Tasarım Örüntüleri
Singleton

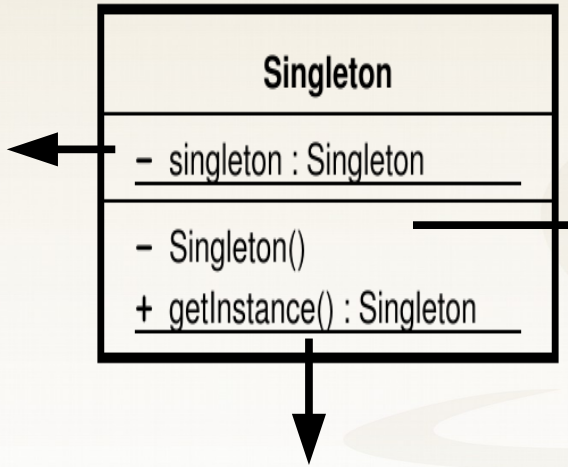
Örüntülerin Temel Prensipleri

- GoF tasarım örüntülerinin altında yatan **temel prensipler**
  - Encapsulation
  - Composition
  - Abstract Data Types

Singleton Örüntüsü

- Singleton
  - Bir sınıftan uygulama genelinde sadece **tek bir nesnenin yaratılması**nı garanti eder
  - Ayrıca bu nesneyi uygulama içerisinde **herhangi bir yerden de erişilebilir** kılar

Singleton Sınıf Diagramı



Nesne kendi sınıfı içinde **private static** bir değişkende tutulur

Constructor private yapılarak sınıf dışından herhangi bir biçimde başka instance oluşturulmasının önüne geçilir

public static getInstance() metodu ile uygulama içerisinde herhangi bir yerden bu instance'a erişmek mümkün olur

Java ve Singleton: Eager Initialization

```
public class Foo {
    public static final Foo INSTANCE = new Foo();

    private Foo() {
    }

    public void doSomeWork() {
        //...
    }
}
```

Java ve Singleton: Eager Initialization

```
public class Foo {
    private static final Foo INSTANCE = new Foo();

    private Foo() {
    }

    public static final Foo getInstance() {
        return INSTANCE;
    }

    public void doSomeWork() {
        //...
    }
}
```

Java ve Singleton: Lazy Initialization

```
public class Foo {
    private static Foo INSTANCE;

    private Foo() {
    }

    public synchronized static final Foo getInstance() {
        if(INSTANCE == null) {
            INSTANCE = new Foo();
        }
        return INSTANCE;
    }

    public void doSomeWork() {
        //...
    }
}
```

Java ve Singleton: Double Checked Locking Idiom


```
public class Foo {
     private static Foo INSTANCE;

     private Foo() {
     }

     public static final Foo getInstance() {
          if(INSTANCE == null) {
               synchronized (Foo.class) {
                    INSTANCE = new Foo();
               }
          }
          return INSTANCE;
     }

     public void doSomeWork() {
          //...
     }
}
```

Java ve Singleton: Double Checked Locking Idiom


```
public class Foo {
    private static Foo INSTANCE;

    private Foo() {
    }

    public static final Foo getInstance() {
        if(INSTANCE == null) {
            synchronized (Foo.class) {
                if(INSTANCE == null) {
                    INSTANCE = new Foo();
                }
            }
        }
        return INSTANCE;
    }

    public void doSomeWork() {
        //...
    }
}
```

Java ve Singleton: Double Checked Locking Idiom

```
public class Foo {
    private static volatile Foo INSTANCE;

    private Foo() {
    }

    public static final Foo getInstance() {
        if(INSTANCE == null) {
            synchronized (Foo.class) {
                if(INSTANCE == null) {
                    INSTANCE = new Foo();
                }
            }
        }
        return INSTANCE;
    }

    public void doSomeWork() {
        //...
    }
}
```

Java ve Singleton: Singleton Holder

```
public class Foo {

    private static final class FooHolder {
        private static final Foo INSTANCE = new Foo();
    }

    private Foo() {
    }

    public static final Foo getInstance() {
        return FooHolder.INSTANCE;
    }

    public void doSomeWork() {
        //...
    }
}
```

Java ve Singleton: Enum Singleton

```
public enum Foo {
    INSTANCE;

    public void doSomeWork() {
        //...
    }
}
```

Java ve Singleton: Serializable Singleton

```
public class Foo implements Serializable {

    private static final Foo INSTANCE = new Foo();

    private Foo() {
    }

    public static final Foo getInstance() {
        return INSTANCE;
    }

    public void doSomeWork() {
        //...
    }
}
```

Java ve Singleton: Serializable Singleton

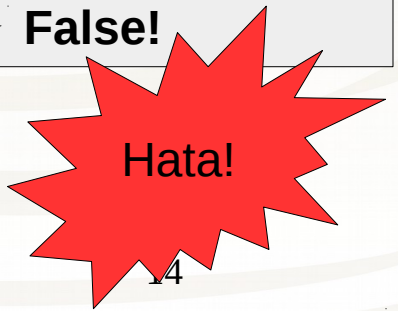
```
Foo f1 = Foo.INSTANCE;
ByteArrayOutputStream bout = new ByteArrayOutputStream();
ObjectOutputStream out = new ObjectOutputStream(bout);
out.writeObject(f1);

ByteArrayInputStream bin = new
ByteArrayInputStream(bout.toByteArray());
ObjectInputStream in = new ObjectInputStream(bin);

Foo f2 = (Foo) in.readObject();

System.out.println(f1 == f2);
```

→ **False!**



Java ve Singleton: Serializable Singleton


```
public class Foo implements Serializable {

     private static final Foo INSTANCE = new Foo();
     private Foo() {
     }
     public static final Foo getInstance() {
          return INSTANCE;
     }

     private Object readResolve()
                         throws ObjectStreamException {
          return INSTANCE;
     }
     private Object writeReplace()
                         throws ObjectStreamException {
          return INSTANCE;
     }

     public void doSomeWork() {
          //...
     }
}
```

Java ve Singleton: ClassLoaders

- Singleton nesneler **sınıf yükleme sırasında** yaratılırlar
- Sınıflar da **ClassLoader** nesneleri tarafından bir kereliğine yüklenir
- Web uygulamalarında ClassLoader hiyerarşilerinde **birden fazla ClassLoader**'ın aynı sınıfı tekrardan yüklemesi söz konusu olabilir

Java ve Singleton: ClassLoaders

- Dolayısı ile Singleton nesnenin **birden fazla kez yaratılması** söz konusu olabilir!
- Uygulama sunucusunda veya kütüphanelerde **yapılacak ayarlar** ile bu durumu engellemek gerekir!

Singleton Örüntüsünün Artı ve Eksileri

- Sistem genelinde bir sınıftan tek bir nesne olması **bazı senaryolar için çok önemli**dir
- Örneğin cache, dosya sistemi yöneticisi gibi yapılardan uygulama genelinde **tek bir tane olması** istenir
- Diğer yandan Singleton nesnenin **statik yaratımı ve statik erişimi birim testler** ile çalışmayı zorlaştırabilir

Spring ve Singleton

- Spring Application Framework'deki **bean tanımları**ndan genellikle tek bir nesne (bean) yaratılır
- Uygulamanın ömrü boyunca da **bu instance kullanılır** ve diğer bean'lara enjekte edilir
- Bu tür bean tanımlarına **singleton scope** adı verilmektedir
- Ancak Spring içerisinde aynı sınıftan **iki veya daha fazla bean tanımı** yapılabilir

Spring ve Singleton

- Dolayısı ile Spring içindeki Singleton örüntüsünün uygulanışı **bean tanımı düzeyinde**dir
- Başka bir ifade ile **singleton scope bir bean tanımı**ndan sadece tek bir bean instance yaratılır
- GOF kitabındaki Singleton tanımı ise **her sınıf için JVM veya ClassLoader düzeyinde**dir
- Ancak Spring'in singleton yaklaşımı testler açısından **daha efektif bir çözüm**dür

İletişim

JAVA Eğitimleri

Kurumsal Java Eğitimleri

www.java-egitimleri.com

info@java-egitimleri.com

@javaegitimleri

youtube.com/c/
KurumsalJavaEğitimleri