APP团队java编程规范

零、说明

* 命名尽量做到见名知意，采用驼峰标识，禁止使用拼音或者表意不明确的标识（比如bianliang、i、j）;
* 命名无法完全表达作者的意思时，才考虑使用注释；注释的目的是帮助读者理解代码，提升代码的可读性；
* 注释应该表达“我的代码为什么要这么做？”，而不是表达“我的代码做了什么？”；
* 尽量降低方法的圈复杂度，遇到不满足条件时直接返回；
* 一个变量只有一个意思；一个方法只做一件事情；业务逻辑的具体实现封装在model层；要保证代码尽量独立，方便单元测试、重构、理解和维护；
* 糟糕的注释示例：

/\* 一个整数 \*/  
int i;  
  
  
/\* i从0循环到8 \*/  
**for** (i=0; i<=8; i++)  
  
  
/\* 输出结果 \*/  
**for** (i=0; i<=9; i++) printf("%d, ", a[i]);

一、排版规范

* 程序块要采用缩进风格编写，缩进为4个空格；
* 较长的语句、表达式或参数（>80字符）要分成多行书写，控制每行在80个字符以内；

\*\*说明：\*\*长表达式要在低优先级操作符处划分新行，操作符放在新行之首，划分出的新行要进行适当的缩进，使排版整齐，语句可读。

* 不允许把多个短语句写在一行中，即一行只写一条语句；
* if、for、do、while、case、switch、default等语句自占一行，且执行语句无论多少都要加括号{}；
* 相对独立的程序块之间、变量说明之后必须加空行；

空行将逻辑相关的代码段分隔开,以提高可读性。下列情况应该使用空行:

1. 一个源文件的两个片段(section)之间；

2. 类声明和接口声明之间；

3. 两个方法之间；

4. 方法内的局部变量和方法的第一条语句之间；

5. 一个方法内的两个逻辑段之间，用以提高可读性；

6. 通常在变量声明区域之后要用空行分隔，常量声明区域之后要有空行分隔，方法声明之前要有空行分隔。

* 构造函数有多个时，要依据参数少的在前面，参数多的在后面的原则编写代码，提高代码可读性；
* 代码排版按照members、interface、inner class、system callback method、public method、protected method、private method、handler、AsynTask、TimerTask的顺序;
* 不允许将业务代码写在系统回调方法中，必须抽离成方法供系统回调调用，eg:

@Override  
protected void onCreate(Bundle savedInstanceState) {  
    super.onCreate(savedInstanceState);  
    setContentView(R.layout.activity\_dictation\_layout);  
  
    // 将初始化代码封装成方法供系统回调调用，不要将实例化代码直接放在系统回调方法里面  
    initDictation();  
}  
  
private void initDictation(){  
    findViews();  
    getData();  
    showContent();  
}

* 为了提高代码的可读性，保证每个方法功能单一，单个方法代码量控制在50行以内，最多不能超过200行，并且应尽量减少方法之间的耦合；
* 左大括号放置在行尾，右大括号单独占一行；
* 静态类的构造方法需要私有化，eg:

public class ArrayUtils {  
    private ArrayUtils(){  
    }  
  
    public static <V> boolean isEmpty(V[] sourceArray) {  
        return (sourceArray == null || sourceArray.length == 0);  
    }  
}

二、注释规范

* JAVA程序有两类注释：

1. 一类是实现性注释（implementation comments）

2. 另一类是文档型注释（documentation comments）。

其中，实现性注释有两种格式，/\*…  */ 和//。文档型注释是JAVA程序所独有的，在/*\*…*/中限定。文档型注释能够被javadoc文件工具提取到HTML文件中。*

\*\*注意：\*\*

1. 过于频繁的注释常常意味着代码质量较低。当你觉得必须频繁添加注释时，应考虑重写该段代码了。

2. 注释内容不应封在星号（\*）或者其它字母围成的矩形框中；注释中不应带有特殊注释字符如制表（form-feed）和退格符（backspace）。

\*\*文档注释基本原则：\*\*

1. 注释应能使代码更加明确；

2. 避免注释部分的过度装饰；

3. 保持注释部分简单、明确；

4. 在编码以前就应开始写注释；

5. 注释应说明设计思路而不是描述代码的行为。

* 类注释

每一个类都必须要包含如下格式的注释，以说明当前类的功能等。

    /\*\*  
    \* @author 作者  
    \* 实现的主要功能。  
    \* 创建日期  
    \* 修改者,修改日期,修改内容。  
    \*/

* 方法注释

关键方法或者需要特别说明的方法都需要添加如下格式的注释，包括当前方法的用途、当前方法参数的含义、当前方法返回值的内容和抛出异常的列表。

    /\*\*  
    \* 方法的一句话概述  
    \* **<p>**方法详述(简单方法可不必详述)**</p>**  
    \* @param s 说明参数含义  
    \* @return 说明返回值含义  
    \* @throws IOException 说明发生此异常的条件  
    \*/

* 类成员变量和常量注释

关键成员变量或者需要说明的成员变量和常量需要使用 java doc 形式的注释，以说明当前变量或常量的含义

    /\*\*  
    \* XXXX含义  
    \*/

* 其他注释；

方法内部的注释如果需要多行，使用/\*??   */形式，如果为单行是用//??形式的注释。不要在方法内部使用 java doc 形式的注释“/*\*??\*/”。

* XML 注释；

如果当前 layout 或资源需要被多处调用，或为公共使用的 layout , 则需要在 xml 写明注释。要求注释清晰易懂。

* 注释与所描述内容进行同样的缩排；
* 与doc相关的标签如下表

1. @author：作者的名称；类、接口；说明特定某一段程序代码的作者。每一个作者各有一个标记。

2. @deprecated：类、方法；说明该类的应用程序编程接口 (API) 已被废弃，因此应不再使用。

3. @exception：name description；说明由方法发出的异常。一个异常采用一个标记，并要给出异常的完整类名。

4. @param：name 参数名的描述；用来说明传递给一个方法的参数，其中包括参数的类型/类和用法。每个参数各有一个标记。

5. @return：方法返回值的描述；若方法有返回值，对该返回值进行说明。应说明返回值的类型/类和可能的用途。

例如：

   @see 类名    类、接口、方法、字段 在文档中生成指向特定类的超文本链接。可以并且应该采用完全合法的类名。  
    @see ClassName#member functionName 类、接口、方法、字段  在文档中生成指向特定方法的超文本链接。可以并且应该采用完全合法的类名。  
    @version 版本号   类、接口  说明特定一段代码的版本信息。

三、命名规范

* 命名的基本原则：

1. 命名的格式：[作用区间]+释义+类型，比如设置界面的布局文件：activity\_setting\_layout，设置按钮的ID：settingBtnId，成员变量设置按钮：mSettingBtn等。

2. 使用完整的英文描述符准确描述变量variable/域field/类class；

3. 好的命名能够减少不必要的注释和理解代码的难度，应尽量做到见名知意，比如settingBtn、feedBackLayout、getUpdateInfo、getCurrentTime、setUserInfo, 而诸如x1、y1、my\*、main\*等命名不能反映任何命名含义而且造成代码难以理解、维护和改进。

4. 采用应用领域相关的术语来命名，软件开发人员应注意软件用户的一些约定术语，不应当随意的创造术语。这会降低软件的易维护性。

5. 采用大小写混合的方式提高命名的可读性一般情况下应该用小写字母来命名，其中类（class）和接口（interface）名称的首字母用大写。

6. 命名中应该慎用缩写命名。如要采用，则应采用统一的缩略规则，并且在文中的相应部分统一采用缩写。例如，采用num作为number的缩写，那么在整个文档中应该始终使用该缩写。

7. 避免太长的命名，命名的长度一般小于30个字符，最好不要超过15个字符。避免采用仅有大小写不同命名。

8. 命名时应避免采用几乎相同的名称。例如，变量名称persistentObject和persistentObjects不应当同时运用；anSqlDatabase和anSQLDatabase也不应同时使用。

9. 有时名称中会含有固定的缩略词，例如SQL代表Standard Query Language. 而在命名时sqlDatabase和SqlDatabase就比sQLDatabase和SQLDatabase易于阅读。

10. 避免以下划线开头的命名。

* 包命名

\*\*命名规则：\*\*字母全部采用小写，以com.eebbk开始，后面跟有模块名称，再后面为层级名称。如:com.eebbk.模块名.层级名：

    com.eebbk.musicplayer.activity

对于功能较多的大模块，可增加功能名一级。如：com.eebbk.模块名.功能名.层级名：

    com.eebbk.english.pointread.view

* 类和接口的命名

\*\*命名规则：\*\*采用大小写混合的方式，每个单词的首字母大写。尽量使你的类名简洁而富于描述。使用完整单词，避免缩写词(除非该缩写词被更广泛使用，像URL，HTML)，由于类是设计用来代表对象的，所以在命名类时应尽量选择名词。如:

**class** PhotoAdapter;  
  
    **class** MusicPlayer;  
  
    **class** VideoInfo;  
  
    **class** ImageSprite;

接口使用I开头，抽象类使用A开头。如:

**class** IVideo;  
  
    **class** ASprite;

* 方法的命名

\*\*命名规则：\*\*方法名是一个动词，采用大小写混合的方式，第一个单词的首字母小写，其后单词的首字母大写（小写字母开头，任意字母、数字组成）。例如:

    public void run();   
  
    public String getBookName();

类中常用方法的命名:

1. 类的获取方法(一般具有返回值)一般要求在被访问的字段名前加上get，如getFirstName()，getLastName()。一般来说，get前缀方法返回的是单个值，find 前缀的方法返回的是列表值；

2. 类的设置方法(一般返回类型为 void)：被访问字段名的前面加上前缀set，如setFirstName()，setLastName()；

3. 类的布尔型的判断方法一般要求方法名使用单词is或has做前缀，如isPersistent()，isString()。或者使用具有逻辑意义的单词，例如equal或 equals。

4. 类的普通方法一般采用完整的英文描述说明成员方法功能，第一个单词尽可能采用动词，首字母小写，如openFile()，addCount()。

5. 构造方法应该用递增的方式写(参数多的写在后面)。

* 非静态成员命名

\*\*命名规则：\*\*使用驼峰规则, m与任意一大写字母开头，由字母、数字组成，如mUserName,mMediaPlayer；

* 常量命名

\*\*命名规则：\*\*类常量的声明，应该全部大写，单词间用下划线隔开（以大写字母开头，由大写字母、数字和下划线组成），例如：

    static final int MIN\_WIDTH = 4;  
  
    static final int MAX\_WIDTH = 999;

* 局部final变量与catch参数:

\*\*命名规则：\*\*小写字母开头，由字母、数字组成，eg：

    final int age；

* 局部非final变量与catch参数:

\*\*命名规则：\*\*小写字母开头，由字母、数字组成，eg:

    int userCount；

* 异常命名

自定义异常的命名必须以Exception为结尾，以明确标示为一个异常。

* layout 命名

layout xml的命名必须以全部单词小写，单词间以下划线分割，并且使用名词或名词词组，即使用 “[作用区间]\_释义\_类型”来命名。

1. layout命名：activity\_释义\_layout.xml，例如：

    activity\_setting\_layout.xml；

2. Dialog命名：dialog\_释义\_layout.xml，例如：

    dialog\_app\_update\_layout.xml

3. Popupwindow命名：popupwindow\_释义\_layout，例如：

    popupwindow\_grade\_list\_layout.xml

4. Adapter中项的布局命名：listview/gridview\_item\_释义\_layout，例如：

    listview\_item\_grade\_layout

5. 包含项的命名：include\_释义\_layout，例如：

    include\_top\_bar\_layout.xml

* id 命名

layout中所使用的id要求能够通过id直接理解当前组件要实现的功能，命名规则为：功能+类型+Id；如:显示课文名称的TextView，应命名为：

    @+id/lessonNameTxtId;

如：输入用户名的EditText，应命名为：

    @+id/userNameEditTxtId;

* 资源命名

layout中所使用的所有资源(如 drawable,style,diem 等)要求能够通过 id 直接理解当前组件要实现的功能，命名规则为：功能+类型+[normal/press]+[索引值]，如： 设置界面的背景图，应命名为：

    setting\_bg;

设置按钮正反选图，应分别命名为：

    setting\_btn\_normal;  
    setting\_btn\_press;

性别勾选checkbox正反选，应命名为：

    sex\_checkbox\_normal;  
    sex\_checkbox\_press;

物体移动AnimationDrawable动画，应命名为：

    dog\_move\_0  
    dog\_move\_1  
    dog\_move\_2

* 常用的反义词组

add / remove；

begin / end；

create / destroy；

insert / delete；

first / last；

get / set；

increment / decrement；

put / get；

add / delete；

lock / unlock；

open / close；

min / max；

old / new ；

start / stop；

next / previous；

source / target；

show / hide；

send / receive；

source / destination ；

cut / paste；

up / down

* 常用的缩写

Button，简写：Btn(btn)；

RadioButton，简写：Rbtn(rbtn)；

ImageButton，简写：Ibtn(ibtn)；

TextView，简写：Tv(tv)；

ImageView，简写：Iv (iv)；

ListView，简写：Lv(lv)；

ProgressBar，简写：Pbar(pbar)；

EditText，简写：Edtv(et)；

ScrollView，简写：Sclv(scly)；

CheckBox，简写：Chk(chk)；

RelativeLayout，简写：Rlyt(rlyt)；

LinearLayout，简写：Llyt(llyt)；

TableLayout，简写：Tlyt(tlyt)；

AbsoluteLayout，简写：ALyt(alyt)；

FrameLayout，简写：Flyt(flyt)；

RadioGroup，简写：Rgp(rgp)；

SharedPreferences，记那些：Sp(sp)。

* 回调函数以必须以on开头；
* 标识符应当使用完整的英文描述，标识符的命名应当符合“min-length && max-information”原则，谨慎使用缩写；

\*\*说明：\*\*

1. 对于标识符应当使用完整的英文进行描述，对于整个描述较长的，可对单词进行缩略;

2. 对单词的缩略可按较短的单词可通过去掉“元音”形成缩写，较长的单词可取单词的头几个字母形成缩写，一些单词有大家公认的缩写，常用单词的缩写必须统一。

3. 协议中的单词的缩写与协议保持一致。对于某个系统使用的专用缩写应该在某处做统一说明。设计命名中应该慎用缩写命名。如要采用，则应采用统一的缩略规则；

* 注意静态常量的声明次序(方括号中的内容为可选)：[作用域]+[static]+[final]+类型+属性名称 = 初始值；比如：

public static final int NAME = 5;

四、JAVA代码标准规则

* 禁止出现多余的修饰符；
* 文件长度控制在 2000 行以内；
* 方法职责要单一，要求方法行数控制在200行以内 ；推荐一个方法尽量不要超过50行,如果方法太长,说明当前方法业务逻辑已经非常复杂,那么就需要进行方法拆分,保证每个方法只作一件事。
* 一个方法的参数尽可能的不要超过5个；
* 控制语句

1. 判断中如有常量,则应将常量置于判断式的右侧。如：

    if ( true == isAdmin())...;

2. 尽量不使用三目条件的嵌套；

3. 所有 if 语句必须用{}包括起来,即便是只有一句：

    if (true){  
       //do something......  
    }  
  
    if (true)  
    i = 0; //不要使用这种

* 异常的捕捉处理

1. 通常的思想是只对错误采用异常处理：逻辑和编程错误，设置错误，被破坏的数据，资源耗尽等等；

2. 通常的法则是系统在正常状态下以及无重载和硬件失效状态下，不应产生任何异常；

3. 最小化从一个给定的抽象类中导出的异常的个数；

4. 对于经常发生的可预计事件不要采用异常；

5. 不要使用异常实现控制结构；

6. 若有finally子句，则不要在try块中直接返回，亦不要在 finally 中直接返回。

* 数据库操作、IO操作等需要使用结束close()的对象必须在try -catch-finally 的finally中close()。示例：

try{  
  
}catch(IOException ioe){  
  
}finally{  
    try{  
       out.close();  
    }catch (IOException ioe){  
  
    }  
}

* 访问控制，仅对需要对外公布的方法、变量采用public，其余的尽量使访问域最小化。
* 禁止使用立即数，用有意义的标识来替代，涉及物理状态或者含有物理意义的常量，不应直接使用数字，必须用有意义的静态变量来代替。如下的程序可读性差：

if (state == 0){  
    state = 1;  
    ...  // program code  
}

应改为如下形式：

    private final static int TRUNK\_IDLE = 0;  
    private final static int TRUNK\_BUSY = 1;  
    private final static int TRUNK\_UNKNOWN = -1;  
  
    if (state == TRUNK\_IDLE){  
       state = TRUNK\_BUSY;  
       ...  // program code  
    }

* 编码格式，所有工程必须是UTF-8编码；

五、代码优化规则

* 避免在循环表达式中调用函数或方法，提高代码性能，反例：

**class** CEL{  
     void **method** (Vector vector){  
       // 违例  
         **for** (int i = 0; i < vector.size (); i++){  
             System.out.println(i);  
         }  
     }  
}

正例：

**class** CELFixed {  
         void **method** (Vector vector){  
             int size = vector.size ();  
             **for** (int i = 0; i < size; i++) {  
             //正确  
                 System.out.println(i);  
             }  
         }  
     }

* 如有打开的数据流，在finally语句块中关闭数据流，避免潜在的bugs，反例：

public class CS {  
     public void method (java.io.File f){  
         try{  
             java.io.FileInputStream fis = new java.io.FileInputStream (f);  
             fis.read ();  
             fis.close ();  
         }catch (java.io.FileNotFoundException e1){  
             System.out.println("File not found");  
         }catch (java.io.IOException e2){  
             System.out.println("I/O Exception");  
         }  
        // 违例: 如果有异常发生'fis'永远不会关闭  
   }  
 }

正例：

    public class CSFixed{  
         public void method (java.io.File f){  
             try{  
                 java.io.FileInputStream fis = new java.io.FileInputStream (f);  
                 fis.read ();  
             }catch (java.io.FileNotFoundException e1) {  
          System.out.println("File not found");  
          }catch (java.io.IOException e2){  
          System.out.println("I/O Exception");  
          }finally{  
          fis.close ();  
          }  
       }  
    }

* 使用’System.arrayCopy()’替代循环拷贝数组，优化性能，反例：

public class IRB{  
     int[] copyArray (int[] array){  
         int length = array.length;  
         int[] copy = new int [length];  
         for (int i = 0; i < length; i++){  
          //违例  
             copy[i] = array[i];                
       }  
  
         return copy;  
     }  
}

正例：

    public class IRB{  
         int[] copyArray (int[] array){  
             int length = array.length;  
             int[] copy = new int [length];  
          //正确  
             System.arraycopy(array, 0, copy, 0, length);   
             return copy;  
         }  
    }

* 避免在循环语句内部出现”try/catch/finally”，优化代码，反例：

public class TRY{  
     void method (FileInputStream fis){  
         for (int i = 0; i < size; i++){  
             try{  
              //违例                   
                 \_sum += fis.read ();  
             }catch (Exception e){  
  
        }  
         }  
     }  
  
     private int \_sum;  
}

正例：

    public class TRY{  
         void method (FileInputStream fis){  
             try{  
           //正确  
                 for (int i = 0; i < size; i++){  
                     \_sum += fis.read ();  
                 }  
             }catch (Exception e) {  
  
         }  
         }  
    }

* 在循环体中避免实例化对象或声明变量，提高代码运行性能，反例：

**public** **class** **LOOP**{  
     public void method(int max){  
         for (int i = 0; i < max; i++){  
          //违例  
             StringBuffer sb = new StringBuffer();    
             sb.append("loop: ");  
             sb.append(i);  
             System.out.println(sb.toString());  
         }  
     }  
}

正例：

**public** **class** **LOOPFixed**{  
         public void method(int max){  
          //正确  
             StringBuffer sb = new StringBuffer();       
             for (int i = 0; i < max; i++){  
                 sb.append("loop: ");  
                 sb.append(i);  
                 System.out.println(sb.toString());  
                 sb.setLength(0);    
             }  
         }  
    }

* try语句块内部尽量精简，try语句块内部代码执行效率比较低，因此尽量只把可能会产生异常的代码书写在try语句块内部，不会产生异常的代码不要写在try语句块内
* 程序异常退出处理，异常处理，程序异常退出统一用胡明明的捕捉工具类；

六、其它规定

* 项目相关的文档必须放在主工程的doc文件夹下；
* 每次发布版本前，要使用Android Studio→Inspect Code…对代码的静态质量做检测，尽早解决编码规范和编码质量的问题；