***Effective Java***

Chapter 1: Introduction

**Chapter 2: Creating and destroying objects**

Item 1: Consider static factory methods instead of constructors.

* Unlike constructors, they have their own names, so that we can different meaningful names for each static factory method to create objects instead of multiple overloaded constructors.
* Static factory methods do not have to create new objects every time, we can control if to create a new object or to return an existing object or modify existing etc.
* Unlike constructors, static factory methods can return an object of any subtype of their return type.
* Con- classes with only static factory methods and no public or protected constructors, will not be able to get subclassed by any other class.
* Naming conventions can be .of(), .getInstance(), getType(), newType(), type().

Item 2: Consider Builder, when faced with many parameters and many constructors’ requirement.

* Builder design pattern is very efficient and useful to maintain readability and reduce complexity for users of the API.
* We can have the mandatory field as parameters of builder (a,b,c….) method and other fields as kind of setters which can be created using chaining of methods.

Item 3: Enforce Singleton property with a private constructor or an Enum type.

Item 4: Enforce non-instantiability using private constructor.

* Trying to make a class abstract will not give us non-instantiatability, the class can be subclassed and instantiated.
* Making private constructors will make it not initializable and not subclassable.

Item 5: Prefer dependency injection to hardwiring resources.

Item 6: Avoid creating unnecessary objects.

* Use static factory methods instead of constructors, so that we can reuse objects.
* Prefer primitives, and avoid unintentional auto-boxing.

Item 7: Eliminate obsolete object references

* Look out for memory leaks, and do nullify the unused objects if needed.

Item 8: Avoid finalizers and cleaners

* Use auto closeable in possible places, to avoid memory leaks.

Item 9: Prefer Try-with-resources to try-finally combo.

* Try-with-resource need or need not have a catch or a finally block, it can just be try(<? Implements AutoCloseable> obj = something){}, if anything fails, the resources will be automatically closed, as it allows objects whish implements Autocloseable interface.

**Chapter 3: Methods common to all objects.**

Item 10: Obey the general contract while overriding equals.

* Equals method should have these properties (RSTC)
  + Reflexive: x.equals(x) should be true.
  + Symmetric: x.equals(y) should be equal to y.equals(x).
  + Transitive: if x.equal(y) is true and y.equals(z) is true, then x.equals(z) should be true.
  + Consistent: multiple invocations of x.equals(y) should return same result every time.
* Recipe for a perfect equals method:
  1. Use == operator to check if the argument is a reference to same object.
  2. Use the instance of operator to check if the argument has the correct type.
  3. Then cast the argument to the correct type.
  4. For each “significant” fields, compare the values.

Sample equals method:

@Override

public boolean equals(Object obj){

if(obj==this)

return true;

if(!(obj instanceof Employee))

return false;

Employee emp = (Employee) obj;

return this.name.equals(emp.name) && this.id==emp.id;

}

Item 11: Always override hashCode() when you override equals().

Item 12: Always override toString(),

* It helps for programmer understanding of the object body while debugging.

Item 13: Override clone() judiciously.

* In practice, a class implementing Cloneable interface is expected to provide a clone() method.
* Immutable classes should never provide a clone() method.

Item 14: Consider implementing comparable interface.

* By providing compareTo(T t) method, we can provide a natural way of sorting of that class objects.

**Chapter 4: Classes and interfaces**

Item 15: Minimize the accessibility of classes and members.

* Make each class or member as inaccessible as possible.
* Instance fields of public classes should rarely be public.
* Classes with mutable fields are not generally thread-safe.
* It is wrong for a class to have a public static final array field.
  + It will not be mutable as expected, the array elements can be modified from the outside.
  + To avoid this, we should have a private static final array and a accessor to provide copy of the array instead of pointing to the same array.

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

Item 17: Minimize mutability.

* To make a class as immutable,
  + Make class as final, to not get subclassed.
  + Make all fields private and final.
  + Don’t provide setter methods(mutators).
  + Provide getter methods(accessors) with defensive copy.
* Immutable objects are simple and thread-safe, so need not require synchronization.
* Immutable objects can be shared freely.
* Immutable objects provide failure atomicity for free.
  + They don’t change state at all, even any process passes or fails.
* Major disadvantage of immutable classes is that they require separate objects created for each distinct value.
  + For example the String class methods, concat(), join(), toUpperCase(), toLowerCase(), etc returns a new String, and not alter the existing String, coz String is immutable.
* Classes should be immutable, unless there is a reason for it to be mutable.
* If a class cannot be made immutable, limit is mutability as much as possible.
* Declare every field private and final, unless you have a reason not to.

Item 18: Favor composition over inheritance.

* Unlike method invocation, inheritance violates encapsulation.

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

* Methods which are available for overriding, should have instructions as comments that how to handle it and whether it has contacts with any other methods.
* Inheriting a class and overriding the methods are dangerous that we never thought of it.
* Constructors must not invoke overridable methods.
* Clone() and readObject() also should not invoke a overridable method directly or indirectly.
* Designing a class for inheritance is much hard and needs great effort.
* Better don’t subclass a class which is not designed and documented to be safely subclassed.

Item 20: Prefer interface to abstract classes.

Item 21: Design interfaces for posterity.

// not sure what this mean, the whole section, I cant understand a good point.