Java Creational Design Pattern - Plural-Sight

Sunday, 8 July 2018 7:51 PM

Design Pattern are of 3 types

1. Creational Pattern
2. Structural
3. Behavioral

Singleton

Concept

- Single Object created
- Thus guarantees control of resource
- Can be Lazily loaded (if needed)
- Examples:
    - Logger (Can be Factory also)
    - Runtime Environment
    - Spring Beans (By default)
    - Graphics Manager

Design

- Class itself is responsible for its lifecycle
- It is static in nature, although not implemented using a static class
- This is done to make it thread safe as static do not guarantee to make it thread safe.
- Private instance of the same class inside the class
- Private constructor, so no one else can call the constructor but class itself.
- No Patterns required for construction (If required usually a factory pattern)
- A getInstance() method

Code

```
package creational.singleton;

/*
* A thread safe and Lazily loaded Singleton class - faster -
* And saving from Reflection
* THREAD SAFE
* */
public class Singleton4 {
```

```
public class Singleton4 {
private static volatile Singleton4 singleton4 = null;  //fixed in java 5
// volatile ensures unsynchronized method when the instance variable is set
private Singleton4() {
if(singleton4 != null) {
throw new RuntimeException("use getInstance() method"); //Reflection Safe
}
};
//Double locking Mechanism used
// Can synchronize whole block but that is costly in computation
public static Singleton4 getInstance() {
if(singleton4 == null)
synchronized (Singleton4.class) {
if(singleton4 == null) {
singleton4 = new Singleton4();
}
}
return singleton4;
}

}
```

Pitfalls

- Overused - peoplemake every object singleton
- Difficult to unit test
- People might wrongly implement and make thread unsafe
- Sometime confused for factory
- Java.util.Calendar is not singleton but prototype.

Contrast b/w factory and singleton design pattern

Contrast

Singleton

- Returns same instance
  - One constructor method - no args
- No Interface

Factory

- Returns various instances
  - Multiple constructors
- Interface driven
- Adaptable to environment more easily

Builder Pattern

Concept

- Handling construction of objects that contain a lot of parameters and we want to make the object immutable once we are done constructing it. Thus it
    - Handles complex constructors
    - Has Large number of parameters
    - It can **force immutability** once object construction is finished.
- It solves a common problem of which constructor to use.
- Examples:
    - StringBuilder
    - DocumentBuilder
    - Locale.Builder

Design

- Creation of multiple constructors with each parameter variation is called **Telescoping Constructor and is a anti-pattern**.
- It is written with a static inner class. (can be used with a different class also).
- Removes the need for every setter we need to make and removes this anti-pattern of having setter of each object.
- Has flexibilty of Bean Approach of all setters and immutability of telescopic Constructor approach.

Code

https://github.com/rishabh1911/Design-Pattern-Tutorial/tree/master/src/main/java/creational/builder

Pitfalls

- The Builder pattern differ from the Abstract Factory pattern and the Factory Method pattern, in that the intention of those two is to enable polymorphism, while the intention of the builder pattern is to find a solution to the Telescoping Constructor anti-pattern
- Builder is used in case of complex constructors and does not require interface but interface can be used if you want.
- This can be implemented in separate class but typically it is used as inner class.
- Difference with prototype Pattern:
    - Builder method is implemented around constructor to get new object while prototype is implemented near clone method.
    - Prototype is difficult to implement in legacy code as cannot be implemented in other class.
    - Builder works with constructor, prototype avoids them.

Prototype Pattern

- Used when type of object to create is determined by proto-typical instance which is cloned to create a new instance.
- Many times it is used to get a unique instance of same object.
- Used when object is expensive to create but we can get what we need by just copying the member variables.
- Prototype pattern provides a mechanism to copy the original object to a new object and then modify it according to our needs. Prototype design pattern uses java cloning to copy the object.
- It guarantees unique instance and can help in performance issues.

Concept

- Avoids costly creation
- Avoids sub-classing
- Typically don't use keyword "new". First instance may be created by using keyword new but after that they are cloned.
- Although it can done without it, it is better to create an interface for prototype instance.
- Usually implemented with some sort of registry. The original object is created and kept in our registry. When another object is needed we create a clone from that registry.

Design

- Typically implements clone/cloneable method and interface.
- Has option to have shallow as well as deep copy.
- Java basic cloneable interface has no knowledge of generics and returns Object which needs to be cast that's why ppl recommend creating your own interface and implementing it.

Pitfalls

- Often not used, as not clear when to use.
- Mostly used with other patterns

Contrast

- Prototype deals with object initialization(by shallow or deep copy) which is then modified while prototype deals with dynamic object
- Both have a new object but factory object don't have initialized copied data.

Factory Pattern

- Opposite of Singleton Pattern

- Second most used design pattern after singleton.
- It is parameter driven and solves parameter driven construction.

Concepts

- It doesn't expose initialization logic.
- The client knows next to nothing even for the type of object created. It only knows about a common interface the client exposes.
- It does this by deferring the logic of initialization to subclasses.
- Examples:
    - Calender
    - ResourceBundle
    - NumberFormat

Design

- Factory is responsible for lifecycle (at least creation part) of object.
- Method to request an object is parameterized, which are used to determine which object to return.
- Common interface is used as placeholder for multiple classes that can be returned.
- A **factory class** has a **static factory() method** which returns a concrete instance based on business logic.
- Factory are usually not refactored into and are usually decided in design time only.

Contrast

Singleton	**Factory**
Returns same instance	Returns new instance
One constructor with no arguements	Compulsory a static factory method with min 1 arguement
No Interface/Subclass	Interface driven(interface or abstract class)

AbstractFactory Pattern

- It is Factory of factories
- Used to group a collection of factories together. <- **IMP**
- The factory is still responsible for lifecycle itself but has common interface that is carried throughout the factories.
- It is typically build using **Composition** (Difference for factory method).
- Ex: Document Builder.

Pitfalls

- Most complex of all creational design patterns
- It is a pattern within a pattern

- It usually happens that code starts with factories and is later refactored into abstract factory.
- It is very complex and highly abstracted.

Difference from Factory Pattern

- It actually hides the actual factory using another factory
- It is built through composition.