2. Code guidelines

2.1 Java language rules

2.1.1 Don't ignore exceptions

절대 해서는 안됩니다!:

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

Exception이 발생하지 않거나, 굳이 처리하지 않아도 괜찮다고 생각할 수 있지만, 이 exception 들을 무시하면 언젠가는 큰 문제가 일어날 수 도 있습니다. 코드에서 모든 exception들을 원칙적으로 처리해야 합니다. 구체적인 처리 방법은 경우에 따라 다를 수 있습니다. - (Android code style guidelines)

자세한 내용은 이 곳을 참고해 주시기 바랍니다.

> checkstyle code

2.1.2 Don't catch generic exception

절대 해서는 안됩니다!:

```
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!
}
```

자세한 내용은 이 곳을 참고해 주시기 바랍니다.

2.1.3 Don't use finalizers

절대 finalizers를 이용해서는 안됩니다!:

자세한 내용은 Android code style guidelines을 참고해 주시기 바랍니다.

> checkstyle code

2.1.4 Fully qualify imports

- Bad: `import foo.*;`
- Good: `import foo.Bar;`

자세한 내용은 이 곳을 참고해 주시기 바랍니다.

> checkstyle code

2.2 Java style rules

2.2.1 Fields definition and naming

필드(Fields)는 **파일의 최상단** 에 정의되어야 하며 아래에 나열된 이름 지정 규칙을 따라야 합니다.

- 소문자로 시작해야 합니다.
- Static final 필드(상수) 들은 대문자로만 구성되어야 합니다.

Example:

```
public class MyClass {
    public static final int SOME_CONSTANT = 42;
    private static MyClass singleton;
    int packagePrivate;
    private int private;
    protected int protected;
    public int publicField;
}
```

> checkstyle code

2.2.2 Method, Class and Interface naming

메서드(Method) 이름은 소문자로 시작해야 합니다.

Example:

```
public class MyClass {

    public int getAge() {
      return 0;
    }
}
```

> checkstyle code

클래스(Class)와 인터페이스(Interface) 이름은 대문자로 시작해야 합니다.

Example:

```
public class MyClass {

}

public interface MyInterface {

}
```

> checkstyle code

수식자(modifiers)의 순서를 지켜야 합니다.

1. public
2. protected
3. private
4. abstract
5. default
6. static
7. final
8. transient

9. volatile
10. synchronized
11. native
12. strictfp

```
public static final int TEMP_VALUE = 0;       // Good
static public final int TEMP_VALUE = 0;       // Bad
final public static int TEMP_VALUE = 0;       // Bad
```

> checkstyle code

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

User **1 space** indent after `comma[,]`, `semi[;]`, `type cast[(String) str]`

```
getData(Context context, String id, int type); // comma[,]
for(int i = 0; i < 10; i++) {                  // semi[;]
View view = (View) findByViewId(R.id.view);    // type cast[(String) str]
```

> checkstyle code

Use **4 space** indents for blocks:

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

Use **8 space** indents for line w raps:

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

> checkstyle code

반대로 공백(space)를 허용하지 않는 부분도 있습니다.

No whitespace before `semi[;]`, `dot[.]`, `post_dec[--]`, `post_inc[++]`.

```
getData(Context context, String id, int type) ; // 허용 안됨 semi[;]
pager .setAdapter(adapter);                     // 허용 안됨 dot[.]
sum --;                                         // 허용 안됨 post_dec[--]
count ++;                                       // 허용 안됨 post_inc[++]
```

> checkstyle code

No whitespace after `dec[--]`, `inc[++]`, `dot[.]`, `lnot[!]`.

```
-- sum;                                    // 허용 안됨 dec[--]
++ count;                                  // 허용 안됨 inc[++]
pager. setAdapter(adapter);                // 허용 안됨 dot[.]
! isFinish;                                // 허용 안됨 lnot[!]
```

> checkstyle code

2.2.5 Use standard line rule

한 줄(line)에 하나의 문장만 있어야 합니다.

```
int var1; int var2;                                          // 허용 안됨
var1 = 1; var2 = 2;                                          // 허용 안됨
int var1 = 1; int var2 = 2;                                  // 허용 안됨
var1++; var2++;                                              // 허용 안됨
Object obj1 = new Object(); Object obj2 = new Object();      // 허용 안됨
import java.io.EOFException; import java.io.BufferedReader; // 허용 안됨
;;                                                           // 허용 안됨

int var1 = 1
; var2 = 2;                                                  // 허용 안됨
int o = 1, p = 2,
r = 5; int t;                                                // 허용 안됨
```

> checkstyle code

`dot[.]`을 기준으로 줄 바꿈을 합니다.

```
builder
  .appModule(new Module())          // Good
  .errorModule(new ErrorModule())
  .build();


builder.
  appModule(new Module()).          // Bad
  errorModule(new ErrorModule()).
  build();
```

`comma[,]`을 기준으로 줄 바꿈을 합니다.

```
public int getValue(int first,    // Good
                    int second,
                    int third) {


public int getValue(int first    // Bad
                    ,int second
                    ,int third) {
```

> checkstyle code

`package, all import declarations, class` 사이에 빈 한줄(Empty line)이 있어야 합니다.

```
package com.puppycrawl.tools.checkstyle.whitespace;
// empty line
import java.io.Serializable;
// empty line
```

```
class Foo {
    public static final int FOO_CONST = 1;
    public void foo() {} //should be separated from previous statement.
}
```

> checkstyle code

2.2.6 Use standard brace style

중괄호`{}`는 앞에 있는 코드와 같은 줄에 있어야 합니다.

```
class MyClass {
    int func() {
        if (something) {
            // ...
        } else if (somethingElse) {
            // ...
        } else {
            // ...
        }
    }
}
```

만약 조건`if-else`와 본문(statements)이 한 줄에 있다면 중괄호가 없어도 괜찮습니다.

```
if (condition) body(); // Good

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

> checkstyle code

2.2.7 Annotations

Classes, Methods and Constructors

클래스, 메서드, 생성자에서는 한 라인별로 annotation을 선언해야 합니다.

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

> checkstyle code

Fields

필드에서는 한 라인에 annotation을 모두 선언해야 합니다.

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

> checkstyle code

2.2.8 Limit variable scope

지역 변수의 범위는 최소한으로 유지되어야합니다 (Effective Java Item 29). 그렇게함으로써 코드의 가독성과 유지 보수성을 높이고 오류의 가능성을 줄일 수 있습니다. 각 변수는 변수의 모든 용도를 둘러싸는 가장 안쪽의 블록에서 선언되어야합니다 .

지역 변수는 처음 사용 된 시점에서 선언해야합니다. 거의 모든 지역 변수 선언에는 초기화(initializer)가 있어야합니다. 변수를 현명하게 초기화하는 데 필요한 정보가 아직 충분하지 않은 경우에는 선언을 연기해야합니다.- (Android code style guidelines)

2.2.9 Logging guidelines

`Log` 클래스가 제공하는 로깅 메서드를 이용하여 오류 메시지나 기타 정보들을 출력합니다.

- `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)

일반적으로 클래스 이름을 TAG로 이용하고 이를 파일의 맨 위에 `static final` 필드로 정의합니다.

```
public class MyClass {
    private static final String TAG = "MyClass";

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

Release 빌드에서는 VERBOSE 및 DEBUG 로그를 **반드시** 비활성화 해야합니다. 또한 INFORMATION, WARNING 및 ERROR 로그도 비활성화하는 것이 좋지만 릴리스 빌드에서 문제를 식별하는 것이 유용 할 수 있다고 생각되는 경우 이를 활성화하는 것이 좋습니다. 활성화한 경우 이메일 주소, 사용자 ID 등과 같은 개인 정보가 유출되지 않도록 주의 해야합니다.

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

2.2.10 Class member ordering

정답이 없는 부분이긴 하지만 **logical** 및 **consistent** 순서를 따르면 가독성이 더 좋아집니다.

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:

```
public class MainActivity extends Activity {

    private String title;
    private TextView textViewTitle;

    public void setTitle(String title) {
        this.title = title;
    }

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

    private void setUpView() {
        ...
    }

    static class AnInnerClass {
```

```
    }

}
```

Activity나 Fragment 같은 **안드로이드 구성 요소** 는 **생명주기(lifecycle)** 를 나열하는게 가독성이 더 좋습니다. 예를 들어, Activity는 아래와 같이 구현할 수 있습니다.

```
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() {}

}
```

Overload methods 들은 함께 그룹화 되어 있어야 합니다.

```
public void foo(int i) {}
public void foo(String s) {}
public void notFoo() {} // Have to be after foo(int i, String s)
public void foo(int i, String s) {}
```

> checkstyle code

2.2.11 Parameter ordering in methods

안드로이드 프로그래밍을 할 때 `Context` 을 취하는 메서드를 정의하는 경우는 빈번하게 발생합니다. 이와 같은 메서드를 작성하는 경우 **Context** 는 **첫 번째** 매개 변수(parameter) 여야 합니다.

반대로 **callback** interface는 **마지막** 매개 변수(parameter) 여야 합니다.

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

// Callbacks always go last
public void loadUserAsync(Context context, int userId, UserCallback callback);
```

2.2.12 String constants, naming, and values

`SharedPreferences` , `Bundle` , `Intent` 와 같은 안드로이드 SDK의 많은 요소들은 키-값 쌍 접근법을 이용하기 때문에 많은 문자열 상수를 작성해야 할 가능성이 높습니다.

이 컴포넌트들 중 하나를 사용할 때 키를 **반드시** `static final` 필드로 정의하십시오. 아래에 표시된 것처럼 접두사를 사용해야합니다.

Element	Field Name Prefix
SharedPreferences	`PREF_`
Bundle	`BUNDLE_`
Fragment Arguments	`ARGUMENT_`

Intent Extra	`EXTRA_`
Intent Action	`ACTION_`

`Fragment.getArguments()`의 인수 또한 Bundle 이지만 이 Bundle은 매우 일반적인 용도이므로 다른 접두사(`BUNDLE_`)를 정의합니다.

Example:

```
// 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";
```

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

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

2.2.13 Line length limit

코드 라인은 **100자** 를 초과하지 않아야합니다. 만약 이 제한보다 길면 길이를 줄이기 위해 일반적으로 두 가지 옵션이 있습니다.

- 로컬 변수 또는 메소드를 추출합니다 (권장).
- 줄 바꿈을 적용하여 한 줄을 여러 줄로 나눕니다.

100자 보다 긴 코드 라인을 가질 수있는 두 가지 **exceptions** 이 있습니다.

- 분할 할 수없는 라인 (ex: 긴 URL)
- `package` 와 `import` 문.

> 안드로이드 스튜디오 `File-Settings-Editor-Code Style-Java-Wrapping and Braces` 메뉴에서 `Ensure right margin is not exceeded`를 checked 하면 자동으로 100자를 체크해서 줄바꿈 해줍니다.
>
> checkstyle code

2.2.13.1 Line-wrapping strategies

줄 바꿈 (line-wrap)하는 방법을 설명하는 정확한 수식이 없지만 일반적인 경우에 적용 할 수있는 몇 가지 규칙이 있습니다.

Break at operators

`+` 이전에 줄 바꿈을 할 수 있습니다.

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

Assignment Operator Exception

`=` 이후에 줄 바꿈을 할 수 있습니다.

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

Method chain case

예를 들어 Builders를 이용할 때 여러 메서드들이 같은 라인에서 연결될 때, `.` 이전에 줄 바꿈을 해야합니다.

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

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

Long parameters case

메소드에 매개 변수가 많거나 매개 변수 이름이 너무 길면 쉼표 `,` 이 후에 줄 바꿈을 해야 합니다.

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

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

2.2.14 RxJava chains styling

Rx chains operators들은 줄 바꿈이 필요합니다. `.` 이전에 줄 바꿈을 해야 합니다.

```
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

XML 요소에 내용이 없으면 **반드시** self closing 태그를 사용하십시오.

This is good:

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

This is **bad** :

```
<!-- 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

리소스 ID 및 이름은 **lowercase_underscore** 로 지정해야 합니다.

2.3.2.1 ID naming

ID 앞에는 소문자로 된 밑줄의 요소 이름이 접두사로 붙어야합니다.

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

Image view example:

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

Menu example:

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

2.3.2.2 Strings

문자열 이름은 자신이 속한 섹션을 식별하는 접두사로 시작합니다. ex) `registration_email_hint` or `registration_name_hint` . 문자열이 어떤 섹션에도 속하지 않으면 다음 규칙을 따라야합니다.

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

나머지 리소스가 없으면 **UpperCamelCase** 로 이름이 작성해야 합니다.

2.3.3 Attributes ordering

일반적으로 유사한 속성을 그룹화해야합니다. 가장 일반적인 속성을 정렬하는 좋은 방법은 다음과 같습니다.

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

테스트 클래스는 테스트 대상 클래스의 이름과 일치해야하며 그 뒤에 `Test` 를 추가합니다. 예를 들어, `DatabaseHelper` 에 대한 테스트를 포함하는 테스트 클래스를 생성한다면, 이름을 `DatabaseHelperTest` 로 지정해야합니다.

테스트 메소드에는 `@Test` annotation이 달려 있으며 일반적으로 테스트 할 메소드의 이름으로 시작해야하며 그 다음에 전제 조건(Precondition) 또는 예상되는 동작(expected behaviour)입니다.

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

사전 조건 또는 예상되는 동작이 항상 필요한 것은 아닙니다.

때때로 클래스에는 많은 양의 메소드가 포함될 수 있으며, 동시에 각 메소드에 대해 여러 테스트가 필요합니다. 테스트 클래스를 여러 클래스로 분할하는 것이 좋습니다. 예를 들어, `DataManager` 에 많은 메소드가 포함되어있는 경우,이 메소드를 `DataManagerSignInTest` , `DataManagerLoadUsersTest` 등으로 나눌 수가 있습니다. 일반적으로 공통된 [test fixture] (https://en.wikipedia.org/wiki/Test_fixture)를 가지고 있기 때문에 어떤 테스트가 속한 것인지 확인할 수 있습니다.

2.4.2 Espresso tests

모든 Espresso 테스트 클래스는 일반적으로 Activity를 대상으로하므로, 이름은 타겟 Activity의 이름과 'Test'가 뒤따라야합니다. `SignInActivityTest`

Espresso API를 사용하는 경우 새 행에 연결된 메소드를 배치하는 것이 일반적입니다.

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