Effective Java Checklist

- ❐ Simplicity is favored over complexity.
- ❐ Objects that support cloning use copy constructors rather than serialization.
- ❐ Objects that can't be extended and initialized have a private constructor.
- ❐ Creation of new objects is avoided as much as possible.
- ❐ Primitives are preferred over boxed primitives, and auto-boxing is avoided.
- ❐ When a class manages its own memory, obsolete object references are eliminated/nulled out when appropriate, as in the case of array elements which are no longer referenced by the class.
- ❐ Static factory methods are used when there aren't many parameters being passed in on class instantiation.
- ❐ Builder constructor pattern is used in instances when many parameters are used on class instantiation.
- ❐ Explicit termination methods are typically used in combination with the try-finally construct to ensure termination. Finalizers aren't used except as a safety net or to terminate noncritical native resources.
- ❐ If a class does not have a super-class which implements equals, and has a notion of logical equality which differs from the Object identity, the equals method is overridden.
- ❐ All classes which override equals have an equals method which implements an equivalence relations that follow: reflexive, symmetric, transitive, consistent, and non-nullity.
- ❐ In general, when creating an equals method:
    - ❐ Use the == operator to check if the argument is a reference to this object.
    - ❐ Use the instanceof operator to check if the argument has the correct type.
    - ❐ Cast the argument to the correct type.
    - ❐ For each “significant” field in the class, check if that field of the argument matches the corresponding field of this object.
    - ❐ Ask yourself three questions: Is it symmetric? Is it transitive? Is it consistent?
- ❐ All object that override equals override hashCode.
- ❐ All equal object have equal hash codes.
- ❐ All unequal objects tend to produce unequal hash codes.
- ❐ Significant components of the object are included as part of the computation for the hash code.
- ❐ If a class is immutable and the cost of computing the hash code is significant, the hash code is cached (lazily initialized) in the object.
- ❐ All classes have a toString method which returns all of the interesting information contained in the object.
- ❐ All information returned by toString can all be programmatically accessed.
- ❐ The implementation never makes the client do anything the library can do for the client.
- ❐ The clone method is never invoked or over-ridden.
- ❐ Object copying is made available through a copy constructor or copy factory rather than a clone method.
- ❐ Classes that implement Cloneable:
    - ❐ Override clone with a public method whose return type is the class itself.

    - ❒ The method first calls super.clone and fixes any fields that need to be fixed. This means copying any mutable objects that comprise the internal “deep structure” of the object being cloned, and replacing the clone’s references to these objects with references to the copies.
- ❒ If a classes instances have a natural ordering, the Comparable interface is implemented.
- ❒ If the comparable interface is implemented, the compareTo method follows the same restrictions/principles as the equals method: reflexivity, symmetry, and transitivity.
- ❒ The accessibility of classes and members are minimized.
    - ❒ For public classes, all instance variables are private.
    - ❒ Public classes always use accessor methods to access fields.
    - ❒ Modules hide the internal implementation details and data from other modules as much as possible.
    - ❒ Objects referenced by public static final fields are immutable.
- ❒ The mutability of all instance information within a class is minimized. Classes should be immutable unless there’s a very good reason to make them mutable. To make a class immutable:
    - ❒ Don’t provide any methods that modify the object’s state.
    - ❒ Ensure that the class can’t be extended.
    - ❒ Make all fields final.
    - ❒ Make all fields private.
- ❒ Ensure exclusive access to any mutable components.
- ❒ If a class has mutable components that it gets from or returns to its clients, the class defensively copies these components.
- ❒ If the cost of the copy is prohibitive and the class trusts its clients not to modify the components inappropriately, documentation is provided outlining the client’s responsibility not to modify the affected components.
- ❒ Composition if favored over inheritance.
- ❒ All classes that allow inheritance are documented for inheritance and tested.
- ❒ When you do have to use inheritance, ensure that:
    - ❒ The subclass IS A super class.
    - ❒ Constructors do not invoke overridable methods, directly or indirectly.
    - ❒ Neither clone() nor readObject() invoke an overridable method, directly or indirectly.
- ❒ The use of interfaces are favored over the use of abstract classes.
- ❒ All non-trivial interfaces have a skeletal implementations.
- ❒ All public interfaces are designed with the utmost care and tested through multiple implementations.
- ❒ Interfaces are only used to define types – never used to export constants.
- ❒ There are no classes with tag fields; all classes with tag fields are refactored into a hierarchy.
- ❒ The primary use of function pointers is to implement the Strategy pattern. To implement this pattern, an interface is used to represent the strategy, and a class that implements this interface exists for each concrete strategy.
- ❒ When a concrete strategy is used only once, it is typically declared and instantiated as an anonymous class.

❒ When a concrete strategy is designed for repeated use, it is generally implemented as a private static member class and exported in a public static final field whose type is the strategy interface.
❒ If a nested class needs to be visible outside of a single method or is too long to fit comfortably inside a method, a member class is used.
❒ If each instance of the member class needs a reference to its enclosing instance, it is non-static; otherwise, it is static.
❒ All unchecked warnings are eliminated.
❒ If there are warnings that cannot be eliminated and it can be proved that the code that provoked the warning is typesafe, the warning is suppressed with an @SuppressWarnings("unchecked") annotation.
❒ All @SuppressWarnings("unchecked") annotations are combined with comments explaining why it is safe to suppress the warning.
❒ Lists are preferred over arrays.
❒ Generics are used as much as possible; use of raw types is avoided.
❒ Generic methods are used as much as possible.
❒ Wildcard types are used on input parameters that represent producers (extend) or consumers (super) in order to maximize flexibility.
❒ Comparable<? super T> is preferred rather than Comparable<T>.
❒ Comparator<? super T> is preferred to Comparator<T>.
❒ Enum types are used for all fixed set of constants (rather than integers).
❒ For enum types, constant specific methods are used rather than switch statements on their own values.
❒ For enum types, the strategy enum pattern is used for constants that share common behaviors.
❒ Ordinal values are never derived from an Enum type, rather, instance fields are used.
❒ EnumSets are used instead of bit fields.
❒ EnumMaps are used instead of ordinal indexing.
❒ Annotations are used instead of naming patterns.
❒ The Override annotation is used on every method declaration which overrides a superclass declaration.
❒ Marker interfaces are used to define types.
❒ The code makes sure that errors are detected as soon as they occur:
    - ❒ For public methods, the validity of method parameters are being checked at the beginning of the method declaration, and appropriate exceptions are thrown if any invalid values are passed in.
    - ❒ Thrown exceptions are documented in the javadoc comments.
    - ❒ Non-public methods check their parameters using assertions.
    - ❒ The method signatures are well designed:

❒ The method names are meaningful (describe what the method does) and follow the same package method naming conventions it is declared in.
❒ There aren't too many convenience methods: every method “pulls its own weight” and there aren't too many methods used throughout the class.
❒ Long parameter lists are avoided: all methods contain fewer than 5 parameters.
❒ There aren't any long sequences of identically typed parameters.

❒ Any methods which require lots of parameters use static helper classes or a builder pattern to reduce the parameter list.
❒ For parameter types, interfaces are favored over static classes.
❒ For parameter types, two element enum types are preferred over boolean values.
❒ There are no confusing overloaded methods: no two overloaded methods have the same number of parameters.
❒ The use of varargs (variable number of arguments) is avoided.
❒ Array or collection valued methods do not return null values; in place of null, an empty array or collection is returned.
❒ All methods are well documented.
❒ Local variables are always initialized.
❒ Local variables always have minimum scope declarations.
❒ For loops are preferred over while loops.
❒ All methods are small and focused. A method should only do one thing.
❒ For-each loops are preferred over for loops.
❒ Built in libraries are used whenever possible rather than re-inventing the wheel.
❒ The use of float and double are avoided when exact values are needed or for monetary calculations; instead, the BigDecimal type is used.
❒ The use of primitives is preferred over boxed primitives.
❒ Strings aren't used to represent objects that other data types could represent (ex. Enums)
❒ String concatenation is avoided; instead, the StringBuilder class is used.
❒ Objects are referenced by their interfaces, or least specific classes rather then their class when possible (to maximize flexibility).
❒ Objects are not accessed reflectively; interfaces are preferred.
❒ Native methods are used judiciously.
❒ The code is optimized judiciously; it is better to write good programs rather than fast ones.
❒ Good algorithms are chosen to implement the needed steps/tasks.
❒ Standard naming conventions are used.
❒ Exceptions are used only for exceptional conditions.
❒ Checked exceptions are used for recoverable conditions; runtime exceptions for programming errors.
❒ Java's standard exceptions are used when possible.
❒ If it isn’t feasible to prevent or to handle exceptions from lower layers, exception translation is used.
❒ When appropriate, exception chaining is used to throw an appropriate higher-level exception, while capturing the underlying cause for failure analysis.
❒ All exceptions are documented using @throws tag.
❒ All implemented exceptions have a toString method which capture the failure and show a detailed message which contains the values of all parameters and fields which contributed to the exception.
❒ A failed method invocation leaves all objects in a state that they were in prior to the invocation when possible (failure atomic).
❒ There are no empty catch blocks in the code. In instances where there are – there are comments explaining why the exception is ignored.
❒ When multiple threads share mutable data, each thread that reads or writes the data

performs synchronization.

- ❐ The volatile modifier is used for inter-thread communication not involving mutual exclusion.
- ❐ Alien methods are never called from within synchronized regions.
- ❐ As little work as possible is done inside of synchronized regions.
- ❐ Mutable classes are synchronized only if there is a good reason to do so.
- ❐ Executors and tasks are preferred to threads.
- ❐ Java.util.concurrent is used instead of wait and notify.
- ❐ If there is code using wait and notify, wait is always invoked within a while look, and notifyAll is used in preference to notify.
- ❐ Every class documents its thread safety properties with a carefully worded description or a thread safety annotation.
- ❐ Normal initialization is preferable to lazy initialization.
- ❐ No parts of the program depend on the thread scheduler.
- ❐ The average number of runnable threads is not significantly greater than the number of processors.
- ❐ Threads do not busy wait – they almost always perform useful work.
- ❐ Avoid the use of Thread.yield – use Thread.sleep(1) instead.
- ❐ Use thread priorities sparingly.
- ❐ The use of thread groups is avoided – thread pool executors are used instead.
- ❐ Classes are rarely made serializable, and if they are, they are handled with care – especially if the class is designed for inheritance.
- ❐ Default serialized form is only used if it is a reasonable description of the logical state of the object; otherwise, a customized serialized form is used which aptly describes the object.
- ❐ readObject methods are written defensively.
- ❐ Enum types are used to enforce instance control invariants whenever possible.
- ❐ On classes that are not extendable by its clients, the serialization proxy pattern is used whenever possible.