Java Design Patterns

Singleton in depth

One note first

- LAZY = when needed
- EAGER = immediately

Singleton Eager Initialization

```
package com.unipi.talepis.singletons;

public class EagerInitializedSingleton {
    private static  final  EagerInitializedSingleton instance =
            new EagerInitializedSingleton();
    private EagerInitializedSingleton(){}
    public static  EagerInitializedSingleton getInstance(){
        return  instance;
    }
}
```

Cons

- Does not support exception handling
- Lets modify it!

Singleton Eager Initialization v2

```
package com.unipi.talepis.singletons;

public class EagerInitializedSingletonV2 {
    private static EagerInitializedSingletonV2 instance;
    private EagerInitializedSingletonV2(){}
    static {
        try {
            instance = new EagerInitializedSingletonV2();
        }catch (Exception e){
        }
    }
    public static  EagerInitializedSingletonV2 getInstance(){
        return  instance;
    }
}
```

Cons

- It is eager…
- Lets build a lazy version!

Singleton Lazy Initialization

```
package com.unipi.talepis.singletons;

public class LazyInitializedSingleton {
    private static LazyInitializedSingleton instance;
    private LazyInitializedSingleton(){}
    public static LazyInitializedSingleton getInstance(){
        if (instance == null){
            instance = new LazyInitializedSingleton();
        }
        return instance;
    }
}
```

Cons

- Has serious problems in multithreading environments
- Lets fix this!

Singleton Lazy Initialization thread safe version

```
package com.unipi.talepis.singletons;

public class LazyInitializedSingletonV2 {
    private static LazyInitializedSingletonV2 instance;
    private LazyInitializedSingletonV2(){}
    public static LazyInitializedSingletonV2 getInstance(){
        if (instance == null){
            synchronized (LazyInitializedSingletonV2.class) {
                if (instance == null)
                    instance = new LazyInitializedSingletonV2();
            }
        }
        return instance;
    }
}
```

Important

- Please pay attention to the conditions inside getInstance() function
- It is called “Double-checked locking” and is used for optimization purposes

One more lazy design

- It seems more “sophisticated”
- Provides safe, highly concurrent lazy initialization with good performance
- Relies on the initialization phase of execution within the Java Virtual Machine (JVM) as specified by the Java Language Specification (JLS)

Singleton Lazy Initialization with on-demand holder

```
package com.unipi.talepis.singletons;

public class OnDemandSingleton {
    private OnDemandSingleton(){}
    private static class SingletonHolder {
        static OnDemandSingleton onDemandSingleton =
                new OnDemandSingleton();
    }
    public static OnDemandSingleton getInstance(){
        return SingletonHolder.onDemandSingleton;
    }
}
```

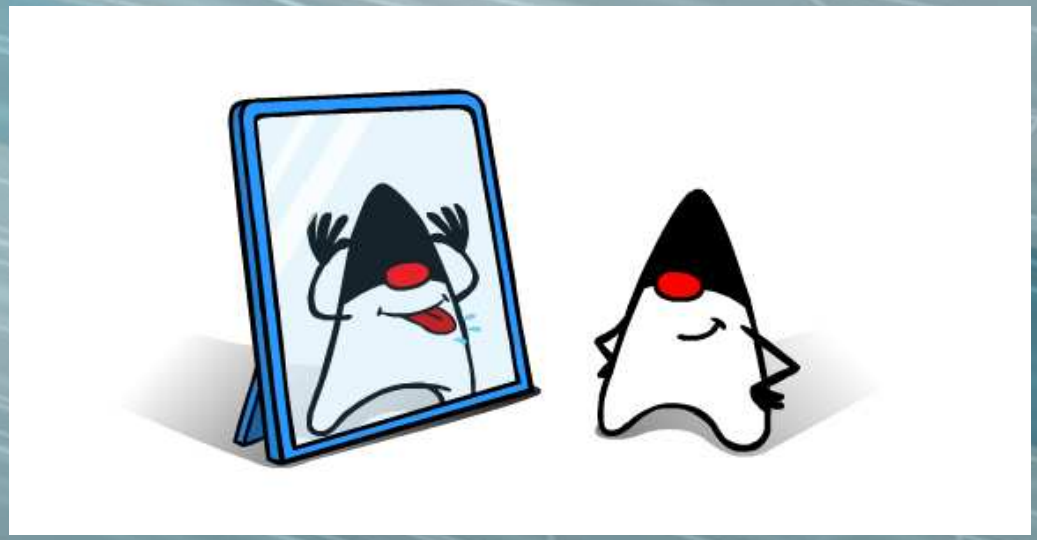
Ok, lets see the previous designs running!

```
System.out.println("Testing EagerInitializedSingleton");
EagerInitializedSingleton e1 = EagerInitializedSingleton.getInstance();
EagerInitializedSingleton e2 = EagerInitializedSingleton.getInstance();
System.out.println(e1.hashCode());
System.out.println(e2.hashCode());

System.out.println("Testing LazyInitializedSingleton");
LazyInitializedSingleton l1 = LazyInitializedSingleton.getInstance();
LazyInitializedSingleton l2 = LazyInitializedSingleton.getInstance();
System.out.println(l1.hashCode());
System.out.println(l2.hashCode());

System.out.println("Testing OnDemandSingleton");
OnDemandSingleton o1 = OnDemandSingleton.getInstance();
OnDemandSingleton o2 = OnDemandSingleton.getInstance();
System.out.println(o1.hashCode());
System.out.println(o2.hashCode());
```

```
"C:\Program Files\Java\jdk1.8.0_91\bin\java" ...
Testing EagerInitializedSingleton
1163157884
1163157884
Testing LazyInitializedSingleton
1956725890
1956725890
Testing OnDemandSingleton
356573597
356573597
Testing EnumSingleton
1735600054
1735600054

Process finished with exit code 0
```

So, are there Cons now?

- Of course…
- Actually, all the above designs suffer from:
  - Reflection!..



Reflection attacks Singletons

```
private static void attackToEager1(){
    EagerInitializedSingleton e1 = EagerInitializedSingleton.getInstance();
    EagerInitializedSingleton e2 = null;
    try {
        Constructor constructor =
             EagerInitializedSingleton.class.getDeclaredConstructor();
        constructor.setAccessible(true);
        e2 = (EagerInitializedSingleton) constructor.newInstance();
    } catch (NoSuchMethodException | IllegalAccessException |
       InvocationTargetException | InstantiationException e) {
    }
    System.out.println("Testing EagerInitializedSingleton");
    System.out.println(e1.hashCode());
    System.out.println(e2.hashCode());
}
```

```
"C:\Program Files\Java\jdk1.8.0_91\bin\java" ...
Testing EagerInitializedSingleton
1163157884
1956725890

Process finished with exit code 0
```

Discussion

- Reflection can “cause” a lot of problems, since it can change private fields to public and vice-versa
- Reflection can also be used in order to change private field values:
  - An already initialized singleton instance can be set to null!
- There are some countermeasures:
  - Final fields cannot be modified
  - A SecurityManager can help…
  - Lets do something about the constructors!

Eager Singleton that defends Reflection

```
package com.unipi.talepis.singletons;

public class EagerDefendReflection {
    private static final EagerDefendReflection instance =
            new EagerDefendReflection();
    private EagerDefendReflection(){
        if (instance != null) {
            throw new IllegalStateException("instance already created!");
        }
    }
    public static  EagerDefendReflection getInstance(){
        return  instance;
    }
}
```

Reflection attacks Singletons

```
private static void attackToEager2(){
    EagerDefendReflection ed1 = EagerDefendReflection.getInstance();
    EagerDefendReflection ed2 = null;
    try {
        Constructor constructor = EagerDefendReflection.class.getDeclaredConstructor();
        constructor.setAccessible(true);
        ed2 = (EagerDefendReflection) constructor.newInstance();
    } catch (NoSuchMethodException | IllegalAccessException |
            InvocationTargetException | InstantiationException e) {
        e.printStackTrace();
    }
    System.out.println("Testing EagerDefendReflection");
    System.out.println(ed1.hashCode());
    if (ed2!=null)
        System.out.println(ed2.hashCode());
    else
        System.out.println("it is null");
}
```

```
"C:\Program Files\Java\jdk1.8.0_91\bin\java" ...
Testing EagerDefendReflection
java.lang.reflect.InvocationTargetException
356573597 <1 internal call>
it is null <3 internal calls>
    at com.unipi.talepis.singletons.ReflectionAttack.attackToEager2(ReflectionAttack.java:32)
    at com.unipi.talepis.singletons.ReflectionAttack.main(ReflectionAttack.java:10)
Caused by: java.lang.IllegalStateException: instance already created!
    at com.unipi.talepis.singletons.EagerDefendReflection.<init>(EagerDefendReflection.java:8)
    ... 6 more

Process finished with exit code 0
```

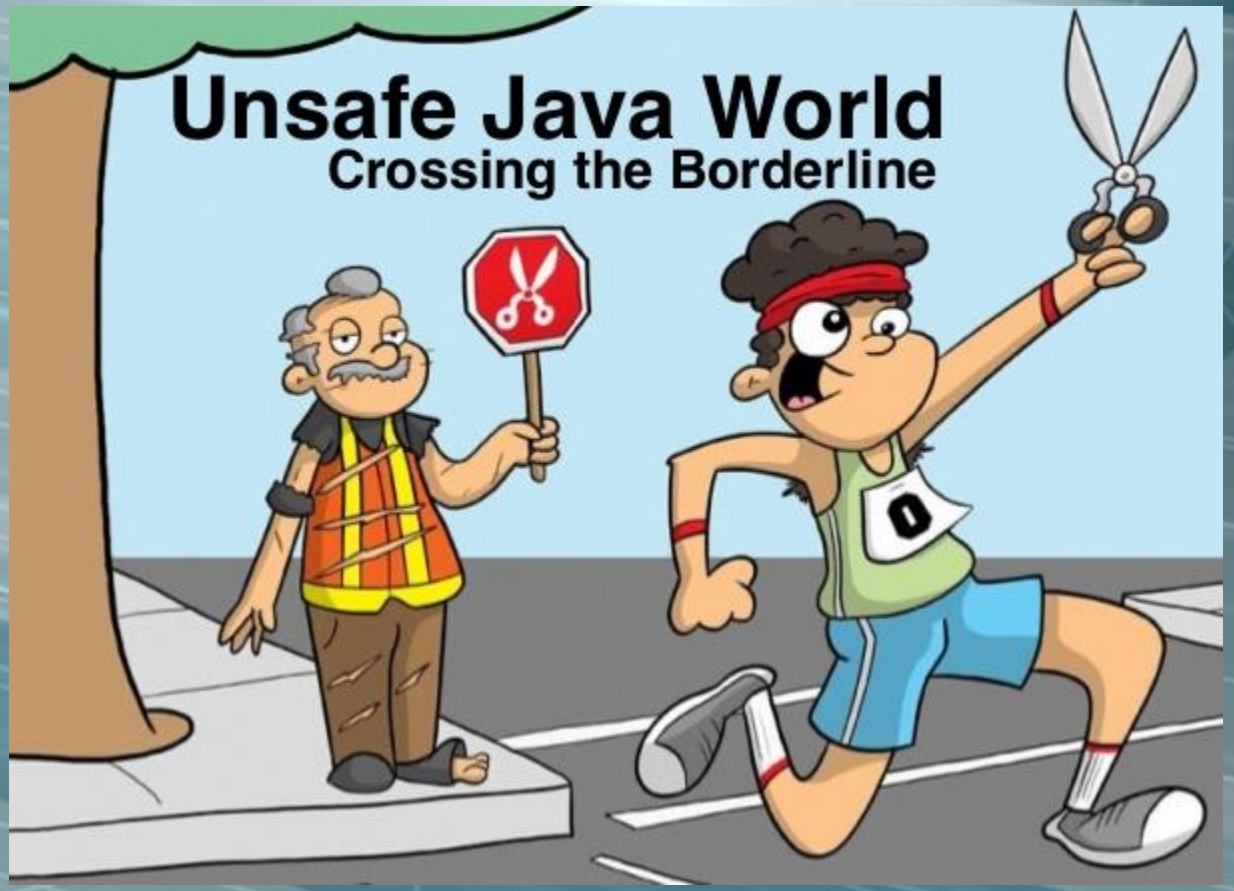
Are we ok now??

- Hmm, sorry
- Not yet!..
- You probably haven't heard about it
- But it is called:
  - Java Unsafe API…

Unsafe attacks Singletons

```
package com.unipi.talepis.singletons;
import sun.misc.Unsafe;
import java.lang.reflect.Field;

public class UnsafeAttack {
    public static void main(String[] args) {
        EagerDefendReflection ed1 = EagerDefendReflection.getInstance();
        EagerDefendReflection ed2 = null;
        try {
            Field f = Unsafe.class.getDeclaredField("theUnsafe");
            f.setAccessible(true);
            Unsafe unsafe = (Unsafe)f.get(null);
            ed2 = (EagerDefendReflection)unsafe.allocateInstance(EagerDefendReflection.class);
        } catch (NoSuchFieldException | IllegalAccessException | InstantiationException e) {
            e.printStackTrace();
        }
        System.out.println("Testing EagerDefendReflection");
        System.out.println(ed1.hashCode());
        System.out.println(ed2.hashCode());
    }
}
```

```
"C:\Program Files\Java\jdk1.8.0_91\bin\java" ...
Testing EagerDefendReflection
1956725890
356573597

Process finished with exit code 0
```



Do we surrender?

- Of course not!..
- Lets Utilize Java Enums!

Singleton through Enums

```
package com.unipi.talepis.singletons;

public enum EnumSingleton {
    INSTANCE;
}
```

What was that?

- Do you think that you can attack this?
- If yes, test it and e-mail me!

One more, final design pattern

- It is called Multiton
- In the “sense” of the Singleton, but with more objects mapped to a key
- Each object is unique for its key

Multiton Design Pattern

```
package com.unipi.talepis.singletons;
import java.util.HashMap;
import java.util.Map;

public class Multiton {
    private Multiton() {}
    private static final Map<String,Multiton> multitonInstance =
            new HashMap<>();
    public static Multiton getInstance(String s){
        Multiton instance = multitonInstance.get(s);
        if (instance == null) {
            synchronized (Multiton.class) {
                if (instance == null) {
                    instance = new Multiton();
                    multitonInstance.put(s, instance);
                }
            }
        }
        return instance;
    }
}
```

Compile and Run

```
private static void multitonExamples(){
    System.out.println("Testing Multiton");
    Multiton m1 = Multiton.getInstance("George");
    Multiton m2 = Multiton.getInstance("Maria");
    Multiton m3 = Multiton.getInstance("George");
    System.out.println(m1.hashCode());
    System.out.println(m2.hashCode());
    System.out.println(m3.hashCode());
}
```

```
"C:\Program Files\Java\jdk1.8.0_91\bin\java" ...
Testing Multiton
1163157884
1956725890
1163157884

Process finished with exit code 0
```