Effective Java for Android (cheatsheet).

cxr Follow
Nov 26, 2016 · 5 min read

Effective Java is considered by many, one of the most important books for writing Java code that is **maintainable** in the long run and **efficient** at the same time. Android is using Java and that means that **all** of the advice given in the book must be applicable, right? Not exactly. Some people consider that *most* of the advice given by the book are not applicable in the Android development world. In my opinion, this is not the case. I believe that there are parts of the book that are not applicable, either because not all Java features are optimized to be used with Android (e.g. enums, serialization, etc) or due to the limitations that a mobile device has (e.g. Dalvik/ART behaves differently than a desktop JVM) . Nevertheless, *most* of the paradigms in the book can be used with little to no modification and can contribute to a healthier, cleaner and more maintainable code base.

This post is an attempt to concentrate what I consider the most important items from the book when developing an Android app. For those who read the book, this post might act as a reminder of the items/principles that are mentioned. For those who didn't read it (yet), it can be used to give them a taste of what the book offers.

Force non-instantiability

If you do not want an object to be created using the *new* keyword, enforce it using a **private constructor**. Especially useful for utility classes that contain only static functions.

```
class MovieUtils {
    private MovieUtils() {}

    static String titleAndYear(Movie movie) {
        [...]
    }
}
```

Static Factories

Instead of using the *new* keyword and the constructor use a **static factory method** (and a private constructor). These factory methods are **named**, are not required to return a **new instance** of the object each time and can return a **different subtype** depending on what's needed.

```
class Movie {
    [...]
    public static Movie create(String title) {
        return new Movie(title);
    }
}
```

***[Update] A useful tip by our reader** @stsdema28: Using static factories can make testing difficult. If that's the case, consider using non-static factories that you can mock during testing (or a factory interface that a fake can implement).*

Builders

When you have an object that requires more than ~3 constructor parameters, use a **builder** to construct the object. It might be a little more verbose to write but it **scales** well and it's very **readable**. If you are creating a value class, consider AutoValue.

```
class Movie {
    static Builder newBuilder() {
        return new Builder();
    }
    static class Builder {
        String title;
        Builder withTitle(String title) {
            this.title = title;
            return this;
        }
        Movie build() {
            return new Movie(title);
        }
    }

    private Movie(String title) {
    [...]
    }
}
// Use like this:
Movie matrix = Movie.newBuilder().withTitle("The Matrix").build();
```

Avoid mutability

Immutable is an object that stays the same for its entire lifetime. All the necessary data of the object are provided during its creation. There are various advantages to this approach like **simplicity**, **thread-safety** and **shareability**.

```
class Movie {
    [...]
    Movie sequel() {
        return Movie.create(this.title + " 2");
    }
}
// Use like this:
```

```
Movie toyStory = Movie.create("Toy Story");
Movie toyStory2 = toyStory.sequel();
```

It might be difficult to make every class immutable. If that's the case, make your class as immutable as possible (eg, fields **private final** and the class **final**). Object creation might be more expensive on mobile, therefore don't over do it.

Static member classes

If you define an inner class that does not depend on the outer class, don't forget to define it as **static**. Failing to do so will result in each instance of the inner class to have references to the outer class.

```
class Movie {
    [...]
    static class MovieAward {
        [...]
    }
}
```

Generics (almost) everywhere

Java provides type-safety and we should be grateful for it (see JS). Try to avoid using raw types or the Object type when possible. Generics provide, most of the times the mechanism to make your code type safe on compile time.

```
// DON'T
List movies = Lists.newArrayList();
movies.add("Hello!");
[...]
String movie = (String) movies.get(0);

// DO
List<String> movies = Lists.newArrayList();
movies.add("Hello!");
[...]
String movie = movies.get(0);
```

Don't forget that you can use generics and in your methods for parameters and returned values.

```
// DON'T
List sort(List input) {
    [...]
}

// DO
<T> List<T> sort(List<T> input) {
    [...]
}
```

To be a little more flexible you can use bounded wildcards to extend the range of types that you accept.

```
// Read stuff from collection - use "extends"
void readList(List<? extends Movie> movieList) {
    for (Movie movie : movieList) {
        System.out.print(movie.getTitle());
        [...]
    }
}

// Write stuff to collection - use "super"
void writeList(List<? super Movie> movieList) {
    movieList.add(Movie.create("Se7en"));
    [...]
}
```

Return empty

When having to return a list/collection with no result avoid *null*. Returning an empty collection leads to a **simpler interface** (no need to document and annotate the null-returning function) and **avoids accidental NPE**. Prefer to return the **same empty collection** rather than creating a new one.

```
List<Movie> latestMovies() {
    if (db.query().isEmpty()) {
        return Collections.emptyList();
    }
    [...]
}
```

No + String

Having to concatenate a few Strings, + operator might do. Never use it for a lot of String concatenations; the performance is really bad. Prefer a StringBuilder instead.

```
String latestMovieOneLiner(List<Movie> movies) {
    StringBuilder sb = new StringBuilder();
    for (Movie movie : movies) {
        sb.append(movie);
    }
    return sb.toString();
}
```

Recoverable exceptions

I am not in favor of throwing exceptions for indicating errors, but if you do so, make sure the **exception is checked** and the catcher of the exception is able to **recover from the error**.

```
List<Movie> latestMovies() throws MoviesNotFoundException {
    if (db.query().isEmpty()) {
        throw new MoviesNotFoundException();
    }
    [...]
}
```

Conclusion

This list is by no mean a complete list of the advice that is given in the book nor the short text explanations full-depth justifications. This was rather a cheat sheet for these useful tips :)

P.S. If you are like me and enjoy being productive throughout the day, check out microtasks, a task-as-undismissable-notification manager that helps you remember the small things that matter :)

Android Java Programming Mobile App Development Android App Development