EFFECTIVE JAVA

**Creating and Destroying Objects**

  Consider static factory methods instead of constructors

**ADVANTAGES**:

* Unlike constructors, they have names
* They are not required to create a new object each time they’re invoked
* They can return an object of any subtype of their return type
* They reduce the verbosity of creating parameterized type instances

**DISADVANTAGES:**

* They are not readily distinguishable from other static methods.
* Since constructors are made private , Classes without public or protected constructors cannot be subclassed

**Consider a builder when faced with many constructor parameters**

**Scenario**:

*You have plenty of parameters to create an Object.*

* Static factories will have huge parameter function.
* Constructors will lead to telescoping constructors(ie Constructor is declared. Another one is declared with optional paramter and calls the earlier one and ..
* JAVABean (Setter and Getter pattern). Client needs to set each property and there are chances of using Incomplete one. Thread safety also matters

public class NutritionFacts {

    private final int servingSize;

    private final int servings;

    private int calories;

    public static class Builder {

        // Required parameters

        private final int servingSize;

        private final int servings;

        // Optional parameters - initialized to default values

        private int calories = 0;

        public Builder(int servingSize, int servings) {

            this.servingSize = servingSize;

            this.servings = servings;

        }

        public Builder calories(int val)

            { calories = val; return this; }

        public NutritionFacts build() {

            return new NutritionFacts(this);

        }

    }

    private NutritionFacts(Builder builder) {

        servingSize = builder.servingSize;

        servings = builder.servings;

        calories = builder.calories;

        fat = builder.fat;

        sodium = builder.sodium;

        carbohydrate = builder.carbohydrate;

    }

}

**CLIENT CODE :**

NutritionFacts cocaCola = new NutritionFacts.Builder(240, 8).calories(100).build();

**DISADVANTAGES:**

        You need to create builder before object and hence performance is affected

**Singleton**

**Method 1:** public static Singleton INSTANCE = new Singleton();

**Method 2:** Static factory method and return instance . You can maintain a pool of instances .Throw exception

                if Reflection tries to create one using setAccessible

**Method 3**: Serialization ? Provide variable as transient and readResolve should return same instance

**Method 4:** ENUM

              public enum Elvis {

                          INSTANCE;

                          public void leaveTheBuilding() { ... }

                }

**Non-instantiable Constructor**

      Private constructor can have exception as insurance . Used for classes that need not have instances

      Ex: Utility class. Side effect: No inheritance

**Avoid unnecessary object creation**

      Reuse if your object doesn't change. Avoid autoboxing

      String s="siva" is better than String s= new String("siva");

      Use static instances

**Eliminate obsolete object references**

Obsolete -> The objects are of no use but the references are still alive .

Scenarios for occurrence:

* Programs that manage their own memory (Stack pop opn)
* Cache
* Listeners and Callbacks

**Finalizers**

* + Finalizer thread runs with low importance and hence time-critical process should not be defined.
  + No guarantee that finalizer will be called
  + Exceptions are ignored in finalizers
  + Performance affected – time taken to destroy is more

**Solutions**:

Use termination methods explicitly (Example: finally clause)

**Uses**:

Second level back-up to call termination (If called, the client has a bug since it didn't call termination method)

Native objects destruction since garbage collector don’t know about it

Call super.finalize -> since there is no chaining

For each instance call finalizer guardian

// Finalizer Guardian idiom

public class Foo {

// Sole purpose of this object is to finalize outer Foo object

private final Object finalizerGuardian = new Object() {

@Override protected void finalize() throws Throwable {

... // Finalize outer Foo object

}

};

... // Remainder omitted

}