Sample extends Object {
    int ivar1;
    int ivar2;

    Sample(int i, int j) {
        ivar1 = i;
        ivar2 = j;
    }

    int emptyMethod() {}
    ...
}
```

Methods are separated by a blank line

Annotations

Annotations applying to a class, method or constructor appear immediately after the documentation block, and each annotation is listed on a line of its own (that is, one annotation per line). These line breaks do not constitute line-wrapping so the indentation level is not increased

```
@Override
@Nullable
public String getNameIfPresent() { ... }
```

Statements

Simple Statements

Each line should contain at most one statement. Example:

```
argv++;         // Correct
argc--;         // Correct
argv++; argc--; // AVOID!
```

Compound Statements

Compound statements are statements that contain lists of statements enclosed in braces "{ statements }". See the following sections for examples.

The enclosed statements should be indented one more level than the compound statement.

The opening brace should be at the end of the line that begins the compound statement; the closing brace should begin a line and be indented to the beginning of the compound statement.

Braces are used around all statements, even single statements, when they are part of a control structure, such as an if-else or for statement. This makes it easier to add statements without accidentally introducing bugs due to forgetting to add braces.

"return" Statements

A return statement with a value should not use parentheses unless they make the return value more obvious in some way. Example:

```
return;
return myDisk.size();
return (size ? size : defaultSize);
```

if, if-else, if else-if else Statements

The if-else class of statements should have the following form:

```
if (condition) {
    statements;
}

if (condition) {
    statements;
} else {
    statements;
}

if (condition) {
    statements;
} else if (condition) {
    statements;
} else {
    statements;
}
```

Note: if statements always use braces, {}. Avoid the following error-prone form:

```
if (condition) //AVOID! THIS OMITS THE BRACES {}!
    statement;
```

`for` Statements

A for statement should have the following form:

```
for (initialization; condition; update) {
    statements;
}
```

An empty for statement (one in which all the work is done in the initialization, condition, and update clauses) should have the following form:

```
for (initialization; condition; update);
```

When using the comma operator in the initialization or update clause of a for statement, avoid the complexity of using more than three variables. If needed, use separate statements before the for loop (for the initialization clause) or at the end of the loop (for the update clause).

`while` statements

A while statement should have the following form:

```
while (condition) {
    statements;
}
```

An empty while statement should have the following form:

```
while (condition);
```

`do-while` statement

A do-while statement should have the following form:

```
do {
    statements;
} while (condition);
```

`switch` statements

A switch statement should have the following form:

```
switch (condition) {
case ABC:
    statements;
    /* falls through */
case DEF:
    statements;
    break;
case XYZ:
    statements;
    break;
default:
    statements;
    break;
}
```

Every time a case falls through (doesn't include a break statement), add a comment where the break statement would normally be. This is shown in the preceding code example with the `/* falls through */` comment.

Every switch statement should include a default case. The break in the default case is redundant, but it prevents a fall-through error if later another case is added.

try-catch Statements

A try-catch statement should have the following format:

```
try {
    statements;
} catch (ExceptionClass e) {
     statements;
}
```

A try-catch statement may also be followed by finally, which executes regardless of whether or not the try block has completed successfully.

```
try {
    statements;
} catch (ExceptionClass e) {
    statements;
} finally {
    statements;
}
```

`for` collection loop

A for collection loop should have following format

```
for (Object obj : objList) {
    …
}
```

White Space

Blank Lines

Blank lines improve readability by setting off sections of code that are logically related.

Two blank lines should always be used in the following circumstances:

- Between sections of a source file
- Between class and interface definitions

One blank line should always be used in the following circumstances:

- Between methods
- Between the local variables in a method and its first statement
- Before a block or single-line comment
- Between logical sections inside a method to improve readability

Blank Spaces

Blank spaces should be used in the following circumstances:

- A keyword followed by a parenthesis should be separated by a space. Example:

```
      while (true) {
           ...
       }

     while  ( true ) {   // Avoid space between ( and true.
       }
```

Note that a blank space should not be used between a method name and its opening parenthesis. This helps to distinguish keywords from method calls.

- A blank space should appear after commas in argument lists.
- All binary operators except . should be separated from their operands by spaces. Blank spaces should never separate unary operators such as unary minus, increment ("++"), and decrement ("--") from their operands. Example:

```
    a += c + d;
    a = (a + b) / (c * d);

    while (d++ = s++) {
        n++;
    }
    printSize("size is " + foo + "\n");
```

- The expressions in a for statement should be separated by blank spaces. Example:

```
for (expr1; expr2; expr3)
```

- Casts should be followed by a blank space. Examples:

```
myMethod((byte) aNum, (Object) x);
    myMethod((int) (cp + 5), ((int) (i + 3))
                          + 1);
```

Naming Conventions

Naming conventions make programs more understandable by making them easier to read. They can also give information about the function of the identifier-for example, whether it's a constant, package, or class-which can be helpful in understanding the code.

Identifier Type	Rules of Naming	Examples
Packages	The prefix of a unique package name is always written in all-lowercase ASCII letters and should be one of the top-level domain names, currently com, edu, gov, mil, net, org, or one of the English two-letter codes identifying countries as specified in ISO Standard 3166, 1981. Subsequent components of the package name vary according to an organization's own internal naming conventions. Such conventions might specify that certain directory name components be division, department, project, machine, or login names.	com.sun.eng com.apple.quicktime.v2 edu.cmu.cs.bovik.cheese

Classes	Class names should be nouns, in mixed case with the first letter of each internal word capitalized. Try to keep your class names simple and descriptive. Use whole words-avoid acronyms and abbreviations (unless the abbreviation is much more widely used than the long form, such as URL or HTML). Implementation class should have suffix Impl	class Raster; class ImageSprite;
Interfaces	Interface names should be capitalized like class names.	interface RasterDelegate; interface Storing;
Methods	Methods should be verbs, in Camel Case The term compute can be used where something is computed The term find can be used when we look up something Is prefix is used for Boolean getter and setters Acronyms such as XML and HTTP should be treated as words, such as getHttp(), getXml(). Not getXML() or getHTTP()	run(); runFast(); getBackground();
Variables	Except for variables, all instance, class, and class constants are in mixed case with a lowercase first letter. Internal words start with capital letters. Variable names should not start with underscore _ or dollar sign $ characters, even though both are allowed. Variable names should be meaningful. The choice of a variable name should be mnemonic- that is, designed to indicate to the casual observer the intent of its use. One-character variable names should be avoided except for loop control variables. Common names for temporary variables are i, j, k, m, and n. Boolean variable names should not be negative, for instance, isNotLoaded, unAttached.	int i; char c; float myWidth;

Constants	The names of variables declared class constants and of ANSI constants should be all uppercase with words separated by underscores ("_").	`static final int MIN_WIDTH = 4;`
		`static final int MAX_WIDTH = 999;`
		`static final int`

```
GET_THE_CPU =
1;
```

Types

Type conversions must always be done explicitly never rely on implicit type conversion

```
    floatValue = (float) intValue // NOT floatValue = intValue
```

Array specifiers must be attached to the type not the variable

```
    int [] a =  new int[20] //NOT int a[] = new int[20]
```

Programming Practices

Providing Access to Instance and Class Variables

Don't make any instance or class variable public without good reason. Just because it’s a class level variable doesn’t mean it needs a getter and setter.

One example of appropriate public instance variables is the case where the class is essentially a data structure, with no behavior

Referring to Class Variables and Methods

Avoid using an object to access a class (static) variable or method. Use a class name instead. For example:

```
classMethod();             //OK
AClass.classMethod();      //OK
anObject.classMethod();    //AVOID!
```

Constants

Numerical constants (literals) should not be coded directly, except for -1, 0, and 1, which can appear in a for loop as counter values.

String constants should be used except for "" or null, some exceptions would be logging and errors.

Variable Assignments

- Avoid assigning several variables to the same value in a single statement. It is hard to read. Example:

```
fooBar.fChar = barFoo.lchar = 'c'; // AVOID!
```

- Do not use embedded assignments in an attempt to improve run-time performance. This is the job of the compiler. Example:

```
d = (a = b + c) + r;        // AVOID!
```

should be written as

```
a = b + c;
d = a + r;
```

Miscellaneous Practices

- Parentheses

It is generally a good idea to use parentheses liberally in expressions involving mixed operators to avoid operator precedence problems. Even if the operator precedence seems clear to you, it might not be to others-you shouldn't assume that other programmers know precedence as well as you do.

```
if (a == b && c == d)     // AVOID!

if ((a == b) && (c == d)) // RIGHT
```

- Returning Values

Try to make the structure of your program match the intent. Example:

```
if (booleanExpression) {
    return true;

} else {
    return false;
}
```

should instead be written as

```
boolean isOkay = complexExpression;
return  isOkay;
```

or

```
return simpleExpression;
```

- Expressions before `?` in the Conditional Operator

If an expression containing a binary operator appears before the ? in the ternary ?: operator, it should be parenthesized. Example:

```
(x >= 0) ? x : -x;
```

! operator and conditional expressions

- There is nothing inherently wrong with the not operator (!). Just be aware that it might be harder to seen when scanning code. Also be careful of double negatives `if (!StringUtils.isNotEmpty(val))`. When in doubt read your if statement aloud and consider if you highschool English teacher would approve.

```
    if (!name.startsWith("Rich")) {  // OK

    }

    if (isUser() == false) {    // If you feel this is more readable...

    }
```

@Override: always used

A method is marked with the @Override annotation whenever it is legal. This includes a class method overriding a superclass method, a class method implementing an interface method, and an interface method respecifying a superinterface method.

Exception:@Override may be omitted when the parent method is @Deprecated.

Caught exceptions: not ignored

Except as noted below, it is very rarely correct to do nothing in response to a caught exception. Exceptions should usually be at least logged at debug or trace level.

When it truly is appropriate to take no action whatsoever in a catch block, the reason this is justified is explained in a comment.

```
try {
  int i = Integer.parseInt(response);
  return handleNumericResponse(i);
} catch (NumberFormatException ok) {
  // it's not numeric; that's fine, just continue
}
return handleTextResponse(response);
```

Exception: In tests, a caught exception may be ignored without comment if it is named expected. The following is a very common idiom for ensuring that the method under test does throw an exception of the expected type, so a comment is unnecessary here.

```
try {
  emptyStack.pop();
  fail();
} catch (NoSuchElementException expected) {
}
```

Logging

- Each project/application should choose and enforce usage of a standard logging library (ex. Log4j)
- Each project/application should document when each logging level should be used
- All exceptions should at least be logged (excluding "expected" exceptions like some parsing exceptions)
- Exceptions should be chained except for good reasons

```
         catch (NullPointerException npe){
             throw new ApiException(npe);
         }
```

- When adding temporary logging for trouble shooting, add a // TODO remove temporary logs comment by set of log messages that should be removed.
- Include relevant data that may have caused the error in the log message. For example, "User id not found: -1" is better than “User id not found.”

Security

Consult these resources for guidance on secure coding practices, particularly CERT standard. Some stylistic considerations are mentioned further on.

- CERT Oracle Coding Standard for Java
- OWASP SCP Quick Reference v2

Readable code is always more secure than unreadable because it is more easily reasoned about and the reader can more easily verify the author’s original intent. Take this example below … it only takes a single typo in the //AVOID section to change a security check into its logical opposite (and likely grant access where it should be denied).. While this is also true for the check in the middle … it is more likely you will notice a == versus != instead of (isUser()) vs (!isUser())

Exercise extreme caution with switch statements, especially fall-through statements. If you must use fall-through, please re-think your design approach. If, after much consideration, you still need fall-through, be judicious and deliberate to mark ALL fall-through points with a comment as indicated earlier in this guide.

See switch statement.

Security checks should be located and maintained in a single place. Be sure to apply "Don't Repeat Yourself" (DRY) principal frequently and comprehensively to all security check logic throughout the application.

Generics

When to Use Generics

- With Collections. The Java Collections framework is excellent and should be used instead of Arrays.
- If you find yourself doing a lot (or perhaps even a little) casting of types you might want to consider Generics
- If you are using variables of type Object to store values form various classes, you might want to use Generics instead.

Naming Convention for Generic types in a class definition

Oracle recommends the following naming convention:

- E - Element (used extensively by the Java Collections Framework)
- K - Key
- N - Number
- T - Type
- V - Value
- S,U,V etc. - 2nd, 3rd, 4th types

public class Box { // T stands for "Type" private T t;

```
public void set(T t) { this.t = t; }
public T get() { return t; }
```

}

Generic Type Inference

When declaring variables using generics, it is not necessary to list the generic data type twice.

```
Box<Integer> integerBox = new Box<>();  // Recommended

Box<Integer> integerBox = new Box<Integer>();  // AVOID
```

Java 8

If you are interested in a learning more about Java 8. A good book covering all of the new features is:

https://www.manning.com/books/java-8-in-action

Streams

- Do not use Parallel Streams unless you have tested to verify that there is actually a speed improvement. For small data sets, Parallel streams add a lot of overhead.
- Formatting for chained stream functions should have each step in the chain on a separate line indented four spaces from the original line of code. For example:

```
List<String> myList = Arrays.asList("a1", "a2", "b1", "c2", "c1");

List<String> newList = myList.stream()

    .filter(s -> s.startsWith("c"))
    .map(String::toUpperCase)

.sorted()
    .toList();
```

Optional Type

There is a good discussion of the new Java 8 Optional data type here:

http://www.oracle.com/technetwork/articles/java/java8-optional-2175753.html

Use of the Optional data type should be discussed by the individual team and an approach selected. The approaches may differ for new code bases vs. legacy (Java 7) code that does specific null checks.

The "pure" view of the Optional data type is that if a method returns an object that might be null, then the return type should be Optional. However, a method that is never supposed to return null should return OriginalDataType. The idea is that the code is semantic. If the developer of the method wants you to be concerned with missing values they signal that by using the Optional data type. If they don’t, then they are indicating that the method is always supposed to return a value and therefore you shouldn’t need to do an explicit Null check.

Lambda Expressions

- When a Method reference is available, it should be used instead of a lambda expression Bad Example: List newList = myList.stream() .map(s -> s.toUpperCase()) .toList();

Good Example: List newList = myList.stream() .map(String::toUpperCase) .toList();

- The variables defined by lambda expressions are typically single character variables.
- Complex lambda expressions should be written as methods or predicates.

Date & Time Objects

The java.util.Date class has been deprecated for a very long time, but until Java 8 has continued to be used because the alternatives were not very good. With Java 8, this has changed and the new java.time.LocalDate and java.time.LocalTime and related classes. Unless you are dealing with legacy code that requires java.util.Date, do NOT use it anymore. Use the new classes.

```
import java.util.Date;
import java.time.LocalDate;

LocalDate myDate1 = new LocalDate();  // Use the java.time classes

Date myDate2 = new Date();  // AVOID
```

References

- https://google.github.io/styleguide/javaguide.html
- http://www.oracle.com/technetwork/java/codeconventions-150003.pdf

1.3 Method Naming Conventions

`/1.0 Conventions/1.3 Method Naming Conventions`

Overview

The following coding standards were collected from various technologies, frameworks and articles that form a collection of best practices for method names and what the purpose and/or implementation is expected.

Each of the identifiers presented by this convention are clauses used in whole or as part of an identifier. For example, 'as' can be used a a stand alone function call or part of a method name:

```
     // Method that converts parameter and returns integer
     Integer i = r.as(Integer.class);

     // Method that returns a value of a integer type.
     Integer i = r.asInteger();
```

Glossary Of Identifiers

As

Return a representation of an instance as a different type.

This clause identifies a method that converts the object to a different type.

Example

```
     var exam1 = Mabye.of(userRepo.findUserById(userId));
     var user = exam1.as(UserDetail.class).getName(); // Cast exam1 to a
```

```
UserDetail and get the users name.
    var str = userId.asString() // Return id value as a string.
```

Find

Return a subset of items matching criteria.

Method searches a collection or database for a specific instance(s) of a item.
The return type should be a Collection, Maybe or Optional.

The *find* clause should follow the Spring Data naming convention. See Spring Data .

Example

Prototype	Description
List findAll()	Return all instance of User
Optional findUserById(Long id)	Find a specific user with the given id
List findUserLikeLastName(String name)	Find all users with last name like name
List findUserByLastNameAndFirstName(..)	Find all users with last name and first name

From

Method that accept a reference to an Object as a parameter and returns a class with a value extracted from object.

Example

```
    var comp = Maybe.from(anOptional); // Creates a Maybe instance from the
value of Optional.
```

Get

Getter method to retrieve the value of a property.

The implementation of a getter should be simple and short; preferably written to qualify for Method Inlining in the JVM.

Of

Method that accepts a reference to an Object as a parameter and returns a wrapper class for the Object.

Example

```
    var response = Maybe.of(resp); // Creates a Maybe instance containing
the value of resp.
```

When

If the predicate is true then return the reference to this, otherwise, return a default instance.

Bisects method chaining depending on the result of a Predicate lambda function.
The return value will be one of two possible values depending on the *truth* returned by the Predicate. The when clause is frequently coupled with the *then* clause.

Example

```
    var someInt = Maybe.of(50); // Create a Maybe<Integer> with the value 50
    var result1 = someInt.when(p -> p > 25).then(p -> p * 10); // result1
will be 500
    var result2 = someInt.when(p -> p <=25).then(p -> p / 10); // result1
will be None<Integer>
```

With

The *with* clause is used to make a call to a lambda that is a {@link Consumer} instance that will receive an instance of an object that the lambda function is expected to build. If the method name is *with* as a stand alone function then the object passed is usually the instance of the parent class.

This clause forms the DSL semantics is "with(x) do (y)".

Via

The *via* clause is used to make a call to a lambda that is a {@link Consumer} instance that will receive an instance of the object.

Project Layout

`/1.0 Conventions/Project Layout`

Package Structures