Table of Contents

Preface

We are pleased to present **Alibaba Java Coding Guidelines**, which consolidates the best programming practices from Alibaba Group's technical teams. A vast number of Java programming teams impose demanding requirements on code quality across projects as we encourage reuse and better understanding of each other's programs. We have seen many programming problems in the past. For example, defective database table structures and index designs may cause software architecture flaws and performance risks. As another example, confusing code structures make it difficult to maintain. Furthermore, vulnerable code without authentication is prone to hackers' attacks. To address those kinds of problems, we developed this document for Java developers in Alibaba.

This document consists of five parts: ***Programming Specification***, ***Exception and Logs***, ***MySQL Specification***, ***Project Specification*** and ***Security Specification***. Based on the severity of the concerns, each specification is classified into three levels: ***Mandatory***, ***Recommended*** and ***Reference***. Further clarification is expressed in: (1) "**Description**", which explains the content;

(2) "**Positive examples**", which describe recommended coding and implementation approaches;
(3) "**Counter examples**", which describe precautions and actual error cases.

The main purpose of this document is to help developers improve code quality. As a result, developers can minimize potential and repetitive code errors. In addition, specification is essential to modern software architectures, which enable effective collaborations. As an analogy, traffic regulations are intended to protect public safety rather than to deprive the rights of driving. It is easy to imagine the chaos of traffic without speed limits and traffic lights. Instead of destroying the creativity and elegance of program, the purpose of developing appropriate specification and standards of software is to improve the efficiency of collaboration by limiting excessive personalization.

We will continue to collect feedback from the community to improve Alibaba Java Coding Guidelines.

1. Programming Specification
Naming Conventions
1\. **[Mandatory]** Names should not start or end with an underline or a dollar sign.
> Counter example: _name / __name / \$Object / name_ / name\$ / Object\$

2\. **[Mandatory]** Using Chinese, Pinyin, or Pinyin-English mixed spelling in naming is strictly prohibited. Accurate English spelling and grammar will make the code readable, understandable, and maintainable.
> Positive example: alibaba / taobao / youku / Hangzhou. In these cases, Chinese proper names in Pinyin are acceptable.

3\. **[Mandatory]** Class names should be nouns in UpperCamelCase except domain models: DO, BO, DTO, VO, etc.
> Positive example: MarcoPolo / UserDO / HtmlDTO / XmlService / TcpUdpDeal / TaPromotion
> Counter example: marcoPolo / UserDo / HTMLDto / XMLService / TCPUDPDeal / TAPromotion

4\. **[Mandatory]** Method names, parameter names, member variable names, and local variable names should be written in lowerCamelCase.
> Positive example: localValue / getHttpMessage() / inputUserId

5\. **[Mandatory]** Constant variable names should be written in upper characters separated by underscores. These names should be semantically complete and clear.
> Positive example: MAX_STOCK_COUNT

> Counter example: MAX_COUNT

6\. **[Mandatory]** Abstract class names must start with *Abstract* or *Base*. Exception class names must end with *Exception*. Test case names shall start with the class names to be tested and end with *Test*.

7\. **[Mandatory]** Brackets are a part of an Array type. The definition could be: *String[] args;*
> Counter example: *String args[];*

8\. **[Mandatory]** Do not add 'is' as prefix while defining Boolean variable, since it may cause a serialization exception in some Java frameworks.
> Counter example: *boolean isSuccess;* The method name will be `isSuccess()` and then RPC framework will deduce the variable name as 'success', resulting in a serialization error since it cannot find the correct attribute.

9\. **[Mandatory]** A package should be named in lowercase characters. There should be only one English word after each dot. Package names are always in singular format while class names can be in plural format if necessary.
> Positive example: `com.alibaba.open.util` can be used as a package name for utils;
`MessageUtils` can be used as a class name.

10\. **[Mandatory]** Uncommon abbreviations should be avoided for the sake of legibility.
> Counter example: AbsClass (AbstractClass); condi (Condition)

11\. **[Recommended]** The pattern name is recommended to be included in the class name if any design pattern is used.
> Positive example: `public class OrderFactory;  `
`public class LoginProxy;` `public class ResourceObserver;`
> Note: Including corresponding pattern names helps readers understand ideas in design patterns quickly.

12\. **[Recommended]** Do not add any modifier, including `public`, to methods in interface classes for coding simplicity. Please add valid *Javadoc* comments for methods. Do not define any variables in the interface except for the common constants of the application.
> Positive example: method definition in the interface: `void f(); `
constant definition: `String COMPANY = "alibaba";`
> Note: In JDK8 it is allowed to define a default implementation for interface methods, which is valuable for all implemented classes.

13\. There are two main rules for interface and corresponding implementation class naming:

  1) **[Mandatory]** All *Service* and *DAO* classes must be interfaces based on SOA principle. Implementation class names should end with *Impl*.
  > Positive example: `CacheServiceImpl` to implement `CacheService`.

  2) **[Recommended]** If the interface name is to indicate the ability of the interface, then its name should be an adjective.
  > Positive example: `AbstractTranslator` to implement `Translatable`.

14\. **[For Reference]** An Enumeration class name should end with *Enum*. Its members should be spelled out in upper case words, separated by underlines.
 > Note: Enumeration is indeed a special constant class and all constructor methods are private by default.
 > Positive example: Enumeration name: `DealStatusEnum`; Member name: `SUCCESS  / UNKOWN_REASON`.

15\. **[For Reference]** Naming conventions for different package layers:
  A) Naming conventions for Service/DAO layer methods
    1) Use `get` as name prefix for a method to get a single object.
    2) Use `list` as name prefix for a method to get multiple objects.
    3) Use `count` as name prefix for a statistical method.
    4) Use `insert` or `save` (recommended) as name prefix for a method to save data.
    5) Use `delete` or `remove` (recommended) as name prefix for a method to remove data.
    6) Use `update` as name prefix for a method to update data.
  B) Naming conventions for Domain models
    1) Data Object: *DO, where * is the table name.
    2) Data Transfer Object: *DTO, where * is a domain-related name.
    3) Value Object: *VO, where * is a website name in most cases.
    4) POJO generally point to DO/DTO/BO/VO but cannot be used in naming as *POJO.

Constant Conventions
1\. **[Mandatory]** Magic values, except for predefined, are forbidden in coding.
> Counter example: String key = "Id#taobao_" + tradeId;

2\. **[Mandatory]** 'L' instead of 'l' should be used for long or Long variable because 'l' is easily to be regarded as number 1 in mistake.
> Counter example: `Long a = 2l`, it is hard to tell whether it is number 21 or Long 2.

3\. **[Recommended]** Constants should be placed in different constant classes based on their functions. For example, cache related constants could be put in `CacheConsts` while configuration related constants could be kept in `ConfigConsts`.
 > Note: It is difficult to find one constant in one big complete constant class.

4\. **[Recommended]** Constants can be shared in the following 5 different layers: *shared in multiple applications; shared inside an application; shared in a sub-project; shared in a package; shared in a class*.
  1) Shared in multiple applications: keep in a library, under `constant` directory in client.jar;
  2) Shared in an application: keep in shared modules within the application, under `constant` directory;
  Counter example: Obvious variable names should also be defined as common shared constants in an application. The following definitions caused an exception in the production environment: it returns *false*, but is expected to return *true* for `A.YES.equals(B.YES)`.
  Definition in Class A: `public static final String YES = "yes";`
  Definition in Class B: `public static final String YES = "y";`
  3) Shared in a sub-project: placed under `constant` directory in the current project;
  4) Shared in a package: placed under `constant` directory in current package;
  5) Shared in a class: defined as 'private static final' inside class.

5\. **[Recommended]** Use an enumeration class if values lie in a fixed range or if the variable has attributes. The following example shows that extra information (which day it is) can be included in enumeration:
 > Positive example: public Enum{ MONDAY(1), TUESDAY(2), WEDNESDAY(3), THURSDAY(4), FRIDAY(5), SATURDAY(6), SUNDAY(7);}

Formatting Style
1\. **[Mandatory]** Rules for braces. If there is no content, simply use *{}* in the same line. Otherwise:
  1) No line break before the opening brace.
  2) Line break after the opening brace.
  3) Line break before the closing brace.
  4) Line break after the closing brace, *only if* the brace terminates a statement or terminates a method body, constructor or named class. There is *no* line break after the closing brace if it is followed by `else` or a comma.

2\. **[Mandatory]** No space is used between the '(' character and its following character. Same for the ')' character and its preceding character. Refer to the *Positive Example* at the 5th rule.

3\. **[Mandatory]** There must be one space between keywords, such as if/for/while/switch, and parentheses.

4\. **[Mandatory]** There must be one space at both left and right side of operators, such as '=', '&&', '+', '-', *ternary operator*, etc.

5\. **[Mandatory]** Each time a new block or block-like construct is opened, the indent increases by four spaces. When the block ends, the indent returns to the previous indent level. Tab characters are not used for indentation.
> Note: To prevent tab characters from being used for indentation, you must configure your IDE. For example, "Use tab character" should be unchecked in IDEA, "insert spaces for tabs" should be checked in Eclipse.
> Positive example:
```java
public static void main(String[] args) {
    // four spaces indent
    String say = "hello";
    // one space before and after the operator
    int flag = 0;
    // one space between 'if' and '(';
    // no space between '(' and 'flag' or between '0' and ')'
    if (flag == 0) {
        System.out.println(say);
    }
    // one space before '{' and line break after '{'
    if (flag == 1) {
        System.out.println("world");
    // line break before '}' but not after '}' if it is followed by 'else'
    } else {
        System.out.println("ok");
    // line break after '}' if it is the end of the block
    }
}
```

6\. **[Mandatory]** Java code has a column limit of 120 characters. Except import statements, any line that would exceed this limit must be line-wrapped as follows:
  1) The second line should be intented at 4 spaces with respect to the first one. The third line and following ones should align with the second line.
  2) Operators should be moved to the next line together with following context.
  3) Character '.' should be moved to the next line together with the method after it.
  4) If there are multiple parameters that extend over the maximum length, a line break should be inserted after a comma.

  5) No line breaks should appear before parentheses.
> Positive example:
```java
StringBuffer sb = new StringBuffer();
// line break if there are more than 120 characters, and 4 spaces indent at
// the second line. Make sure character '.' moved to the next line
// together.  The third and fourth lines are aligned with the second one.
sb.append("zi").append("xin").
    .append("huang")...
    .append("huang")...
    .append("huang");
```
> Counter example:
```java
StringBuffer sb = new StringBuffer();
// no line break before '('
sb.append("zi").append("xin")...append
    ("huang");
// no line break before ',' if there are multiple params
invoke(args1, args2, args3, ...
    , argsX);
```

7\. **[Mandatory]** There must be one space between a comma and the next parameter for methods with multiple parameters.
> Positive example: One space is used after the *','* character in the following method definition.
```java
f("a", "b", "c");
```

8\. **[Mandatory]** The charset encoding of text files should be *UTF-8* and the characters of line breaks should be in *Unix* format, instead of *Windows* format.

9\. **[Recommended]** It is unnecessary to align variables by several spaces.
> Positive example:
```java
int a = 3;
long b = 4L;
float c = 5F;
StringBuffer sb = new StringBuffer();
```
> Note: It is cumbersome to insert several spaces to align the variables above.

10\. **[Recommended]** Use a single blank line to separate sections with the same logic or semantics.
> Note: It is unnecessary to use multiple blank lines to do that.

OOP Rules
1\. **[Mandatory]** A static field or method should be directly referred to by its class name instead of its corresponding object name.

2\. **[Mandatory]** An overridden method from an interface or abstract class must be marked with `@Override` annotation.
 > Counter example: For `getObject()` and `get0bject()`, the first one has a letter 'O', and the second one has a number '0'. To accurately determine whether the overriding is successful, an `@Override` annotation is necessary. Meanwhile, once the method signature in the abstract class is changed, the implementation class will report a compile-time error immediately.

3\. **[Mandatory]** *varargs* is recommended only if all parameters are of the same type and semantics. Parameters with `Object` type should be avoided.
 > Note: Arguments with the *varargs* feature must be at the end of the argument list. (Programming with the *varargs* feature is not recommended.)
 > Positive example:
```java
public User getUsers(String type, Integer... ids);
```

4\. **[Mandatory]** Modifying the method signature is forbidden to avoid affecting the caller. A `@Deprecated` annotation with an explicit description of the new service is necessary when an interface is deprecated.

5\. **[Mandatory]** Using a deprecated class or method is prohibited.
 > Note: For example, `decode(String source, String encode)` should be used instead of the deprecated method `decode(String encodeStr)`. Once an interface has been deprecated, the interface provider has the obligation to provide a new one. At the same time, client programmers have the obligation to use the new interface.

6\. **[Mandatory]** Since `NullPointerException` can possibly be thrown while calling the *equals* method of `Object`, *equals* should be invoked by a constant or an object that is definitely not *null*.
 > Positive example: `"test".equals(object);`
 > Counter example: `object.equals("test");`
 > Note: `java.util.Objects#equals` (a utility class in JDK7) is recommended.

7\. **[Mandatory]** Use the `equals` method, rather than reference equality '==', to compare primitive wrapper classes.
 > Note: Consider this assignment: `Integer var = ?`. When it fits the range from -128 to 127, we can use `==` directly for a comparison. Because the `Integer` object will be generated by `IntegerCache.cache`, which reuses an existing object. Nevertheless, when it fits the complementary set of the former range, the `Integer` object will be allocated in the heap, which does not reuse an existing object. This is a pitfall. Hence the `equals` method is recommended.

8\. **[Mandatory]** Rules for using primitive data types and wrapper classes:
  1) Members of a POJO class must be wrapper classes.
  2) The return value and arguments of a RPC method must be wrapper classes.
  3) **[Recommended]** Local variables should be primitive data types.
  Note: In order to remind the consumer of explicit assignments, there are no initial values for members in a POJO class. As a consumer, you should check problems such as `NullPointerException` and warehouse entries for yourself.
  Positive example: As the result of a database query may be *null*, assigning it to a primitive date type will cause a risk of `NullPointerException` because of autoboxing.
  Counter example: Consider the output of a transaction volume's amplitude, like *±x%*. As a primitive data, when it comes to a failure of calling a RPC service, the default return value: *0%* will be assigned, which is not correct. A hyphen like *-* should be assigned instead. Therefore, the *null* value of a wrapper class can represent additional information, such as a failure of calling a RPC service, an abnormal exit, etc.

9\. **[Mandatory]** While defining POJO classes like DO, DTO, VO, etc., do not assign any default values to the members.

10\. **[Mandatory]** To avoid a deserialization failure, do not change the *serialVersionUID* when a serialized class needs to be updated, such as adding some new members. If a completely incompatible update is needed, change the value of *serialVersionUID* in case of a confusion when deserialized.
> Note: The inconsistency of *serialVersionUID* may cause an `InvalidClassException` at runtime.

11\. **[Mandatory]** Business logic in constructor methods is prohibited. All initializations should be implemented in the `init` method.

12\. **[Mandatory]** The `toString` method must be implemented in a POJO class. The `super.toString` method should be called in in the beginning of the implementation if the current class extends another POJO class.

> Note: We can call the `toString` method in a POJO directly to print property values in order to check the problem when a method throws an exception in runtime.

13\. **[Recommended]** When accessing an array generated by the split method in String using an index, make sure to check the last separator whether it is null to avoid `IndexOutOfBoundException`.
> Note:
``` java
String str = "a,b,c,,";
String[] ary = str.split(",");
// The expected result exceeds 3. However it turns out to be 3.
System.out.println(ary.length);
```

14\. **[Recommended]** Multiple constructor methods or homonymous methods in a class should be put together for better readability.

15\. **[Recommended]** The order of methods declared within a class is:
public or protected methods -> private methods -> getter/setter methods.
> Note: As the most concerned ones for consumers and providers, *public* methods should be put on the first screen. *Protected* methods are only cared for by the subclasses, but they have chances to be vital when it comes to Template Design Pattern. *Private* methods, the black-box approaches, basically are not significant to clients. *Getter/setter* methods of a Service or a DAO should be put at the end of the class implementation because of the low significance.

16\. **[Recommended]** For a *setter* method, the argument name should be the same as the field name. Implementations of business logics in *getter/setter* methods, which will increase difficulties of the troubleshooting, are not recommended.
> Counter example:
```java
public Integer getData() {
    if (true) {
        return data + 100;
    } else {
        return data - 100;
    }
}
```

17\. **[Recommended]** Use the `append` method in `StringBuilder` inside a loop body when concatenating multiple strings.
> Counter example:

```java
String str = "start";
for (int i = 0; i < 100; i++) {
    str = str + "hello";
}
```

> Note: According to the decompiled bytecode file, for each loop, it allocates a `StringBuilder` object, appends a string, and finally returns a `String` object via the `toString` method. This is a tremendous waste of memory.

18\. **[Recommended]** Keyword *final* should be used in the following situations:
  1) A class which is not allow to be inherited, or a local variable not to be reassigned.
  2) An argument which is not allow to be modified.
  3) A method which is not allow to be overridden.

19\. **[Recommended]** Be cautious to copy an object using the `clone` method in `Object`.
> Note: The default implementation of `clone` in `Object` is a shallow (not deep) copy, which copies fields as pointers to the same objects in memory.

20\. **[Recommended]** Define the access level of members in class with severe restrictions:
  1) Constructor methods must be `private` if an allocation using `new` keyword outside of the class is forbidden.
  2) Constructor methods are not allowed to be `public` or `default` in a utility class.
  3) Nonstatic class variables that are accessed from inheritants must be `protected`.
  4) Nonstatic class variables that no one can access except the class that contains them must be `private`.
  5) Static variables that no one can access except the class that contains them must be `private`.
  6) Static variables should be considered in determining whether they are `final`.
  7) Class methods that no one can access except the class that contains them must be `private`.
  8) Class methods that are accessed from inheritants must be `protected`.
> Note: We should strictly control the access for any classes, methods, arguments and variables. Loose access control causes harmful coupling of modules. Imagine the following situations. For a `private` class member, we can remove it as soon as we want. However, when it comes to a `public` class member, we have to think twice before any updates happen to it.

Collection
1\. **[Mandatory]** The usage of *hashCode* and *equals* should follow:
  1) Override *hashCode* if *equals* is overridden.

  2) These two methods must be overridden for elements of a `Set` since they are used to ensure that no duplicate object will be inserted in `Set`.
  3) These two methods must be overridden for any object that is used as the key of `Map`.
> Note: `String` can be used as the key of `Map` since `String` defines these two methods.

2\. **[Mandatory]** Do not add elements to collection objects returned by `keySet()`/`values()`/`entrySet()`, otherwise `UnsupportedOperationException` will be thrown.

3\. **[Mandatory]** Do not add nor remove to/from immutable objects returned by methods in `Collections`, e.g. `emptyList()`/`singletonList()`.
> Counter example: Adding elements to `Collections.emptyList()` will throw `UnsupportedOperationException`.

4\. **[Mandatory]** Do not cast *subList* in class `ArrayList`, otherwise `ClassCastException` will be thrown: `java.util.RandomAccessSubList` cannot be cast to `java.util.ArrayList`.
> Note: `subList` of `ArrayList` is an inner class, which is a view of `ArrayList`. All operations on the `Sublist` will affect the original list.

5\. **[Mandatory]** When using *subList*, be careful when modifying the size of original list. It might cause `ConcurrentModificationException` when performing traversing, adding or deleting on the *subList*.

6\. **[Mandatory]** Use `toArray(T[] array)` to convert a list to an array. The input array type should be the same as the list whose size is `list.size()`.
> Counter example: Do not use `toArray` method without arguments. Since the return type is `Object[]`, `ClassCastException` will be thrown when casting it to a different array type.
> Positive example:
```java
List<String> list = new ArrayList<String>(2);
list.add("guan");
list.add("bao");
String[] array = new String[list.size()];
array = list.toArray(array);
```
> Note: When using `toArray` method with arguments, pass an input with the same size as the list. If input array size is not large enough, the method will re-assign the size internally, and then return the address of new array. If the size is larger than needed, extra elements (`index[list.size()]` and later) will be set to *null*.

7\. **[Mandatory]** Do not use methods which will modify the list after using `Arrays.asList` to convert array to list, otherwise methods like *add/remove/clear* will throw `UnsupportedOperationException`.
> Note: The result of `asList` is the inner class of `Arrays`, which does not implement methods to modify itself. `Arrays.asList` is only a transferred interface, data inside which is stored as an array.
```java
String[] str = new String[] { "a", "b" };
List<String> list = Arrays.asList(str);
```
Case 1: `list.add("c");` will throw a runtime exception.
Case 2: `str[0]= "gujin";` `list.get(0)` will be modified.

8\. **[Mandatory]** Method `add` cannot be used for generic wildcard with `<? Extends T>`, method `get` cannot be used with `<? super T>`, which probably goes wrong.
> Note: About PECS (Producer Extends Consumer Super) principle:
1) `Extends` is suitable for frequently reading scenarios.
2) `Super` is suitable for frequently inserting scenarios.

9\. **[Mandatory]** Do not remove or add elements to a collection in a *foreach* loop. Please use `Iterator` to remove an item. `Iterator` object should be synchronized when executing concurrent operations.
> Counter example:
```java
List<String> a = new ArrayList<String>();
a.add("1");
a.add("2");
for (String temp : a) {
    if ("1".equals(temp)){
        a.remove(temp);
    }
}
```
> Note: If you try to replace "1" with "2", you will get an unexpected result.
> Positive example:
```java
Iterator<String> it = a.iterator();
while (it.hasNext()) {
    String temp =  it.next();
    if (delete condition) {
        it.remove();
    }
```

```
}
```

10\. **[Mandatory]** In JDK 7 and above version, `Comparator` should meet the three requirements listed below, otherwise `Arrays.sort` and `Collections.sort` will throw `IllegalArgumentException`.
> Note:
  1) Comparing x,y and y,x should return the opposite result.
  2) If x>y and y>z, then x>z.
  3) If x=y, then comparing x with z and comparing y with z should return the same result.
> Counter example: The program below cannot handle the case if o1 equals to o2, which might cause an exception in a real case:
```java
new Comparator<Student>() {
    @Override
    public int compare(Student o1, Student o2) {
        return o1.getId() > o2.getId() ? 1 : -1;
    }
}
```

11\. **[Recommended]** Set a size when initializing a collection if possible.
> Note: Better to use `ArrayList(int initialCapacity)` to initialize `ArrayList`.

12\. **[Recommended]** Use `entrySet` instead of `keySet` to traverse KV maps.
> Note: Actually, `keySet` iterates through the map twice, firstly convert to `Iterator` object, then get the value from the `HashMap` by key. `EntrySet` iterates only once and puts keys and values in the entry which is more efficient. Use `Map.foreach` method in JDK8.
> Positive example: `values()` returns a list including all values, `keySet()` returns a set including all values, `entrySet()` returns a k-v combined object.

13\. **[Recommended]** Carefully check whether a *k/v collection* can store *null* value, refer to the table below:

Collection	Key	Value	Super	Note
Hashtable	Null is not allowed	Null is not allowed	Dictionary	Thread-safe
ConcurrentHashMap	Null is not allowed	Null is not allowed	AbstractMap	Segment lock

| TreeMap | Null is not allowed| Null is allowed| AbstractMap| Thread-unsafe |
| HashMap | Null is allowed| Null is allowed| AbstractMap| Thread-unsafe |

> Counter example: Confused by `HashMap`, lots of people think *null* is allowed in `ConcurrentHashMap`. Actually, `NullPointerException` will be thrown when putting in *null* value.

14\. **[For Reference]** Properly use sort and order of a collection to avoid negative influence of unsorted and unordered one.
> Note: *Sorted* means that its iteration follows specific sorting rule. *Ordered* means the order of elements in each traverse is stable. e.g. `ArrayList` is ordered and unsorted, `HashMap` is unordered and unsorted, `TreeSet` is ordered and sorted.

15\. **[For Reference]** Deduplication operations could be performed quickly since set stores unique values only. Avoid using method *contains* of `List` to perform traverse, comparison and de-duplication.

Concurrency
1\. **[Mandatory]** Thread-safe should be ensured when initializing singleton instance, as well as all methods in it.
> Note: Resource driven class, utility class and singleton factory class are all included.

2\. **[Mandatory]** A meaningful thread name is helpful to trace the error information, so assign a name when creating threads or thread pools.
> Positive example:
```java
public class TimerTaskThread extends Thread {
        public TimerTaskThread() {
                super.setName("TimerTaskThread");
            …
        }
}
```

3\. **[Mandatory]** Threads should be provided by thread pools. Explicitly creating threads is not allowed.
> Note: Using thread pool can reduce the time of creating and destroying thread and save system resource. If we do not use thread pools, lots of similar threads will be created which lead to "running out of memory" or over-switching problems.

4\. **[Mandatory]** A thread pool should be created by `ThreadPoolExecutor` rather than `Executors`. These would make the parameters of the thread pool understandable. It would also reduce the risk of running out of system resource.
 > Note: Below are the problems created by usage of `Executors` for thread pool creation:
  1) `FixedThreadPool` and `SingleThreadPool`:
  Maximum request queue size `Integer.MAX_VALUE`. A large number of requests might cause OOM.
  2) `CachedThreadPool` and `ScheduledThreadPool`:
  The number of threads which are allowed to be created is `Integer.MAX_VALUE`. Creating too many threads might lead to OOM.

5\. **[Mandatory]** `SimpleDateFormat` is unsafe, do not define it as a *static* variable. If have to, lock or `DateUtils` class must be used.
 > Positive example: Pay attention to thread-safety when using `DateUtils`. It is recommended to use as below:
 ```java
private static final ThreadLocal<DateFormat> df = new ThreadLocal<DateFormat>() {
    @Override
    protected DateFormat initialValue() {
        return new SimpleDateFormat("yyyy-MM-dd");
    }
};
 ```
 > Note: In JDK8, `Instant` can be used to replace `Date`, `Calendar` is replaced by `LocalDateTime`, `Simpledateformatter` is replaced by `DateTimeFormatter`.

6\. **[Mandatory]** `remove()` method must be implemented by `ThreadLocal` variables, especially when using thread pools in which threads are often reused. Otherwise, it may affect subsequent business logic and cause unexpected problems such as memory leak.
> Positive example:
```java
objectThreadLocal.set(someObject);
try {
    ...
} finally {
    objectThreadLocal.remove();
}
```

7\. **[Mandatory]** In highly concurrent scenarios, performance of `Lock` should be considered in synchronous calls. A block lock is better than a method lock. An object lock is better than a class lock.

8\. **[Mandatory]** When adding locks to multiple resources, tables in the database and objects at the same time, locking sequence should be kept consistent to avoid deadlock.
> Note: If thread 1 does update after adding lock to table A, B, C accordingly, the lock sequence of thread 2 should also be A, B, C, otherwise deadlock might happen.

9\. **[Mandatory]** A lock needs to be used to avoid update failure when modifying one record concurrently. Add lock either in application layer, in cache, or add optimistic lock in the database by using version as update stamp.
> Note: If access confliction probability is less than 20%, recommend to use optimistic lock, otherwise use pessimistic lock. Retry number of optimistic lock should be no less than 3.

10\. **[Mandatory]** Run multiple `TimeTask` by using `ScheduledExecutorService` rather than `Timer` because `Timer` will kill all running threads in case of failing to catch exceptions.

11\. **[Recommended]** When using `CountDownLatch` to convert asynchronous operations to synchronous ones, each thread must call `countdown` method before quitting. Make sure to catch any exception during thread running, to let `countdown` method be executed. If main thread cannot reach `await` method, program will return until timeout.
> Note: Be careful, exception thrown by sub-thread cannot be caught by main thread.

12\. **[Recommended]** Avoid using `Random` instance by multiple threads. Although it is safe to share this instance, competition on the same seed will damage performance.
> Note: `Random` instance includes instances of `java.util.Random` and `Math.random()`.
> Positive example: After JDK7, API `ThreadLocalRandom` can be used directly. But before JDK7, instance needs to be created in each thread.

13\. **[Recommended]** In concurrent scenarios, one easy solution to optimize the lazy initialization problem by using double-checked locking (referred to The Double-checked locking is broken Declaration), is to declare the object type as `volatile`.
> Counter example:
```java
class Foo {
    private Helper helper = null;
    public Helper getHelper() {
        if (helper == null) {
            synchronized(this) {
                if (helper == null)
                helper = new Helper();
            }
```

```
        }
        return helper;
    }
    // other functions and members...
}
```

14\. **[For Reference]** `volatile` is used to solve the problem of invisible memory in multiple threads. *Write-Once-Read-Many* can solve variable synchronization problem. But *Write-Many* cannot settle thread safe problem. For `count++`, use below example:
```java
AtomicInteger count = new AtomicInteger();
count.addAndGet(1);
```
> Note: In JDK8, `LongAdder` is recommended which reduces retry times of optimistic lock and has better performance than `AtomicLong`.

15\. **[For Reference]** Resizing `HashMap` when its capacity is not enough might cause dead link and high CPU usage because of high-concurrency. Avoid this risk in development.

16\. **[For Reference]** `ThreadLocal` cannot solve update problems of shared object. It is recommended to use a *static* `ThreadLocal` object which is shared by all operations in the same thread.

Flow Control Statements
1\. **[Mandatory]** In a `switch` block, each case should be finished by *break/return*. If not, a note should be included to describe at which case it will stop. Within every `switch` block, a default statement must be present, even if it is empty.

2\. **[Mandatory]** Braces are used with *if*, *else*, *for*, *do* and *while* statements, even if the body contains only a single statement. Avoid using the following example:
```java
    if (condition) statements;
```

3\. **[Recommended]** Use `else` as less as possible, `if-else` statements could be replaced by:
```java
if (condition) {
    ...
    return obj;
}
// other logic codes in else could be moved here
```

> Note: If statements like `if()...else if()...else...` have to be used to express the logic, **[Mandatory]** nested conditional level should not be more than three.
> Positive example: `if-else` code with over three nested conditional levels can be replaced by *guard statements* or *State Design Pattern*. Example of *guard statement*:

```java
public void today() {
    if (isBusy()) {
        System.out.println("Change time.");
        return;
    }

    if (isFree()) {
        System.out.println("Go to travel.");
        return;
    }

    System.out.println("Stay at home to learn Alibaba Java Coding Guidelines.");
    return;
}
```

4\. **[Recommended]** Do not use complicated statements in conditional statements (except for frequently used methods like *getXxx/isXxx*). Use *boolean* variables to store results of complicated statements temporarily will increase the code's readability.
> Note: Logic within many `if` statements are very complicated. Readers need to analyze the final results of the conditional expression to decide what statement will be executed in certain conditions.
> Positive example:
```java
// please refer to the pseudo-code as follows
boolean existed = (file.open(fileName, "w") != null) && (...) || (...);
if (existed) {
    ...
}
```
 > Counter example:
```java
if ((file.open(fileName, "w") != null) && (...) || (...)) {
    ...
}
```

5\. **[Recommended]** Performance should be considered when loop statements are used. The following operations are better to be processed outside the loop statement, including object and variable declaration, database connection, `try-catch` statements.

6\. **[Recommended]** Size of input parameters should be checked, especially for batch operations.

7\. **[For Reference]** Input parameters should be checked in following scenarios:
  1) Low-frequency implemented methods.
  2) Overhead of parameter checking could be ignored in long-time execution methods, but if illegal parameters lead to exception, the loss outweighs the gain. Therefore, parameter checking is still recommended in long-time execution methods.
  3) Methods that needs extremely high stability or availability.
  4) Open API methods, including RPC/API/HTTP.
  5) Authority related methods.

8\. **[For Reference]** Cases that input parameters do not require validation:
  1) Methods very likely to be implemented in loops. A note should be included informing that parameter check should be done externally.
  2) Methods in bottom layers are very frequently called so generally do not need to be checked. e.g. If *DAO* layer and *Service* layer is deployed in the same server, parameter check in *DAO* layer can be omitted.
  3) *Private* methods that can only be implemented internally, if all parameters are checked or manageable.

Code Comments
1\. **[Mandatory]** *Javadoc* should be used for classes, class variables and methods. The format should be '/** comment **/', rather than '// xxx'.
> Note: In IDE, *Javadoc* can be seen directly when hovering, which is a good way to improve efficiency.

2\. **[Mandatory]** Abstract methods (including methods in interface) should be commented by *Javadoc*. *Javadoc* should include method instruction, description of parameters, return values and possible exceptions.

3\. **[Mandatory]** Every class should include information of author(s) and date.

4\. **[Mandatory]** Single line comments in a method should be put above the code to be commented, by using `//` and multiple lines by using `/* */`. Alignment for comments should be noticed carefully.

5\. **[Mandatory]** All enumeration type fields should be commented as *Javadoc* style.

6\. **[Recommended]** Local language can be used in comments if English cannot describe the problem properly. Keywords and proper nouns should be kept in English.
 > Counter example: To explain "TCP connection overtime" as "Transmission Control Protocol connection overtime" only makes it more difficult to understand.

7\. **[Recommended]** When code logic changes, comments need to be updated at the same time, especially for the changes of parameters, return value, exception and core logic.

8\. **[For Reference]** Notes need to be added when commenting out code.
> Note: If the code is likely to be recovered later, a reasonable explanation needs to be added. If not, please delete directly because code history will be recorded by *svn* or *git*.

9\. **[For Reference]** Requirements for comments:
  1) Be able to represent design ideas and code logic accurately.
  2) Be able to represent business logic and help other programmers understand quickly. A large section of code without any comment is a disaster for readers. Comments are written for both oneself and other people. Design ideas can be quickly recalled even after a long time. Work can be quickly taken over by other people when needed.

10\. **[For Reference]** Proper naming and clear code structure are self-explanatory. Too many comments need to be avoided because it may cause too much work on updating if code logic changes.
 > Counter example:
```java
// put elephant into fridge
put(elephant, fridge);
```
The name of method and parameters already represent what does the method do, so there is no need to add extra comments.

11\. **[For Reference]** Tags in comments (e.g. TODO, FIXME) need to contain author and time. Tags need to be handled and cleared in time by scanning frequently. Sometimes online breakdown is caused by these unhandled tags.
  1) TODO: TODO means the logic needs to be done, but not finished yet. Actually, TODO is a member of *Javadoc*, although it is not realized in *Javadoc* yet, but has already been widely used. TODO can only be used in class, interface and methods, since it is a *Javadoc* tag.
  2) FIXME: FIXME is used to represent that the code logic is not correct or does not work, should be fixed in time.

Other
1\. **[Mandatory]** When using regex, precompile needs to be done in order to increase the matching performance.

> Note: Do not define `Pattern pattern = Pattern.compile(.);` within method body.

2\. **[Mandatory]** When using attributes of POJO in velocity, use attribute names directly. Velocity engine will invoke `getXxx()` of POJO automatically. In terms of *boolean* attributes, velocity engine will invoke `isXxx()` (Do not use *is* as prefix when naming boolean attributes).
> Note: For wrapper class `Boolean`, velocity engine will invoke `getXxx()` first.

3\. **[Mandatory]** Variables must add exclamatory mark when passing to velocity engine from backend, like \$!{var}.
 > Note: If attribute is *null* or does not exist, \${var} will be shown directly on web pages.

4\. **[Mandatory]** The return type of `Math.random()` is double, value range is *0<=x<1* (0 is possible). If a random integer is required, do not multiply x by 10 then round the result. The correct way is to use `nextInt` or `nextLong` method which belong to Random Object.

5\. **[Mandatory]** Use `System.currentTimeMillis()` to get the current millisecond. Do not use `new Date().getTime()`.
> Note: In order to get a more accurate time, use `System.nanoTime()`. In JDK8, use `Instant` class to deal with situations like time statistics.

6\. **[Recommended]** It is better not to contain variable declaration, logical symbols or any complicated logic in velocity template files.

7\. **[Recommended]** Size needs to be specified when initializing any data structure if possible, in order to avoid memory issues caused by unlimited growth.

8\. **[Recommended]** Codes or configuration that is noticed to be obsoleted should be resolutely removed from projects.
> Note: Remove obsoleted codes or configuration in time to avoid code redundancy.
 > Positive example: For codes which are temporarily removed and likely to be reused, use **`///`** to add a reasonable note.
```java
public static void hello() {
    /// Business is stopped temporarily by the owner.
    // Business business = new Business();
    // business.active();
    System.out.println("it's finished");
}
```

2. Exception and Logs

Exception
1\. **[Mandatory]** Do not catch *Runtime* exceptions defined in *JDK*, such as `NullPointerException` and `IndexOutOfBoundsException`. Instead, pre-check is recommended whenever possible.
> Note: Use try-catch only if it is difficult to deal with pre-check, such as `NumberFormatException`.
> Positive example: ```if (obj != null) {...}```
> Counter example: ```try { obj.method() } catch(NullPointerException e){…}```

2\. **[Mandatory]** Never use exceptions for ordinary control flow. It is ineffective and unreadable.

3\. **[Mandatory]** It is irresponsible to use a try-catch on a big chunk of code. Be clear about the stable and unstable code when using try-catch. The stable code that means no exception will throw. For the unstable code, catch as specific as possible for exception handling.

4\. **[Mandatory]** Do not suppress or ignore exceptions. If you do not want to handle it, then re-throw it. The top layer must handle the exception and translate it into what the user can understand.

5\. **[Mandatory]** Make sure to invoke the rollback if a method throws an Exception.

6\. **[Mandatory]** Closeable resources (stream, connection, etc.) must be handled in *finally* block. Never throw any exception from a *finally* block.
> Note: Use the *try-with-resources* statement to safely handle closeable resources (Java 7+).

7\. **[Mandatory]** Never use *return* within a *finally* block. A *return* statement in a *finally* block will cause exceptions or result in a discarded return value in the *try-catch* block.

8\. **[Mandatory]** The *Exception* type to be caught needs to be the same class or superclass of the type that has been thrown.

9\. **[Recommended]** The return value of a method can be *null*. It is not mandatory to return an empty collection or object. Specify in *Javadoc* explicitly when the method might return *null*. The caller needs to make a *null* check to prevent `NullPointerException`.
> Note: It is caller's responsibility to check the return value, as well as to consider the possibility that remote call fails or other runtime exception occurs.

10\. **[Recommended]** One of the most common errors is `NullPointerException`. Pay attention to the following situations:
  1) If the return type is primitive, return a value of wrapper class may cause `NullPointerException`.
    Counter example: `public int f() { return Integer }` Unboxing a *null* value will throw a `NullPointerException`.
  2) The return value of a database query might be *null*.
  3) Elements in collection may be *null*, even though `Collection.isEmpty()` returns *false*.
  4) Return values from an RPC might be *null*.
  5) Data stored in sessions might by *null*.
  6) Method chaining, like `obj.getA().getB().getC()`, is likely to cause `NullPointerException`.
    Positive example: Use `Optional` to avoid null check and NPE (Java 8+).

11\. **[Recommended]** Use "throw exception" or "return error code". For HTTP or open API providers, "error code" must be used. It is recommended to throw exceptions inside an application. For cross-application RPC calls, result is preferred by encapsulating *isSuccess*, *error code* and brief error messages.
> Note: Benefits to return Result for the RPC methods:
  1) Using the 'throw exception' method will occur a runtime error if the exception is not caught.
  2) If stack information it not attached, allocating custom exceptions with simple error message is not helpful to solve the problem. If stack information is attached, data serialization and transmission performance loss are also problems when frequent error occurs.

12\. **[Recommended]** Do not throw `RuntimeException`, `Exception`, or `Throwable` directly. It is recommended to use well defined custom exceptions such as `DAOException`, `ServiceException`, etc.

13\. **[For Reference]** Avoid duplicate code (Do not Repeat Yourself, also known as DRY principle).
> Note: Copying and pasting code arbitrarily will inevitably lead to duplicated code. If you keep logic in one place, it is easier to change when needed. If necessary, extract common codes to methods, abstract classes or even shared modules.
> Positive example: For a class with a number of public methods that validate parameters in the same way, it is better to extract a method like:
```java
private boolean checkParam (DTO dto) {
    ...
}
```

Logs
1\. **[Mandatory]** Do not use API in log system (Log4j, Logback) directly. API in log framework SLF4J is recommended to use instead, which uses *Facade* pattern and is conducive to keep log processing consistent.
```java
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
private static final Logger logger = LoggerFactory.getLogger(Abc.class);
```

2\. **[Mandatory]** Log files need to be kept for at least 15 days because some kinds of exceptions happen weekly.

3\. **[Mandatory]** Naming conventions of extended logs of an Application (such as RBI, temporary monitoring, access log, etc.):
appName_logType_logName.log
logType: Recommended classifications are *stats*, *desc*, *monitor*, *visit*, etc.
logName: Log description.
Benefits of this scheme: The file name shows what application the log belongs to, type of the log and what purpose is the log used for. It is also conducive for classification and search.
> Positive example: Name of the log file for monitoring the timezone conversion exception in *mppserver* application:
mppserver_monitor_timeZoneConvert.log
> Note: It is recommended to classify logs. Error logs and business logs should be stored separately as far as possible. It is not only easy for developers to view, but also convenient for system monitoring.

4\. **[Mandatory]** Logs at *TRACE / DEBUG / INFO* levels must use either conditional outputs or placeholders.
> Note: `logger.debug ("Processing trade with id: " + id + " symbol: " + symbol);` If the log level is warn, the above log will not be printed. However, it will perform string concatenation operator. `toString()` method of *symbol* will be called, which is a waste of system resources.
> Positive example:
```java
if (logger.isDebugEnabled()) {
    logger.debug("Processing trade with id: " + id + " symbol: " + symbol);
}
```
> Positive example:
```java
logger.debug("Processing trade with id: {} and symbol : {} ", id, symbol);
```

5\. **[Mandatory]** Ensure that additivity attribute of a Log4j logger is set to *false*, in order to avoid redundancy and save disk space.
> Positive example:
`<logger name="com.taobao.ecrm.member.config" additivity="false" \>`

6\. **[Mandatory]** The exception information should contain two types of information: the context information and the exception stack. If you do not want to handle the exception, re-throw it.
> Positive example:
```java
logger.error(various parameters or objects toString + "_" + e.getMessage(), e);
```

7\. **[Recommended]** Carefully record logs. Use *Info* level selectively and do not use *Debug* level in production environment. If *Warn* is used to record business behavior, please pay attention to the size of logs. Make sure the server disk is not over-filled, and remember to delete these logs in time.
> Note: Outputting a large number of invalid logs will have a certain impact on system performance, and is not conducive to rapid positioning problems. Please think about the log: do you really have these logs? What can you do to see this log? Is it easy to troubleshoot problems?

8\. **[Recommended]** Level *Warn* should be used to record invalid parameters, which is used to track data when problem occurs. Level *Error* only records the system logic error, abnormal and other important error messages.

3. MySQL Rules
Table Schema Rules
1\. **[Mandatory]** Columns expressing the concept of *True* or *False*, must be named as *is_xxx*, whose data type should be *unsigned tinyint* (1 is *True*, 0 is *False*).
> Note: All columns with non-negative values must be *unsigned*.

2\. **[Mandatory]** Names of tables and columns must consist of lower case letters, digits or underscores. Names starting with digits and names which contain only digits (no other characters) in between two underscores are not allowed. Columns should be named cautiously, as it is costly to change column names and cannot be released in pre-release environment.
> Positive example: *getter_admin, task_config, level3_name*
> Counter example: *GetterAdmin, taskConfig, level_3_name*

3\. **[Mandatory]** Plural nouns are not allowed as table names.

4\. **[Mandatory]** Keyword, such as *desc*, *range*, *match*, *delayed*, etc., should not be used. It can be referenced from MySQL official document.

5\. **[Mandatory]** The name of primary key index should be prefixed with *pk_*, followed by column name; Unique index should be named by prefixing its column name with *uk_*; And normal index should be formatted as *idx_[column_name]*.
> Note: *pk* means primary key, *uk* means unique key, and *idx* is short for index.

6\. **[Mandatory]** Decimals should be typed as *decimal*. *float* and *double* are not allowed.
> Note: It may have precision loss when float and double numbers are stored, which in turn may lead to incorrect data comparison result. It is recommended to store integral and fractional parts separately when data range to be stored is beyond the range covered by decimal type.

7\. **[Mandatory]** Use *char* if lengths of information to be stored in that column are almost the same.

8\. **[Mandatory]** The length of *varchar* should not exceed 5000, otherwise it should be defined as `text`. It is better to store them in a separate table in order to avoid its effect on indexing efficiency of other columns.

9\. **[Mandatory]** A table must include three columns as following: *id*, *gmt_create* and *gmt_modified*.
> Note: *id* is the primary key, which is *unsigned bigint* and self-incrementing with step length of 1. The type of *gmt_create* and *gmt_modified* should be *DATE_TIME*.

10\. **[Recommended]** It is recommended to define table name as *[table_business_name]_[table_purpose]*.
 > Positive example: *tiger_task* / *tiger_reader* / *mpp_config*

11\. **[Recommended]** Try to define database name same with the application name.

12\. **[Recommended]** Update column comments once column meaning is changed or new possible status values are added.

13\. **[Recommended]** Some appropriate columns may be stored in multiple tables redundantly to improve search performance, but consistency must be concerned. Redundant columns should not be:
  1) Columns with frequent modification.
  2) Columns typed with very long *varchar* or *text*.

> Positive example: Product category names are short, frequently used and with almost never changing/fixed values. They may be stored redundantly in relevant tables to avoid joined queries.

14\. **[Recommended]** Database sharding may only be recommended when there are more than 5 million rows in a single table or table capacity exceeds 2GB.
> Note: Please do not shard during table creation if anticipated data quantity is not to reach this grade.

15\. **[For Reference]** Appropriate *char* column length not only saves database and index storing space, but also improves query efficiency.
> Positive example: Unsigned types could avoid storing negative values mistakenly, but also may cover bigger data representative range.

Object	Age	Recommended data type	Range
human	within 150 years old	unsigned tinyint	unsigned integers: 0 to 255
turtle	hundreds years old	unsigned smallint	unsigned integers: 0 to 65,535
dinosaur fossil	tens of millions years old	unsigned int	unsigned integers: 0 to around 4.29 billion
sun	around 5 billion years old	unsigned bigint	unsigned integers: 0 to around 10^19

Index Rules
1\. **[Mandatory]** Unique index should be used if business logic is applicable.
> Note: Negative impact of unique indices on insert efficiency is neglectable, but it improves query speed significantly. Additionally, even if complete check is done at the application layer, as per Murphy's Law, dirty data might still be produced, as long as there is no unique index.

2\. **[Mandatory]** *JOIN* is not allowed if more than three tables are involved. Columns to be joined must be with absolutely similar data types. Make sure that columns to be joined are indexed.
> Note: Indexing and SQL performance should be considered even if only 2 tables are joined.

3\. **[Mandatory]** Index length must be specified when adding index on *varchar* columns. The index length should be set according to the distribution of data.
> Note: Normally for *char* columns, an index with the length of 20 can distinguish more than 90% data, which is calculated by *count(distinct left(column_name, index_length)) / count(*)*.

4\. **[Mandatory]** *LIKE '%...'* or *LIKE '%...%'* are not allowed when searching with pagination. Search engine can be used if it is really needed.

> Note: Index files have B-Tree's *left most prefix matching* characteristic. Index cannot be applied if left prefix value is not determined.

5\. **[Recommended]** Make use of the index order when using *ORDER BY* clauses. The last columns of *ORDER BY* clauses should be at the end of a composite index. The reason is to avoid the *file_sort* issue, which affects the query performance.
> Positive example: *where a=? and b=? order by c;* Index is: *a_b_c* </br>
> Counter example: The index order will not take effect if the query condition contains a range, e.g., *where a>10 order by b;* Index *a_b* cannot be activated.

6\. **[Recommended]** Make use of *Covering Index* for query to avoid additional query after searching index.
> Note: If we need to check the title of Chapter 11 of a book, do we need turn to the page where Chapter 11 starts? No, because the table of contents actually includes the title, which serves as a covering index.
> Positive example: Index types include *primary key index*, *unique index* and *common index*. *Covering index* pertains to a query effect. When refer to explain result, *using index* may appear in extra columns.

7\. **[Recommended]** Use *late join* or *sub-query* to optimize scenarios with many pages.
> Note: Instead of bypassing *offset* rows, MySQL retrieves totally *offset+N* rows, then drops off offset rows and returns N rows. It is very inefficient when offset is very big. The solution is either limiting the number of pages to be returned, or rewriting SQL statement when page number exceeds a predefined threshold.
> Positive example: Firstly locate the required *id* range quickly, then join:
select a.* from table1 a, (select id from table1 where *some_condition* LIMIT 100000, 20) b where a.id=b.id;

8\. **[Recommended]** The target of SQL performance optimization is that the result type of *EXPLAIN* reaches *REF* level, or *RANGE* at least, or CONSTS if possible.
> Counter example: Pay attention to the type of *INDEX* in *EXPLAIN* result because it is very slow to do a full scan to the database index file, whose performance nearly equals to an all-table scan.
CONSTS: There is at most one matching row, which is read by the optimizer. It is very fast.
REF: The normal index is used.
RANGE: A given range of index are retrieved, which can be used when a key column is compared to a constant by using any of the =, <>, >, >=, <, <=, IS NULL, <=>, BETWEEN, or IN() operators.

9\. **[Recommended]** Put the most discriminative column to the left most when adding a composite index.
> Positive example: For the sub-clause *where a=? and b=?*, if data of column *a* is nearly unique, adding index *idx_a* is enough.
> Note: When *equal* and *non-equal* check both exist in query conditions, put the column in *equal* condition first when adding an index. For example, *where a>? and b=?*, *b* should be put as the 1st column of the index, even if column *a* is more discriminative.

10\. **[For Reference]** Avoid listed below misunderstandings when adding index:
  1) It is false that each query needs one index.
  2) It is false that index consumes story space and degrades *update*, *insert* operations significantly.
  3) It is false that *unique index* should all be achieved from application layer by "check and insert".

SQL Rules
1\. **[Mandatory]** Do not use COUNT(column_name) or COUNT(constant_value) in place of COUNT(*). COUNT(*) is *SQL92* defined standard syntax to count the number of rows. It is not database specific and has nothing to do with *NULL* and *non-NULL*.
> Note: COUNT(*) counts NULL row in, while COUNT(column_name) does not take *NULL* valued row into consideration.

2\. **[Mandatory]** COUNT(distinct column) calculates number of rows with distinct values in this column, excluding *NULL* values. Please note that COUNT(distinct column1, column2) returns *0* if all values of one of the columns are *NULL*, even if the other column contains distinct *non-NULL* values.

3\. **[Mandatory]** When all values of one column are *NULL*, COUNT(column) returns *0*, while SUM(column) returns *NULL*, so pay attention to `NullPointerException` issue when using SUM().
> Positive example: NPE issue could be avoided in this way: *SELECT IF(ISNULL(SUM(g)), 0, SUM(g)) FROM table;*

4\. **[Mandatory]** Use *ISNULL()* to check *NULL* values. Result will be *NULL* when comparing *NULL* with any other values.
> Note:
  1) *NULL<>NULL* returns *NULL*, rather than *false*.
  2) *NULL=NULL* returns *NULL*, rather than *true*.
  3) *NULL<>1* returns *NULL*, rather than *true*.

5\. **[Mandatory]** When coding on DB query with paging logic, it should return immediately once count is *0*, to avoid executing paging query statement followed.

6\. **[Mandatory]** *Foreign key* and *cascade update* are not allowed. All foreign key related logic should be handled in application layer.
> Note: e.g. Student table has *student_id* as primary key, score table has *student_id* as foreign key. When *student.student_id* is updated, *score.student_id update* is also triggered, this is called a *cascading update*. *Foreign key* and *cascading update* are suitable for single machine, low parallel systems, not for distributed, high parallel cluster systems. *Cascading updates* are strong blocked, as it may lead to a DB update storm. *Foreign key* affects DB insertion efficiency.

7\. **[Mandatory]** Stored procedures are not allowed. They are difficult to debug, extend and not portable.

8\. **[Mandatory]** When correcting data, delete and update DB records, *SELECT* should be done first to ensure data correctness.

9\. **[Recommended]** *IN* clause should be avoided. Record set size of the *IN* clause should be evaluated carefully and control it within 1000, if it cannot be avoided.

10\. **[For Reference]** For globalization needs, characters should be represented and stored with *UTF-8*, and be cautious of character number counting.
> Note:
SELECT LENGTH("轻松工作"); returns *12*.
SELECT CHARACTER_LENGTH("轻松工作"); returns *4*.
Use *UTF8MB4* encoding to store emoji if needed, taking into account of its difference from *UTF-8*.

11\. **[For Reference]** *TRUNCATE* is not recommended when coding, even if it is faster than *DELETE* and uses less system, transaction log resource. Because *TRUNCATE* does not have transaction nor trigger DB *trigger*, problems might occur.
> Note: In terms of Functionality, *TRUNCATE TABLE* is similar to *DELETE* without *WHERE* sub-clause.

ORM Rules
1\. **[Mandatory]** Specific column names should be specified during query, rather than using *.
> Note:
1) * increases parsing cost.
2) It may introduce mismatch with *resultMap* when adding or removing query columns.

2\. **[Mandatory]** Name of *Boolean* property of *POJO* classes cannot be prefixed with *is*, while DB column name should prefix with *is*. A mapping between properties and columns is required.

> Note: Refer to rules of *POJO* class and DB column definition, mapping is needed in *resultMap*. Code generated by *MyBatis Generator* might need to be adjusted.

3\. **[Mandatory]** Do not use *resultClass* as return parameters, even if all class property names are the same as DB columns, corresponding DO definition is needed.
> Note: <resultMap>Mapping configuration is needed, to decouple DO definition and table columns, which in turn facilitates maintenance.

4\. **[Mandatory]** Be cautious with parameters in xml configuration. Do not use `${}` in place of `#{}`, `#param#`. SQL injection may happen in this way.

5\. **[Mandatory]** *iBatis* built in *queryForList(String statementName, int start, int size)* is not recommended.
> Note: It may lead to *OOM* issue because its implementation is to retrieve all DB records of statementName's corresponding SQL statement, then start, size subset is applied through subList.
> Positive example: Use *#start#*, *#size#* in *sqlmap.xml*.
```java
Map<String, Object> map = new HashMap<String, Object>();
map.put("start", start);
map.put("size", size);
```

6\. **[Mandatory]** Do not use *HashMap* or *HashTable* as DB query result type.

7\. **[Mandatory]** *gmt_modified* column should be updated with current timestamp simultaneously with DB record update.

8\. **[Recommended]** Do not define a universal table updating interface, which accepts POJO as input parameter, and always update table set *c1=value1, c2=value2, c3=value3, ...* regardless of intended columns to be updated. It is better not to update unrelated columns, because it is error prone, not efficient, and increases *binlog* storage.

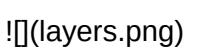
9\. **[For Reference]** Do not overuse *@Transactional*. Because transaction affects QPS of DB, and relevant rollbacks may need be considered, including cache rollback, search engine rollback, message making up, statistics adjustment, etc.

10\. **[For Reference]** *compareValue* of *\<isEqual>* is a constant (normally a number) which is used to compared with property value. *\<isNotEmpty>* means executing corresponding logic when property is not empty and not null. *\<isNotNull>* means executing related logic when property is not null.

4. Project Specification

Application Layers

1\. **[Recommended]** The upper layer depends on the lower layer by default. Arrow means direct dependent. For example: Open Interface can depend on Web Layer, it can also directly depend on Service Layer, etc.:



- **Open Interface:** In this layer service is encapsulate to be exposed as RPC interface, or HTTP interface through Web Layer; The layer also implements gateway security control, flow control, etc.
- **View:** In this layer templates of each terminal render and execute. Rendering approaches include velocity rendering, JS rendering, JSP rendering, and mobile display, etc.
- **Web Layer:** This layer is mainly used to implement forward access control, basic parameter verification, or non-reusable services.
- **Service Layer:** In this layer concrete business logic is implemented.
- **Manager Layer:** This layer is the common business process layer, which contains the following features:
  1) Encapsulates third-party service, to preprocess return values and exceptions;
  2) The precipitation of general ability of Service Layer, such as caching solutions, middleware general processing;
  3) Interacts with the *DAO layer*, including composition and reuse of multiple *DAO* classes.
- **DAO Layer**: Data access layer, data interacting with MySQL, Oracle and HBase.
- **External interface or third-party platform:** This layer includes RPC open interface from other departments or companies.

2\. **[For Reference]** Many exceptions in the *DAO Layer* cannot be caught by using a fine-grained exception class. The recommended way is to use `catch (Exception e)`, and `throw new DAOException(e)`. In these cases, there is no need to print the log because the log should have been caught and printed in *Manager Layer/Service Layer*.
   Logs about exception in *Service Layer* must be recorded with as much information about the parameters as possible to make debugging simpler.
   If *Manager Layer* and *Service Layer* are deployed in the same server, log logic should be consistent with *DAO Layer*. If they are deployed separately, log logic should be consistent with each other.
    In *Web Layer* Exceptions cannot be thrown, because it is already on the top layer and there is no way to deal with abnormal situations. If the exception is likely to cause failure when rendering the page, the page should be redirected to a friendly error page with the friendly error information.
   In *Open Interface* exceptions should be handled by using *error code* and *error message*.

3\. **[For Reference]** Layers of Domain Model:

- **DO (Data Object):** Corresponding to the database table structure, the data source object is transferred upward through *DAO* Layer.
- **DTO (Data Transfer Object):** Objects which are transferred upward by *Service Layer* and Manager Layer.
- **BO (Business Object):** Objects that encapsulate business logic, which can be outputted by *Service Layer*.
- **Query**: Data query objects that carry query request from upper layers. Note: Prohibit the use of `Map` if there are more than 2 query conditions.
- **VO (View Object):** Objects that are used in *Display Layer*, which is normally transferred from *Web Layer*.

Library Specification
1\. **[Mandatory]** Rules of defining GAV:
  1) **G**roupID: `com.{company/BU}.{business line}.{sub business line}`, at most 4 levels
> Note: {company/BU} for example: such business unit level as Alibaba, Taobao, Tmall, Aliexpress and so on; sub-business line is optional.
> Positive example: com.taobao.tddl
  com.alibaba.sourcing.multilang

  2) **A**rtifactID: Product name - module name.
> Positive example: `tc-client` / `uic-api` / `tair-tool`

  3) **V**ersion: Please refer to below

2\. **[Mandatory]** Library naming convention: `prime version number`.`secondary version number`.`revision number`
  1) **prime version number**: Need to change when there are incompatible API modification, or new features that can change the product direction.
  2) **secondary version number**: Changed for backward compatible modification.
  3) **revision number**: Changed for fixing bugs or adding features that do not modify the method signature and maintain API compatibility.
 > Note: The initial version must be 1.0.0, rather than 0.0.1.

3\. **[Mandatory]** Online applications should not depend on *SNAPSHOT* versions (except for security packages); Official releases must be verified from central repository, to make sure that the RELEASE version number has continuity. Version numbers are not allowed to be overridden.

> Note: Avoid using *SNAPSHOT* is to ensure the idempotent of application release. In addition, it can speed up the code compilation when packaging. If the current version is *1.3.3*, then the number for the next version can be *1.3.4*, *1.4.0* or *2.0.0*.

4\. **[Mandatory]** When adding or upgrading libraries, remain the versions of dependent libraries unchanged if not necessary. If there is a change, it must be correctly assessed and checked. It is recommended to use command *dependency:resolve* to compare the information before and after. If the result is identical, find out the differences with the command *dependency:tree* and use *\<excludes\>* to eliminate unnecessary libraries.

5\. **[Mandatory]** Enumeration types can be defined or used for parameter types in libraries, but cannot be used for interface return types (POJO that contains enumeration types is also not allowed).

6\. **[Mandatory]** When a group of libraries are used, a uniform version variable need to be defined to avoid the inconsistency of version numbers.
> Note: When using `springframework-core`, `springframework-context`, `springframework-beans` with the same version, a uniform version variable \${spring.version} is recommended to be used.

7\. **[Mandatory]** For the same *GroupId* and *ArtifactId*, *Version* must be the same in sub-projects.
> Note: During local debugging, version number specified in sub-project is used. But when merged into a war, only one version number is applied in the lib directory. Therefore, such case might occur that offline debugging is correct, but failure occurs after online launch.

8\. **[Recommended]** The declaration of dependencies in all *POM* files should be placed in *\<dependencies\>* block. Versions of dependencies should be specified in *\<dependencyManagement\>* block.
> Note: In *\<dependencyManagement\>* versions of dependencies are specified, but dependencies are not imported. Therefore, in sub-projects dependencies need to be declared explicitly, version and scope of which are read from the parent *POM*. All declarations in *\<dependencies\>* in the main *POM* will be automatically imported and inherited by all subprojects by default.

9\. **[Recommended]** It is recommended that libraries do not include configuration, at least do not increase the configuration items.

10\. **[For Reference]** In order to avoid the dependency conflict of libraries, the publishers should follow the principles below:
  1) **Simple and controllable:** Remove all unnecessary API and dependencies, only contain Service API, necessary domain model objects, Utils classes, constants, enumerations, etc. If other libraries must be included, better to make the scope as *provided*

and let users to depend on the specific version number. Do not depend on specific log implementation, only depend on the log framework instead.
  2) **Stable and traceable:** Change log of each version should be recorded. Make it easy to check the library owner and where the source code is. Libraries packaged in the application should not be changed unless the user updates the version proactively.

Server Specification
1\. **[Recommended]** It is recommended to reduce the *time_wait* value of the *TCP* protocol for high concurrency servers.
> Note: By default the operating system will close connection in *time_wait* state after 240 seconds. In high concurrent situation, the server may not be able to establish new connections because there are too many connections in *time_wait* state, so the value of *time_wait* needs to be reduced.
> Positive example: Modify the default value (Sec) by modifying the *_etcsysctl.conf* file on Linux servers:
`net.ipv4.tcp\_fin\_timeout = 30`

2\. **[Recommended]** Increase the maximum number of *File Descriptors* supported by the server.
> Note: Most operating systems are designed to manage *TCP/UDP* connections as a file, i.e. one connection corresponds to one *File Descriptor*. The maximum number of *File Descriptors* supported by the most Linux servers is 1024. It is easy to make an "open too many files" error because of the lack of *File Descriptor* when the number of concurrent connections is large, which would cause that new connections cannot be established.

3\. **[Recommended]** Set `-XX:+HeapDumpOnOutOfMemoryError` parameter for *JVM*, so *JVM* will output dump information when *OOM* occurs.
> Note: *OOM* does not occur very often, only once in a few months. The dump information printed when error occurs is very valuable for error checking.

4\. **[For Reference]** Use *forward* for internal redirection and URL assembly tools for external redirection. Otherwise there will be problems about URL maintaining inconsistency and potential security risks.

5. Security Specification

1\. **[Mandatory]** User-owned pages or functions must be authorized.
> Note: Prevent the access and manipulation of other people's data without authorization check, e.g. view or modify other people's orders.

2\. **[Mandatory]** Direct display of user sensitive data is not allowed. Displayed data must be desensitized.

> Note: Personal phone number should be displayed as: 158****9119. The middle 4 digits are hidden to prevent privacy leaks.

3\. **[Mandatory]** *SQL* parameter entered by users should be checked carefully or limited by *METADATA*, to prevent *SQL* injection. Database access by string concatenation *SQL* is forbidden.

4\. **[Mandatory]** Any parameters input by users must go through validation check.
> Note: Ignoring parameter check may cause:
>
 * memory leak because of excessive page size
 * slow database query because of malicious *order by*
 * arbitrary redirection
 * *SQL* injection
 * deserialize injection
 * *ReDoS*

> Note: In Java *regular expressions* is used to validate client input. Some *regular expressions* can validate common user input without any problem, but it could lead to a dead cycle if the attacker uses a specially constructed string to verify.

5\. **[Mandatory]** It is forbidden to output user data to *HTML* page without security filtering or proper escaping.

6\. **[Mandatory]** Form and *AJAX* submission must be filtered by *CSRF* security check.
> Note: *CSRF* (Cross-site Request Forgery) is a kind of common programming flaw. For applications/websites with *CSRF* leaks, attackers can construct *URL* in advance and modify the user parameters in database as long as the victim user visits without notice.

7\. **[Mandatory]** It is necessary to use the correct anti-replay restrictions, such as number restriction, fatigue control, verification code checking, to avoid abusing of platform resources, such as text messages, e-mail, telephone, order, payment.
> Note: For example, if there is no limitation to the times and frequency when sending verification codes to mobile phones, users might be bothered and *SMS* platform resources might be wasted.

8\. **[Recommended]** In scenarios when users generate content (e.g., posting, comment, instant messages), anti-scam word filtering and other risk control strategies must be applied.

Table of Contents

Preface

We are pleased to present **Alibaba Java Coding Guidelines**, which consolidates the best programming practices from Alibaba Group's technical teams. A vast number of Java programming teams impose demanding requirements on code quality across projects as we encourage reuse and better understanding of each other's programs. We have seen many programming problems in the past. For example, defective database table structures and index designs may cause software architecture flaws and performance risks. As another example, confusing code structures make it difficult to maintain. Furthermore, vulnerable code without authentication is prone to hackers' attacks. To address those kinds of problems, we developed this document for Java developers in Alibaba.

This document consists of five parts: ***Programming Specification***, ***Exception and Logs***, ***MySQL Specification***, ***Project Specification*** and ***Security Specification***. Based on the severity of the concerns, each specification is classified into three levels: ***Mandatory***, ***Recommended*** and ***Reference***. Further clarification is expressed in:
 (1) "**Description**", which explains the content;
 (2) "**Positive examples**", which describe recommended coding and implementation approaches;
 (3) "**Counter examples**", which describe precautions and actual error cases.

The main purpose of this document is to help developers improve code quality. As a result, developers can minimize potential and repetitive code errors. In addition, specification is essential to modern software architectures, which enable effective collaborations. As an analogy, traffic regulations are intended to protect public safety rather than to deprive the rights of driving. It is easy to imagine the chaos of traffic without speed limits and traffic lights. Instead of destroying the creativity and elegance of program, the purpose of developing appropriate specification and standards of software is to improve the efficiency of collaboration by limiting excessive personalization.

We will continue to collect feedback from the community to improve Alibaba Java Coding Guidelines.

1. Programming Specification
Naming Conventions
1\. **[Mandatory]** Names should not start or end with an underline or a dollar sign.
> Counter example: _name / __name / \$Object / name_ / name\$ / Object\$

2\. **[Mandatory]** Using Chinese, Pinyin, or Pinyin-English mixed spelling in naming is strictly prohibited. Accurate English spelling and grammar will make the code readable, understandable, and maintainable.
> Positive example: alibaba / taobao / youku / Hangzhou. In these cases, Chinese proper names in Pinyin are acceptable.

3\. **[Mandatory]** Class names should be nouns in UpperCamelCase except domain models: DO, BO, DTO, VO, etc.
> Positive example: MarcoPolo / UserDO / HtmlDTO / XmlService / TcpUdpDeal / TaPromotion
> Counter example: marcoPolo / UserDo / HTMLDto / XMLService / TCPUDPDeal / TAPromotion

4\. **[Mandatory]** Method names, parameter names, member variable names, and local variable names should be written in lowerCamelCase.
> Positive example: localValue / getHttpMessage() / inputUserId

5\. **[Mandatory]** Constant variable names should be written in upper characters separated by underscores. These names should be semantically complete and clear.
> Positive example: MAX_STOCK_COUNT
> Counter example: MAX_COUNT

6\. **[Mandatory]** Abstract class names must start with *Abstract* or *Base*. Exception class names must end with *Exception*. Test case names shall start with the class names to be tested and end with *Test*.

7\. **[Mandatory]** Brackets are a part of an Array type. The definition could be: *String[] args;*
> Counter example: *String args[];*

8\. **[Mandatory]** Do not add 'is' as prefix while defining Boolean variable, since it may cause a serialization exception in some Java frameworks.
> Counter example: *boolean isSuccess;* The method name will be `isSuccess()` and then RPC framework will deduce the variable name as 'success', resulting in a serialization error since it cannot find the correct attribute.

9\. **[Mandatory]** A package should be named in lowercase characters. There should be only one English word after each dot. Package names are always in singular format while class names can be in plural format if necessary.
> Positive example: `com.alibaba.open.util` can be used as a package name for utils;
`MessageUtils` can be used as a class name.

10\. **[Mandatory]** Uncommon abbreviations should be avoided for the sake of legibility.
> Counter example: AbsClass (AbstractClass); condi (Condition)

11\. **[Recommended]** The pattern name is recommended to be included in the class name if any design pattern is used.
> Positive example: `public class OrderFactory;  `
`public class LoginProxy;` `public class ResourceObserver;`
> Note: Including corresponding pattern names helps readers understand ideas in design patterns quickly.

12\. **[Recommended]** Do not add any modifier, including `public`, to methods in interface classes for coding simplicity. Please add valid *Javadoc* comments for methods. Do not define any variables in the interface except for the common constants of the application.
> Positive example: method definition in the interface: `void f(); ` constant definition: `String COMPANY = "alibaba";`
> Note: In JDK8 it is allowed to define a default implementation for interface methods, which is valuable for all implemented classes.

13\. There are two main rules for interface and corresponding implementation class naming:
  1) **[Mandatory]** All *Service* and *DAO* classes must be interfaces based on SOA principle. Implementation class names should end with *Impl*.

> Positive example: `CacheServiceImpl` to implement `CacheService`.

  2) **[Recommended]** If the interface name is to indicate the ability of the interface, then its name should be an adjective.
> Positive example: `AbstractTranslator` to implement `Translatable`.

14\. **[For Reference]** An Enumeration class name should end with *Enum*. Its members should be spelled out in upper case words, separated by underlines.
> Note: Enumeration is indeed a special constant class and all constructor methods are private by default.
> Positive example: Enumeration name: `DealStatusEnum`; Member name: `SUCCESS / UNKOWN_REASON`.

15\. **[For Reference]** Naming conventions for different package layers:
  A) Naming conventions for Service/DAO layer methods
    1) Use `get` as name prefix for a method to get a single object.
    2) Use `list` as name prefix for a method to get multiple objects.
    3) Use `count` as name prefix for a statistical method.
    4) Use `insert` or `save` (recommended) as name prefix for a method to save data.
    5) Use `delete` or `remove` (recommended) as name prefix for a method to remove data.
    6) Use `update` as name prefix for a method to update data.
  B) Naming conventions for Domain models
    1) Data Object: *DO, where * is the table name.
    2) Data Transfer Object: *DTO, where * is a domain-related name.
    3) Value Object: *VO, where * is a website name in most cases.
    4) POJO generally point to DO/DTO/BO/VO but cannot be used in naming as *POJO.

Constant Conventions
1\. **[Mandatory]** Magic values, except for predefined, are forbidden in coding.
> Counter example: String key = "Id#taobao_" + tradeId;

2\. **[Mandatory]** 'L' instead of 'l' should be used for long or Long variable because 'l' is easily to be regarded as number 1 in mistake.
> Counter example: `Long a = 2l`, it is hard to tell whether it is number 21 or Long 2.

3\. **[Recommended]** Constants should be placed in different constant classes based on their functions. For example, cache related constants could be put in `CacheConsts` while configuration related constants could be kept in `ConfigConsts`.
 > Note: It is difficult to find one constant in one big complete constant class.

4\. **[Recommended]** Constants can be shared in the following 5 different layers: *shared in multiple applications; shared inside an application; shared in a sub-project; shared in a package; shared in a class*.
  1) Shared in multiple applications: keep in a library, under `constant` directory in client.jar;
  2) Shared in an application: keep in shared modules within the application, under `constant` directory;
  Counter example: Obvious variable names should also be defined as common shared constants in an application. The following definitions caused an exception in the production environment: it returns *false*, but is expected to return *true* for `A.YES.equals(B.YES)`.
  Definition in Class A: `public static final String YES = "yes";`
  Definition in Class B: `public static final String YES = "y";`
  3) Shared in a sub-project: placed under `constant` directory in the current project;
  4) Shared in a package: placed under `constant` directory in current package;
  5) Shared in a class: defined as 'private static final' inside class.

5\. **[Recommended]** Use an enumeration class if values lie in a fixed range or if the variable has attributes. The following example shows that extra information (which day it is) can be included in enumeration:
 > Positive example: public Enum{ MONDAY(1), TUESDAY(2), WEDNESDAY(3), THURSDAY(4), FRIDAY(5), SATURDAY(6), SUNDAY(7);}

Formatting Style
1\. **[Mandatory]** Rules for braces. If there is no content, simply use *{}* in the same line. Otherwise:
  1) No line break before the opening brace.
  2) Line break after the opening brace.
  3) Line break before the closing brace.
  4) Line break after the closing brace, *only if* the brace terminates a statement or terminates a method body, constructor or named class. There is *no* line break after the closing brace if it is followed by `else` or a comma.

2\. **[Mandatory]** No space is used between the '(' character and its following character. Same for the ')' character and its preceding character. Refer to the *Positive Example* at the 5th rule.

3\. **[Mandatory]** There must be one space between keywords, such as if/for/while/switch, and parentheses.

4\. **[Mandatory]** There must be one space at both left and right side of operators, such as '=', '&&', '+', '-', *ternary operator*, etc.

5\. **[Mandatory]** Each time a new block or block-like construct is opened, the indent increases by four spaces. When the block ends, the indent returns to the previous indent level. Tab characters are not used for indentation.
> Note: To prevent tab characters from being used for indentation, you must configure your IDE. For example, "Use tab character" should be unchecked in IDEA, "insert spaces for tabs" should be checked in Eclipse.
> Positive example:
```java
public static void main(String[] args) {
    // four spaces indent
    String say = "hello";
    // one space before and after the operator
    int flag = 0;
    // one space between 'if' and '(';
    // no space between '(' and 'flag' or between '0' and ')'
    if (flag == 0) {
        System.out.println(say);
    }
    // one space before '{' and line break after '{'
    if (flag == 1) {
        System.out.println("world");
    // line break before '}' but not after '}' if it is followed by 'else'
    } else {
        System.out.println("ok");
    // line break after '}' if it is the end of the block
    }
}
```

6\. **[Mandatory]** Java code has a column limit of 120 characters. Except import statements, any line that would exceed this limit must be line-wrapped as follows:
  1) The second line should be intented at 4 spaces with respect to the first one. The third line and following ones should align with the second line.
  2) Operators should be moved to the next line together with following context.
  3) Character '.' should be moved to the next line together with the method after it.
  4) If there are multiple parameters that extend over the maximum length, a line break should be inserted after a comma.

  5) No line breaks should appear before parentheses.
> Positive example:
```java
StringBuffer sb = new StringBuffer();
// line break if there are more than 120 characters, and 4 spaces indent at
// the second line. Make sure character '.' moved to the next line
// together.  The third and fourth lines are aligned with the second one.
sb.append("zi").append("xin").
    .append("huang")...
    .append("huang")...
    .append("huang");
```
> Counter example:
```java
StringBuffer sb = new StringBuffer();
// no line break before '('
sb.append("zi").append("xin")...append
    ("huang");
// no line break before ',' if there are multiple params
invoke(args1, args2, args3, ...
    , argsX);
```

7\. **[Mandatory]** There must be one space between a comma and the next parameter for methods with multiple parameters.
> Positive example: One space is used after the *','* character in the following method definition.
```java
f("a", "b", "c");
```

8\. **[Mandatory]** The charset encoding of text files should be *UTF-8* and the characters of line breaks should be in *Unix* format, instead of *Windows* format.

9\. **[Recommended]** It is unnecessary to align variables by several spaces.
> Positive example:
```java
int a = 3;
long b = 4L;
float c = 5F;
StringBuffer sb = new StringBuffer();
```
> Note: It is cumbersome to insert several spaces to align the variables above.

10\. **[Recommended]** Use a single blank line to separate sections with the same logic or semantics.
> Note: It is unnecessary to use multiple blank lines to do that.

OOP Rules
1\. **[Mandatory]** A static field or method should be directly referred to by its class name instead of its corresponding object name.

2\. **[Mandatory]** An overridden method from an interface or abstract class must be marked with `@Override` annotation.
 > Counter example: For `getObject()` and `get0bject()`, the first one has a letter 'O', and the second one has a number '0'. To accurately determine whether the overriding is successful, an `@Override` annotation is necessary. Meanwhile, once the method signature in the abstract class is changed, the implementation class will report a compile-time error immediately.

3\. **[Mandatory]** *varargs* is recommended only if all parameters are of the same type and semantics. Parameters with `Object` type should be avoided.
 > Note: Arguments with the *varargs* feature must be at the end of the argument list. (Programming with the *varargs* feature is not recommended.)
 > Positive example:
```java
public User getUsers(String type, Integer... ids);
```

4\. **[Mandatory]** Modifying the method signature is forbidden to avoid affecting the caller. A `@Deprecated` annotation with an explicit description of the new service is necessary when an interface is deprecated.

5\. **[Mandatory]** Using a deprecated class or method is prohibited.
 > Note: For example, `decode(String source, String encode)` should be used instead of the deprecated method `decode(String encodeStr)`. Once an interface has been deprecated, the interface provider has the obligation to provide a new one. At the same time, client programmers have the obligation to use the new interface.

6\. **[Mandatory]** Since `NullPointerException` can possibly be thrown while calling the *equals* method of `Object`, *equals* should be invoked by a constant or an object that is definitely not *null*.
 > Positive example: `"test".equals(object);`
 > Counter example: `object.equals("test");`
 > Note: `java.util.Objects#equals` (a utility class in JDK7) is recommended.

7\. **[Mandatory]** Use the `equals` method, rather than reference equality '==', to compare primitive wrapper classes.
 > Note: Consider this assignment: `Integer var = ?`. When it fits the range from -128 to 127, we can use `==` directly for a comparison. Because the `Integer` object will be generated by `IntegerCache.cache`, which reuses an existing object. Nevertheless, when it fits the complementary set of the former range, the `Integer` object will be allocated in the heap, which does not reuse an existing object. This is a pitfall. Hence the `equals` method is recommended.

8\. **[Mandatory]** Rules for using primitive data types and wrapper classes:
  1) Members of a POJO class must be wrapper classes.
  2) The return value and arguments of a RPC method must be wrapper classes.
  3) **[Recommended]** Local variables should be primitive data types.
  Note: In order to remind the consumer of explicit assignments, there are no initial values for members in a POJO class. As a consumer, you should check problems such as `NullPointerException` and warehouse entries for yourself.
  Positive example: As the result of a database query may be *null*, assigning it to a primitive date type will cause a risk of `NullPointerException` because of autoboxing.
  Counter example: Consider the output of a transaction volume's amplitude, like *±x%*. As a primitive data, when it comes to a failure of calling a RPC service, the default return value: *0%* will be assigned, which is not correct. A hyphen like *-* should be assigned instead. Therefore, the *null* value of a wrapper class can represent additional information, such as a failure of calling a RPC service, an abnormal exit, etc.

9\. **[Mandatory]** While defining POJO classes like DO, DTO, VO, etc., do not assign any default values to the members.

10\. **[Mandatory]** To avoid a deserialization failure, do not change the *serialVersionUID* when a serialized class needs to be updated, such as adding some new members. If a completely incompatible update is needed, change the value of *serialVersionUID* in case of a confusion when deserialized.
> Note: The inconsistency of *serialVersionUID* may cause an `InvalidClassException` at runtime.

11\. **[Mandatory]** Business logic in constructor methods is prohibited. All initializations should be implemented in the `init` method.

12\. **[Mandatory]** The `toString` method must be implemented in a POJO class. The `super.toString` method should be called in in the beginning of the implementation if the current class extends another POJO class.

> Note: We can call the `toString` method in a POJO directly to print property values in order to check the problem when a method throws an exception in runtime.

13\. **[Recommended]** When accessing an array generated by the split method in String using an index, make sure to check the last separator whether it is null to avoid `IndexOutOfBoundException`.
> Note:
``` java
String str = "a,b,c,,";
String[] ary = str.split(",");
// The expected result exceeds 3. However it turns out to be 3.
System.out.println(ary.length);
```

14\. **[Recommended]** Multiple constructor methods or homonymous methods in a class should be put together for better readability.

15\. **[Recommended]** The order of methods declared within a class is:
public or protected methods -> private methods -> getter/setter methods.
> Note: As the most concerned ones for consumers and providers, *public* methods should be put on the first screen. *Protected* methods are only cared for by the subclasses, but they have chances to be vital when it comes to Template Design Pattern. *Private* methods, the black-box approaches, basically are not significant to clients. *Getter/setter* methods of a Service or a DAO should be put at the end of the class implementation because of the low significance.

16\. **[Recommended]** For a *setter* method, the argument name should be the same as the field name. Implementations of business logics in *getter/setter* methods, which will increase difficulties of the troubleshooting, are not recommended.
> Counter example:
```java
public Integer getData() {
    if (true) {
        return data + 100;
    } else {
        return data - 100;
    }
}
```

17\. **[Recommended]** Use the `append` method in `StringBuilder` inside a loop body when concatenating multiple strings.
> Counter example:

```java
String str = "start";
for (int i = 0; i < 100; i++) {
    str = str + "hello";
}
```

> Note: According to the decompiled bytecode file, for each loop, it allocates a `StringBuilder` object, appends a string, and finally returns a `String` object via the `toString` method. This is a tremendous waste of memory.

18\. **[Recommended]** Keyword *final* should be used in the following situations:
  1) A class which is not allow to be inherited, or a local variable not to be reassigned.
  2) An argument which is not allow to be modified.
  3) A method which is not allow to be overridden.

19\. **[Recommended]** Be cautious to copy an object using the `clone` method in `Object`.
> Note: The default implementation of `clone` in `Object` is a shallow (not deep) copy, which copies fields as pointers to the same objects in memory.

20\. **[Recommended]** Define the access level of members in class with severe restrictions:
  1) Constructor methods must be `private` if an allocation using `new` keyword outside of the class is forbidden.
  2) Constructor methods are not allowed to be `public` or `default` in a utility class.
  3) Nonstatic class variables that are accessed from inheritants must be `protected`.
  4) Nonstatic class variables that no one can access except the class that contains them must be `private`.
  5) Static variables that no one can access except the class that contains them must be `private`.
  6) Static variables should be considered in determining whether they are `final`.
  7) Class methods that no one can access except the class that contains them must be `private`.
  8) Class methods that are accessed from inheritants must be `protected`.
> Note: We should strictly control the access for any classes, methods, arguments and variables. Loose access control causes harmful coupling of modules. Imagine the following situations. For a `private` class member, we can remove it as soon as we want. However, when it comes to a `public` class member, we have to think twice before any updates happen to it.

Collection
1\. **[Mandatory]** The usage of *hashCode* and *equals* should follow:
  1) Override *hashCode* if *equals* is overridden.

  2) These two methods must be overridden for elements of a `Set` since they are used to ensure that no duplicate object will be inserted in `Set`.
  3) These two methods must be overridden for any object that is used as the key of `Map`.
> Note: `String` can be used as the key of `Map` since `String` defines these two methods.

2\. **[Mandatory]** Do not add elements to collection objects returned by `keySet()`/`values()`/`entrySet()`, otherwise `UnsupportedOperationException` will be thrown.

3\. **[Mandatory]** Do not add nor remove to/from immutable objects returned by methods in `Collections`, e.g. `emptyList()`/`singletonList()`.
> Counter example: Adding elements to `Collections.emptyList()` will throw `UnsupportedOperationException`.

4\. **[Mandatory]** Do not cast *subList* in class `ArrayList`, otherwise `ClassCastException` will be thrown: `java.util.RandomAccessSubList` cannot be cast to `java.util.ArrayList`.
> Note: `subList` of `ArrayList` is an inner class, which is a view of `ArrayList`. All operations on the `Sublist` will affect the original list.

5\. **[Mandatory]** When using *subList*, be careful when modifying the size of original list. It might cause `ConcurrentModificationException` when performing traversing, adding or deleting on the *subList*.

6\. **[Mandatory]** Use `toArray(T[] array)` to convert a list to an array. The input array type should be the same as the list whose size is `list.size()`.
> Counter example: Do not use `toArray` method without arguments. Since the return type is `Object[]`, `ClassCastException` will be thrown when casting it to a different array type.
> Positive example:
```java
List<String> list = new ArrayList<String>(2);
list.add("guan");
list.add("bao");
String[] array = new String[list.size()];
array = list.toArray(array);
```
> Note: When using `toArray` method with arguments, pass an input with the same size as the list. If input array size is not large enough, the method will re-assign the size internally, and then return the address of new array. If the size is larger than needed, extra elements (`index[list.size()]` and later) will be set to *null*.

7\. **[Mandatory]** Do not use methods which will modify the list after using `Arrays.asList` to convert array to list, otherwise methods like *add/remove/clear* will throw `UnsupportedOperationException`.
> Note: The result of `asList` is the inner class of `Arrays`, which does not implement methods to modify itself. `Arrays.asList` is only a transferred interface, data inside which is stored as an array.
```java
String[] str = new String[] { "a", "b" };
List<String> list = Arrays.asList(str);
```
Case 1: `list.add("c");` will throw a runtime exception.
Case 2: `str[0]= "gujin";` `list.get(0)` will be modified.

8\. **[Mandatory]** Method `add` cannot be used for generic wildcard with `<? Extends T>`, method `get` cannot be used with `<? super T>`, which probably goes wrong.
> Note: About PECS (Producer Extends Consumer Super) principle:
1) `Extends` is suitable for frequently reading scenarios.
2) `Super` is suitable for frequently inserting scenarios.

9\. **[Mandatory]** Do not remove or add elements to a collection in a *foreach* loop. Please use `Iterator` to remove an item. `Iterator` object should be synchronized when executing concurrent operations.
> Counter example:
```java
List<String> a = new ArrayList<String>();
a.add("1");
a.add("2");
for (String temp : a) {
    if ("1".equals(temp)){
        a.remove(temp);
    }
}
```
> Note: If you try to replace "1" with "2", you will get an unexpected result.
> Positive example:
```java
Iterator<String> it = a.iterator();
while (it.hasNext()) {
    String temp =  it.next();
    if (delete condition) {
        it.remove();
    }
```

```
}
```

10\. **[Mandatory]** In JDK 7 and above version, `Comparator` should meet the three requirements listed below, otherwise `Arrays.sort` and `Collections.sort` will throw `IllegalArgumentException`.
> Note:
  1) Comparing x,y and y,x should return the opposite result.
  2) If x>y and y>z, then x>z.
  3) If x=y, then comparing x with z and comparing y with z should return the same result.
> Counter example: The program below cannot handle the case if o1 equals to o2, which might cause an exception in a real case:
```java
new Comparator<Student>() {
    @Override
    public int compare(Student o1, Student o2) {
        return o1.getId() > o2.getId() ? 1 : -1;
    }
}
```

11\. **[Recommended]** Set a size when initializing a collection if possible.
> Note: Better to use `ArrayList(int initialCapacity)` to initialize `ArrayList`.

12\. **[Recommended]** Use `entrySet` instead of `keySet` to traverse KV maps.
> Note: Actually, `keySet` iterates through the map twice, firstly convert to `Iterator` object, then get the value from the `HashMap` by key. `EntrySet` iterates only once and puts keys and values in the entry which is more efficient. Use `Map.foreach` method in JDK8.
> Positive example: `values()` returns a list including all values, `keySet()` returns a set including all values, `entrySet()` returns a k-v combined object.

13\. **[Recommended]** Carefully check whether a *k/v collection* can store *null* value, refer to the table below:

Collection	Key	Value	Super	Note
Hashtable	Null is not allowed	Null is not allowed	Dictionary	Thread-safe
ConcurrentHashMap	Null is not allowed	Null is not allowed	AbstractMap	Segment lock

| TreeMap | Null is not allowed| Null is allowed| AbstractMap| Thread-unsafe |
| HashMap | Null is allowed| Null is allowed| AbstractMap| Thread-unsafe |

> Counter example: Confused by `HashMap`, lots of people think *null* is allowed in `ConcurrentHashMap`. Actually, `NullPointerException` will be thrown when putting in *null* value.

14\. **[For Reference]** Properly use sort and order of a collection to avoid negative influence of unsorted and unordered one.
> Note: *Sorted* means that its iteration follows specific sorting rule. *Ordered* means the order of elements in each traverse is stable. e.g. `ArrayList` is ordered and unsorted, `HashMap` is unordered and unsorted, `TreeSet` is ordered and sorted.

15\. **[For Reference]** Deduplication operations could be performed quickly since set stores unique values only. Avoid using method *contains* of `List` to perform traverse, comparison and de-duplication.

Concurrency
1\. **[Mandatory]** Thread-safe should be ensured when initializing singleton instance, as well as all methods in it.
> Note: Resource driven class, utility class and singleton factory class are all included.

2\. **[Mandatory]** A meaningful thread name is helpful to trace the error information, so assign a name when creating threads or thread pools.
> Positive example:
```java
public class TimerTaskThread extends Thread {
        public TimerTaskThread() {
                super.setName("TimerTaskThread");
            …
        }
}
```

3\. **[Mandatory]** Threads should be provided by thread pools. Explicitly creating threads is not allowed.
> Note: Using thread pool can reduce the time of creating and destroying thread and save system resource. If we do not use thread pools, lots of similar threads will be created which lead to "running out of memory" or over-switching problems.

4\. **[Mandatory]** A thread pool should be created by `ThreadPoolExecutor` rather than `Executors`. These would make the parameters of the thread pool understandable. It would also reduce the risk of running out of system resource.
 > Note: Below are the problems created by usage of `Executors` for thread pool creation:
  1) `FixedThreadPool` and `SingleThreadPool`:
  Maximum request queue size `Integer.MAX_VALUE`. A large number of requests might cause OOM.
  2) `CachedThreadPool` and `ScheduledThreadPool`:
  The number of threads which are allowed to be created is `Integer.MAX_VALUE`. Creating too many threads might lead to OOM.

5\. **[Mandatory]** `SimpleDateFormat` is unsafe, do not define it as a *static* variable. If have to, lock or `DateUtils` class must be used.
 > Positive example: Pay attention to thread-safety when using `DateUtils`. It is recommended to use as below:
 ```java
private static final ThreadLocal<DateFormat> df = new ThreadLocal<DateFormat>() {
    @Override
    protected DateFormat initialValue() {
        return new SimpleDateFormat("yyyy-MM-dd");
    }
};
 ```
 > Note: In JDK8, `Instant` can be used to replace `Date`, `Calendar` is replaced by `LocalDateTime`, `Simpledateformatter` is replaced by `DateTimeFormatter`.

6\. **[Mandatory]** `remove()` method must be implemented by `ThreadLocal` variables, especially when using thread pools in which threads are often reused. Otherwise, it may affect subsequent business logic and cause unexpected problems such as memory leak.
> Positive example:
```java
objectThreadLocal.set(someObject);
try {
    ...
} finally {
    objectThreadLocal.remove();
}
```

7\. **[Mandatory]** In highly concurrent scenarios, performance of `Lock` should be considered in synchronous calls. A block lock is better than a method lock. An object lock is better than a class lock.

8\. **[Mandatory]** When adding locks to multiple resources, tables in the database and objects at the same time, locking sequence should be kept consistent to avoid deadlock.
> Note: If thread 1 does update after adding lock to table A, B, C accordingly, the lock sequence of thread 2 should also be A, B, C, otherwise deadlock might happen.

9\. **[Mandatory]** A lock needs to be used to avoid update failure when modifying one record concurrently. Add lock either in application layer, in cache, or add optimistic lock in the database by using version as update stamp.
> Note: If access confliction probability is less than 20%, recommend to use optimistic lock, otherwise use pessimistic lock. Retry number of optimistic lock should be no less than 3.

10\. **[Mandatory]** Run multiple `TimeTask` by using `ScheduledExecutorService` rather than `Timer` because `Timer` will kill all running threads in case of failing to catch exceptions.

11\. **[Recommended]** When using `CountDownLatch` to convert asynchronous operations to synchronous ones, each thread must call `countdown` method before quitting. Make sure to catch any exception during thread running, to let `countdown` method be executed. If main thread cannot reach `await` method, program will return until timeout.
> Note: Be careful, exception thrown by sub-thread cannot be caught by main thread.

12\. **[Recommended]** Avoid using `Random` instance by multiple threads. Although it is safe to share this instance, competition on the same seed will damage performance.
> Note: `Random` instance includes instances of `java.util.Random` and `Math.random()`.
> Positive example: After JDK7, API `ThreadLocalRandom` can be used directly. But before JDK7, instance needs to be created in each thread.

13\. **[Recommended]** In concurrent scenarios, one easy solution to optimize the lazy initialization problem by using double-checked locking (referred to The Double-checked locking is broken Declaration), is to declare the object type as `volatile`.
> Counter example:
```java
class Foo {
    private Helper helper = null;
    public Helper getHelper() {
        if (helper == null) {
            synchronized(this) {
                if (helper == null)
                helper = new Helper();
            }
```

```
        }
        return helper;
    }
    // other functions and members...
}
```

14\. **[For Reference]** `volatile` is used to solve the problem of invisible memory in multiple threads. *Write-Once-Read-Many* can solve variable synchronization problem. But *Write-Many* cannot settle thread safe problem. For `count++`, use below example:
```java
AtomicInteger count = new AtomicInteger();
count.addAndGet(1);
```
> Note: In JDK8, `LongAdder` is recommended which reduces retry times of optimistic lock and has better performance than `AtomicLong`.

15\. **[For Reference]** Resizing `HashMap` when its capacity is not enough might cause dead link and high CPU usage because of high-concurrency. Avoid this risk in development.

16\. **[For Reference]** `ThreadLocal` cannot solve update problems of shared object. It is recommended to use a *static* `ThreadLocal` object which is shared by all operations in the same thread.

Flow Control Statements
1\. **[Mandatory]** In a `switch` block, each case should be finished by *break/return*. If not, a note should be included to describe at which case it will stop. Within every `switch` block, a default statement must be present, even if it is empty.

2\. **[Mandatory]** Braces are used with *if*, *else*, *for*, *do* and *while* statements, even if the body contains only a single statement. Avoid using the following example:
```java
    if (condition) statements;
```

3\. **[Recommended]** Use `else` as less as possible, `if-else` statements could be replaced by:
```java
if (condition) {
    ...
    return obj;
}
// other logic codes in else could be moved here
```

> Note: If statements like `if()...else if()...else...` have to be used to express the logic, **[Mandatory]** nested conditional level should not be more than three.
> Positive example: `if-else` code with over three nested conditional levels can be replaced by *guard statements* or *State Design Pattern*. Example of *guard statement*:

```java
public void today() {
    if (isBusy()) {
        System.out.println("Change time.");
        return;
    }

    if (isFree()) {
        System.out.println("Go to travel.");
        return;
    }

    System.out.println("Stay at home to learn Alibaba Java Coding Guidelines.");
    return;
}
```

4\. **[Recommended]** Do not use complicated statements in conditional statements (except for frequently used methods like *getXxx/isXxx*). Use *boolean* variables to store results of complicated statements temporarily will increase the code's readability.
> Note: Logic within many `if` statements are very complicated. Readers need to analyze the final results of the conditional expression to decide what statement will be executed in certain conditions.
> Positive example:
```java
// please refer to the pseudo-code as follows
boolean existed = (file.open(fileName, "w") != null) && (...) || (...);
if (existed) {
    ...
}
```
 > Counter example:
```java
if ((file.open(fileName, "w") != null) && (...) || (...)) {
    ...
}
```

5\. **[Recommended]** Performance should be considered when loop statements are used. The following operations are better to be processed outside the loop statement, including object and variable declaration, database connection, `try-catch` statements.

6\. **[Recommended]** Size of input parameters should be checked, especially for batch operations.

7\. **[For Reference]** Input parameters should be checked in following scenarios:
  1) Low-frequency implemented methods.
  2) Overhead of parameter checking could be ignored in long-time execution methods, but if illegal parameters lead to exception, the loss outweighs the gain. Therefore, parameter checking is still recommended in long-time execution methods.
  3) Methods that needs extremely high stability or availability.
  4) Open API methods, including RPC/API/HTTP.
  5) Authority related methods.

8\. **[For Reference]** Cases that input parameters do not require validation:
  1) Methods very likely to be implemented in loops. A note should be included informing that parameter check should be done externally.
  2) Methods in bottom layers are very frequently called so generally do not need to be checked. e.g. If *DAO* layer and *Service* layer is deployed in the same server, parameter check in *DAO* layer can be omitted.
  3) *Private* methods that can only be implemented internally, if all parameters are checked or manageable.

Code Comments
1\. **[Mandatory]** *Javadoc* should be used for classes, class variables and methods. The format should be '/** comment **/', rather than '// xxx'.
> Note: In IDE, *Javadoc* can be seen directly when hovering, which is a good way to improve efficiency.

2\. **[Mandatory]** Abstract methods (including methods in interface) should be commented by *Javadoc*. *Javadoc* should include method instruction, description of parameters, return values and possible exceptions.

3\. **[Mandatory]** Every class should include information of author(s) and date.

4\. **[Mandatory]** Single line comments in a method should be put above the code to be commented, by using `//` and multiple lines by using `/* */`. Alignment for comments should be noticed carefully.

5\. **[Mandatory]** All enumeration type fields should be commented as *Javadoc* style.

6\. **[Recommended]** Local language can be used in comments if English cannot describe the problem properly. Keywords and proper nouns should be kept in English.
 > Counter example: To explain "TCP connection overtime" as "Transmission Control Protocol connection overtime" only makes it more difficult to understand.

7\. **[Recommended]** When code logic changes, comments need to be updated at the same time, especially for the changes of parameters, return value, exception and core logic.

8\. **[For Reference]** Notes need to be added when commenting out code.
> Note: If the code is likely to be recovered later, a reasonable explanation needs to be added. If not, please delete directly because code history will be recorded by *svn* or *git*.

9\. **[For Reference]** Requirements for comments:
  1) Be able to represent design ideas and code logic accurately.
  2) Be able to represent business logic and help other programmers understand quickly. A large section of code without any comment is a disaster for readers. Comments are written for both oneself and other people. Design ideas can be quickly recalled even after a long time. Work can be quickly taken over by other people when needed.

10\. **[For Reference]** Proper naming and clear code structure are self-explanatory. Too many comments need to be avoided because it may cause too much work on updating if code logic changes.
 > Counter example:
```java
// put elephant into fridge
put(elephant, fridge);
```
The name of method and parameters already represent what does the method do, so there is no need to add extra comments.

11\. **[For Reference]** Tags in comments (e.g. TODO, FIXME) need to contain author and time. Tags need to be handled and cleared in time by scanning frequently. Sometimes online breakdown is caused by these unhandled tags.
  1) TODO: TODO means the logic needs to be done, but not finished yet. Actually, TODO is a member of *Javadoc*, although it is not realized in *Javadoc* yet, but has already been widely used. TODO can only be used in class, interface and methods, since it is a *Javadoc* tag.
  2) FIXME: FIXME is used to represent that the code logic is not correct or does not work, should be fixed in time.

Other
1\. **[Mandatory]** When using regex, precompile needs to be done in order to increase the matching performance.

> Note: Do not define `Pattern pattern = Pattern.compile(.);` within method body.

2\. **[Mandatory]** When using attributes of POJO in velocity, use attribute names directly. Velocity engine will invoke `getXxx()` of POJO automatically. In terms of *boolean* attributes, velocity engine will invoke `isXxx()` (Do not use *is* as prefix when naming boolean attributes).
> Note: For wrapper class `Boolean`, velocity engine will invoke `getXxx()` first.

3\. **[Mandatory]** Variables must add exclamatory mark when passing to velocity engine from backend, like \$!{var}.
 > Note: If attribute is *null* or does not exist, \${var} will be shown directly on web pages.

4\. **[Mandatory]** The return type of `Math.random()` is double, value range is *0<=x<1* (0 is possible). If a random integer is required, do not multiply x by 10 then round the result. The correct way is to use `nextInt` or `nextLong` method which belong to Random Object.

5\. **[Mandatory]** Use `System.currentTimeMillis()` to get the current millisecond. Do not use `new Date().getTime()`.
> Note: In order to get a more accurate time, use `System.nanoTime()`. In JDK8, use `Instant` class to deal with situations like time statistics.

6\. **[Recommended]** It is better not to contain variable declaration, logical symbols or any complicated logic in velocity template files.

7\. **[Recommended]** Size needs to be specified when initializing any data structure if possible, in order to avoid memory issues caused by unlimited growth.

8\. **[Recommended]** Codes or configuration that is noticed to be obsoleted should be resolutely removed from projects.
> Note: Remove obsoleted codes or configuration in time to avoid code redundancy.
 > Positive example: For codes which are temporarily removed and likely to be reused, use **`///`** to add a reasonable note.
```java
public static void hello() {
    /// Business is stopped temporarily by the owner.
    // Business business = new Business();
    // business.active();
    System.out.println("it's finished");
}
```

2. Exception and Logs

Exception
1\. **[Mandatory]** Do not catch *Runtime* exceptions defined in *JDK*, such as `NullPointerException` and `IndexOutOfBoundsException`. Instead, pre-check is recommended whenever possible.
> Note: Use try-catch only if it is difficult to deal with pre-check, such as `NumberFormatException`.
> Positive example: ```if (obj != null) {...}```
> Counter example: ```try { obj.method() } catch(NullPointerException e){…}```

2\. **[Mandatory]** Never use exceptions for ordinary control flow. It is ineffective and unreadable.

3\. **[Mandatory]** It is irresponsible to use a try-catch on a big chunk of code. Be clear about the stable and unstable code when using try-catch. The stable code that means no exception will throw. For the unstable code, catch as specific as possible for exception handling.

4\. **[Mandatory]** Do not suppress or ignore exceptions. If you do not want to handle it, then re-throw it. The top layer must handle the exception and translate it into what the user can understand.

5\. **[Mandatory]** Make sure to invoke the rollback if a method throws an Exception.

6\. **[Mandatory]** Closeable resources (stream, connection, etc.) must be handled in *finally* block. Never throw any exception from a *finally* block.
> Note: Use the *try-with-resources* statement to safely handle closeable resources (Java 7+).

7\. **[Mandatory]** Never use *return* within a *finally* block. A *return* statement in a *finally* block will cause exceptions or result in a discarded return value in the *try-catch* block.

8\. **[Mandatory]** The *Exception* type to be caught needs to be the same class or superclass of the type that has been thrown.

9\. **[Recommended]** The return value of a method can be *null*. It is not mandatory to return an empty collection or object. Specify in *Javadoc* explicitly when the method might return *null*. The caller needs to make a *null* check to prevent `NullPointerException`.
> Note: It is caller's responsibility to check the return value, as well as to consider the possibility that remote call fails or other runtime exception occurs.

10\. **[Recommended]** One of the most common errors is `NullPointerException`. Pay attention to the following situations:
  1) If the return type is primitive, return a value of wrapper class may cause `NullPointerException`.
    Counter example: `public int f() { return Integer }` Unboxing a *null* value will throw a `NullPointerException`.
  2) The return value of a database query might be *null*.
  3) Elements in collection may be *null*, even though `Collection.isEmpty()` returns *false*.
  4) Return values from an RPC might be *null*.
  5) Data stored in sessions might by *null*.
  6) Method chaining, like `obj.getA().getB().getC()`, is likely to cause `NullPointerException`.
    Positive example: Use `Optional` to avoid null check and NPE (Java 8+).

11\. **[Recommended]** Use "throw exception" or "return error code". For HTTP or open API providers, "error code" must be used. It is recommended to throw exceptions inside an application. For cross-application RPC calls, result is preferred by encapsulating *isSuccess*, *error code* and brief error messages.
> Note: Benefits to return Result for the RPC methods:
  1) Using the 'throw exception' method will occur a runtime error if the exception is not caught.
  2) If stack information it not attached, allocating custom exceptions with simple error message is not helpful to solve the problem. If stack information is attached, data serialization and transmission performance loss are also problems when frequent error occurs.

12\. **[Recommended]** Do not throw `RuntimeException`, `Exception`, or `Throwable` directly. It is recommended to use well defined custom exceptions such as `DAOException`, `ServiceException`, etc.

13\. **[For Reference]** Avoid duplicate code (Do not Repeat Yourself, also known as DRY principle).
> Note: Copying and pasting code arbitrarily will inevitably lead to duplicated code. If you keep logic in one place, it is easier to change when needed. If necessary, extract common codes to methods, abstract classes or even shared modules.
> Positive example: For a class with a number of public methods that validate parameters in the same way, it is better to extract a method like:
```java
private boolean checkParam (DTO dto) {
    ...
}
```

Logs
1\. **[Mandatory]** Do not use API in log system (Log4j, Logback) directly. API in log framework SLF4J is recommended to use instead, which uses *Facade* pattern and is conducive to keep log processing consistent.
```java
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
private static final Logger logger = LoggerFactory.getLogger(Abc.class);
```

2\. **[Mandatory]** Log files need to be kept for at least 15 days because some kinds of exceptions happen weekly.

3\. **[Mandatory]** Naming conventions of extended logs of an Application (such as RBI, temporary monitoring, access log, etc.):
appName_logType_logName.log
logType: Recommended classifications are *stats*, *desc*, *monitor*, *visit*, etc.
logName: Log description.
Benefits of this scheme: The file name shows what application the log belongs to, type of the log and what purpose is the log used for. It is also conducive for classification and search.
> Positive example: Name of the log file for monitoring the timezone conversion exception in *mppserver* application:
mppserver_monitor_timeZoneConvert.log
> Note: It is recommended to classify logs. Error logs and business logs should be stored separately as far as possible. It is not only easy for developers to view, but also convenient for system monitoring.

4\. **[Mandatory]** Logs at *TRACE / DEBUG / INFO* levels must use either conditional outputs or placeholders.
> Note: `logger.debug ("Processing trade with id: " + id + " symbol: " + symbol);` If the log level is warn, the above log will not be printed. However, it will perform string concatenation operator. `toString()` method of *symbol* will be called, which is a waste of system resources.
> Positive example:
```java
if (logger.isDebugEnabled()) {
    logger.debug("Processing trade with id: " + id + " symbol: " + symbol);
}
```
> Positive example:
```java
logger.debug("Processing trade with id: {} and symbol : {} ", id, symbol);
```

5\. **[Mandatory]** Ensure that additivity attribute of a Log4j logger is set to *false*, in order to avoid redundancy and save disk space.
> Positive example:
`<logger name="com.taobao.ecrm.member.config" additivity="false" \>`

6\. **[Mandatory]** The exception information should contain two types of information: the context information and the exception stack. If you do not want to handle the exception, re-throw it.
> Positive example:
```java
logger.error(various parameters or objects toString + "_" + e.getMessage(), e);
```

7\. **[Recommended]** Carefully record logs. Use *Info* level selectively and do not use *Debug* level in production environment. If *Warn* is used to record business behavior, please pay attention to the size of logs. Make sure the server disk is not over-filled, and remember to delete these logs in time.
> Note: Outputting a large number of invalid logs will have a certain impact on system performance, and is not conducive to rapid positioning problems. Please think about the log: do you really have these logs? What can you do to see this log? Is it easy to troubleshoot problems?

8\. **[Recommended]** Level *Warn* should be used to record invalid parameters, which is used to track data when problem occurs. Level *Error* only records the system logic error, abnormal and other important error messages.

3. MySQL Rules
Table Schema Rules
1\. **[Mandatory]** Columns expressing the concept of *True* or *False*, must be named as *is_xxx*, whose data type should be *unsigned tinyint* (1 is *True*, 0 is *False*).
> Note: All columns with non-negative values must be *unsigned*.

2\. **[Mandatory]** Names of tables and columns must consist of lower case letters, digits or underscores. Names starting with digits and names which contain only digits (no other characters) in between two underscores are not allowed. Columns should be named cautiously, as it is costly to change column names and cannot be released in pre-release environment.
> Positive example: *getter_admin, task_config, level3_name*
> Counter example: *GetterAdmin, taskConfig, level_3_name*

3\. **[Mandatory]** Plural nouns are not allowed as table names.

4\. **[Mandatory]** Keyword, such as *desc*, *range*, *match*, *delayed*, etc., should not be used. It can be referenced from MySQL official document.

5\. **[Mandatory]** The name of primary key index should be prefixed with *pk_*, followed by column name; Unique index should be named by prefixing its column name with *uk_*; And normal index should be formatted as *idx_[column_name]*.
> Note: *pk* means primary key, *uk* means unique key, and *idx* is short for index.

6\. **[Mandatory]** Decimals should be typed as *decimal*. *float* and *double* are not allowed.
> Note: It may have precision loss when float and double numbers are stored, which in turn may lead to incorrect data comparison result. It is recommended to store integral and fractional parts separately when data range to be stored is beyond the range covered by decimal type.

7\. **[Mandatory]** Use *char* if lengths of information to be stored in that column are almost the same.

8\. **[Mandatory]** The length of *varchar* should not exceed 5000, otherwise it should be defined as `text`. It is better to store them in a separate table in order to avoid its effect on indexing efficiency of other columns.

9\. **[Mandatory]** A table must include three columns as following: *id*, *gmt_create* and *gmt_modified*.
> Note: *id* is the primary key, which is *unsigned bigint* and self-incrementing with step length of 1. The type of *gmt_create* and *gmt_modified* should be *DATE_TIME*.

10\. **[Recommended]** It is recommended to define table name as *[table_business_name]_[table_purpose]*.
 > Positive example: *tiger_task* / *tiger_reader* / *mpp_config*

11\. **[Recommended]** Try to define database name same with the application name.

12\. **[Recommended]** Update column comments once column meaning is changed or new possible status values are added.

13\. **[Recommended]** Some appropriate columns may be stored in multiple tables redundantly to improve search performance, but consistency must be concerned. Redundant columns should not be:
  1) Columns with frequent modification.
  2) Columns typed with very long *varchar* or *text*.

> Positive example: Product category names are short, frequently used and with almost never changing/fixed values. They may be stored redundantly in relevant tables to avoid joined queries.

14\. **[Recommended]** Database sharding may only be recommended when there are more than 5 million rows in a single table or table capacity exceeds 2GB.
> Note: Please do not shard during table creation if anticipated data quantity is not to reach this grade.

15\. **[For Reference]** Appropriate *char* column length not only saves database and index storing space, but also improves query efficiency.
> Positive example: Unsigned types could avoid storing negative values mistakenly, but also may cover bigger data representative range.

Object	Age	Recommended data type	Range
human	within 150 years old	unsigned tinyint	unsigned integers: 0 to 255
turtle	hundreds years old	unsigned smallint	unsigned integers: 0 to 65,535
dinosaur fossil	tens of millions years old	unsigned int	unsigned integers: 0 to around 4.29 billion
sun	around 5 billion years old	unsigned bigint	unsigned integers: 0 to around 10^19

Index Rules
1\. **[Mandatory]** Unique index should be used if business logic is applicable.
> Note: Negative impact of unique indices on insert efficiency is neglectable, but it improves query speed significantly. Additionally, even if complete check is done at the application layer, as per Murphy's Law, dirty data might still be produced, as long as there is no unique index.

2\. **[Mandatory]** *JOIN* is not allowed if more than three tables are involved. Columns to be joined must be with absolutely similar data types. Make sure that columns to be joined are indexed.
> Note: Indexing and SQL performance should be considered even if only 2 tables are joined.

3\. **[Mandatory]** Index length must be specified when adding index on *varchar* columns. The index length should be set according to the distribution of data.
> Note: Normally for *char* columns, an index with the length of 20 can distinguish more than 90% data, which is calculated by *count(distinct left(column_name, index_length)) / count(*)*.

4\. **[Mandatory]** *LIKE '%...'* or *LIKE '%...%'* are not allowed when searching with pagination. Search engine can be used if it is really needed.

> Note: Index files have B-Tree's *left most prefix matching* characteristic. Index cannot be applied if left prefix value is not determined.

5\. **[Recommended]** Make use of the index order when using *ORDER BY* clauses. The last columns of *ORDER BY* clauses should be at the end of a composite index. The reason is to avoid the *file_sort* issue, which affects the query performance.
> Positive example: *where a=? and b=? order by c;* Index is: *a_b_c* </br>
> Counter example: The index order will not take effect if the query condition contains a range, e.g., *where a>10 order by b;* Index *a_b* cannot be activated.

6\. **[Recommended]** Make use of *Covering Index* for query to avoid additional query after searching index.
> Note: If we need to check the title of Chapter 11 of a book, do we need turn to the page where Chapter 11 starts? No, because the table of contents actually includes the title, which serves as a covering index.
> Positive example: Index types include *primary key index*, *unique index* and *common index*. *Covering index* pertains to a query effect. When refer to explain result, *using index* may appear in extra columns.

7\. **[Recommended]** Use *late join* or *sub-query* to optimize scenarios with many pages.
> Note: Instead of bypassing *offset* rows, MySQL retrieves totally *offset+N* rows, then drops off offset rows and returns N rows. It is very inefficient when offset is very big. The solution is either limiting the number of pages to be returned, or rewriting SQL statement when page number exceeds a predefined threshold.
> Positive example: Firstly locate the required *id* range quickly, then join:
select a.* from table1 a, (select id from table1 where *some_condition* LIMIT 100000, 20) b where a.id=b.id;

8\. **[Recommended]** The target of SQL performance optimization is that the result type of *EXPLAIN* reaches *REF* level, or *RANGE* at least, or CONSTS if possible.
> Counter example: Pay attention to the type of *INDEX* in *EXPLAIN* result because it is very slow to do a full scan to the database index file, whose performance nearly equals to an all-table scan.
CONSTS: There is at most one matching row, which is read by the optimizer. It is very fast.
REF: The normal index is used.
RANGE: A given range of index are retrieved, which can be used when a key column is compared to a constant by using any of the =, <>, >, >=, <, <=, IS NULL, <=>, BETWEEN, or IN() operators.

9\. **[Recommended]** Put the most discriminative column to the left most when adding a composite index.
> Positive example: For the sub-clause *where a=? and b=?*, if data of column *a* is nearly unique, adding index *idx_a* is enough.
> Note: When *equal* and *non-equal* check both exist in query conditions, put the column in *equal* condition first when adding an index. For example, *where a>? and b=?*, *b* should be put as the 1st column of the index, even if column *a* is more discriminative.

10\. **[For Reference]** Avoid listed below misunderstandings when adding index:
  1) It is false that each query needs one index.
  2) It is false that index consumes story space and degrades *update*, *insert* operations significantly.
  3) It is false that *unique index* should all be achieved from application layer by "check and insert".

SQL Rules
1\. **[Mandatory]** Do not use COUNT(column_name) or COUNT(constant_value) in place of COUNT(*). COUNT(*) is *SQL92* defined standard syntax to count the number of rows. It is not database specific and has nothing to do with *NULL* and *non-NULL*.
> Note: COUNT(*) counts NULL row in, while COUNT(column_name) does not take *NULL* valued row into consideration.

2\. **[Mandatory]** COUNT(distinct column) calculates number of rows with distinct values in this column, excluding *NULL* values. Please note that COUNT(distinct column1, column2) returns *0* if all values of one of the columns are *NULL*, even if the other column contains distinct *non-NULL* values.

3\. **[Mandatory]** When all values of one column are *NULL*, COUNT(column) returns *0*, while SUM(column) returns *NULL*, so pay attention to `NullPointerException` issue when using SUM().
> Positive example: NPE issue could be avoided in this way: *SELECT IF(ISNULL(SUM(g)), 0, SUM(g)) FROM table;*

4\. **[Mandatory]** Use *ISNULL()* to check *NULL* values. Result will be *NULL* when comparing *NULL* with any other values.
> Note:
  1) *NULL<>NULL* returns *NULL*, rather than *false*.
  2) *NULL=NULL* returns *NULL*, rather than *true*.
  3) *NULL<>1* returns *NULL*, rather than *true*.

5\. **[Mandatory]** When coding on DB query with paging logic, it should return immediately once count is *0*, to avoid executing paging query statement followed.

6\. **[Mandatory]** *Foreign key* and *cascade update* are not allowed. All foreign key related logic should be handled in application layer.
> Note: e.g. Student table has *student_id* as primary key, score table has *student_id* as foreign key. When *student.student_id* is updated, *score.student_id update* is also triggered, this is called a *cascading update*. *Foreign key* and *cascading update* are suitable for single machine, low parallel systems, not for distributed, high parallel cluster systems. *Cascading updates* are strong blocked, as it may lead to a DB update storm. *Foreign key* affects DB insertion efficiency.

7\. **[Mandatory]** Stored procedures are not allowed. They are difficult to debug, extend and not portable.

8\. **[Mandatory]** When correcting data, delete and update DB records, *SELECT* should be done first to ensure data correctness.

9\. **[Recommended]** *IN* clause should be avoided. Record set size of the *IN* clause should be evaluated carefully and control it within 1000, if it cannot be avoided.

10\. **[For Reference]** For globalization needs, characters should be represented and stored with *UTF-8*, and be cautious of character number counting.
> Note:
SELECT LENGTH("轻松工作"); returns *12*.
SELECT CHARACTER_LENGTH("轻松工作"); returns *4*.
Use *UTF8MB4* encoding to store emoji if needed, taking into account of its difference from *UTF-8*.

11\. **[For Reference]** *TRUNCATE* is not recommended when coding, even if it is faster than *DELETE* and uses less system, transaction log resource. Because *TRUNCATE* does not have transaction nor trigger DB *trigger*, problems might occur.
> Note: In terms of Functionality, *TRUNCATE TABLE* is similar to *DELETE* without *WHERE* sub-clause.

ORM Rules
1\. **[Mandatory]** Specific column names should be specified during query, rather than using *.
> Note:
1) * increases parsing cost.
2) It may introduce mismatch with *resultMap* when adding or removing query columns.

2\. **[Mandatory]** Name of *Boolean* property of *POJO* classes cannot be prefixed with *is*, while DB column name should prefix with *is*. A mapping between properties and columns is required.

> Note: Refer to rules of *POJO* class and DB column definition, mapping is needed in *resultMap*. Code generated by *MyBatis Generator* might need to be adjusted.

3\. **[Mandatory]** Do not use *resultClass* as return parameters, even if all class property names are the same as DB columns, corresponding DO definition is needed.
> Note: <resultMap>Mapping configuration is needed, to decouple DO definition and table columns, which in turn facilitates maintenance.

4\. **[Mandatory]** Be cautious with parameters in xml configuration. Do not use `${}` in place of `#{}`, `#param#`. SQL injection may happen in this way.

5\. **[Mandatory]** *iBatis* built in *queryForList(String statementName, int start, int size)* is not recommended.
> Note: It may lead to *OOM* issue because its implementation is to retrieve all DB records of statementName's corresponding SQL statement, then start, size subset is applied through subList.
> Positive example: Use *#start#*, *#size#* in *sqlmap.xml*.
```java
Map<String, Object> map = new HashMap<String, Object>();
map.put("start", start);
map.put("size", size);
```

6\. **[Mandatory]** Do not use *HashMap* or *HashTable* as DB query result type.

7\. **[Mandatory]** *gmt_modified* column should be updated with current timestamp simultaneously with DB record update.

8\. **[Recommended]** Do not define a universal table updating interface, which accepts POJO as input parameter, and always update table set *c1=value1, c2=value2, c3=value3, ...* regardless of intended columns to be updated. It is better not to update unrelated columns, because it is error prone, not efficient, and increases *binlog* storage.

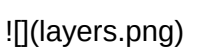
9\. **[For Reference]** Do not overuse *@Transactional*. Because transaction affects QPS of DB, and relevant rollbacks may need be considered, including cache rollback, search engine rollback, message making up, statistics adjustment, etc.

10\. **[For Reference]** *compareValue* of *\<isEqual>* is a constant (normally a number) which is used to compared with property value. *\<isNotEmpty>* means executing corresponding logic when property is not empty and not null. *\<isNotNull>* means executing related logic when property is not null.

4. Project Specification

Application Layers

1\. **[Recommended]** The upper layer depends on the lower layer by default. Arrow means direct dependent. For example: Open Interface can depend on Web Layer, it can also directly depend on Service Layer, etc.:



- **Open Interface:** In this layer service is encapsulate to be exposed as RPC interface, or HTTP interface through Web Layer; The layer also implements gateway security control, flow control, etc.
- **View:** In this layer templates of each terminal render and execute. Rendering approaches include velocity rendering, JS rendering, JSP rendering, and mobile display, etc.
- **Web Layer:** This layer is mainly used to implement forward access control, basic parameter verification, or non-reusable services.
- **Service Layer:** In this layer concrete business logic is implemented.
- **Manager Layer:** This layer is the common business process layer, which contains the following features:
  1) Encapsulates third-party service, to preprocess return values and exceptions;
  2) The precipitation of general ability of Service Layer, such as caching solutions, middleware general processing;
  3) Interacts with the *DAO layer*, including composition and reuse of multiple *DAO* classes.
- **DAO Layer**: Data access layer, data interacting with MySQL, Oracle and HBase.
- **External interface or third-party platform:** This layer includes RPC open interface from other departments or companies.

2\. **[For Reference]** Many exceptions in the *DAO Layer* cannot be caught by using a fine-grained exception class. The recommended way is to use `catch (Exception e)`, and `throw new DAOException(e)`. In these cases, there is no need to print the log because the log should have been caught and printed in *Manager Layer/Service Layer*.
   Logs about exception in *Service Layer* must be recorded with as much information about the parameters as possible to make debugging simpler.
   If *Manager Layer* and *Service Layer* are deployed in the same server, log logic should be consistent with *DAO Layer*. If they are deployed separately, log logic should be consistent with each other.
    In *Web Layer* Exceptions cannot be thrown, because it is already on the top layer and there is no way to deal with abnormal situations. If the exception is likely to cause failure when rendering the page, the page should be redirected to a friendly error page with the friendly error information.
   In *Open Interface* exceptions should be handled by using *error code* and *error message*.

3\. **[For Reference]** Layers of Domain Model:

- **DO (Data Object):** Corresponding to the database table structure, the data source object is transferred upward through *DAO* Layer.
- **DTO (Data Transfer Object):** Objects which are transferred upward by *Service Layer* and Manager Layer.
- **BO (Business Object):** Objects that encapsulate business logic, which can be outputted by *Service Layer*.
- **Query**: Data query objects that carry query request from upper layers. Note: Prohibit the use of `Map` if there are more than 2 query conditions.
- **VO (View Object):** Objects that are used in *Display Layer*, which is normally transferred from *Web Layer*.

Library Specification
1\. **[Mandatory]** Rules of defining GAV:
  1) **G**roupID: `com.{company/BU}.{business line}.{sub business line}`, at most 4 levels
> Note: {company/BU} for example: such business unit level as Alibaba, Taobao, Tmall, Aliexpress and so on; sub-business line is optional.
> Positive example: com.taobao.tddl
  com.alibaba.sourcing.multilang

  2) **A**rtifactID: Product name - module name.
> Positive example: `tc-client` / `uic-api` / `tair-tool`

  3) **V**ersion: Please refer to below

2\. **[Mandatory]** Library naming convention: `prime version number`.`secondary version number`.`revision number`
  1) **prime version number**: Need to change when there are incompatible API modification, or new features that can change the product direction.
  2) **secondary version number**: Changed for backward compatible modification.
  3) **revision number**: Changed for fixing bugs or adding features that do not modify the method signature and maintain API compatibility.
 > Note: The initial version must be 1.0.0, rather than 0.0.1.

3\. **[Mandatory]** Online applications should not depend on *SNAPSHOT* versions (except for security packages); Official releases must be verified from central repository, to make sure that the RELEASE version number has continuity. Version numbers are not allowed to be overridden.

> Note: Avoid using *SNAPSHOT* is to ensure the idempotent of application release. In addition, it can speed up the code compilation when packaging. If the current version is *1.3.3*, then the number for the next version can be *1.3.4*, *1.4.0* or *2.0.0*.

4\. **[Mandatory]** When adding or upgrading libraries, remain the versions of dependent libraries unchanged if not necessary. If there is a change, it must be correctly assessed and checked. It is recommended to use command *dependency:resolve* to compare the information before and after. If the result is identical, find out the differences with the command *dependency:tree* and use *\<excludes\>* to eliminate unnecessary libraries.

5\. **[Mandatory]** Enumeration types can be defined or used for parameter types in libraries, but cannot be used for interface return types (POJO that contains enumeration types is also not allowed).

6\. **[Mandatory]** When a group of libraries are used, a uniform version variable need to be defined to avoid the inconsistency of version numbers.
> Note: When using `springframework-core`, `springframework-context`, `springframework-beans` with the same version, a uniform version variable \${spring.version} is recommended to be used.

7\. **[Mandatory]** For the same *GroupId* and *ArtifactId*, *Version* must be the same in sub-projects.
> Note: During local debugging, version number specified in sub-project is used. But when merged into a war, only one version number is applied in the lib directory. Therefore, such case might occur that offline debugging is correct, but failure occurs after online launch.

8\. **[Recommended]** The declaration of dependencies in all *POM* files should be placed in *\<dependencies\>* block. Versions of dependencies should be specified in *\<dependencyManagement\>* block.
> Note: In *\<dependencyManagement\>* versions of dependencies are specified, but dependencies are not imported. Therefore, in sub-projects dependencies need to be declared explicitly, version and scope of which are read from the parent *POM*. All declarations in *\<dependencies\>* in the main *POM* will be automatically imported and inherited by all subprojects by default.

9\. **[Recommended]** It is recommended that libraries do not include configuration, at least do not increase the configuration items.

10\. **[For Reference]** In order to avoid the dependency conflict of libraries, the publishers should follow the principles below:
  1) **Simple and controllable:** Remove all unnecessary API and dependencies, only contain Service API, necessary domain model objects, Utils classes, constants, enumerations, etc. If other libraries must be included, better to make the scope as *provided*

and let users to depend on the specific version number. Do not depend on specific log implementation, only depend on the log framework instead.
  2) **Stable and traceable:** Change log of each version should be recorded. Make it easy to check the library owner and where the source code is. Libraries packaged in the application should not be changed unless the user updates the version proactively.

Server Specification
1\. **[Recommended]** It is recommended to reduce the *time_wait* value of the *TCP* protocol for high concurrency servers.
> Note: By default the operating system will close connection in *time_wait* state after 240 seconds. In high concurrent situation, the server may not be able to establish new connections because there are too many connections in *time_wait* state, so the value of *time_wait* needs to be reduced.
> Positive example: Modify the default value (Sec) by modifying the *_etcsysctl.conf* file on Linux servers:
`net.ipv4.tcp\_fin\_timeout = 30`

2\. **[Recommended]** Increase the maximum number of *File Descriptors* supported by the server.
> Note: Most operating systems are designed to manage *TCP/UDP* connections as a file, i.e. one connection corresponds to one *File Descriptor*. The maximum number of *File Descriptors* supported by the most Linux servers is 1024. It is easy to make an "open too many files" error because of the lack of *File Descriptor* when the number of concurrent connections is large, which would cause that new connections cannot be established.

3\. **[Recommended]** Set `-XX:+HeapDumpOnOutOfMemoryError` parameter for *JVM*, so *JVM* will output dump information when *OOM* occurs.
> Note: *OOM* does not occur very often, only once in a few months. The dump information printed when error occurs is very valuable for error checking.

4\. **[For Reference]** Use *forward* for internal redirection and URL assembly tools for external redirection. Otherwise there will be problems about URL maintaining inconsistency and potential security risks.

5. Security Specification

1\. **[Mandatory]** User-owned pages or functions must be authorized.
> Note: Prevent the access and manipulation of other people's data without authorization check, e.g. view or modify other people's orders.

2\. **[Mandatory]** Direct display of user sensitive data is not allowed. Displayed data must be desensitized.

> Note: Personal phone number should be displayed as: 158****9119. The middle 4 digits are hidden to prevent privacy leaks.

3\. **[Mandatory]** *SQL* parameter entered by users should be checked carefully or limited by *METADATA*, to prevent *SQL* injection. Database access by string concatenation *SQL* is forbidden.

4\. **[Mandatory]** Any parameters input by users must go through validation check.
> Note: Ignoring parameter check may cause:
>
 * memory leak because of excessive page size
 * slow database query because of malicious *order by*
 * arbitrary redirection
 * *SQL* injection
 * deserialize injection
 * *ReDoS*

> Note: In Java *regular expressions* is used to validate client input. Some *regular expressions* can validate common user input without any problem, but it could lead to a dead cycle if the attacker uses a specially constructed string to verify.

5\. **[Mandatory]** It is forbidden to output user data to *HTML* page without security filtering or proper escaping.

6\. **[Mandatory]** Form and *AJAX* submission must be filtered by *CSRF* security check.
> Note: *CSRF* (Cross-site Request Forgery) is a kind of common programming flaw. For applications/websites with *CSRF* leaks, attackers can construct *URL* in advance and modify the user parameters in database as long as the victim user visits without notice.

7\. **[Mandatory]** It is necessary to use the correct anti-replay restrictions, such as number restriction, fatigue control, verification code checking, to avoid abusing of platform resources, such as text messages, e-mail, telephone, order, payment.
> Note: For example, if there is no limitation to the times and frequency when sending verification codes to mobile phones, users might be bothered and *SMS* platform resources might be wasted.

8\. **[Recommended]** In scenarios when users generate content (e.g., posting, comment, instant messages), anti-scam word filtering and other risk control strategies must be applied.

1. Project guidelines

1.1 Project structure

New projects should follow the Android Gradle project structure that is defined on the [Android Gradle plugin user guide](http://tools.android.com/tech-docs/new-build-system/user-guide#TOC-Project-Structure). The [ribot Boilerplate](https://github.com/ribot/android-boilerplate) project is a good reference to start from.

1.2 File naming

1.2.1 Class files
Class names are written in [UpperCamelCase](http://en.wikipedia.org/wiki/CamelCase).

For classes that extend an Android component, the name of the class should end with the name of the component; for example: `SignInActivity`, `SignInFragment`, `ImageUploaderService`, `ChangePasswordDialog`.

1.2.2 Resources files

Resources file names are written in __lowercase_underscore__.

1.2.2.1 Drawable files

Naming conventions for drawables:

Asset Type	Prefix	Example
Action bar	`ab_`	`ab_stacked.9.png`
Button	`btn_`	`btn_send_pressed.9.png`
Dialog	`dialog_`	`dialog_top.9.png`
Divider	`divider_`	`divider_horizontal.9.png`
Icon	`ic_`	`ic_star.png`
Menu	`menu_ `	`menu_submenu_bg.9.png`
Notification	`notification_`	`notification_bg.9.png`
Tabs	`tab_`	`tab_pressed.9.png`

Naming conventions for icons (taken from [Android iconography guidelines](http://developer.android.com/design/style/iconography.html)):

Asset Type	Prefix	Example

Icons	`ic_`	`ic_star.png`
Launcher icons	`ic_launcher`	`ic_launcher_calendar.png`
Menu icons and Action Bar icons	`ic_menu`	`ic_menu_archive.png`
Status bar icons	`ic_stat_notify`	`ic_stat_notify_msg.png`
Tab icons	`ic_tab`	`ic_tab_recent.png`
Dialog icons	`ic_dialog`	`ic_dialog_info.png`

Naming conventions for selector states:

State	Suffix	Example
Normal	`_normal`	`btn_order_normal.9.png`
Pressed	`_pressed`	`btn_order_pressed.9.png`
Focused	`_focused`	`btn_order_focused.9.png`
Disabled	`_disabled`	`btn_order_disabled.9.png`
Selected	`_selected`	`btn_order_selected.9.png`

1.2.2.2 Layout files

Layout files should match the name of the Android components that they are intended for but moving the top level component name to the beginning. For example, if we are creating a layout for the `SignInActivity`, the name of the layout file should be `activity_sign_in.xml`.

Component	Class Name	Layout Name
Activity	`UserProfileActivity`	`activity_user_profile.xml`
Fragment	`SignUpFragment`	`fragment_sign_up.xml`
Dialog	`ChangePasswordDialog`	`dialog_change_password.xml`
AdapterView item	---	`item_person.xml`
Partial layout	---	`partial_stats_bar.xml`

A slightly different case is when we are creating a layout that is going to be inflated by an `Adapter`, e.g to populate a `ListView`. In this case, the name of the layout should start with `item_`.

Note that there are cases where these rules will not be possible to apply. For example, when creating layout files that are intended to be part of other layouts. In this case you should use the prefix `partial_`.

1.2.2.3 Menu files

Similar to layout files, menu files should match the name of the component. For example, if we are defining a menu file that is going to be used in the `UserActivity`, then the name of the file should be `activity_user.xml`

A good practice is to not include the word `menu` as part of the name because these files are already located in the `menu` directory.

1.2.2.4 Values files

Resource files in the values folder should be __plural__, e.g. `strings.xml`, `styles.xml`, `colors.xml`, `dimens.xml`, `attrs.xml`

2 Code guidelines

2.1 Java language rules

2.1.1 Don't ignore exceptions

You must never do the following:

```java
void setServerPort(String value) {
    try {
        serverPort = Integer.parseInt(value);
    } catch (NumberFormatException e) { }
}
```

While you may think that your code will never encounter this error condition or that it is not important to handle it, ignoring exceptions like above creates mines in your code for someone else to trip over some day. You must handle every Exception in your code in some principled way. The specific handling varies depending on the case. - ([Android code style guidelines](https://source.android.com/source/code-style.html))

See alternatives [here](https://source.android.com/source/code-style.html#dont-ignore-exceptions).

2.1.2 Don't catch generic exception

You should not do this:

```java
try {
    someComplicatedIOFunction();        // may throw IOException
```

```
    someComplicatedParsingFunction();   // may throw ParsingException
    someComplicatedSecurityFunction();  // may throw SecurityException
    // phew, made it all the way
} catch (Exception e) {                 // I'll just catch all exceptions
    handleError();                      // with one generic handler!
}
```

See the reason why and some alternatives [here](https://source.android.com/source/code-style.html#dont-catch-generic-exception)

2.1.3 Don't use finalizers

We don't use finalizers. There are no guarantees as to when a finalizer will be called, or even that it will be called at all. In most cases, you can do what you need from a finalizer with good exception handling. If you absolutely need it, define a `close()` method (or the like) and document exactly when that method needs to be called. See `InputStream` for an example. In this case it is appropriate but not required to print a short log message from the finalizer, as long as it is not expected to flood the logs. - ([Android code style guidelines](https://source.android.com/source/code-style.html#dont-use-finalizers))

2.1.4 Fully qualify imports

This is bad: `import foo.*;`

This is good: `import foo.Bar;`

See more info [here](https://source.android.com/source/code-style.html#fully-qualify-imports)

2.2 Java style rules

2.2.1 Fields definition and naming

Fields should be defined at the __top of the file__ and they should follow the naming rules listed below.

* Private, non-static field names start with __m__.
* Private, static field names start with __s__.
* Other fields start with a lower case letter.
* Static final fields (constants) are ALL_CAPS_WITH_UNDERSCORES.

Example:

```java
public class MyClass {
    public static final int SOME_CONSTANT = 42;
    public int publicField;
    private static MyClass sSingleton;
    int mPackagePrivate;
    private int mPrivate;
    protected int mProtected;
}
```

2.2.3 Treat acronyms as words

Good	Bad
`XmlHttpRequest`	`XMLHTTPRequest`
`getCustomerId`	`getCustomerID`
`String url`	`String URL`
`long id`	`long ID`

2.2.4 Use spaces for indentation

Use __4 space__ indents for blocks:

```java
if (x == 1) {
    x++;
}
```

Use __8 space__ indents for line wraps:

```java
Instrument i =
        someLongExpression(that, wouldNotFit, on, one, line);
```

2.2.5 Use standard brace style

Braces go on the same line as the code before them.

```java
class MyClass {
    int func() {
```

```
        if (something) {
            // ...
        } else if (somethingElse) {
            // ...
        } else {
            // ...
        }
    }
}
```

Braces around the statements are required unless the condition and the body fit on one line.

If the condition and the body fit on one line and that line is shorter than the max line length, then braces are not required, e.g.

```java
if (condition) body();
```

This is __bad__:

```java
if (condition)
    body();  // bad!
```

2.2.6 Annotations

2.2.6.1 Annotations practices

According to the Android code style guide, the standard practices for some of the predefined annotations in Java are:

* `@Override`: The @Override annotation __must be used__ whenever a method overrides the declaration or implementation from a super-class. For example, if you use the @inheritdocs Javadoc tag, and derive from a class (not an interface), you must also annotate that the method @Overrides the parent class's method.

* `@SuppressWarnings`: The @SuppressWarnings annotation should only be used under circumstances where it is impossible to eliminate a warning. If a warning passes this "impossible to eliminate" test, the @SuppressWarnings annotation must be used, so as to ensure that all warnings reflect actual problems in the code.

More information about annotation guidelines can be found [here](http://source.android.com/source/code-style.html#use-standard-java-annotations).

2.2.6.2 Annotations style

__Classes, Methods and Constructors__

When annotations are applied to a class, method, or constructor, they are listed after the documentation block and should appear as __one annotation per line__ .

```java
/* This is the documentation block about the class */
@AnnotationA
@AnnotationB
public class MyAnnotatedClass { }
```

__Fields__

Annotations applying to fields should be listed __on the same line__, unless the line reaches the maximum line length.

```java
@Nullable @Mock DataManager mDataManager;
```

2.2.7 Limit variable scope

The scope of local variables should be kept to a minimum (Effective Java Item 29). By doing so, you increase the readability and maintainability of your code and reduce the likelihood of error. Each variable should be declared in the innermost block that encloses all uses of the variable.

Local variables should be declared at the point they are first used. Nearly every local variable declaration should contain an initializer. If you don't yet have enough information to initialize a variable sensibly, you should postpone the declaration until you do. - ([Android code style guidelines](https://source.android.com/source/code-style.html#limit-variable-scope))

2.2.8 Order import statements

If you are using an IDE such as Android Studio, you don't have to worry about this because your IDE is already obeying these rules. If not, have a look below.

The ordering of import statements is:

1. Android imports
2. Imports from third parties (com, junit, net, org)
3. java and javax
4. Same project imports

To exactly match the IDE settings, the imports should be:

* Alphabetically ordered within each grouping, with capital letters before lower case letters (e.g. Z before a).
* There should be a blank line between each major grouping (android, com, junit, net, org, java, javax).

More info [here](https://source.android.com/source/code-style.html#limit-variable-scope)

2.2.9 Logging guidelines

Use the logging methods provided by the `Log` class to print out error messages or other information that may be useful for developers to identify issues:

* `Log.v(String tag, String msg)` (verbose)
* `Log.d(String tag, String msg)` (debug)
* `Log.i(String tag, String msg)` (information)
* `Log.w(String tag, String msg)` (warning)
* `Log.e(String tag, String msg)` (error)

As a general rule, we use the class name as tag and we define it as a `static final` field at the top of the file. For example:

```java
public class MyClass {
    private static final String TAG = MyClass.class.getSimpleName();

    public myMethod() {
        Log.e(TAG, "My error message");
    }
}
```

VERBOSE and DEBUG logs __must__ be disabled on release builds. It is also recommended to disable INFORMATION, WARNING and ERROR logs but you may want to keep them enabled if you think they may be useful to identify issues on release builds. If you decide to leave them enabled, you have to make sure that they are not leaking private information such as email addresses, user ids, etc.

To only show logs on debug builds:

```java
if (BuildConfig.DEBUG) Log.d(TAG, "The value of x is " + x);
```

2.2.10 Class member ordering

There is no single correct solution for this but using a __logical__ and __consistent__ order will improve code learnability and readability. It is recommendable to use the following order:

1. Constants
2. Fields
3. Constructors
4. Override methods and callbacks (public or private)
5. Public methods
6. Private methods
7. Inner classes or interfaces

Example:

```java
public class MainActivity extends Activity {

    private static final String TAG = MainActivity.class.getSimpleName();

    private String mTitle;
    private TextView mTextViewTitle;

    @Override
    public void onCreate() {
        ...
    }

    public void setTitle(String title) {
            mTitle = title;
    }

    private void setUpView() {
        ...
    }

    static class AnInnerClass {
```

```
    }

}
```

If your class is extending an __Android component__ such as an Activity or a Fragment, it is a good practice to order the override methods so that they __match the component's lifecycle__. For example, if you have an Activity that implements `onCreate()`, `onDestroy()`, `onPause()` and `onResume()`, then the correct order is:

```java
public class MainActivity extends Activity {

        //Order matches Activity lifecycle
    @Override
    public void onCreate() {}

    @Override
    public void onResume() {}

    @Override
    public void onPause() {}

    @Override
    public void onDestroy() {}

}
```

2.2.11 Parameter ordering in methods

When programming for Android, it is quite common to define methods that take a `Context`. If you are writing a method like this, then the __Context__ must be the __first__ parameter.

The opposite case are __callback__ interfaces that should always be the __last__ parameter.

Examples:

```java
// Context always goes first
public User loadUser(Context context, int userId);

// Callbacks always go last
```

public void loadUserAsync(Context context, int userId, UserCallback callback);
```

### 2.2.13 String constants, naming, and values

Many elements of the Android SDK such as `SharedPreferences`, `Bundle`, or `Intent` use a key-value pair approach so it's very likely that even for a small app you end up having to write a lot of String constants.

When using one of these components, you __must__ define the keys as a `static final` fields and they should be prefixed as indicated below.

| Element | Field Name Prefix |
| ---------- | ---------- |
| SharedPreferences | `PREF_` |
| Bundle | `BUNDLE_` |
| Fragment Arguments | `ARGUMENT_` |
| Intent Extra | `EXTRA_` |
| Intent Action | `ACTION_` |

Note that the arguments of a Fragment - `Fragment.getArguments()` - are also a Bundle. However, because this is a quite common use of Bundles, we define a different prefix for them.

Example:

```java
// Note the value of the field is the same as the name to avoid duplication issues
static final String PREF_EMAIL = "PREF_EMAIL";
static final String BUNDLE_AGE = "BUNDLE_AGE";
static final String ARGUMENT_USER_ID = "ARGUMENT_USER_ID";

// Intent-related items use full package name as value
static final String EXTRA_SURNAME = "com.myapp.extras.EXTRA_SURNAME";
static final String ACTION_OPEN_USER = "com.myapp.action.ACTION_OPEN_USER";
```

2.2.14 Arguments in Fragments and Activities

When data is passed into an `Activity` or `Fragment` via an `Intent` or a `Bundle`, the keys for the different values __must__ follow the rules described in the section above.

When an `Activity` or `Fragment` expects arguments, it should provide a `public static` method that facilitates the creation of the relevant `Intent` or `Fragment`.
```

In the case of Activities the method is usually called `getStartIntent()`:

```java
public static Intent getStartIntent(Context context, User user) {
        Intent intent = new Intent(context, ThisActivity.class);
        intent.putParcelableExtra(EXTRA_USER, user);
        return intent;
}
```

For Fragments it is named `newInstance()` and handles the creation of the Fragment with the right arguments:

```java
public static UserFragment newInstance(User user) {
        UserFragment fragment = new UserFragment();
        Bundle args = new Bundle();
        args.putParcelable(ARGUMENT_USER, user);
        fragment.setArguments(args)
        return fragment;
}
```

__Note 1__: These methods should go at the top of the class before `onCreate()`.

__Note 2__: If we provide the methods described above, the keys for extras and arguments should be `private` because there is not need for them to be exposed outside the class.

2.2.15 Line length limit

Code lines should not exceed __100 characters__. If the line is longer than this limit there are usually two options to reduce its length:

* Extract a local variable or method (preferable).
* Apply line-wrapping to divide a single line into multiple ones.

There are two __exceptions__ where it is possible to have lines longer than 100:

* Lines that are not possible to split, e.g. long URLs in comments.
* `package` and `import` statements.

2.2.15.1 Line-wrapping strategies

There isn't an exact formula that explains how to line-wrap and quite often different solutions are valid. However there are a few rules that can be applied to common cases.

__Break at operators__

When the line is broken at an operator, the break comes __before__ the operator. For example:

```java
int longName = anotherVeryLongVariable + anEvenLongerOne - thisRidiculousLongOne
        + theFinalOne;
```

__Assignment Operator Exception__

An exception to the `break at operators` rule is the assignment operator `=`, where the line break should happen __after__ the operator.

```java
int longName =
        anotherVeryLongVariable + anEvenLongerOne - thisRidiculousLongOne + theFinalOne;
```

__Method chain case__

When multiple methods are chained in the same line - for example when using Builders - every call to a method should go in its own line, breaking the line before the `.`

```java
Picasso.with(context).load("http://ribot.co.uk/images/sexyjoe.jpg").into(imageView);
```

```java
Picasso.with(context)
        .load("http://ribot.co.uk/images/sexyjoe.jpg")
        .into(imageView);
```

__Long parameters case__

When a method has many parameters or its parameters are very long, we should break the line after every comma `,`

```java
```

```
loadPicture(context, "http://ribot.co.uk/images/sexyjoe.jpg", mImageViewProfilePicture,
clickListener, "Title of the picture");
```

```java
loadPicture(context,
        "http://ribot.co.uk/images/sexyjoe.jpg",
        mImageViewProfilePicture,
        clickListener,
        "Title of the picture");
```

2.2.16 RxJava chains styling

Rx chains of operators require line-wrapping. Every operator must go in a new line and the line should be broken before the `.`

```java
public Observable<Location> syncLocations() {
    return mDatabaseHelper.getAllLocations()
            .concatMap(new Func1<Location, Observable<? extends Location>>() {
                @Override
                public Observable<? extends Location> call(Location location) {
                    return mRetrofitService.getLocation(location.id);
                }
            })
            .retry(new Func2<Integer, Throwable, Boolean>() {
                @Override
                public Boolean call(Integer numRetries, Throwable throwable) {
                    return throwable instanceof RetrofitError;
                }
            });
}
```

2.3 XML style rules

2.3.1 Use self closing tags

When an XML element doesn't have any contents, you __must__ use self closing tags.

This is good:

```xml
```

```
<TextView
        android:id="@+id/text_view_profile"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content" />
```

This is __bad__ :

```xml
<!-- Don\'t do this! -->
<TextView
   android:id="@+id/text_view_profile"
   android:layout_width="wrap_content"
   android:layout_height="wrap_content" >
</TextView>
```

2.3.2 Resources naming

Resource IDs and names are written in __lowercase_underscore__.

2.3.2.1 ID naming

IDs should be prefixed with the name of the element in lowercase underscore. For example:

Element	Prefix
`TextView`	`text_`
`ImageView`	`image_`
`Button`	`button_`
`Menu`	`menu_`

Image view example:

```xml
<ImageView
   android:id="@+id/image_profile"
   android:layout_width="wrap_content"
   android:layout_height="wrap_content" />
```

Menu example:

```xml
<menu>
        <item
      android:id="@+id/menu_done"
      android:title="Done" />
</menu>
```

2.3.2.2 Strings

String names start with a prefix that identifies the section they belong to. For example `registration_email_hint` or `registration_name_hint`. If a string __doesn't belong__ to any section, then you should follow the rules below:

Prefix	Description
`error_`	An error message
`msg_`	A regular information message
`title_`	A title, i.e. a dialog title
`action_`	An action such as "Save" or "Create"

2.3.2.3 Styles and Themes

Unlike the rest of resources, style names are written in __UpperCamelCase__.

2.3.3 Attributes ordering

As a general rule you should try to group similar attributes together. A good way of ordering the most common attributes is:

1. View Id
2. Style
3. Layout width and layout height
4. Other layout attributes, sorted alphabetically
5. Remaining attributes, sorted alphabetically

2.4 Tests style rules

2.4.1 Unit tests

Test classes should match the name of the class the tests are targeting, followed by `Test`. For example, if we create a test class that contains tests for the `DatabaseHelper`, we should name it `DatabaseHelperTest`.

Test methods are annotated with `@Test` and should generally start with the name of the method that is being tested, followed by a precondition and/or expected behaviour.

* Template: `@Test void methodNamePreconditionExpectedBehaviour()`
* Example: `@Test void signInWithEmptyEmailFails()`

Precondition and/or expected behaviour may not always be required if the test is clear enough without them.

Sometimes a class may contain a large amount of methods, that at the same time require several tests for each method. In this case, it's recommendable to split up the test class into multiple ones. For example, if the `DataManager` contains a lot of methods we may want to divide it into `DataManagerSignInTest`, `DataManagerLoadUsersTest`, etc. Generally you will be able to see what tests belong together because they have common [test fixtures](https://en.wikipedia.org/wiki/Test_fixture).

2.4.2 Espresso tests

Every Espresso test class usually targets an Activity, therefore the name should match the name of the targeted Activity followed by `Test`, e.g. `SignInActivityTest`

When using the Espresso API it is a common practice to place chained methods in new lines.

```java
onView(withId(R.id.view))
        .perform(scrollTo())
        .check(matches(isDisplayed()))
```

License

```

```

Unless required by applicable law or agreed to in writing, software
distributed under the License is distributed on an "AS IS" BASIS,
WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
See the License for the specific language governing permissions and
limitations under the License.
```

Erlang Coding Standards & Guidelines
==========

Suggested reading material: http://www.erlang.se/doc/programming_rules.shtml

***

Table of Contents:
```

Contact Us

If you find any **bugs** or have a **problem** while using this library, please [open an issue](https://github.com/inaka/erlang_guidelines/issues/new) in this repo (or a pull request :)).

And you can check all of our open-source projects at inaka.github.io

Conventions & Rules

These are _"Things that may be used as reason to reject a Pull Request"_.

Source Code Layout

***
Maintain existing style
> When editing a module written by someone else, stick to the style in which it was written. If a project has an overall style, stick to that when writing new modules as well.

Examples: [existing_style](src/existing_style.erl)

Reasoning: It's better to keep a module that just looks ugly to you than to have a module that looks half ugly to you, half ugly to somebody else.

***
Spaces over tabs
> Spaces over tabs, 2 space indentation.

Examples: [indent](src/indent.erl)

Reasoning: This is *not* intended to allow deep nesting levels in the code. 2 spaces are enough if the code is clean enough, and the code looks more concise, allowing more characters in the same line.

***

Use your spacebar
> Surround operators and commas with spaces.

Examples: [spaces](src/spaces.erl)

Reasoning: It produces cleaner code that's easier to find / read / etc.

***
No Trailing Whitespace
> Remove trailing whitespaces from your lines

Examples: [trailing_whitespace](src/trailing_whitespace.erl)

Reasoning: It's commit noise. As a reference, check out [this long argument](https://programmers.stackexchange.com/questions/121555/why-is-trailing-whitespace-a-big-deal).

***
100 column per line
> Stick to 100 chars per line, maximum.

Examples: [col_width](src/col_width.erl)

Reasoning: Excessively long lines are a pain to deal with: you either have to scroll horizontally while editing, or live with ugly line wrapping at arbitrary points.
The 100 character limit also keeps lines short enough that you can comfortably work with two source files side by side on a typical laptop screen, or three on a 1080p display.

***

More, smaller functions over case expressions
> Use pattern-maching in clause functions rather than case's. Specially important if the case is:
> - the top-level expression of the function
> - huge

Examples: [smaller_functions](src/smaller_functions.erl)

Reasoning: it is usually the case that a case in a function body represents some sort of decision, and functions should be as simple as possible. If each branch of a decision's outcome is implemented as a function clause instead of as a case clause, the decision may be given a meaningful name. In other words, the case is acting as an 'anonymous function', which unless they are being used in the context of a higher-order function, merely obscure meaning.

***

Group functions logically
> Try to always separate **unexported** and **exported** functions in groups, with the exported ones first, unless it helps readability and code discovery.

Examples: [grouping_functions](src/grouping_functions)

Reasoning: Well structured code is easier to read/understand/modify.

***
Get your types together
> Place all types at the beginning of the file

Examples: [type_placement](src/type_placement.erl)

Reasoning: Types are used to define data structures that will most likely be used by multiple functions on the module, so their definition can not be tied to just one of them. Besides it's a good practice to place them in code in a similar way as the documentation presents them and edoc puts types at the beginning of each module documentation

***
No God modules
> Don't design your system using **god** modules (modules that have a huge number of functions and/or deal with very unrelated things)

Examples: [god](src/god.erl)

Reasoning: God modules, like god objects, are modules that do too much or know too much. God modules usually come into existence by feature accretion. A beautiful, to-the-point module with one job, one responsibility done well, gains a function. Then another, which does the same thing but with different parameters. Then one day, you have a 6000-line module with 500 functions. Having modules (and functions) that do one and only one thing well makes it easy to explore and reason about code, and thus maintain it.

***
Simple unit tests
> Single responsibility applies to tests as well. When writing **unit** tests, keep them short and don't put more than 1 or 2 asserts per test

Examples: [test_SUITE](src/test_SUITE.erl)

Reasoning: Multiple tests can identify multiple errors in one run, if you put all the things you want to test into one test you'll have to fix one thing at a time until the test passes.

***

Honor DRY
> Don't write the same code in many places, use functions and variables for that

Examples: [dry](src/dry.erl)

Reasoning: This convention is specifically put in this list (instead of treat it as a [great idea](#great-ideas)) so that reviewers can reject PRs that include the same code several times or PRs that re-implement something that they know it's already done somewhere else.

***
Group modules in subdirectories by functionality
> When having lots of modules, use subdirectories for them, named with a nice descriptive name for what that "package" does.

Reasoning: That way it's easier to find what you need and determine what a certain module does.

Note: Remember to properly configure your ``Emakefile`` to handle that, if you use it.

***
Header files
> Header files:
> - SHOULD NOT include type definitions nor record definitions nor function definitions.
> - MAY include macros definitions, although macros should be [avoided](#no-macros).

Examples: [headers](src/headers.hrl)

Reasoning:
Type definitions should be located in the modules where the data and its associated functions are defined. In type specs types may be module-prefixed which also makes it clear where the data type is defined, so there is no reason to need to share them via headers.

Including record definitions in header files promotes sharing internal details of those records across modules, increasing coupling and preventing encapsulation, in turn making it more difficult to change and maintain the code. Records should be defined in their own modules which should provide an opaque data type and functions to access and manipulate the record.

Function definitions should most definitely not be included in header files because it leads to code duplication.

Syntax

Erlang syntax is horrible amirite? So you might as well make the best of it, right? _Right_?

***
Don't write spaghetti code
> Don't write spaghetti code (A list comprehension with a case inside, or blocks with begin/end, and nested stuff)

Examples: [spaghetti](src/spaghetti.erl)

Reasoning: Spaghetti code is harder to read, understand and edit. The function callgraph for your program should strive to be a directed acyclic graph.

***
Avoid dynamic calls
> If there is no specific need for it, don't use dynamic function calling.

Examples: [dyn_calls](src/dyn_calls.erl)

Reasoning: Dynamic calls can't be checked by [``xref``](http://erlang.org/doc/apps/tools/xref_chapter.html), one of the most useful tools in the Erlang world. ``xref`` is a cross reference checking/observing tool.

***
Avoid deep nesting
> Try not to nest more than 3 levels deep.

Examples: [nesting](src/nesting.erl)

Reasoning: Nested levels indicate deep logic in a function, too many decisions taken or things done in a single function. This hinders not only readability, but also maintainability (making changes) and debugging, and writing unit tests.
See also: [More, smaller functions over case expressions](#more-smaller-functions-over-case-expressions).

***
Avoid if expressions
> Don't use `if`.

Examples: [no_if](src/no_if.erl)

Reasoning: In some circumstances `if` introduces static boolean logic in your code, reducing code flexibility. In other cases, a `case` or a function call with pattern matching in its clauses is just more declarative. For newcommers (that have learned to use `if` in other languages), Erlang's `if` can be either hard to understand or easily abused.

Debate:

- [From OOP world](http://antiifcampaign.com/)
- [In this repo](issues/14)
- [In erlang-questions](http://erlang.org/pipermail/erlang-questions/2014-September/080827.html)

***
Avoid nested try...catches
> Don't nest `try…catch` clauses

Examples: [nested_try_catch](src/nested_try_catch.erl)

Reasoning: Nesting `try…catch` blocks defeats the whole purpose of them, which is to isolate the code that deals with error scenarios from the nice and shiny code that deals with the expected execution path.

Naming

***
Be consistent when naming concepts
> Use the same variable name for the same concept everywhere (even in different modules).

Examples: [consistency](src/consistency.erl)

Reasoning: When trying to figure out all the places where an ``OrgID`` is needed (e.g. if we want to change it from ``string`` to ``binary``), it's way easier if we can just grep for ``OrgID`` instead of having to check all possible names.

***
Explicit state should be explicitly named
> Name your state records ``#mod_state`` and use ``-type state():: #mod_state{}`` in all your modules that implement OTP behaviors.

Examples: [state](src/state)

Reasoning: OTP behaviours implementations usually require a state, and if it has a recognizable name it makes it more easily identifiable. Defining a type for it, helps _dialyzer_ detect leaks (where an internal type as the state is used outside of the module).
The usage of the module prefix in the record name has the goal of distinguishing different state tuples while debugging: Since records are just tuples when one is dumped into the shell it is easier to read `{good_state, att1, attr2}` than `{state, attr1, attr2, attr3}` or `{state, attr1, att2}`. At a glance you know that the tuple/record can be found in the `good.erl`module.

***

Don't use _Ignored variables
> Variables beginning with _ are still variables, and are matched and bound, the _ just keeps the compiler from warning when you don't use them. If you add the _ to a variable's name, don't use it.

Examples: [ignored_vars](src/ignored_vars.erl)

Reasoning: They are **not** supposed to be used.

***
Avoid boolean parameters
> Don't use boolean parameters (i.e. `true` and `false`) to control clause selection.

Examples: [boolean_params](src/boolean_params.erl)

Reasoning: Clarity of intention and not requiring the reader to check the function definition to understand what it does.

***
Stick to one convention for naming modules
> Stick to one convention when naming modules (i.e: ik_something vs iksomething vs something).

Examples: [naming_modules](src/naming_modules)

Reasoning: It gives coherence to your system.

***
Lowercase atoms
> Atoms should use only lowercase characters. Words in atom names should be separated with `_`. Special cases are allowed (like `'GET'`, `'POST'`, etc.) but should be properly justified.

Examples: [atoms](src/atoms.erl)

Reasoning: Adhering to one convention makes it easier not to have "duplicated" atoms all around the code. Also, not using caps or special characters reduces the need for `'` around atoms.

***
Function Names
> Function names must use only lowercase characters or digits. Words in function names must be separated with `_`.

Examples: [function_names](src/function_names.erl)

Reasoning: Function names are atoms, they should follow the same rules that apply to them.

***
Variable Names
> CamelCase must be used for variables. Don’t separate words in variables with `_`.

Examples: [variable_names](src/variable_names.erl)

Reasoning: Adhering to one convention makes it easier not to have "duplicated" variables all around the code. Camel-case makes variable names more visually distinguishable from atoms and it matches the OTP standard.

Strings

***
IOLists over string concatenation
> Use iolists instead of string concatenation whenever possible

Examples: [iolists](src/iolists.erl)

Reasoning: Performance and errors during conversion.
[iolists](http://www.erlangpatterns.org/iolist.html) are just deeply nested lists of integers and binaries to represent IO data to avoid copying when concatenating strings or binaries.

Macros

***
No Macros
> Don't use macros, except for very specific cases, that include
> * Predefined ones: ``?MODULE``, ``?MODULE_STRING`` and ``?LINE``
> * Literal constants: ``?DEFAULT_TIMEOUT``

Examples: [macros](src/macros.erl)

Reasoning: Macros make code harder to debug. If you're trying to use them to avoid repeating the same block of code over and over, you can use functions for that.
See [related blog post](https://medium.com/@erszcz/when-not-to-use-macros-in-erlang-1d3f10d377f#.xc9b4bsl9) by [@erszcz](https://github.com/erszcz).

***
Uppercase macros
> Macros should be named in ALL_UPPER_CASE:

Examples: [macro_names](src/macro_names.erl)

Reasoning: It makes it easier not to duplicate macro names, to find them using grep, etc.

***
No module or function name macros
> Don't use macros for module or function names

Examples: [macro_mod_names](src/macro_mod_names.erl)

Reasoning: Copying lines of code to the console for debugging (something that happens *a lot*) becomes a really hard task if we need to manually replace all the macros.

Records

***
Record names
> Record names must use only lowercase characters. Words in record names must be separated with `_`. Same rule applies to record field names

Examples: [record_names](src/record_names.erl)

Reasoning: Record and field names are atoms, they should follow the same rules that apply to them.

***
Records go first
> Records that are used within a module should be defined before any function bodies.

Examples: [record_placement](src/record_placement.erl)

Reasoning: Records are used to define data types that will most likely be used by multiple functions on the module, so their definition can not be tied to just one. Also, since records will be associated to types, it's a good practice to place them in code in a similar way as the documentation does (and edoc puts types at the beginning of each module documentation)

***
Don't share your records
> Records should not be shared among multiple modules. If you need to share _objects_ that are represented as records, use opaque exported types and provide adequate accesor functions in your module.

Examples: [record_sharing](src/record_sharing.erl)

Reasoning: Records are used for data structure definitions. Hiding those structures aids encapsulation and abstraction. If a record structure needs to be changed and its definition is written in a .hrl file, the developer should find all the files where that .hrl and verify that his change hasn't broken anything. That's not needed if the record structure is internal to the module that manages it.

***
Avoid records in specs
> Avoid using records in your specs, use types.

Examples: [record_spec](src/record_spec.erl)

Reasoning: Types can be exported, which aids documentation and, using ``opaque`` types it also helps with encapsulation and abstraction.

***
Types in records
> Always add type definitions to your record fields

Examples: [record_types](src/record_types.erl)

Reasoning: Records define data structures, and one of the most important parts of that definition is the type of the constituent pieces.

Misc

***
Write function specs
> Write the **-spec**'s for your exported fun's, and for unexported fun's when it adds real value for documentation purposes. Define as many types as needed.

Examples: [specs](src/specs.erl)

Reasoning: Dialyzer output is complicated as is, and it is improved with good type names. In general, having semantically loaded type names for arguments makes reasoning about possible type failures easier, as well as the function's purpose.

***
Use -callback attributes over behaviour_info/1.
> Unless you know your project will be compiled with R14 or lower, use ``-callback`` instead of ``behavior_info/1`` for your behavior definitions.

Examples: [callbacks](src/callbacks)

Reasoning: Avoid deprecated functionality

***
Use atoms or tagged tuples for messages
> When sending a message between processes, you should typically either send a single, human-readable atom, or a tuple with a human-readable atom placed in element 1. This includes messages being sent via ``gen_server:call`` and the like.

Examples: [message-formatting](src/message_formatting.erl)

Reasoning: Tagging messages with a distinctive, human-readable atom helps clarify the purpose of a message for anyone reading or debugging the code. Using element 1 of the tuple makes code more consistent and predictable, and improves readability when browsing through multiple clauses of functions like ``handle_call``.

This pattern also helps avoid bugs where different messages get confused with one another, or where messages get sent to the wrong recipient; it's much easier to find the source of an unexpected message if it looks like ``{set_foobar_worker_pid, <0.312.0>}`` than if you just find a bare pid in your mailbox.

***
No nested header inclusion
> When having many _nested_ "include files", use -ifndef(HEADER_FILE_HRL) .... -endif so they can be included in any order without conflicts.

Examples: [nested](include/nested.hrl)

Reasoning: ``-include`` directives in included headers may lead to duplication of inclusions and/or other conflicts and it also hides things from the developer view.

***
No types in include files
> No `-type` in hrl files

Examples: [types](src/types.erl)

Reasoning: Defining types in public header files (especially those intended for inclusion via `-include_lib()`) might lead to type name clashes between projects and even modules of a single big project.
Instead, types should be defined in modules which they correspond to (with `-export_type()`) and this way take advantage of the namespacing offered by module names.
In other words, "no type definitions in header files" rule means that we will always need to use `some_mod:some_type()` unless referring to a type from the same module it's defined in.

Following this rule you also get the benefits that `-opaque` types provide, for instance, to dialyzer.

***
Don't import
> Do not use the `-import` directive

Examples: [import](src/import.erl)

Reasoning: Importing functions from other modules makes the code harder to read and debug since you cannot directly distinguish local from external functions. In appropriately named functions, the module is _part_ of the function name, it gives meaning to it.

***
Don't export_all
> Do not use the `-compile(export_all)` directive

Examples: [export_all](src/export_all.erl)

Reasoning: It's generally considered best to only export the specific functions that make up your module's known and documented external API. Keeping this list of functions small and consistent encourages good encapsulation and allows for more aggressive refactoring and internal improvements without altering the experience for those who make use of your module.

***
Encapsulate OTP server APIs
> Never do raw ``gen_server`` calls across module boundaries; the call should be encapsulated in an API function in the same module that implements the corresponding ``handle_call`` function. The same goes for other such OTP constructs (``gen_server`` casts, ``gen_fsm`` events, etc).

Examples: [otp_encapsulation](src/otp_encapsulation.erl)

Reasoning: By sticking to this pattern of encapsulation, we make it _much_ easier to find out where calls/events might originate from.
Instead of having to search through the entire source tree for e.g. ``gen_server`` calls that look like they might send a certain message to a given process, we can just search for calls to the corresponding API function.
This makes it much easier to modify APIs, and also allows us to benefit more from Dialyzer's checks, assuming our API functions have appropriate type specs on them.
We can also change the underlying message format without disturbing any code outside of the module in question, and we can more easily avoid issues with RPC calls when running a mixed cluster.

With good encapsulation, you can even do things like convert a ``gen_server`` to a ``gen_fsm`` without any code changes beyond just the one module.

***
No debug calls
> Unless your project is meant to be run as an escript, there should be no `io:format` nor `ct:pal` calls in your production code (i.e. in the modules inside the `src` folder). Same rule applies for `lager` or `error_logger` calls if they're used just for debugging purposes during test stages.

Examples: [debug_calls](src/debug_calls.erl)

Reasoning: Leaving unnecessary logs on production code impacts performance. It increases the processing time for the functions you're debugging and also consumes disk space if the logs are written to a file (as they usually are). Besides, more often than not the log messages are only understood in the context of the test or debugging round in which they were created, therefore the become useless pretty fast.

***
Don't Use Case Catch
> Don't capture errors with `case catch`, use `try ... of ... catch` instead.

Examples: [case-catch](src/case_catch.erl)

Reasoning: `case catch ...` mixes good results with errors which is confusing. By using `try ... of ... catch` the golden path is kept separate from the error handling.

Tools

***
Lock your dependencies
> In your rebar.config or Erlang.mk, specify a tag or commit, but not master.

Examples:
- [erlang.mk](priv/Makefile)
- [rebar.config](priv/rebar.config)

Reasoning: You don't want to be suddenly affected by a change in one of your dependencies. Once you've found the right version for you, stick to it until you *need* to change.

***
Loud errors

> Don't let errors and exceptions go unlogged. Even when you handle them, write a log line with the stack trace.

Examples: [loud_errors](src/loud_errors.erl)

Reasoning: The idea is that somebody watching the logs has enough info to understand what's happening.

***
Properly use logging levels
> When using lager, use the different logging levels with the following meanings:

Meanings:
  * ``debug``: Very low-level info, that may cover your screen and don't let you type in it :P
  * ``info``: The system's life, in some detail. Things that happen usually, but not all the time. You should be able to use the console with acceptable interruptions in this level.
  * ``notice``: Meaningful things that are worth noticing, like the startup or termination of supervisors or important gen_servers, etc…
  * ``warning``: Handled errors, the system keeps working as usual, but something out of the ordinary happened
  * ``error``: Something bad and unexpected happen, usually an exception or error (**DO** log the **stack trace** here)
  * ``critical``: The system (or a part of it) crashed and somebody should be informed and take action about it
  * ``alert``: _There is no rule on when to use this level_
  * ``emergency``: _There is no rule on when to use this level_

***
Prefer the https protocol over others when specifying dependency URLs
> When specifying dependencies in erlang.mk Makefiles or rebar.config, prefer using the https protocol to download the dependency repository.

Examples:
 * [makefile example](src/dependency_protocol/dep_protocol.makefile)
 * [rebar example](src/dependency_protocol/dep_protocol.config)

Reasoning: HTTPS is recommended by GitHub and is easier for CI.

* [Git on the Server - The Protocols](http://git-scm.com/book/ch4-1.html)
* [GitHub Official Recommendation](https://help.github.com/articles/which-remote-url-should-i-use/)
* [GitHub Protocol Comparison](https://gist.github.com/grawity/4392747#file-github-protocol-comparison-md)

***
No implicit functions with mixer
> Don't implicitly include all functions from a module when using the [mixer](https://github.com/chef/mixer) library. Explicitly list all mixed-in functions.

Examples: [mixer](src/mixer.erl)

Reasoning: Knowing all the functions that are included in a module makes it easier to reason about it. If any number of functions are implicitly brought from another module, it introduces an extra level of unnecessary indirection that requires jumping back and forth between files. The less information we have to keep in our heads the better.

Suggestions & Great Ideas

Things that should be considered when writing code, but do not cause a PR rejection, or are too vague to consistently enforce.

***
Prefer pattern-matching over testing for equality
> When you want to write a conditional statement based on a comparison of two values, don't use equality and then switch according to the boolean result value. Use pattern matching instead.

Examples: [pattern matching](src/prefer_pm.erl)

Reasoning:
From a semantic standpoint, _boolean switches_ after _equality_ introduce static boolean logic in your code, reducing its flexibility. Besides, pattern matching is just more declarative. And, specially in the case where there is a function involved, using pattern matching you get a chance to _do something_ with the result of such a function call.

***
Favor higher-order functions over manual use of recursion
> Occasionally recursion is the best way to implement a function, but often a fold or a list comprehension will yield safer, more comprehensible code.

Examples: [alternatives to recursion](src/recursion.erl)

Reasoning: Manually writing a recursive function is error-prone, and mistakes can be costly. In the wrong circumstances, a buggy recursive function can miss its base case, spiral out of control, and take down an entire node. This tends to counteract one of the main benefits of Erlang, where an error in a single process does not normally cause the entire node to crash.

Additionally, to an experienced Erlang developer, folds and list comprehensions are much easier to understand than complex recursive functions. Such contstructs behave predictably: they always perform an action for each element in a list. A recursive function may work similarly, but it often requires careful scrutiny to verify what path the control flow will actually take through the code in practice.

***
CamelCase over Under_Score
> Symbol naming: Use variables in CamelCase and atoms, function and module names with underscores.

Examples: [camel_case](src/camel_case.erl)

Reasoning: It helps a lot with the next issue in this list ;)

***
Prefer shorter (but still meaningful) variable names
> As long as it's easy to read and understand, keep variable names short

Examples: [var_names](src/var_names.erl)

Reasoning: It helps reducing line lengths, which is also described above

***
Comment levels
> Module comments go with **%%%**, function comments with **%%**, and code comments with **%**.

Examples: [comment_levels](src/comment_levels.erl)

Reasoning: It clearly states what the comment is about, also helpful to search for specific comments, like "%% @".

***
Keep functions small
> Try to write functions with a small number of expressions, and that do only one thing. **12** expressions per function except for integration tests is a good measure.

Examples: [small_funs](src/small_funs.erl)

Reasoning: From 3 different sources:
- Small functions aid readability and composeability. Readability aids maintainability. This cannot be stressed enough. The smaller your code, the easier it is to fix and change.

- A small function allows one to see its purpose clearly, so that you need to only understand the small subset of operations it performs, which makes it very simple to verify it works correctly.
- These are all compeling reasons:
  + a function should do one thing, if it's too large you are likely to be doing work better suited for multiple functions
  + clarity, it's easier to see what a function does when it's short and concise
  + reuse, keeping them short means you can use them later for something else (specially true for Erlang)
  + screen size: you want to be able to see the whole function if you want to connect via ssh to a server for whatever reason

Notes:

This guideline, together with **[Avoid deep nesting](#avoid-deep-nesting)** and **[More, smaller functions over case expressions](#more-smaller-functions-over-case-expressions)**, can be well followed by structuring your functions as follows:

```erlang
some_fun(#state{name=foo} = State) ->
  do_foo_thing(),
  continue_some_fun(State);
some_fun(#state{name=bar} = State) ->
  do_bar_thing(),
  continue_some_fun(State).

continue_some_fun(State) ->
  ...,
  ok.

```

Remember:

- There is no cost for a tail call like that.
- This pattern is efficient, compact, clear.
- It "resets" indentation so the code doesn't wander off the right edge of the screen.

Most importantly:

- It's easier to test because the functions delineate the testing hinge points.
- It gives more surface for tracing, so one can get very specific about where the computation goes off the rails. Nested cases are opaque at runtime.

***

Use behaviours.
> Encapsulate reusable code in behaviors.

Examples: [behavior](src/behavior.erl)

Reasoning: It's the OTP way ;)

***
When programming defensively, do so on client side
Do validations on the outmost layers of your code.

Examples: [validations](src/validations.erl)

Reasoning: One aspect of choosing where want you to crash is how you design your API: A function that checks the input before calling the gen_server behind it will avoid a full roundtrip to the gen_server and maybe even a gen_server crash.
do_it(Pid, X) when is_integer(X) -> gen_server:call(Pid, {do_it, X}).
If you design this way, the caller crashes if the arg is wrong.
If you don't tighten up the function head, the gen_server will crash.

***
Avoid unnecesary calls to length/1
> Lots of use cases of length/1 can be replaced by pattern matching, this is specially true when checking if the list has at least one element.

Examples: [pattern matching](src/pattern_matching.erl)

Reasoning: Pattern matching is one of the core aspects of Erlang and as such it's both performant and readable. Pattern matching is also more flexible so changes to the logic get simpler.

***
Move stuff to independent applications
> When you identify a block of functionality that is self-contained (it may be several modules or just a big one) and actually independent of the main purpose of your application, place that in a separate application. And consider open-sourcing it.

Reasoning: It's easier to share among apps. If open-sourced, you're sharing it with the community and you get the benefits of the community being involved in it.

Note: Do **not** create highly specific libraries that are too coupled with the project you're working on. Use this rule for libraries that will likely be reused in other projects.

***

Use the facade pattern on libraries
> [The facade pattern](http://en.wikipedia.org/wiki/Facade_pattern) is great to simplify library usage and serves as a form of self-documentation.

Examples: [kafkerl](https://github.com/inaka/kafkerl/blob/master/src/kafkerl.erl)

Reasoning: Having the relevant functions in a single module means that the end user doesn't have a hard time figuring out which functions to call. Note that to avoid making it too complex, you probably want to carefully consider which functionality you wish to support here; exposing fewer functions (the ones that show the basic use of the library) as opposed to just creating a dummy module containing every single exported function in the library is prefered.
This greatly reduces the learning curve of the library and therefore makes it more tempting to use.

***
Types in exported functions
> Custom data types used in exported functions should be defined with Erlang type declarations and exported from the module

Examples: [data_types](src/data_types.erl)

Reasoning: It helps with function documentation and, when using opaque types, we ensure encapsulation.

***
Separate responsibilities in sumo_db
> When using sumo_db you should separate the responsibilities clearly, creating for each entity:
> - one module (usually called MODELs) to describe the entity and allow administrating instances of the model in memory
> - one module (usually called MODEL_repo) to handle the various operations that require business logic relating to the entity

Examples: [separate responsibilities in sumo_db](https://github.com/inaka/fiar/tree/master/src/models)

Reasoning: By dividing the functions into two different modules we increase understandability of the functionality especially if these are called from external modules. It also allows us to better organize the code and have smaller modules.