Design Patterns

- They describe recurring solution approaches used to solve common problems encountered in the software development process.

- They are divided into three groups, these are:
    1. **Creational Patterns:** These patterns are used to determine how objects will be created, constructed, and managed.

    2. **Structural Patterns:** These patterns are used to define relationships between classes and objects and show how they can be put together.

    3. **Behavioral Patterns:** These patterns define how objects will work and cooperate.

Creational Patterns

- These patterns abstract object creation and prevent object creation codes from being spread around.

- There are five, these are:

    1. **Singleton:** It is used to create a class that has only one instance and to access this instance from a single point.

        Examples of situations where it can be used include database connection, resource management (file processing, network connections), cache, thread pool, logging.

        To create a singleton class, these steps can be followed:
        1. Make the class constructor private.
        2. To use the class, define the object to be accessed from outside as “private static ClassName variableName”.
        3. Create the method that returns the object.

```java
/*
    The preceding implementation works fine in the case of the single-threaded environment,
    but when it comes to multi-threaded systems, it can cause issues if multiple threads are inside the condition at the same time.
 */
3 usages
public class LazyInitializedSingleton {

    3 usages
    private static LazyInitializedSingleton instance;

    1 usage
    private LazyInitializedSingleton(){}

    no usages
    public static LazyInitializedSingleton getInstance() {
        if (instance == null) {
            instance = new LazyInitializedSingleton();
        }
        return instance;
    }

}
```

Figure 1.1 LazyInitializedSingleton Class

When multi-threaded is used, the structure in ***Figure 1.1*** **will cause problems because it is not synchronized.** We can overcome this by making the getInstance() method synchronized.

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

Figure 1.2 Synchronized getInstance()

Currently, our method works properly, and we do not encounter any problems. However, when you think about it a little, this solution is very slow and contains optimization problems.

We use synchronized to prevent the function in Figure 1.2 from entering the if block only in the first few calls. In other words, we make this check throughout the life cycle of the program for any problems that may occur in the first few calls. This creates a huge cost since we synchronize the function.

We can overcome this problem with ***DoubleLocking*** as follows.

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

Figure 1.3 Double Locking getInstance()

Here we use an if block for the instance variable, which will be null in the first few thread calls. Then, we determine a synchronized block for the threads that can enter and ensure that they continue the process sequentially. Then, the first thread that comes to the second if block creates the object, and other threads that take the lock cannot enter this block. In this way, the function is synchronized for the first few thread calls, and since the instance cannot be null in the remaining calls, the threads are not synchronized, thus ensuring optimization.

Bill Pugh Singleton Approach

- This method is used to create a thread-safe singleton class in a lazy way and without using any synchronized mechanism.

```
public class BillPughSingleton {

    1 usage
    private BillPughSingleton(){}

    1 usage
    private static class SingletonHelper {
        1 usage
        private static final BillPughSingleton INSTANCE = new BillPughSingleton();
    }

    no usages
    public static BillPughSingleton getInstance() {
        return SingletonHelper.INSTANCE;
    }
}
```

Figure 1.4 BillPughSingleton Class

Here, when the ***BillPughSingleton.getInstance()*** method is called by any thread for the first time, the SingletonHelper class is created, and the INSTANCE variable is created and assign operation is performed. It is then returned. When other threads access it, only the INSTANCE variable is returned.

This is related to JVM's class loading and creation mechanism. Therefore, we create both a lazy and thread-safe singleton without any additional processing.

Disabling Singleton Pattern with Reflection

- Reflection can be used to disable all previous singleton implementation approaches and produce a new object.

```
public class ReflectionSingletonTest {

    public static void main(String[] args) {
        EagerInitializedSingleton instanceOne = EagerInitializedSingleton.getInstance();
        EagerInitializedSingleton instanceTwo = null;
        try {
            Constructor[] constructors = EagerInitializedSingleton.class.getDeclaredConstructors();
            for (Constructor constructor : constructors) {
                // This code will destroy the singleton pattern
                constructor.setAccessible(true);
                instanceTwo = (EagerInitializedSingleton) constructor.newInstance();
                break;
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
        System.out.println(instanceOne.hashCode());
        System.out.println(instanceTwo.hashCode());
    }

}
```

Figure 1.5 ReflectionSingletonTest Class

*To handle disabling with Reflection, **Enum Singleton** can be used.*

Enum Singleton

- Since Enums in Java are classes and can have their own constructors and methods, they can be used as a singleton class.

  Due to the structure of the enum, while enum classes are loaded into memory, each enum value is loaded into memory by calling its constructor. Thus, the singularity situation is ensured due to the structure of the enum.

  Enum values **cannot be recreated with reflection**. However, this usage has a slight disadvantage because **it is not lazy**.

```
public enum EnumSingleton {

    no usages
    INSTANCE;

    1 usage
    public static void doSomething() {
        // do something
    }
}
```

Figure 1.6 EnumSingleton Class

Serialization and Singleton

- Sometimes we need to implement the Serializable interface in the singleton class so that we can store the singleton in the file system and retrieve it at a later time.

Since a new object will be produced in the Deserialization process, we can prevent this with the **readResolve()** method.

```
public class SerializedSingleton implements Serializable {

    no usages
    private static final long serialVersionUID = -7604766932017737115L;

    1 usage
    private SerializedSingleton(){}

    1 usage
    private static class SingletonHelper {
        1 usage
        private static final SerializedSingleton instance = new SerializedSingleton();
    }

    2 usages
    public static SerializedSingleton getInstance() {
        return SingletonHelper.instance;
    }

    no usages
    @Serial
    protected Object readResolve() {
        return getInstance();
    }

}
```

Figure 1.7 SerializedSingleton Class