Effective Java 3rd Edition Summary

Disclaimer

This is a summary of the book “Effective Java 3rd Edition” by Joshua Bloch (https://twitter.com/joshbloch (https://twitter.com/joshbloch))

The book is awesome and has a lot of interesting views in it. This document is only my take on what I want to really keep in mind when working with Java.

I hope it’s not a copyright infringement. If it is, please contact me in order to remove this file from github.

Double disclaimer

This summary was done by David Sauvage (https://github.com/david-sauvage (https://github.com/david-sauvage)).

I simply copy-pasted the markdown here and converted the `items` to markdown headers so that sections can be accessed using the side bar.

Creating and destroying objects

Static factory methods

Pros

- They have a name
- You can use them to control the number of instance (Example : Boolean.valueOf)
- You can return a subtype of the return class

Cons

- You can’t subclass a class without public or protected constructor
- It can be hard to find the static factory method for a new user of the API

Example :

```
public static Boolean valueOf(boolean b){
        return b ? Boolean.TRUE :  Boolean.FALSE;
}
```

Builder

Pros

- Builder are interesting when your constructor may need many arguments
- It’s easier to read and write
- Your class can be immutable (Instead of using a java bean)
- You can prevent inconsistent state of you object

Example :

```
public class NutritionFacts {
        private final int servingSize;
        private final int servings;
        private final int calories;
        private final int fat;
        private final int sodium;

        public static class Builder {
                //Required parameters
                private final int servingSize:
                private final int servings;
                //Optional parameters - initialized to default values
                private int calories          = 0;
                private int fat                       = 0;
                private int sodium                    = 0;

                public Builder (int servingSize, int servings) {
                        this.servingSize = servingSize;
                        this.servings = servings;
                }
                public Builder calories (int val) {
                        calories = val;
                        return this;
                }
                public Builder fat (int val) {
                        fat = val;
                        return this;
                }
                public Builder sodium (int val) {
                        sodium = val;
                        return this;
                }
                public NutritionFacts build(){
                        return new NutritionFacts(this);
                }
        }
        private NutritionFacts(Builder builder){
                servingSize           = builder.servingSize;
                servings              = builder.servings;
                calories              = builder.calories;
                fat                   = builder.fat;
                sodium                = builder.sodium;
        }
}
```

Think of Enum to implement the Singleton pattern

Example :

```
public enum Elvis(){
        INSTANCE;
        ...
        public void singASong(){...}
}
```

Utility class should have a private constructor

A utility class with only static methods will never be instantiated. Make sure it's the case with a private constructor to prevent the construction of a useless object.

Example :

```
public class UtilityClass{
        // Suppress default constructor for noninstantiability
        private UtilityClass(){
                throw new AssertionError();
        }
        ...
}
```

Dependency Injection

A common mistake is the use of a singleton or a static utility class for a class that depends on underlying ressources.
The use of dependency injection gives us more flexibility, testability and reusability

Example :

```
public class SpellChecker {
        private final Lexicon dictionary;
        public SpellChecker (Lexicon dictionary) {
                this.dictionary = Objects.requireNonNull(dictionary);
        }
        ...
}
```

Avoid creating unnecessary objects

When possible use the static factory method instead of constructor (Example : Boolean)
Be vigilant on autoboxing. The use of the primitive and his boxed primitive type can be harmful.
Most of the time use primitives.

Eliminate obsolete object references

Memory leaks can happen in :

- A class that managed its own memory
- Caching objects
- The use of listeners and callback

In those three cases the programmer needs to think about nulling object references that are known to be obsolete

Example :
In a stack implementation, the pop method could be implemented this way :

```
public pop(){
        if (size == 0) {
                throw new EmptyStackException();
        }
        Object result = elements[--size];
        elements[size] = null; // Eliminate obsolete references.
        return result;
}
```

Avoid finalizers and cleaners

Finalizers and cleaners are not guaranteed to be executed. It depends on the garbage collector and it can be executed long after the object is not referenced anymore.
If you need to let go of resources, think about implementing the *AutoCloseable* interface.

Try with resources

When using try-finally blocks exceptions can occur in both the try and finally block. It results in non clear stacktraces.
Always use try with resources instead of try-finally. It's clearer and the exceptions that can occured will be clearer.

Example :

```
static void copy(String src, String dst) throws IOException {
        try (InputStream in = new InputStream(src);
                OutputStream out = new FileOutputStream(dst)) {
                byte[] buf = new byte[BUFFER_SIZE];
                int n;
                while ((n = in.read(buf)) >= 0) {
                        out.write(buf, 0, n);
                }
        }
}
```

Methods of the Object class

equals

The equals method needs to be overriden when the class has a notion of logical equality.
This is generally the case for value classes.

The equals method must be :

- Reflexive (x = x)
- Symmetric (x = y => y = x)
- Transitive (x = y and y = z => x = z)
- Consistent
- For non null x, x.equals(null) should return false

Not respecting those rules will have impact on the use of List, Set or Map.

hashCode

The hashCode method needs to be overriden if the equals method is overriden.

Here is the contract of the hashCode method :

- hashCode needs to be consistent
- if a.equals(b) is true then a.hashCode() == b.hashCode()
- if a.equals(b) is false then a.hashCode() doesn't have to be different of b.hashCode()

If you don't respect this contract, HashMap or HashSet will behave erratically.

toString

Override toString in every instantiable classes unless a superclass already did it.
Most of the time it helps when debugging.
It needs to be a full representation of the object and every informations contained in the toString representation should be accessible in some other way in order to avoid programmers to parse the String representation.

clone

When you implement Cloneable, you should also override clone with a public method whose return type is the class itself.
This method should start by calling super.clone and then also clone all the mutable objects of your class.

Also, when you need to provide a way to copy classes, you can think first of copy constructor or copy factory except for arrays.

Implementing Comparable

If you have a value class with an obvious natural ordering, you should implement Comparable.

Here is the contract of the compareTo method :

- signum(x.compareTo(y)) == -signum(y.compareTo(x))
- x.compareTo(y) > 0 && y.compareTo(z) > 0 => x.compareTo(z) > 0
- x.compareTo(y) == 0 => signum(x.compareTo(z)) == signum(y.compareTo(z))

It's also recommended that (x.compareTo(y) == 0) == x.equals(y).
If it's not, it has to be documented that the natural ordering of this class is inconsistent with equals.

When confronted to different types of Object, compareTo can throw ClassCastException.

Classes and Interfaces

Accessibility

Make accessibility as low as possible. Work on a public API that you want to expose and try not to give access to implementation details.

Accessor methods

Public classes should never expose its fields. Doing this will prevent you to change its representation in the future.
Package private or private nested classes, can, on the contrary, expose their fields since it won't be part of the API.

Immutability

To create an immutable class :

- Don't provide methods that modify the visible object's state
- Ensure that the class can't be extended
- Make all fields final
- Make all fields private
- Don't give access to a reference of a mutable object that is a field of your class

As a rule of thumb, try to limit mutability.

Favor composition over inheritance

With inheritance, you don't know how your class will react with a new version of its superclass. For example, you may have added a new method whose signature will happen to be the same than a method of its superclass in the next release.

You will then override a method without even knowing it.

Also, if there is a flaw in the API of the superclass you will suffer from it too.
With composition, you can define your own API for your class.

As a rule of thumb, to know if you need to choose inheritance over composition, you need to ask yourself if B is really a subtype of A.

Example :

```
// Wrapper class – uses composition in place of inheritance
public class InstrumentedSet<E> extends ForwardingSet<E> {
        private int addCount = 0;
        public InstrumentedSet (Set<E> s){
                super(s)
        }

        @Override
        public boolean add(E e){
                addCount++;
                return super.add(e);
        }

        @Override
        public boolean addAll (Collection< ? extends E> c){
                addCount += c.size();
                return super.addAll(c);
        }

        public int getAddCount() {
                return addCount;
        }
}

// Reusable forwarding class
public class ForwardingSet<E> implements Set<E> {
        private final Set<E> s; // Composition
        public ForwardingSet(Set<E> s) { this.s = s ; }

        public void clear() {s.clear();}
        public boolean contains(Object o) { return s.contains(o);}
        public boolean isEmpty() {return s.isEmpty();}
        ...
}
```

Create classes for inheritance or forbid it

First of all, you need to document all the uses of overridable methods.
Remember that you'll have to stick to what you documented.
The best way to test the design of your class is to try to write subclasses.
Never call overridable methods in your constructor.

If a class is not designed and documented for inheritance it should be me made forbidden to inherit her, either by making it final, or making its constructors private (or package private) and use static factories.

Interfaces are better than abstract classes

Since Java 8, it's possible to implements default mechanism in an interface.
Java only permits single inheritance so you probably won't be able to extends your new abstract class to exising classes when you always will be permitted to implements a new interface.

When designing interfaces, you can also provide a Skeletal implementation. This type of implementation is an abstract class that implements the interface.
It can help developers to implement your interfaces and since default methods are not permitted to override Object methods, you can do it in your Skeletal implementation.
Doing both allows developers to use the one that will fit their needs.

Design interfaces for posterity

With Java 8, it's now possible to add new methods in interfaces without breaking old implemetations thanks to default methods.
Nonetheless, it needs to be done carefully since it can still break old implementations that will fail at runtime.

Interfaces are mean't to define types

Interfaces must be used to define types, not to export constants.

Example :

```
//Constant interface antipattern. Don't do it !
public interface PhysicalConstants {
        static final double AVOGADROS_NUMBER = 6.022_140_857e23;
        static final double BOLTZMAN_CONSTANT = 1.380_648_52e-23;
        ...
}
//Instead use
public class PhysicalConstants {
        private PhysicalConstants() {} //prevents instatiation

        public static final double AVOGADROS_NUMBER = 6.022_140_857e23;
        public static final double BOLTZMAN_CONSTANT = 1.380_648_52e-23;
        ...
}
```

Tagged classes

Those kinds of classes are clutted with boilerplate code (Enum, switch, useless fields depending on the enum).
Don't use them. Create a class hierarchy that will fit you needs better.

Nested classes

If a member class does not need access to its enclosing instance then declare it static.
If the class is non static, each intance will have a reference to its enclosing instance. That can result in the enclosing instance not being garbage collected and memory leaks.

One single top level class by file

Even thow it's possible to write multiple top level classes in a single file, don't !
Doing so can result in multiple definition for a single class at compile time.

Generics

Raw types

A raw type is a generic type without its type parameter (Example : *List* is the raw type of *List<E>*)
Raw types shouldn't be used. They exist for compatibility with older versions of Java.

We want to discover mistakes as soon as possible (compile time) and using raw types will probably result in error during runtime.
We still need to use raw types in two cases :

- Usage of class litrals (List.class)
- Usage of instanceof

Examples :

```
//Use of raw type : don't !
private final Collection stamps = ...
stamps.add(new Coin(...)); //Erroneous insertion. Does not throw any error
Stamp s = (Stamp) stamps.get(i); // Throws ClassCastException when getting the Coin

//Common usage of instance of
if (o instanceof Set){
        Set<?> = (Set<?>) o;
}
```

Unchecked warnings

Working with generics can often create warnings about them. Not having those warnings assure you that your code is typesafe.
Try as hard as possible to eliminate them. Those warnings represents a potential ClassCastException at runtime.
When you prove your code is safe but you can’t remove this warning use the annotation @SuppressWarnings(“unchecked”) as close as possible to the declaration.
Also, comment on why it is safe.

List and arrays

Arrays are covariant and generics are invariant meaning that Object[] is a superclass of String[] when List<Object> is not for List<String>.
Arrays are reified when generics are erased. Meaning that array still have their typing right at runtime when generics don’t. In order to assure retrocompatibility with previous version List<String> will be a List at runtime.
Typesafety is assured at compile time with generics. Since it’s always better to have our coding errors the sooner (meaning at compile time), prefer the usage of generics over arrays.

Generic types_

Generic types are safer and easier to use because they won’t require any cast from the user of this type.
When creating new types, always think about generics in order to limit casts.

Generic methods

Like types, methods are safer and easier to use it they are generics.
If you don’t use generics, your code will require users of your method to cast parameters and return values which will result in non typesafe code.

Bounded wildcards

Bounded wildcards are important in order to make our code as generic as possible.
They allow more than a simple type but also all their sons (? extends E) or parents (? super E)

Examples :

If we have a stack implementation and we want to add two methods pushAll and popAll, we should implement it this way :

```
//We want to push in everything that is E or inherits E
public void pushAll(Iterable<? Extends E> src){
        for (E e : src) {
                push(e);
        }
}

//We want to pop out in any Collection that can welcome E
public void popAll(Collection<? super E> dst){
        while(!isEmpty()) {
                dst.add(pop());
        }
}
```

Generics and varargs

Even thow it's not legal to declare generic arrays explicitly, it's still possible to use varargs with generics.
This inconsistency has been a choice because of its usefulness (Exemple : Arrays.asList(T... a)).
This can, obviously, create problems regarding type safety.
To make a generic varargs method safe, be sure :

- it doesn't store anything in the varargs array
- it doesn't make the array visible to untrusted code
  When those two conditions are met, use the annotation @SafeVarargs to remove warnings that you took care of and show users of your methods that it is typesafe.

Typesafe heterogeneous container

Example :

```
public class Favorites{
        private Map<Class<?>, Object> favorites = new HashMap<Class<?>, Object>();

        public <T> void putFavorites(Class<T> type, T instance){
                if(type == null)
                        throw new NullPointerException("Type is null");
                favorites.put(type, type.cast(instance));//runtime safety with a dyna
        }

        public <T> getFavorite(Class<T> type){
                return type.cast(favorites.get(type));
        }
}
```

Enums and annotations

Enums instead of int constants

Prior to enums it was common to use int to represent enum types. Doing so is now obsolete and enum types must be used.
The usage of int made them difficult to debug (all you saw was int values).

Enums are classes that export one instance for each enumeration constant. They are instance controlled. They provide type safety and a way to iterate over each values.

If you need a specific behavior for each value of your enum, you can declare an abstract method that you will implement for each value.

Enums have an automatically generated valueOf(String) method that translates a constant's name into the constant. If the toString method is overriden, you should write a fromString method.

Example :

```
public enum Operation{
        PLUS("+") { double apply(double x, double y){return x + y;}},
        MINUS("-") { double apply(double x, double y){return x - y;}},
        TIMES("*") { double apply(double x, double y){return x * y;}},
        DIVIDE("/") { double apply(double x, double y){return x / y;}};

        private final String symbol;
        private static final Map<String, Operation>   stringToEnum = Stream.of(value

        Operation(String symbol) {
                this.symbol = symbol;
        }

        public static Optional<Operation> fromString(String symbol) {
                return Optional.ofNullable(stringToEnum.get(symbol);
        }

        @Override
        public String toString() {
                return symbol;
        }

        abstract double apply(double x, double y);
}
```

Instance fields instead of ordinals

Never use the ordinal method to calculate a value associated with an enum.

Example :

```
//Never do this !
public enum Ensemble{
        SOLO, DUET, TRIO, QUARTET;
        public int numberOfMusicians(){
                return ordinal() + 1;
        }
}

//Instead, do this :
public enum Ensemble{
        SOLO(1), DUET(2), TRIO(3), QUARTET(4);

        private final int numberOfMusicians;

        Ensemble(int size) {
                this.numberOfMusicians = size;
        }

        public int numberOfMusicians(){
                return numberOfMusicians;
        }
}
```

EnumSet instead of bit fields

Before enums existed, it was common to use bit fields for enumerated types that would be used in sets. This would allow you to combine them but they have the same issues than int constants we saw in item 34.
Instead use EnumSet to combine multiple enums.

Example :

```
public class Text {
        public enum Style {BOLD, ITALIC, UNDERLINE}
        public void applyStyle(Set<Style> styles) {...}
}

//Then you would use it like this :
text.applyStyle(EnumSet.of(Style.BOLD, Style.ITALIC));
```

EnumMap instead of ordinal

You may want to store data by a certain enum. For that you could have the idea to use the ordinal method. This is a bad practice.
Instead, prefer the use of EnumMaps.

Emulate extensible enums with interfaces

The language doesn't allow us to write extensible enums. In the few cases that we would want an enum type to be extensible, we can emulate it with an interface written for the basic enum. Users of the api will be able to implements this interface in order to "extend" your enum.

Annotations instead of naming patterns

Prior to JUnit 4, you needed to name you tests by starting with the word "test". This is a bad practice since the compiler will never complain if, by mistake, you've names a few of them "tset*".
Annotations are a good way to avoid this kind of naming patterns and gives us more security.

Example :

```
//Annotation with array parameter
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
public @interface ExceptionTest {
 Class<? extends Exception>[] value();
}

//Usage of the annotation
@ExceptionTest( {IndexOutOfBoundsException.class, NullPointerException.class})
public void myMethod() { ... }

//By reflexion you can use the annotation this way
m.isAnnotationPresent(ExceptionTest.class);
//Or get the values this way :
Class<? extends Exception>[] excTypes = m.getAnnotation(ExceptionTest.class).value()
```

Use @Override

You should use the @Override for every method declaration that you believe to override a superclass declaration.

Example :

```
//Following code won't compile. Why ?
@Override
public boolean equals(Bigram b) {
        return b.first == first && b.second == second;
}

/**
This won't compile because we aren't overriding the Object.equals method. We are ove
The annotation allows the compiler to warn us of this mistake. That's why @Override
**/
```

Marker interfaces

A marker interface is an interface that contains no method declaration. It only “marks” a class that implements this interface. One common example in the JDK is Serializable.
Using marker interface results in compile type checking.

Lambdas and streams

Lambdas are clearer than anonymous classes

Lambdas are the best way to represent function objects. As a rule of thumb, lambdas needs to be short to be readable. Three lines seems to be a reasonnable limit.
Also the lambdas needs to be self-explanatory since it lacks name or documentation. Always think in terms of readability.

Method references

Most of the time, method references are shorter and then clearer. The more arguments the lambdas has, the more the method reference will be clearer.
When a lambda is too long, you can refactor it to a method (which will give a name and documentation) and use it as a method reference.

They are five kinds of method references :

Method ref type	Example	Lambda equivalent
Static	Integer::parseInt	str -> Integer.parseInt(str)
Bound	Instant.now()::isAfter	Instant then = Instant.now(); t->then.isAfter(t)
Unbound	String::toLowerCase	str -> str.toLowerCase()
Class Constructor	TreeMap<K,V>::new	() -> new TreeMap<K,V>
Array Constructor	int[]::new	len -> new int[len]

Standard functional interfaces

java.util.Function provides a lot of functional interfaces. If one of those does the job, you should use it

Here are more common interfaces :

Interface	Function signature	Example
UnaryOperator<T>	T apply(T t)	String::toLowerCase
BinaryOperator<T>	T apply(T t1, T t2)	BigInteger::add
Predicate<T>	boolean test(T t)	Collection::isEmpty
Function<T,R>	R apply(T t)	Arrays::asList
Supplier<T>	T get()	Instant::now
Consumer<T>	void accept(T t)	System.out::println

When creating your own functional interfaces, always annotate with @FunctionalInterfaces so that it won't compile unless it has exactly one abstract method.

Streams

Carefully name parameters of lambda in order to make your stream pipelines readable. Also, use helper methods for the same purpose.

Streams should mostly be used for tasks like :

- Transform a sequence of elements
- Filter a sequence of elements
- Combine sequences of elements
- Accumulate a sequence of elements inside a collection (perhaps grouping them)
- Search for en element inside of a sequence

Prefer side-effect-free functions in streams

Programming with stream pipelines should be side effect free.
The terminal forEach method should only be used to report the result of a computation not to perform the computation itself.
In order to use streams properly, you need to know about collectors. The most importants are toList, toSet, toMap, groupingBy and joining.

Return collections instead of streams

The collection interface is a subtype of Iterable and has a stream method. It provides both iteration and stream access.
If the collection in too big memory wise, return what seems more natural (stream or iterable)

Parallelization

Parallelizing a pipeline is unlikely to increase its performance if it comes from a Stream.iterate or the limit method is used.
As a rule of thumb, parallelization should be used on ArrayList, HashMap, HashSet, ConcurrentHashMap, arrays, int ranges and double ranges. Those structure can be divided in any desired subranged and so on, easy to work among parrallel threads.

Methods

Check parameters for validity

When writing a public or protected method, you should begin by checking that the parameters are not enforcing the restrictions that you set.
You should also document what kind of exception you will throw if a parameter enforce those restrictions.
The *Objects.requireNonNull* method should be used for nullability checks.

Defensive copies

If a class has mutable components that comes from or goes to the client, the class needs to make defensive copies of those components.

Example :

```
//This example is a good example but since java 8, we would use Instant instead of D
public final class Period{
        private final Date start;
        private final Date end;
        /**
        * @param start the beginning of the period
        * @param end the end of the period; must not precede start;
        * @throws IllegalArgumentException if start is after end
        * @throws NullPointerException if start or end is null
        */
        public Period(Date start, Date end) {
                this.start = new Date(start.getTime());
                this.end = new Date(end.getTime());
                if(start.compare(end) > 0) {
                        throw new IllegalArgumentException(start + " after " + end );
                }
        }

        public Date start(){
                return new Date(start.getTime());
        }

        public Date end(){
                return new Date(end.getTime());
        }
        ...
}
```

Method signature

Few rules to follow when designing you API :

- Choose your methode name carefully. Be explicit and consistent.
- Don’t provide too many convenience methods. A small API is easier to learn and use.
- Avoid long parameter lists. Use helper class if necessary.
- Favor interfaces over classes for parameter types.
- Prefer enum types to boolean parameters when it makes the parameter more explicit.

Overloading

Example :

```
// Broken! - What does this program print?
public class CollectionClassifier {
        public static String classify(Set<?> s) {
                return "Set";
        }
        public static String classify(List<?> lst) {
                return "List";
        }
        public static String classify(Collection<?> c) {
                return "Unknown Collection";
        }
        public static void main(String[] args) {
                Collection<?>[] collections = {
                        new HashSet<String>(),
                        new ArrayList<BigInteger>(),
                        new HashMap<String, String>().values()
                };
        for (Collection<?> c : collections)
                System.out.println(classify(c)); // Returns "Unknown Collection" 3 t
        }
}
```

As shown in the previous example overloading can be confusing. It is recommanded to never export two overloadings with the same number of parameters.
If you have to, consider giving different names to your methods. (writeInt, writeLong...)

Varargs

Varargs are great when you need to define a method with a variable number of arguments.
Always precede the varargs parameter with any required parameter.

Return empty collections or arrays instead of null

Returning null when you don't have elements to return makes the use of your methods more difficult. Your client will have to check if you object is not null.
Instead always return an empty array of collection.

Return of Optionals

You should declare a method to return Optional<T> if it might not be able to return a result and clients will have to perform special processing if no result is returned.
You should never use an optional of a boxed primitive. Instead use OptionalInt, OptionalLong etc...

Documentation

Documentation should be mandatory for exported API.

General programming

Minimize the scope of local variables

To limit the scope of your variables, you should :

- declare them when first used
- use for loops instead of while when doable
- keep your methods small and focused

```
//Idiom for iterating over a collection
for (Element e : c) {
        //Do something with e
}

//Idiom when you need the iterator
for (Iterator<Element> i = c.iterator() ; i.hasNext() ; ) {
        Element e = i.next();
        //Do something with e
}

//Idiom when the condition of for is expensive
for (int i = 0, n = expensiveComputation() ; i < n ; i++) {
        //Do something with i
}
```

For each loops instead of traditional for loops

The default for loop must be a for each loop. It’s more readable and can avoid you some mistakes.

Unfortunately, there are situations where you can’t use this kind of loops :

- When you need to delete some elements
- When you need to replace some elements
- When you need to traverse multiple collections in parallel

Use the standard libraries

When using a standard library you take advantage of the knowledge of experts and the experience of everyone who used it before you.
Don’t reinvent the wheel. Library code is probably better than code that we would write simply because this code receives more attention than what we could afford.

Avoid float and double for exact answers

Float and double types are not suited for monetary calculations. Use BigDecimal, int or long for this kind of calculation.

Prefer primitives to boxed primitives

Use primitives whenever you can. The use of boxed primitives is essentially for type parameters in parameterized types (example : keys and values in collections)

```
//Can you spot the object creation ?
Long sum = 0L;
for (long i = 0 ; i < Integer.MAX_VALUE ; i++) {
        sum += i;
}
System.out.println(sum);

//sum is repeatably boxed and unboxed which cause a really slow running time.
```

Avoid Strings when other types are more appropriate

Avoid natural tendency to represent objects as Strings when there is better data types available.

String concatenation

Don’t use the String concatenation operator to combine more than a few strings. Instead, use a StringBuilder.

Refer to objects by their interfaces

If an interface exists, parameters, return values, variables and fields should be declared using this interface to insure flexibility.
If there is no appropriate interface, use the least specific class that provides the functionality you need.

Prefer interfaces to reflection

Reflextion is a powerful tool but has many disadvantages.
When you need to work with classes unknown at compile time, try to only use it to instantiate object and then access them by using an interface of superclass known at compile time.

Native methods

It’s really rare that you will need to use native methods to improve performances. If it’s needed to access native libraries use as little native code as possible.

Optimization

Write good programs rather than fast one. Good programs localize design decisions within individual components so those individuals decisions can be changed easily if performance becomes an issue.
Good designs decisions will give you good performances.
Measure performance before and after each attempted optimization.

Naming conventions

Indentifier Type	Examples
Package	org.junit.jupiter, com.google.common.collect
Class or Interface	Stream, FutureTask, LinkedHashMap, HttpServlet
Method or Field	remove, groupBy, getCrc
Constant Field	MIN_VALUE, NEGATIVE_INFINITY
Local Variable	i, denom, houseNum
Type Parameter	T, E, K, V, X, R, U, V, T1, T2

Exceptions

Exceptions are for exceptional conditions

Exceptions should never be used for ordinary control flow. They are designed for exceptional conditions and should be used accordingly.

Checked exceptions and runtime exceptions

Use checked exceptions for conditions from which the caller can reasonably recover.
Use runtime exceptions to indicate programming errors.
By convention, *errors* are only used by the JVM to indicate conditions that make execution

impossible.
Therefore, all the unchecked throwables you implement must be a subclass of RuntimeException.

Avoid unnecessary use of checked exceptions

When used sparingly, checked exceptions increase the reliability of programs. When overused, they make APIs painful to use.
Use checked exceptions only when you want the callers to handle the exceptional condition.
Remember that a method that throws a checked exception can't be used directly in streams.

Standard exceptions

When appropriate, use the exceptions provided by the jdk. Here's a list of the most common exceptions :

Exception	Occasion for Use
IllegalArgumentException	Non-null parameter value is inappropriate
IllegalStateException	Object state is inappropriate for method invocation
NullPointerException	Parameter value is null where prohibited
IndexOutOfBoundsException	Index parameter value is out of range
ConcurrentModificationException	Concurrent modification of an object has been detected where it is prohibited
UnsupportedOperationException	Object does not support method

Throw exceptions that are appropriate to the abstraction

Higher layers should catch lower level exceptions and throw exceptions that can be explained at their level of abstraction.
While doing so, don't forget to use chaining in order to provide the underlying cause for failure.

Document thrown exceptions

Document every exceptions that can be thrown by your methods, checked or unchecked. This documentation should be done by using the @throws tag.
Nonetheless, only checked exceptions must be declared as thrown in your code.

Include failure capture information in detail messages

The detailed message of an exception should contain the values of all parameters that lead to such failure.

Example :

```
public IndexOutOfBoundsException(int lowerBound, int upperBound, int index) {
        super(String.format("Lower bound : %d, Upper bound : %d, Index : %d", lowerB

        //Save for programmatic access
        this.lowerBound = lowerBound;
        this.upperBound = upperBound;
        this.index = index;
}
```

Failure atomicity

A failed method invocation should leave the object in the state that it was before the invocation.

Don’t ignore exceptions

An empty catch block defeats the purpose of exception which is to force you to handle exceptional conditions.
When you decide with *very* good reasons to ignore an exception the catch block should contain a comment explaining those reasons and the variable should be named ignored.

Concurrency

Synchronize access to shared mutable data

Synchronization is not guaranteed to work unless both read and write operations are synchronized.
When multiple threads share mutable data, each of them that reads or writes this data must perform synchronization.

Avoid excessive synchronization

As a rule, you should do as little work as possible inside synchronized regions.
When designing a mutable class think about whether or not it should be synchronized.

Executors, tasks and streams

The java.util.concurrent package added a the executor framework. It contains class such as ExecutorService that can help you run Tasks in other threads.
You should refrain from using Threads and now using this framework in order to parallelize computation when needed.

Prefer concurrency utilities to wait and notify

Using wait and notify is quite difficult. You should then use the higher level concurrency utilities such as the Executor Framework, concurrent collections and synchronizers.

- Common concurrent collections : ConcurrentHashMap, BlockingQueue
- Common synchronizers : CountdownLatch, Semaphore

Document thread safety

Every class should document its thread safety. When writing and unconditionnally thread safe class, consider using a private lock object instead of synchronized methods. This will give you more flexibility.

Example :

```
// Private lock object idiom - thwarts denial-of-service attack
private final Object lock = new Object();

public void foo() {
        synchronized(lock) {
                ...
        }
}
```

Lazy initialization

In the context of concurrency, lazy initialization is tricky. Therefore, normal initialization is preferable to lazy initialization.

On a static field you can use the lazy initialization holder class idiom :

```
// Lazy initialization holder class idiom for static fields
private static class FieldHolder {
        static final FieldType field = computeFieldValue();
}
static FieldType getField() { return FieldHolder.field; }
```

On an instance field you can use the double-check idiom :

```
// Double-check idiom for lazy initialization of instance fields
private volatile FieldType field;
FieldType getField() {
        FieldType result = field;
        if (result == null) { // First check (no locking)
                synchronized(this) {
                        result = field;
                        if (result == null) // Second check (with locking)
                                field = result = computeFieldValue();
                }
        }
        return result;
}
```

Don’t depend on the thread scheduler

The best way to write a robust and responsive program is to ensure that the average number of *runnable* threads is not significantly greater than the number of processors.
Thread priorities are among the least portable features of Java.

Serialization

Prefer alternatives to Java serialization

Serialization is dangerous and should be avoided. Alternatives such as JSON should be used. If working with serialization, try not deserialize untrusted data. If you have no other choice, use object deserialization filtering.

Implement *Serializable* with great caution

Unless a class will only be used in a protected environment where versions will never have to interoperate and servers will never be exposed to untrusted data, implementing Serializable should be decided with great care.

Custom serialized form

Use the default serialized form only if it’s a reasonable description of the logical state of the object. Otherwise, write your own implementation in order to only have its logical state.

Write readObject methods defensively

When writing a readObject method, keep in mind that you are writing a public constructor and it must produce a valid instance regardless of the stream it is given.

For instance control, prefer enum types to readResolve

When you need instance control (such a Singleton) use enum types whenever possible.

Serialization proxies

The serialization proxy pattern is probably the easiest way to robustly serialize objects if those objects can't be extendable or does not contain circularities.

```
// Serialization proxy for Period class
private static class SerializationProxy implements Serializable {
        private final Date start;
        private final Date end;

        SerializationProxy(Period p) {
                this.start = p.start;
                this.end = p.end;
        }

        private static final long serialVersionUID = 234098243823485285L; // Any num
}

// writeReplace method for the serialization proxy pattern
private Object writeReplace() {
        return new SerializationProxy(this);
}

// readObject method for the serialization proxy pattern
private void readObject(ObjectInputStream stream) throws InvalidObjectException {
        throw new InvalidObjectException("Proxy required");
}

// readResolve method for Period.SerializationProxy
private Object readResolve() {
        return new Period(start, end); // Uses public constructor
}
```