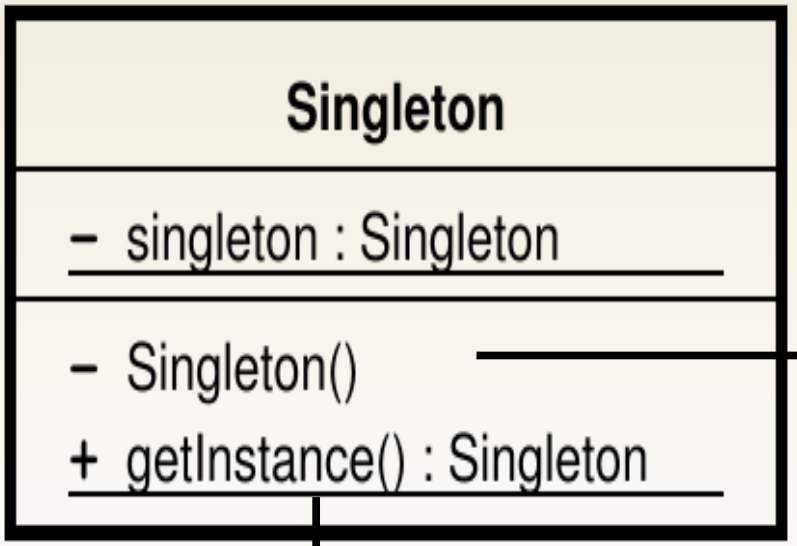
Singleton Örüntüsü

Singleton

- Bir sınıftan **uygulama genelinde tek bir nesne**nin olmasını sağlar
- Ayrıca bu nesneyi uygulama içerisinde **herhangi bir yerden de erişilebilir** kılar

Singleton Sınıf Diagramı



Nesne **private static** 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() {
        //...
    }
}
```

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

    private Foo() {
    }

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

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

Klasik eager initialization yöntemi ile singleton oluşturma örnekleridir

İlk örnekte statik final değişken public yapılarak dış dünyanın doğrudan erişimine açılmıştır

İkinci örnekte ise singleton instance static final getInstance() metodu ile erişilebilir kılınmıştır

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() {
        //...
    }
}
```

Burada ise getInstance() metodunda lazy initialization yapılamaktadır.
Metodun synchronized olması önemlidir!
Ancak bütün metodun synchronized yapılması maliyetlidir.

Java ve Singleton: Double Checked Locking Idiom

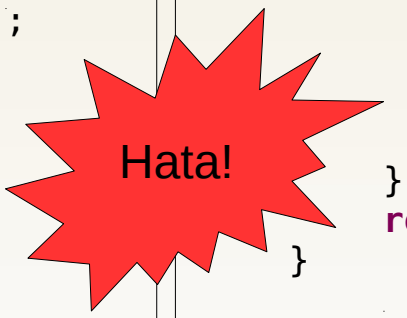

```
public class Foo {
     private static Foo INSTANCE;

     private Foo() {
     }

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

               }
          }
          return INSTANCE;
     }

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




```
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() {
          //...
     }
}
```

Bütün metodu synchronized yapmak yerine sadece singleton instance'ın yaratıldığı bölümü synchronized yapmak daha efektif bir çözüm olabilir. Ancak yukarıdaki iki çözüm de problemlidir!

Java ve Singleton: Double Checked Locking Idiom

JAVA Eğitimleri

```
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() {
        //...
    }
}
```

Foo içerisideki bütün diğer alanlarda volatile olmalıdır!

JVM'in shared/global hafıza alanının yanı sıra, her bir Thread'in kendine özel hafıza alanı(CPU/Thread cache) vardır. Thread'ler global hafıza alanındaki değerleri kendi cache'lerine kopyalarlar, cache üzerinde işlemleri yaparlar ve daha sonra cache'deki değişiklikleri global alana yazarlar. JVM'de nesne oluşturma, nesne adresinin değişkene yazılması gibi işlemlerde atomik olmayan birden fazla adımda yürütülebilirler. Örneğin, singleton instance'ın hafıza alanının ayrılması ve adresin değişkene atanması ve constructor'ın çalıştırılarak instance'ın yaratılması farklı adımlarla gerçekleşebilir. Dolayısı ile global hafıza alanında değişkenin NULL olmadığı ama nesne'nin de tam olarak construct edilmediği bir durumda diğer Thread bu değişkene erişip işlem yapabilir. Volatile anahtar kelimesi ile JVM'e değişkenin global hafıza alanında tutulacağı,okuma/yazmaların doğrudan buradan yapılacağı söylenebilir.

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'da inner sınıflar erişilmedikleri müddetçe yüklenmezler. Dolayısı ile FooHolder inner sınıfı da getInstance() metodu çağrılmadığı müddetçe yüklenmeyecek, böylece INSTANCE değişkeni de initialize olmayacaktır. Bu da Java'da az bilinen bir lazy initialization yöntemidir.

Java ve Singleton: Enum Singleton

```
public enum Foo {
    INSTANCE;

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

- Java 5 ile birlikte gelen Enum kabiliyeti ile tek instance'a sahip enum tipleri oluşturulabilir

Java ve Singleton: Serializable Singleton

- Singleton sınıfları **Serializable** veya **Externalizable** arayüzlerini implement edebilirler
- Serialize/deserialize işlemi ile Singleton instance'tan **yeni bir kopya** elde etmek mümkündür
- Bu durumda uygulama genelinde **tek bir instance kuralı ihlal edilmiş** olur
- Bunun önüne geçmek için **readResolve/writeReplace** metotları kullanılabilir

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() {
        //...
    }
}
```


```
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);
```

f1 ve f2 instance'larının birbirlerinden farklı oldukları görülecektir!

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() {
        //...
    }
}
```

ObjectInputStream'den okunan nesnenin bu metot tarafından dönülen nesne ile değiştirilmesini sağlar. Böylece deserialization sırasında oluşan farklı nesne yerine asıl singleton instance dönülebilir.

ObjectOutputStream'e yazılan nesnenin bu metot tarafından dönülen nesne ile değiştirilmesini sağlar. Böylece İstenirse asıl nesneden farklı bir Nesnenin asıl nesne yerine serialize edilmesi mümkündür.

Java ve Singleton: ClassLoaders

- JVM'de sınıflar **ClassLoader** nesneleri tarafından yüklenmektedir
- Dolayısı ile bir singleton sınıf **iki farklı ClassLoader nesnesi** tarafından yüklenirse **iki singleton instance** yaratılması kaçınılmazdır
- **Standalone uygulamalar**da genellikle tek ClassLoader olduğu için bu sorun teşkil etmez

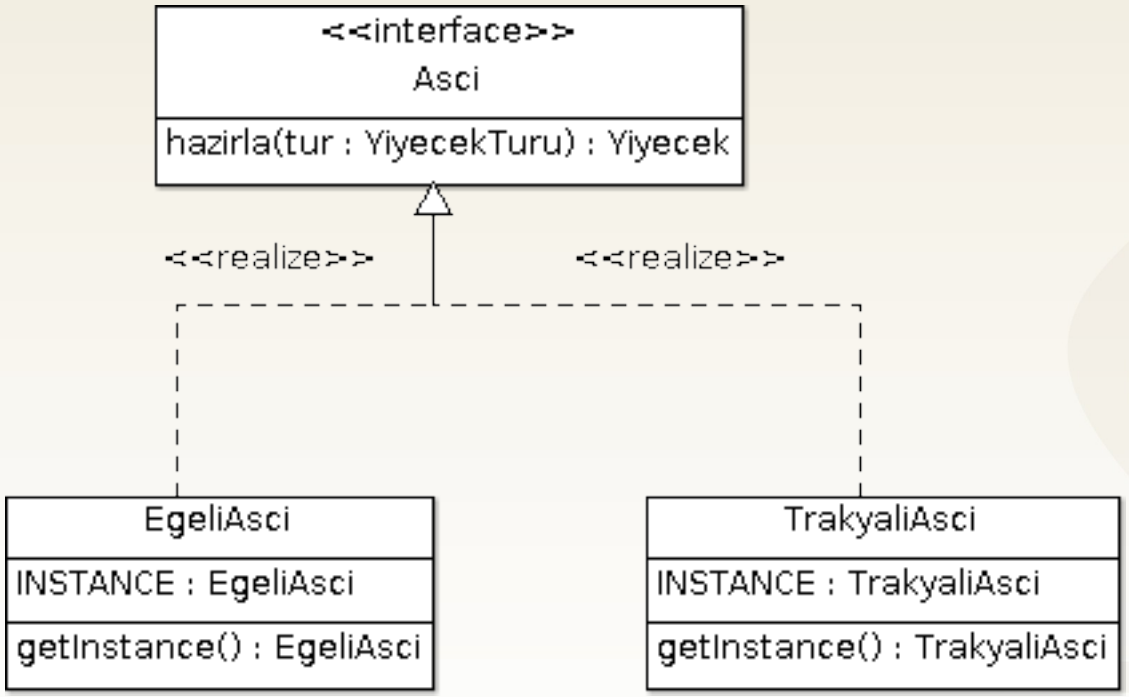
Java ve Singleton: ClassLoaders

- **Web uygulamaları**nda ise uygulama sunucusu birden fazla ClassLoader ile çalışmaktadır
- Bu yüzden **farklı ClassLoader instance'ları**nın aynı singleton sınıfı birden fazla yükleme ihtimali ortaya çıkabilir
- Web sunucusunda veya kütüphanelerde yapılacak düzenlemeler ile bu durumun **önüne geçilmesi** gerekir

LAB ÇALIŞMASI: Singleton

- Kafede her yöreden sadece bir tane aşçının olması sağlanmalıdır

LAB ÇALIŞMASI: Singleton



Singleton Örüntüsünün Sonuçları

- Sistem genelinde **bir sınıftan tek bir nesne** olmasını garanti eder
- Bazı senaryolar için **bu çok önemli**dir. Örneğin, cache, file sistem yöneticisi vb.
- **Birim testler** ile çalışmayı **zor**laştırabilir
- Tek instance'ı garanti etmek için **class loading** ve **serialization** gibi konuları da dikkate almak gerekir

İletişim

www.harezmi.com.tr

www.java-egitimleri.com

info@harezmi.com.tr

info@java-egitimleri.com

@HarezmiBilisim

@JavaEgitimleri