Effective Java

Chapter 2. Creating and destroying objects

Item 1. Consider static factory methods over constructors:

The static factory method is not the same as the Factory Design pattern.

Advantages:

1. **It is easier to name.**

***In cases where a class seems to require multiple constructors with the same signature, replace the constructors with static factory methods and carefully chosen names to highlight their differences.***

1. **No need to create a new instance, when the method is invoked, unlike constructors.**

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

1. **Can return an object of any subtype of their return type.**
2. **Returned object can vary from call to call as a function of the input params.** Can return other implementations. based on different input parameters.

*The EnumSet class has no public constructors, only static factories. In the OpenJDK implementation, they return an instance of one of two subclasses, depending on the size of the underlying enum type: if it has sixty-four or fewer elements, as most enum types do, the static factories return a RegularEnumSet instance, which is backed by a single long; if the enum type has sixty-five or more elements, the factories return a JumboEnumSet instance, backed by a long array.*

1. **The class of the returned object need not exist when the class containing the method is written**.

*The return type can be an interface, the implementation of it can be generated later. The actual implementation can be loaded dynamically*

*Like JDBC API service provider, decouples implementation*

Disadvantages:

1. **Classes without public or protected constructors can be subclassed**

Even though, u provide factory method without proper constructors, you won’t be able to create subclasses of that class.

1. **static factory methods is that they are hard for programmers to find.**

Item 2. Consider using a builder when faced with many constructor params

Case: class has a lot of properties some of them are optional, like for ex. NutritionFacts

1. **Telescoping constructor pattern** suggests to have a lot of overloaded constructors

Disadvantage: It does not look clean

1. **Java Beans patterns** proposes to use setters in that case, but it's not thread-safe and may have inconsistent behavior partway through the constructor, it rejects immutability, although it looks clean
2. **Builder pattern**, allows us to have a nice and clean code, with its safety, good for classes which have multiple varargs, also can aggregate the params passed into multiple calls.

But in performance-critical situations, it can take some time to create the builder, every class should have its own builder

**Covariant return typing –** subclass returns type of subclass

Item 3. Enforce the singleton prop with a private constructor or an enum type

The class which is instantiated only once called singleton

Represents a stateless object such as a function, or system component that is unique

This can be achieved in 2 ways:

1. Public field, but privileged client can invoke private constructor reflectively with *AccessibleObject.setAccessible* method. To defend against this attack, modify the constructor to make it throw an exception
2. Static factory method: pros factory can be generic, method reference can be used as a supplier
3. Enum singleton preferred approach, provides serialization, the best way. But if your singleton should extend a superclass other than Enum, u can’t

To make the class Serializable using 1st 2 approaches, besides implementing a Serializable interface, we need to make all fields transient, and provide readResolve(). Otherwise, when serialized every time, a new instance will be created.

Ways how it can be broken can be found here. <https://www.geeksforgeeks.org/prevent-singleton-pattern-reflection-serialization-cloning/>

**Breaking the singleton pattern in the application can cause DoS of a system, Unauthorized access, inconsistent logging, etc.**

ITEM 4: ENFORCE NONINSTANTIABILITY WITH A PRIVATE CONSTRUCTOR

By making a class abstract, we can not guarantee that a constructor will be created.

so a class can be made noninstantiable by including a private constructor:

// Noninstantiable utility class public class UtilityClass { // Suppress default constructor for noninstantiability private UtilityClass() { throw new AssertionError(); } ... // Remainder omitted }

utility classes were not designed to be instantiated, as they have static fields and methods

*like Arrays, and Collections classes which have private constructors.*

As a side effect, this idiom also prevents the class from being subclassed.

ITEM 5: PREFER DEPENDENCY INJECTION TO HARDWIRING RESOURCES