Effective java

# CH-1 Classes and interfaces

*Item 13: Minimize the accessibility of classes and members:*

* The rule of thumb is simple: make each class or member as inaccessible as possible. In other words, use the lowest possible access level consistent with the proper functioning of the software that you are writing.
* It is wrong for a class to have a public static final array field, or an accessor that returns such a field.  
   Example:

// Potential security hole!

public static final Thing[] VALUES = { ... };

//Correct ways

private static final Thing[] PRIVATE\_VALUES = { ... };

public static final List<Thing> VALUES = Collections.unmodifiableList(Arrays.asList(PRIVATE\_VALUES));

private static final Thing[] PRIVATE\_VALUES = { ... };

public static final Thing[] values() { return PRIVATE\_VALUES.clone(); }

* Ensure that objects referenced by public static final fields are immutable.

*Item 14: In public classes, use accessor methods, not public fields*

* + Public classes should never expose mutable fields.
  + However, sometimes desirable for package-private or private nested classes to expose fields, whether mutable or immutable

*Item 15: Minimize mutability*

* Immutable objects are inherently thread-safe; they require no synchronization.
* Apply Immutable rules
* Ref:- <http://javarevisited.blogspot.in/2013/03/how-to-create-immutable-class-object-java-example-tutorial.html>
  + Immutable objects can be shared freely.
  + The only real disadvantage of immutable classes is that they require a separate object for each distinct value.
  + If a class can- not be made immutable, limit its mutability as much as possible.

*Item 16: Favor composition over inheritance*

* + ­One of the disadvantages of Inheritance is that it breaks encapsulation.
  + TODO: Decorator pattern

*Item 17: Design and document for inheritance or else prohibit it*

* + TODO

*Item* 18: Prefer interfaces to abstract classes

* + Classes can implement interfaces easily.
  + Interfaces are ideal for defining mixins. An interface is called a mixin when it allows the optional functionality to be “mixed in” to the type’s primary functionality. Example comparable interface which allows to add the functionality to compare the objects.