Joshua Bloch - Effective Java (3rd) – 2018

Programing language
- core language: is it algorithmic, functional, object-oriented
- vocabulary: what data structures, operations, and facilities are provided by the standard libraries
- customary and effective ways to structure your code.

The Java programming language, for example, is object-oriented with single inheritance and supports an imperative (statement-oriented) coding style within each method. The libraries address graphic display support, networking, distributed computing, and security

Many items contain one or more program examples illustrating some practice to be avoided. Such examples, sometimes known as **antipatterns**,

You should not slavishly follow these rules, but violate them only occasionally and with good reason. Learning the art of programming, like most other disciplines, consists of first learning the rules and then learning when to break them.

See terms as they are defined in *The Java Language Specification, Java SE 8 Edition*

Creating and Destroying Objects

Item 1: Consider static factory methods instead of constructors

A class can provide a public *static factory method*, which is simply a static method that returns an instance of the class.
- Unlike constructors they have names
- They are not required to create a new object each time they're invoked.
- They can return an objec of any subtype
- Class of the returned object can vary from call to call as a function of the input parameters
- Class of the returned object need not exist when the class containing the method is written (5)
- Classes without public or protected constructors cannot be subclassed

Also consider **interface-based frameworks** where interfaces provide natural return types for static factory methods. (requires the client to refer to the returned object by interface rather than implementation class, which is generally good practice)

(5) Such flexible static factory methods form the basis of **service provider frameworks**, like the Java Database Connectivity API (JDBC). A service provider framework is a system in which providers implement a service, and the system makes the implementations available to clients, decoupling the clients from the implementations.

Some common names for static factory method
- from
- of
- valueOf
- getInstance / instance / new instance
- create
- getType / newType /type

Do not use them if you are going to do inheritence

Item 2: Consider a builder when faced with many constructor parameters

The builder is typically a static member class of the class it builds.'
The Builder pattern simulates named optional parameters

In order to create an object, you must first create its builder.

Use it if if many of the parameters are optional-

Item 3: Enforce the singleton property with a private constructor or an enum type

Item 4: Enforce noninstantiability with a private constructor

Class that is just a grouping of static methods and static fields. Such *utility classes* were not designed to be instantiated- class can be made noninstantiable by including a private constructor: Also consider making it final,

Item 5: Prefer dependency injection to hardwiring resources

Pass the resource into the constructor when creating a new instance.

Use a dependency injection framework, such as Dagger, Guice, or Spring.

Item 6: Avoid creating unnecessary objects

String s = new String("bikini"); // **DON'T DO THIS!**
The statement creates a new String instance each time it is executed**.**
String s = "bikini"; // **DO THIS**

```
// Reusing expensive object for improved performance
public class RomanNumerals {
private static final Pattern ROMAN = Pattern.compile(
"^(?=.)M*(C[MD]|D?C{0,3})"
+ "(X[CL]|L?X{0,3})(I[XV]|V?I{0,3})$");
static boolean isRomanNumeral(String s) {
return ROMAN.matcher(s).matches();
}
}
```

Prefer primitives to boxed primitives, and watch out for unintentional autoboxing.

This item should not be misconstrued to imply that object creation is expensive and should be avoided. On the contrary, the creation and reclamation of small objects whose constructors do little explicit work is cheap, especially on modern JVM implementations.

Avoiding object creation by maintaining your own *object pool* is a bad idea unless the objects in the pool are extremely heavyweight

Don’t create a new object when you should reuse an existing one but also Don’t reuse an existing object when you should create a new one

Item 7: Eliminate obsolete object references

Nulling out object references should be the exception rather than the norm.
The best way to eliminate an obsolete reference is to let the variable that contained the reference fall out of scope. This occurs naturally if you define each variable in the narrowest possible scope.

Whenever a class manages its own memory, the programmer should be alert for memory leaks TODO investigate and practice this.

Another common source of memory leaks is caches.

A third common source of memory leaks is listeners and other callbacks.
If you implement an API where clients register callbacks but don't deregister them explicitly, they will accumulate unless you take some action. One way to ensure that callbacks are garbage collected promptly is to store only **weak references** to them, for instance, by storing them only as keys in a WeakHashMap.

Use debugging tool known as a heap profiler.

Item 8: Avoid finalizers and cleaners

So what should you do instead of writing a finalizer or cleaner for a class whose objects encapsulate resources that require termination, such as files or threads? Just have your class implement AutoCloseable

Item 9: Prefer try-with-resources to try-finally

Methods Common to All Objects

Object is a concrete class**, it is designed primarily for extension.** All of its nonfinal methods (equals, hashCode, toString, clone, and finalize) have explicit **general contracts** because they are designed to be overridden. It is the responsibility of any class overriding these methods to obey their general contracts; failure to do so will prevent other classes that depend on the contracts (such as HashMap and HashSet) from functioning properly in conjunction with the class.

Item 10: Obey the general contract when overriding equals

Do not override equals
- each instance of class is unique (represents activity, rather than value ex: Thread)
- no nned for the class to provide a logical equality test (Pattern)
- superclass has already overridden equal this can be good
- class is private and you are shure that equal will never be invoked

Equlas contract
- reflexive x.equals(x) return true, x not null
- symmetric x.equals(y) and y.equals(x) same result, xy, not null
- tranzitive x.equals(y), y.equals(z) then x.equal(z) return true, x y z not null
- consistent x.eqauls(y) alwys same result, x y not null
- x.equlas(null) return false

Do not write an equals method that depends on unreliable resources

Steps:
1. use == to check reference
2. use instance of to check type
3. For each "significant" field in the class, check if that field of the argument
matches the corresponding field of this object

Some object reference fields may legitimately contain **null**. To avoid the possibility of a NullPointerException, check such fields for equality using the static method **Objects.equals(**Object, Object).

If field comparaion is more complex, you may want to store a **canonical form** of the field so the equals method can do a cheap exact comparison on canonical forms rather than a more costly nonstandard comparison

When you are finished writing your equals method, ask yourself three questions: Is it symmetric? Is it transitive? Is it consistent? And don't just ask yourself; write unit tests to check. Writing and testing equals (and hashCode) methods is tedious, and the resulting code is mundane. An excellent alternative to writing and testing these methods manually is to use Google’s open source AutoValue framework,

Don’t substitute another type for Object in the equals declaration

```
public boolean equals(MyClass o) {
...
}
```

Item 11: Always override hashCode when you override equals

Objects.hash(lineNum, prefix, areaCode);

Item 12: Always override toString

Whether or not you decide to specify the format, you should clearly document your intentions.

```
/**
* Returns the string representation of this phone number.
* The string consists of twelve characters whose format is
* "XXX-YYY-ZZZZ", where XXX is the area code, YYY is the
* prefix, and ZZZZ is the line number. Each of the capital
* letters represents a single decimal digit.
*
* If any of the three parts of this phone number is too small
* to fill up its field, the field is padded with leading zeros.
* For example, if the value of the line number is 123, the last
* four characters of the string representation will be "0123".
*/
@Override public String toString() {
return String.format("%03d-%03d-%04d",
areaCode, prefix, lineNum);
}
```

You should, however, write a toString method in any abstract class whose subclasses share a common string representation.

Item 13: Override clone judiciously

Do not override clone.

Item 14: Consider implementing Comparable

By implementing Comparable, a class indicates that its instances have a natural ordering.

Because the Comparable interface is parameterized, the compareTo method is statically typed, so you don’t need to type check or cast its argument..

If the argument is null, the invocation should throw a NullPointer-Exception, and it will, as soon as the method attempts to access its members.

Use of the relational operators < and > in compareTo methods is verbose and error-prone and no longer recommended

In Java 8, the Comparator interface was outfitted with a set of **comparator construction methods,** which enable fluent construction of comparators. These comparators can then be used to implement a compareTo method, as required by the Comparable interface. (but it is slower)

return o1.hashCode() - o2.hashCode(); **// BROKEN difference-based comparator - violates transitivity!**
You can use this Integer.compare(o1.hashCode(), o2.hashCode());

Classes and Interfaces

Item 15: Minimize the accessibility of classes and members

Information hiding increases software reuse because components that aren’t tightly coupled often prove useful in other contexts besides the ones for which they were developed

Item 16: In public classes, use accessor methods, not public fields

Public classes should never expose mutable fields

Item 17: Minimize mutability

Immutable classes are easier to design, implement, and use than mutable classes. They are less prone to error and are more secure.

Steps
make class final or private constructor and static create method
make all fields final
make all fields private
If class has any fields that refer to mutable objects ensure that client of the class cannot obtain reference to these objects. **Make defensive copies**

```
// inside complex object
public Complex times(Complex c) {
        return new Complex(re * c.re - im * c.im,
        re * c.im + im * c.re);
}
```

This pattern is known as the **functional approach** because methods return the result of applying a function to their operand, without modifying it. Emphasizes the fact that methods don’t change the values of the objects.
Immutable classes should therefore encourage clients to reuse existing instances wherever possible.
Also An immutable class can provide static factories (Item 1) that cache frequently requested instances to avoid creating new instances when existing ones would do.
The major disadvantage of immutable classes is that they require a separate object for each distinct value. Creating these objects can be costly, especially if they are large.

Item 18: Favor composition over inheritance

Inheritance violates encapsulation. Subclass depends on the implementaion details of its superclass for its proper function.

Item 19: Design and document for inheritance or else prohibit it

First, the class must document precisely the effects of overriding any method. In other words, the class must document its *self-use* of overridable methods**.** For each public or protected method, the documentation must indicate which overridable methods the method invokes, in what sequence, and how the results of each invocation affect subsequent processing

A method that invokes overridable methods contains a description of these invocations at the end of its documentation comment. The description is in a special section of the specification, labeled “Implementation Requirements,” which is generated by the Javadoc tag **@implSpec**. This section describes the inner workings of the method.

See: AbstractCollection: public boolean remove(Object o) documentation.

You must test your class by writing subclasses *before* you release it.

Item 20: Prefer interfaces to abstract classes

Existing classes cannot, in general, be retrofitted to extend a new abstract class.

Interfaces allow for the construction of nonhierarchical type frameworks.

You can, however, combine the advantages of interfaces and abstract classes by providing an abstract **skeletal** implementation class to go with an interface (Template method pattern). By convention, skeletal implementation classes are called Abstract*Interface*, where *Interface* is the name of the interface they implement (AbstractMap, AbstractCollection…)

Item 21: Design interfaces for posterity

It is not always possible to write a default method that maintains all invariants of every conceivable implementation.

It is also worth noting that default methods were not designed to support removing methods from interfaces or changing the signatures of existing methods.

It is still of the utmost importance to design interfaces with great care.

Therefore, it is critically important to test each new interface before you release it. Multiple programmers should implement each interface in different ways. At a minimum, you should aim for three diverse implementations. Equally important is to write multiple client programs that use instances of each new interface to perform various tasks.

Item 22: Use interfaces only to define types

When a class implements an interface, the interface serves as a *type* that can be used to refer to instances of the class.

Do not use interfaces to declare constants.

Item 23: Prefer class hierarchies to tagged classes

Occasionally you may run across a class whose instances come in two or more flavors and contain a *tag* field indicating the flavor of the instance.

```
Class X {
        ShapeType types; // do not do this
}
```

Such *tagged classes* have numerous shortcomings. They are cluttered with boilerplate, including enum declarations, tag fields, and switch statements. Solution is to create class hierarchy.

To transform a tagged class into a class hierarchy, first define an abstract class containing an abstract method for each method in the tagged class whose behavior depends on the tag value.

Item 24: Favor static member classes over nonstatic

One common use of a static member class is as a **public helper class**, useful only in conjunction with its outer class.

One common use of a nonstatic member class is to define an **Adapter** that allows an instance of the outer class to be viewed as an instance of some unrelated class.

Item 25: Limit source files to a single top-level class

The risks stem from the fact that defining multiple top-level classes in a source file makes it possible to provide multiple definitions for a class. Which definition gets used is affected by the order in which the source files are passed to the compiler.

If you are tempted to put multiple top-level classes into a single source file, consider using static member classes

Generics