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:

```
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:

```
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:

```
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.

```
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:

```
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:

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:

```
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:

```
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:

```
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:

```
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.

```
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:

```
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:

```
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:

```
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:

```
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:

```
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:

```
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:

```
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:

```
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:

```
  if (condition) statements;
```

3. **[Recommended]** Use else as less as possible, if-else statements could be replaced by:

```
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*:

```
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:

```
// please refer to the pseudo-code as follows
boolean existed = (file.open(fileName, "w") != null) && (...) || (...);
if (existed) {
    ...
}
```

Counter example:

```
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:

```
// 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.

```
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:

```
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.

```
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:

```
if (logger.isDebugEnabled()) {
   logger.debug("Processing trade with id: " + id + " symbol: " + symbol);
}
```

> Positive example:

```
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:

```
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*

    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: 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*.

```
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.

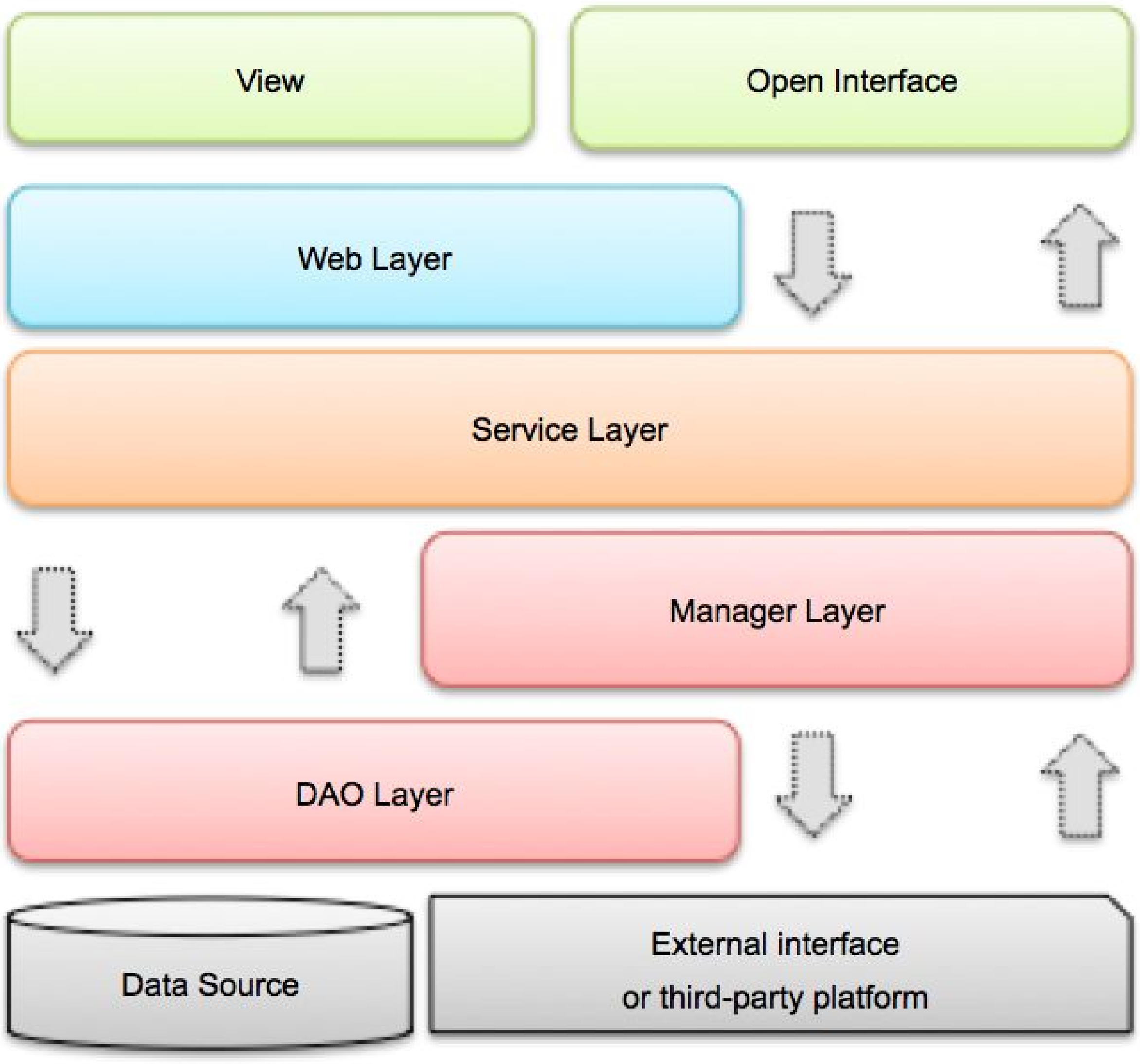
9. **[For Reference]** Do not overuse @*Transactional*. Because transaction affects QPS of DB, and relevant rollbacks may need be considered, including cache rollback, search engine rollback, message making up, statistics adjustment, etc.

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.