Singleton

Static Factory Method

```
public class MySingleton {

    private static MySingleton instance;

    private MySingleton() {
        System.out.println("Creating Singleton");
    }

    public static MySingleton getInstance() {
        if (instance == null) {
            instance = new MySingleton();
        }
        return instance;
    }
}
```

Singleton

Static Factory Method

Pros :

One advantage of the factory-method approach is that it gives you the flexibility to change your mind about whether the class should be a singleton without changing its API. The factory method returns the sole instance but could easily be modified to return. A second advantage, concerning generic types.

Cons :

More than one instance danger

Singleton

Static Final Field

```
public class MySingleton2 {

    public static final MySingleton2 INSTANCE;

    static {
        INSTANCE = new MySingleton2();
    }

    private MySingleton2() {
        System.out.println("Creating MySingleton 2");
    }
}
```

Singleton

Static Final Field

Pros :

The main advantage of the public field approach is that the declarations make it clear that the class is a singleton: the public static field is final, so it will always contain the same object reference.

Cons :

Doesn’t have advantages of static factory method.

- ***“Often neither of these advantages is relevant, and the final-field approach is simpler.”***
  Effective Java - item3

Singleton

Reflection Attack

```
MySingleton instance1 = MySingleton.getInstance();
MySingleton instance2 = null;
Constructor[] constructors =
        MySingleton.class.getDeclaredConstructors();
for (Constructor constructor : constructors) {
    // Below code will destroy the singleton pattern
    constructor.setAccessible(true);
    instance2 = (MySingleton) constructor.newInstance();
    break;
}

System.out.println(instance1.hashCode());
System.out.println(instance2.hashCode());
```

Singleton

Serialization Attack

```
MySerializableSingleton instance1 = MySerializableSingleton.getInstance();
ObjectOutput out
        = new ObjectOutputStream(new FileOutputStream("file.text"));
out.writeObject(instance1);
out.close();

// deserialize from file to object
ObjectInput in
        = new ObjectInputStream(new FileInputStream("file.text"));

MySerializableSingleton instance2 = (MySerializableSingleton) in.readObject();
in.close();

System.out.println(instance1.hashCode());
System.out.println(instance2.hashCode());
```

Singleton

Clone Attack

```
MyCloneableSingleton instance1 = MyCloneableSingleton.getInstance();
MyCloneableSingleton instance2 = (MyCloneableSingleton) instance1.clone();

System.out.println(instance1.hashCode());
System.out.println(instance2.hashCode());
```

Singleton

Overcome Reflection Attack

```
public enum ReflectionProtectedSingleton {
    INSTANCE;

    /**
     * enum constructor is private by default
     * below code is for the sake of explicit
     */
    private ReflectionProtectedSingleton() {
        System.out.println("Creating ReflectionProtectedSingleton");
    }

    public void foo() {
        System.out.println("foo");
    }

    public void bar() {
        System.out.println("bar");
    }
}
```

Singleton

Overcome Reflection Attack

To overcome issue raised by reflection, enums are used because java ensures internally that enum value is instantiated only once. Since java Enums are globally accessible, they can be used for singletons. Its only drawback is that it is not flexible i.e it does not allow lazy initialization. We can’t invoke enum constructors by ourself. JVM handles the creation and invocation of enum constructors internally. As enums don’t give their constructor definition to the program, it is not possible for us to access them by Reflection also. Hence, reflection can’t break singleton property in case of enums.

Singleton

Overcome Serialization Attack

```java
public class SerializationProtectedSingleton implements Serializable {
    private static SerializationProtectedSingleton instance;

    private SerializationProtectedSingleton() {
        System.out.println("Creating SerializationProtectedSingleton");
    }

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

    /**
     * For Serializable and Externalizable classes,
     * the readResolve method allows a class to replace/resolve the
     * object read from the stream before it is returned to the caller.
     * By implementing the readResolve method, a class can directly control
     * the types and instances of its own instances being deserialized
     */
    protected Object readResolve(){
        return instance;
    }
}
```

Singleton

Overcome Serialization Attack (readResolve close up)

```
    /**
     * For Serializable and Externalizable classes,
     * the readResolve method allows a class to replace/resolve the
     * object read from the stream before it is returned to the caller.
     * By implementing the readResolve method, a class can directly control
     * the types and instances of its own instances being deserialized
     */
    protected Object readResolve(){
        return instance;
    }
```

Singleton

Overcome Cloning Attack

```
public class CloningProtectedSingleton extends CloneableSuperClass {

    private static CloningProtectedSingleton instance;

    private CloningProtectedSingleton() {
        System.out.println("Creating Singleton");
    }

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


    @Override
    protected Object clone() throws CloneNotSupportedException {
        throw new CloneNotSupportedException();
    }
}
```

Singleton

Codes and Slides :
https://github.com/zeynaliAli/Singleton

ali.zzeynail@gmail.com