

# SPRING FRAMEWORK'ÜN KULLANDIĞI DESIGN PATTERN'LAR NELERDİR?

Spring Framework içerisinde kullanılan pattern'lar den bazıları ve kullanıldıkları yerler;

- **Factory Method:** BeanFactory bu pattern üzerine bina edilmiştir. Spring managed beanların yaratılması ve bağımlılıkların sağlanmasında kullanılır.
- **Prototype:** İstendiği takdirde ApplicationContext herhangi bir bean tanımından her `getBean('beanName')` ile erişimde yeni bir instance yaratmaktadır. Bean tanımı burada tam bir prototype örüntüsüdür.
- **Proxy:** Spring Application Framework'ün en çok faydalandığı pattern diyebiliriz. Scoped bean oluşturmada, Spring AOP kabiliyetinin sunulmasında, TX kabiliyetinin implemantasyonunda hep bu pattern kullanılmaktadır.
- **Template Method:** Spring'in veri erişim altyapısı bu pattern üzerine kuruludur. `JdbcTemplate`, `HibernateTemplate`, `JpaTemplate`, `RestTemplate`...
- **Observer:** ApplicationContext'in event yönetimi tam bir publish subscribe örneğidir. ApplicationContext bir event medium rolündedir. Bir grup spring managed bean ApplicationContext vasıtası ile `ApplicationListener` tipindeki diğer bir grup bean'lere event notifikasyonunda bulunabilirler.
- **Mediator:** Bir önceki pattern'da ApplicationContext'in event medium rolünde olduğunu söylemiştik. Bu sayede birbirleri ile haberleşmek isteyen bean'lar loosely coupled halde kalabilmektedirler. Sadece bildikleri ApplicationContext'in kendisidir, yani mediator.
- **Singleton:** Bazı şartlar altında bir sınıftan sadece bir nesnenin oluşturulması ve oluşturulan bu nesnenin tüm sistemde kullanılması gerekebilir. Örneğin bilgibankası için bir connection pool (bilgibankası bağlantı havuzu) sadece bir defa oluşturulmalı ve kullanılmalıdır. Bu durumda Singleton tasarım şablonu kullanılarak, bir sınıftan sadece bir nesnenin oluşturulması sağlanabilir.

Singleton yapıya sahip bir sınıfı inceliyelim:

```
public class Singleton
{
    /*
     * Singleton sınıfından oluşturulabilecek
     * tek nesne static sınıf degiskeni olarak
     * tanimlaniyor.
     */
    private static Singleton instance = null;

    /*
     * Double check locking yapabilmek
     * icin kullanilan nesne.
     */
    private static Object lock = new Object();

    /**
```

```

    * Başka sınıfların new Singleton()
    * şeklinde nesne oluşturmalarını,
    * sınıf konstruktörünü private
    * yaparak engellemiş oluyoruz.
    */
private Singleton()
{
    System.out.println("singleton init()");
}

/**
 * Singleton sınıfından oluşturulabilen
 * tek nesneye ulaşmak için instance()
 * metodu kullanılır.
 *
 * @return Singleton static Singleton nesnesi
 */
public static Singleton instance()
{
    if(instance == null)
    {
        // Double checked locking
        synchronized (lock)
        {
            if(instance == null)
            {
                instance = new Singleton();
            }
        }
    }
    return instance;
}

/**
 * Singleton sınıfında bulunan
 * bir metod.
 */
public void printThis()
{
    System.out.println(this);
}
}

```

---

Bir singleton sınıfın taşıması gereken bazı özellikler vardır. Bunlar:

- Sınıf konstruktörlerinin private olması gerekiyor. Konstruktörleri private olan bir sınıftan, başka bir sınıf new operatörü ile nesne oluşturamaz.
- Singleton sınıfından sadece bir tane nesne oluşturulması gerektiği için, oluşturulması gereken nesneyi sınıfın static değişkeni olarak tanımlamamız gerekiyor. Yukardaki örnekte private static Singleton instance = null; şeklinde bu tanımlamayı yapıyoruz.
- Singleton sınıfında instance() isminde static bir metodun olması ve bu metodun static olarak tanımlanmış nesneyi geri vermesi gerekiyor. instance() metodu içinde sınıfın tek nesnesi olacak değişken oluşturulur. Sınıf bünyesinde bulunan instance() static metodu büyük önem taşımaktadır. Amatörce yazılmış singleton sınıflarında genelde aşağıdaki gibi bir instance() metodu görülebilir:

```

public static Singleton instance()
{
    if(instance == null)
    {
        instance = new Singleton();
    }
    return instance;
}

```

---

Java multi threaded bir sistem olduğu için, yukardaki instance() metodunda new ile birden fazla Singleton nesne oluşturulabilir. Gözümüzde çalışan iki thread canlandırarak, bu sorunun nasıl oluştuğunu inceleyelim: Sistemde T1 ve T2 isimlerinde iki thread mevcut. T1 ilk olarak instance() metoduna girdi ve if(instance == null) satırını geçtikten sonra, java thread scheduler tarafından bloke edildi. Bu noktada T1 instance = new Singleton() yapmadan devre dışı kalmış oldu. Akabinde T2 çalışmaya başladı ve if(instance == null) satırını geçti ve zamanı yeterli olduğu için instance = new Singleton() yaptı. Zamanı dolduğu için T2 thread scheduler tarafından bloke edildi. Kontrol tekrar T1 threadine geçti. T1, T2'nin daha önce new Singleton() yaptığından habersiz olduğu için, instance = new Singleton() yaparak tekrar instance değişkenini oluşturdu. Bu durumda aynı nesne arka arkaya iki sefer oluşturulmuş oldu. Bu durum, sistem tarafından kullanılan kıymetli kaynakların oluşturulmasında sorun yaratabilir. Singleton sınıfı ve barındırdığı nesne sadece ve sadece bir defa oluşturulmalı. Yukarda yer alan sorunu ortadan kaldırmak için double checked locking1 mekanizması kullanılabilir. Bizim örneğimizde yer alan instance() metodu double checked locking kullanmaktadır:

```
/**
 * Singleton sinifindan olusturulabilen
 * tek nesneye ulasmak icin instance()
 * metodu kullanilir.
 *
 * @return Singleton static Singleton nesnesi
 */
public static Singleton instance()
{
    if(instance == null)
    {
        // Double checked locking
        synchronized (lock)
        {
            if(instance == null)
            {
                instance = new Singleton();
            }
        }
    }
    return instance;
}
```

---

Tekrar T1 ve T2 threadlerimizi kullanarak, nasıl double checked locking metodunun, iki sefer new Singleton() yapılmasını önlediğini görelim: T1 instance() metoduna girer ve if(instance == null) satırını değerlendirir. Burada T1 thread scheduler tarafından bloke edilebilir. Biz edilmediğini ve T1'in devam ettiğini düşünelim. T1 synchronized bloğuna girdiği andan itibaren, bu bloğa T2'nin girmesi imkansızdır. Şimdi T1'in synchronized bloğuna girdikten sonra bloke edildiğini düşünelim. Bu durumda T2 sadece synchronized metodunun önüne kadar gelip, T1 bu bloktan çıkana kadar beklemek zorundadır, çünkü T1 lock nesnesinin synchronized lock mekanizmasını elinde tutmaktadır. T1 bunu geri vermediği sürece, yani synchronized bloğundan çıkmadığı sürece, diğer threadler bu bloğa girmezler. T1 kontrolü tekrar eline aldıktan sonra, emin olmak için bir daha if(instance == null) satırı ile, nesnenin oluşturulup, oluşturulmadığını kontrol eder. Eğer T1 synchronized metoduna girmeden bloke olmuşsa, T2 bloğa girmiş ve new Singleton() yapmış olabilir. Tekrar kontrol edildiği için bu mekanizmanın ismi double checked locking'dir. Metoda ilk girişte if(instance == null) ile kontrol edilir ve kontrol synchronized bloğunda tekrarlanır. Bu şekilde singleton nesnesinin birden fazla init edilmesi önlenmiş olur. Bir test sınıfı ile, singleton nesnesi kontrol edilebilir.

```

package org.javatasarim.pattern.singleton;

/**
 * Singleton tasarim sablonu test sinifi
 *
 * @author Oezcan Acar
 */
public class Test
{
    /**
     * main
     * @param args
     */
    public static void main(String[] args)
    {
        for(int i =0; i < 10; i ++)
        {
            Singleton.instance().printThis();
        }
    }
}

```

---

Ekran çıktısı:

```

singleton init()
org.javatasarim.pattern.singleton.Singleton@1add2dd
org.javatasarim.pattern.singleton.Singleton@1add2dd
org.javatasarim.pattern.singleton.Singleton@1add2dd
org.javatasarim.pattern.singleton.Singleton@1add2dd
org.javatasarim.pattern.singleton.Singleton@1add2dd
org.javatasarim.pattern.singleton.Singleton@1add2dd
org.javatasarim.pattern.singleton.Singleton@1add2dd
org.javatasarim.pattern.singleton.Singleton@1add2dd

```

---

Ekran çıktısında görüldüğü gibi, singleton sınıfından sadece bir nesne oluşturuluyor. Bunu singleton init() çıktısı ile görüyoruz. Daha sonraki satırlarda aynı nesne kullanılarak, printThis() metodunun çıktısı ekrana geliyor