

Devender's Effective Java Reference Sheet

NOTE: All of these tips are from the book *Effective Java (Second Edition)* which is a great book and highly recommended, this sheet is meant to be a quick reference for those who have already read the book.

• Creating and Destroying Objects

- **Consider static factory methods instead of constructors**
 - Ex: `public static Boolean valueOf(boolean b) .`
 - Advantages: They have meaningful names, not required to create a new object each time they are invoked, instance controlled, can return any subtype, forms basis of service provider framework (such as the JDBC api).
- **Consider a builder (builder pattern) when faced with many constructor parameters:** use a static member builder class call setter like methods on the builder and finally call build.
 - Ex `NutritionFacts coke = new NutritionFacts.Builder(240).calories(100).sodium(35).build();`
- **Enforce the singleton property with private constructor or an enum type** :NOTE: if serializing use the readResolve method or simply use Enum .
 - A single Element Enum type is the best way to implement a Singleton.
- **Enforce non-instantiability with a private constructor.**
- **Avoid creating unnecessary objects:** An object can always be reused if it is immutable.
 - Ex: Do not do this **BAD** `String s = new String("string");` Do this instead `String s = "string";`
 - Prefer primitives to boxed primitives and watch out for unintentional auto boxing.
- **Eliminate obsolete object references**
 - Nulling out object references should be an exception instead of a norm, preferred way is to let reference fall out of scope.
 - Common source of memory leaks, whenever a class manages its own memory, caches, listeners and other callbacks.
- **Avoid finalizers**
 - Severe performance penalty for using finalizers.
 - No guarantee that it will actually run.

• Methods Common to all Objects

- **Obey the general contract when overriding equals**
 - The equals contract : Should be reflexive (`x.equals(x) == true`), symmetric (`x.equals(y) == y.equals(x)`), transitive (`x == y, y == z` then `x == z`), consistent and when `x != null` `x.equals(null) == false`.

- When implementing equals instead of testing `o.getClass == this.getClass` (since it will violate the Liskov substitution principle) use `o instanceof MyClass`.
- There is no way you can extend a class with a value component while still preserving the equals contract, *favor composition over inheritance*.
- Use `Float.compare` and `Double.compare` when comparing floats and doubles.
- Don't overload `equals(Object o)` with `equals(MyClass c)`.
- **Always override hashCode when you override equals**
 - Whenever invoked in the same run, must return the same integer provided no information used in equals is modified.
 - If two objects are equal according to equals then they must return the same hashCode.
 - It is not required that if two object are unequal according to the equals method they should produce different hashcodes, but it will improve the performance in hash tables to do so.
 - There are library classes that depend on this principle so by breaking it you will break the functionality of the library classes.
- **Always override toString**
- **Override clone judiciously**
 - Never make the client do anything the library can do for the client.
 - Too complex, better off providing an alternative means of object copying.
 - Consider using a copy constructor.
- **Consider implementing Comparable**
 - Classes that depend on comparison include `TreeSet` and `TreeMap` and the utility classes `Collections` and `Arrays`.
 - Use `Float.compare` or `Double.compare` when working with floats and doubles.

• Classes and Interfaces

- **Minimize the accessibility of classes and members**
 - Make each class or member as inaccessible as possible.
 - Top level Classes and interfaces have only two possible levels package-private and public.
 - Members can be private, package-private (default) ,protected,public.
 - Instance fields should never be public.
 - If you have a public static final collection of things, try using the `Collections.unmodifiable` .
 - Do not have a public static final array of things (cause it is not immutable).In general make sure public static final fields are immutable.

- **In public classes, use accessor methods, not public fields**
 - Exception to this rule is, if a class is package private or is a private nested class, you could expose its data fields.
- **Minimize mutability**
 - An immutable class is simply a class whose instances cannot be modified, they are less prone to error and more secure.
 - To make 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.
 - Take a more functional approach (return a new instance of fields instead of modifying) when designing methods.
 - Could also achieve immutable instance by using a private or package private constructor with static factories.
 - Immutable objects are inherently thread-safe.
 - Classes should be immutable unless there's a very good reason to make them mutable.
 - Make every field final, unless there is a compelling reason not to.
- **Favor composition over inheritance**
 - Inheritance violates encapsulation, subclass depend on internals of super class. Use composition and forwarding instead.
 - Don't extend classes that were not meant to be extended.
- **Design and document for inheritance or else prohibit it**
 - You must test your class by writing subclasses before you release it, this will make it obvious what methods need to be private and protected.
 - Designing a class for inheritance places substantial limitation on the class, such as constructors must not invoke overridable methods, if `Cloneable` or `Serializable` interfaces are implemented `clone` and `readObject` methods may not invoke overridable method and have to make `readResolve` and `writeReplace` methods `protected` else subclasses will ignore them.
- **Prefer interface to abstract classes**
 - Existing classes can easily be retrofitted to implement a new interface.
 - Interfaces are ideal for defining mixins (a type a class implements in addition to its primary type, to declare that it provides some optional behavior).
 - Interfaces allow the construction of non hierarchical type frameworks.

- Consider providing an Interface in combination with a skeletal implementation, a pre-existing class may either extend the skeletal implementation or just forward invocations to it.
- **Use interfaces only to define types.**
- **Prefer class hierarchies to tagged classes.**
 - Instead of having the same class behave different based on some field being set (leads to lot of `if else` statements in the code), use two different classes and choose one at runtime.
- **Use function objects to represent strategies**
 - Though Java does not provide function pointers or lambda expressions, a similar effect can be achieved by defining an object whose methods perform operations on other objects passed explicitly to the methods.
 - Primary use of function pointers is to implement the strategy pattern (allows the switching of strategy/algorithm at anytime ex: descending order, ascending order).
- **Favor static member classes over non static**
 - A nested class should only exist to serve its enclosing class.

• Generics

- **Don't use raw types in new code**
 - `List<String>` is a sub type of the `List` but not of the parameterized type `List<Object>`.
 - If you want to use a generic type but don't know or care what the actual type parameter is you can use a question mark instead ex:
`Set<?>`
 - You cannot put any element other than null into a `Collection<?>`
 - You must use raw types in class literals.
- **Eliminate unchecked warnings**
 - Eliminate every unchecked exception that you can and only after you prove that the code that provoked warning is type-safe suppress the warning with a `@SuppressWarnings("unchecked")` annotation.
- **Prefer lists to arrays.**
 - `Long[]` is a subtype of `Object[]`, arrays are covariant where as `List` are invariant for any two types `Type1` and `Type2` `List<Type1>` is neither subtype nor super type of `List<Type2>`.
 - Arrays enforce their type at runtime, generics enforce their type only at compile time.

- Arrays and generics don't mix well since they have very different type rules.
- **Favor generic types**
 - Generic types are safer and easier to use than types that require casts in client code. When you design new types make sure that they can be used without casts.
- **Favor generic methods**
 - You can exploit the type inference provided by generic method invocation to ease the process of creating parameterized type instances.
 - ex: Instead of writing `Map<String, List<String>>`
`something = new HashMap<String, List<String>>();` you could provide a static factory method such as below
 - ```
public static <K, V> HashMap<K, V> newHashMap() {
 return new HashMap<K, V>(); }
```
  - Generic methods, like generic types are safer and easier to use than methods that require clients to cast input parameters and return values.
- **Use bounded wildcards to increase API flexibility.**
  - Simply a token that limits what types can be presented `<?>` is unbounded, `<? super E>` or `<? Extends E>` and `Class<T>` are bounded wild cards.
  - ex: of wild card type for parameter that servers as an E consumer  
`void popAll(Collection<? super E> dst),` accepts collection of some super-type of E (E is supertype of itself).
  - ex: of wild card type for parameter that servers as an E producer  
`public <E> E reduce(List<? extends E> list)`
  - PECS: producer extends, consumer super.
  - Do not use wild car types as return types.
- **Consider type safe heterogeneous containers**
  - if `<T>`, `t.cast` is the dynamic analog of Java's cast operator.
  - You can place type parameter on the key rather than the container, you could use Class objects as key for such a typesafe heterogeneous containers ex: `Map<Class<?>>, Object> favorites = new HashMap<Class<?>, Object>();`

## • Enums and Annotations

- **Use enums instead of int constants**

- Enums are classes that export one instance for each enumeration constant via a public static final field. Clients can neither create instances nor extend it.
- They implement Serializable and Comparable.
- To associate different behaviour with each enum declare an abstract method in the enum type and override in each constant.
- Consider strategy enum pattern (enum with nested enum) when multiple enums constants share common behaviors.
- **Use instance fields instead of ordinals**
  - Ordinal of an enum represents the numerical position of the constant, don't depend on this.
- **Use EnumSet instead of bit fields**
  - Internally each EnumSet is represented as a bit vector, if the underlying enum type has 64 or fewer elements the entire EnumSet is represented with a single long.
- **Use EnumMap instead of ordinal indexing**
  - EnumMap is a very fast Map implementation designed for use with enum keys.
- **Emulate extensible enums with interfaces**
  - While Enum types cannot be extended Enum can implement arbitrary interfaces.
- **Prefer annotations to naming patterns**
- **Consistently use the Override annotation**
- **Use marker interface to define types**
  - A marker interface defines a type, a marker annotation does not, by using marker interface rather than a annotation you get compile time checking and tool support. By making the marker interface extend some other type you can even restrict what types the marker interface can be applied to.
  - If you create an annotation that is only applied to Types, take time to think if it should be an interface instead.

## • Methods

- **Check parameters for validity**

- It is best to fail fast and harder to detect the error in the midst of processing.
- **Make defensive copies when needed**
  - When creating immutable types, it is essential to make defensive copy of each mutable parameter to the constructor and return defensive copies of mutable internal fields.
  - Do not use the clone method to make defensive copy of a parameter whose type is subclassable by untrusted parties.
  - Where ever possible try to use immutable objects.
- **Design method signature carefully**
  - Choose method names carefully.
  - Too many methods make a class difficult to learn, use, document and maintain. WHEN IN DOUBT, LEAVE IT OUT.
  - Avoid long parameter lists. Techniques for shortening
    - Break it up into multiple methods.
    - Create a helper class to hold groups of parameters.
    - Builder pattern
  - For parameter type, favor interfaces over classes
  - Prefer two element enum types to boolean parameters.
- **Use overloading judiciously**
  - Choice of which overloading method to invoke is made at compile time and the choice of which overriding method is done at runtime.
  - Never export two overloadings with the same number of parameters.
- **User varargs judiciously**
  - If you are always expecting at least on parameter use this `method(int first, int... remaining)` instead of `method(int...array)` and checking length.
  - Every invocation of varargs will cause creation of a new array.
- **Return empty arrays or collections, not nulls**
  - There is no reason ever to return null from an array or collection valued method instead of returning an empty array or collection.
- **Write doc comments for all exposed API elements**
  - New doc tags added in 1.5 are `{@literal}` and `{@code}`

## • General Programming

- **Minimize the scope of local variables**
- **Prefer for-each loops to traditional for loops**
  - Situations where you can't use for-each loop
    - Filtering, when you need to traverse and remove
    - Transforming, traverse and replace
    - Parallel iteration, if you need to traverse multiple collection in parallel
- **Know and use the libraries**
  - Should be familiar with the contents of `java.lang` and `java.util`
  - If you need to do something that seems like it should be reasonably common, there may already be a class in the libraries that does it.
- **Avoid float and double if exact answers are required**
  - Use `BigDecimal`, it gives you full control over rounding.
- **Prefer primitive types to boxed primitives**
  - If you must use boxed primitives do it with caution it could result in performance degradation and errors, such as `==` on boxed primitives checks for identity of objects and not equality and unboxing can throw a `NullPointerException` if it is not initialized.
- **Avoid strings when other types are more appropriate**
  - Strings are poor substitutes for other values types
  - Strings are poor substitutes for enum types.
  - Strings are poor substitutes for aggregate types.
- **Beware the performance of string concatenation**
  - Using string concatenation operator repeatedly to concatenate `n` strings requires time quadratic in `n` use `StringBuilder` instead.
- **Refer to objects by there interfaces**
- **Prefer interfaces to reflection**
  - Objects should not be accessed reflectively in normal applications at runtime. Reflection uses verbose code and does not perform well.
- **Use native methods judiciously**
- **Optimize judiciously**

- Strive to avoid design decisions that limit performance. Use interfaces break system down into modules so that performance changes in module does not effect others.
- Consider the performance consequences of your API design decisions.
- Examine your choice of algorithms no amount of low level optimization can make up for a poor choice of algorithm.
- **Adhere to generally accepted naming conventions**
  - Such as `edu.institutename.` or `com.companyname.` Names of class `MyClassName` and so on.

## • Exceptions

- **Use exceptions only for exceptional conditions**
  - Placing code inside a try-catch block inhibits certain optimizations that modern JVM implementations might otherwise perform.
  - Should never be used for ordinary control flow.
  - A well designed API must not force its clients to use exceptions for ordinary control flow.
- **Use checked exceptions for recoverable conditions and runtime exception for programming errors.**
  - Since checked exceptions are recoverable it is especially important for such exceptions to provide methods that furnish information that could help the caller to recover.
  - Parsing the string representation of exceptions to get additional information is an extremely bad practice.
- **Avoid unnecessary use of checked exceptions**
- **Favor the use of standard exceptions**
- **Throw exceptions appropriate to the abstraction**
  - Higher layers should catch lower level exceptions and in their place throw exceptions that can be explained in terms of the higher level abstraction, this is also know as exception translation
  - If higher layers can debug/recover then exception chaining should be used.
- **Document all exceptions thrown by each method**
- **Include failure capture information in detail messages**

- The detail message of an exception should contain the values of all parameters and fields that contributed to the exception, length prose descriptions are generally superfluous.
- You can add your own constructors to the exception that takes all the values that contributed to the exception, rather than just implementing the default constructor.
- **Strive for failure atomicity**
  - A failed method invocation should leave the object in the state that it was in prior to the invocation.
- **Don't ignore exceptions**
- **Concurrency**
  - **Synchronize access to shared mutable data**
    - Reading and writing a variable is atomic unless the variable is of type `long` or `double`, synchronization is still required when accessing shared variables.
    - Synchronization is required for reliable communication between threads as well as for mutual exclusion.
    - Synchronization has no effect unless both read and write operations are synchronized.
    - Do not use `Thread.stop`
    - `++` is not an atomic operator.
    - Confine mutable data to a single thread.
    - A `safety failure` is when a program computes the wrong results due to accessing unprotected shared data.
    - A `liveness failure`, is when a program fails to make progress.
  - **Avoid excessive synchronization**
    - Excessive synchronization can cause reduced performance, deadlock and even non deterministic behavior.
    - To avoid liveness and safety failures, never cede control to the client within a synchronized method or block.
    - Do as little work as possible inside synchronized regions.
    - Over Synchronization limits VM's ability to optimize.
    - Make mutable classes thread safe if it is intended for concurrent use.

- **Prefer executors and tasks to threads**
  - The executor framework is a flexible interface based task execution facility. With the executor framework you can wait for a particular task to complete.
  - `java.util.concurrent.Executors` class contains static factories that provide most of the executors you'll ever need.
  - You should generally refrain from working directly with threads. The key abstraction is no longer `Thread`, which served as both the unit of work and the mechanism for executing it. Now the unit of work is called task.
  - There are two kinds of tasks `Runnable` and `Callable` with returns a value.
- **Prefer concurrency utilities to wait and notify**
  - Given the difficulty of using `wait` and `notify` correctly, you should use the higher level concurrency utilities instead.
  - The higher level concurrency utilities provided in `java.util.concurrent` fall into three categories
    - The Executor Framework
    - Concurrent Collections : provides high performance concurrent implementations of standard collection interfaces such as `List`, `Queue` and `Map`
    - Synchronizers : Objects that enable threads to wait for one another, allowing them to coordinate their activities ex `CountDownLatch` and `Semaphore`
  - For interval timing, always use `System.nanoTime` in preference to `System.currentTimeMillis`
  - Always use the wait loop idiom to invoke the `wait` method; never invoke it outside the of a loop.
- **Document thread safety**
  - Document is a class is immutable/unconditionally thread safe/conditionally thread safe/not thread safe/thread hostile.
- **Use lazy initialization judiciously**
  - Under most circumstances, normal initialization is preferable to lazy initialization.

- if lazy initialization is used, use the synchronized accessor.
- if needed on a static field, use the lazy initialization holder class idiom.
- if needed to use on an instance field, use the double check idiom, prior to the 1.5 the double check did not work reliably, but with the memory model introduced in 1.5 fixed this problem.
- **Don't depend on the thread scheduler**
  - Any program that depends on thread scheduler correctness or performance is likely to be non portable.
  - Ensure that the average number of runnable threads is not significantly greater than the number of processors.
  - `Thread.yield` has no testable semantics.
  - Thread priorities are among the least portable features of the java platform.
- **Avoid thread groups**
- **Serialization**
  - **Implement `Serializable` judiciously**
    - It decreases the flexibility to change a class's implementation once it has been released.
    - It is possible to change the internal representation while maintaining the original serialized form using `ObjectOutputStream.putFields` and `ObjectInputStream.readFields`.
    - Increases the likelihood of bugs and security holes.
    - Classes designed for inheritance should rarely implement `Serializable` and interfaces should rarely extend it.
    - Inner classes should not implement `Serializable`, a static member class can.
  - **Consider using a custom serialized form**
    - Do not accept the default serialized form without first considering whether it is appropriate.
  - **Write `readObject` methods defensively**
  - **For instance control, prefer enum types to `readResolve`**
  - **Consider serialization proxies instead of serialized instances**