**Effective Java Summary – 2022**

**Item 1: Consider static factory methods instead of Constructors**

* Static factory methods have more informative names than constructors
* Not required to create new objects, could return cached instance or preconstructed instance.
* Static factory methods could return object subtype.
* One advantage is controlling access to a limited resource such as connections. This a way to implement [pools of reusable objects](http://en.wikipedia.org/wiki/Object_pool) - instead of building, using, and tearing down an object.
* Static factory can choose from many subclasses (or implementers of an interface) and return that

The **Boolean.valueOf(boolean)** method illustrates this technique: it never creates an object. This technique is similar to the Flyweight patten.

**public static Boolean valueOf(boolean b) {**

**return b ? Boolean.TRUE : Boolean.FALSE;**

**}**

The difference between static factory method and Factory Design is that in case Factory Design you will get a new object at runtime. Using such static factories causes clients to share instances instead of creating new ones

Optional<String> value1 = Optional.empty(); Optional<String> value2 = Optional.of("Baeldung"); Optional<String> value3 = Optional.ofNullable(null);

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

* Builder Pattern is flexible and right way to handle optional parameters

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

* Throw an exception in a private constructor to avoid re㓐ection call to constructor
* If standard serialization is needed make all ᜀelds transient and override readResolve method
* Best way to use single element enum as a singleton

Item 5: Avoid creating unnecessary objects

* "hello" is better than new String("hello")
* Boolean.valueOf("true") is better than new Boolean("true")
* Immutable objects could be reused for free

A good case for using the 'new String(String)' constructor is when you don't want to hold a reference to a literal string. When you declare a string in the following form... **String s = "a string";** "a string" will be cached in permgen space.

**Item 6: Eliminate obsolete object references**

* Garbage collector is not savior from memory leaks
* Nullify obsolete references
* Invalidate cache periodically
* Deregister outdated listeners and callbacks

**Item 7: Avoid finalizers**

* Finalizers are not destructors
* No guarantee finalizers will be executed promptly
* No guarantee ᜀ finalizers will be executed at all
* System.gc just a hint, not a gc call
* Finalizers cause severe performance penalty
* Use own explicit methods for finalization like close()

**Item 11: Override clone judiciously**

* If you override the clone method in a normal class, you should return an object

obtained by invoking super.clone.

* Provide copy constructor or copy factory instead of implementing clone

**Item 12: Consider implementing Comparable**

* Implementing Comparable indicates that objects have natural ordering
* Implement compareTo that returns ‐1, 0 and 1 and do not cause integer overflow

**Item 18: Prefer interfaces to abstract classes**

* Interfaces enable safe, powerful functionality enhancements via wrapper classes
* Abstract classes are useful for skeletal implementation
* You could safely add a method to abstract class with default implementation (the same applies to interfaces since Java 8 release, with help of default methods)
* Once an interface is released and widely implemented, it is almost impossible to change

**Item 25: Prefer lists to arrays**

* Arrays are covariant
* Generics are invariant (List<Sub> is not a subtype of List<Super>)
* Arrays are refiᜀed (enforce their element types at runtime)

**Item 28: Use bounded wildcards to increase API flexibility**

* Generics are invariant (List<Integer> is not a subtype of List<Number>)
* PECS: Producer - Extends , Consumer - Super
* Producer: add(List<? extends Number>)
* Consumer: get(List<? super Number>)
* Comparable and Comparator are consumers

**Item 31: Use instance ᜀelds instead of ordinals**

* Every enum constant associated with int value via ordinal() method
* Reordering, adding or deleting enum constant cause problems if you depend on
* ordinal()
* Use instance ᜀelds for enum (APPLE(1) instead of APPLE.ordinal())

**Item 37: Use marker interfaces to define types**

* Marker interfaces deᜀne a type that is implemented by instances of the marked class; marker annotations do not

**Item 41: Use overloading judiciously**

* A safe, conservative policy is never to export two overloadings with the same number of Parameters

**Item 43: Return empty arrays or collections, not nulls**

* Do not return nulls!
* Return empty collection (Collections.emptyList()), or zero-length array (new int[0]) instead of nulls

**Item 46: Prefer for-each loops to traditional for loops**

* foreach introduces no performance penalty

**Item 48: Avoid float and double if exact answers are required**

* Use BigDecimal, int, or long for monetary calculations

**Item 49: Prefer primitive types to boxed primitives**

* Boxed primitive may have null value
* Primitives more time and space-efficient than boxed primitives
* Applying the == operator to boxed primitives is almost always wrong

**Item 51: Beware the performance of string concatenation**

* Using the string concatenation operator repeatedly to concatenate n strings requires O(n^2) time
* To achieve acceptable performance, use a StringBuilder instead

**Item 57: Use exceptions only for exceptional conditions**

* Exceptions slower than normal checks
* Placing code inside a try-catch block inhibits certain optimizations that modern JVM implementations might otherwise perform
* A well-designed API must not force its clients to use exceptions for ordinary control flow

**Item 58: Use checked exceptions for recoverable conditions and runtime exceptions for programming errors**

**Item 67: Avoid excessive synchronization**

* Java locks are reentrant
* CopyOnWriteArrayList is a variant of ArrayList in which all write operations are

implemented by making a fresh copy of the entire underlying array

**Item 68: Prefer executors and tasks to threads**

* Thread is no longer a key abstraction, use Runnable or Callable
* Executors.newCachedThreadPool good choice for lightly-loaded server, if no threads available for submitted task, new one will be created
* Executors.newFixedThreadPool(n) good choice for heavily-loaded server, Only n

threads will be created

**Item 69: Prefer concurrency utilities to wait and notify**

* ConcurrentHashMap is optimized for retrieval operations
* Use ConcurrentHashMap in preference to Collections.synchronizedMap or Hashtable
* BlockingQueue used for producer-consumer queues
* CountdownLatch is a single-use barrier that allow one or more threads to wait for one or more other threads to do something
* For interval timing, always use System.nanoTime in preference to System.currentTimeMillis. System.nanoTime is both more accurate and more precise, and it is not a㥖ected by adjustments to the system's real-time clock.

**Item 74: Implement Serializable judiciously**

* Define serialVersionUID to avoid InvalidClassException

**Item 76: Write readObject methods defensively**

* If an entire object graph must be validated a\_ter it is deserialized, use the ObjectInputValidation interface

**Item 77: For instance control, prefer enum types to readResolve**

* To satisfy singleton property for serializable object, implement readResolve
* If you depend on readResolve for instance control, all instance ᜀelds with object reference types must be declared transient readResolve is not obsolete. It is needed for writing a serializable instance-controlled class whose instances are not known at compile time
* Use enum types to enforce instance control invariants wherever possible

**Item 78: Consider serialization proxies instead of serialized Instances**

* Serialization proxy is a private static inner class implements Serializable and reflects serializable data for original object
* Add writeReplace method to proxy class. Serialization system emits a proxy instance instead of an instance of the enclosing class.
* Add readObject method to proxy class. Attacker wouldn't be able to violate class invariants.
* Add readResolve method to proxy class that returns logically equialent instance of the enclosing class.